**《操作系统》实验指导书**

**袁健 张幸 魏赟**

**光电信息与计算机工程学院计算机工程系**

**2018年 9月**

目 录

[操作系统实验大纲 1](#_Toc468960814)

[实验目的 1](#_Toc468960815)

[实验教学的基本要求 1](#_Toc468960816)

[实验报告要求 2](#_Toc468960817)

[实验项目与学时分配 2](#_Toc468960818)

[实验一 进程调度 3](#_Toc468960819)

[实验二 资源分配算法 8](#_Toc468960820)

[实验三 存储管理分区分配算法 12](#_Toc468960821)

[实验四 Linux的安装与使用 16](#_Toc468960822)

[实验五 Linux的常用命令操作 23](#_Toc468960823)

[实验六 Linux的Vi操作 31](#_Toc468960824)

[实验七 Linux下的C语言编程 36](#_Toc468960825)

[实验八 Linux的Shell编程 43](#_Toc468960826)

[实验九 Linux的进程管理 58](#_Toc468960827)

[实验十 Linux的进程通信 74](#_Toc468960828)

[实验十一Linux的内存管理 94](#_Toc468960829)

操作系统实验大纲

## 实验目的

实验是操作系统原理课程中不可缺少的重要教学环节，实验目的是使学生理论联系实际，使学生在实践探索中去发现问题，去解决问题，提高学生获取知识和应用技术的能力，培养学生分析问题和解决问题的能力。

操作系统原理课程要求理论与实践相结合，本门实验课程就是对理论知识的一个补充，与理论学习起着相辅相成的作用。通过本实验课程的学习，借助实例分析，使同学了解计算机系统的工作过程，资源管理策略以及并发活动的处理方法，以便为今后的课程学习和高级程序设计（如进程，线程，同步）打好基础。

## 实验教学的基本要求

1. 进一步熟悉和掌握结构化程序设计语言编制方法。
2. 进一步熟悉和掌握汇编语言程序设计方法。
3. 掌握操作系统进程的调度算法。
4. 掌握操作系统存储管理分区分配算法。
5. 加深对I/O传输控制的理解。
6. 学习Linux操作系统下的常用命令的操作。比如文件的基本操作，使用vi编辑器编辑文件，在Linux下进行C语言的编程和调试。
7. 了解和熟悉Linux下的Shell编程。
8. 学习Linux下进程管理命令，理解进程创建的基本原理。
9. 理解Linux下如何编程利用信号量实现进程同步和进程通信的原理。

10、学习Linux内存管理命令，理解页式管理页面置换算法的程序。

## 实验报告要求

每个实验均应编写实验报告，学生实验后要写出严谨的，实事求是、文字通顺的实验报告。实验报告应包括以下内容：

1、实验题目

2、实验目的

3、实验内容（流程图、源程序清单并附上注释）

4、实验结果及分析

* + - 运行结果（必须是上面程序运行输出的结果）。
    - 如果程序未能通过，则应分析其原因。
    - 调试本程序的心得体会。

## 实验项目与学时分配

|  |  |  |
| --- | --- | --- |
| 序号 | 实验项目 | 学时 |
| 1 | 进程调度 | 4 |
| 2 | 存储管理调度算法 | 3 |
| 3 | 资源分配算法 | 3 |
| 4 | Linux的安装与使用 | 2 |
| 5 | Linux常用命令操作 | 2 |
| 6 | Linux的vi操作 | 4 |
| 7 | Linux下的C语言编程 | 4 |
| 8 | Linux的Shell编程 | 4 |
| 9 |  |  |
| 10 |  |  |
| 11 |  |  |
| 合计 |  | 26 |

实验一 进程调度

实验二 资源分配算法

实验三 存储管理分区分配算法

实验四 Linux的安装与使用

**一、 实验目的**

了解虚拟机软件的功能和特点，学习和动手安装linux 操作系统，了解Linux的目录结构。

**二、预备知识**

**1、Linux简介**

Linux操作系统是一个基于POSIX和UNIX的多用户、多任务、支持多线程和多CPU的操作系统。它是一个类UNIX的操作系统，能免费使用并能自由传播。它诞生于1991年的10月5日，这一天Linux0.01版正式向外公布，后来借助于Internet的传播，并经过全世界各地众多编程高手的共同改进、扩充和完善，现已成为世界上使用最多的一种UNIX操作系统，并且使用人数还在不断增长。

Linux内核并非是操作系统，它只是一个完整操作系统的组成部分，负责提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。

Linux内核分为稳定版和开发版。一些新特性、实验性改进等首先在开发版中进行，在开发版中测试以后，再在稳定版中进行相同的修改。一旦开发版经过足够的发展，开发版就会成为新的稳定版。两种版本相互关联，相互循环。

Linux内核版本号格式如major.minor.patchlevel，其中各部分说明如下。

（1）major：表示主版本号，有结构性变化时才变更。

（2）minor：表示次版本号，新增功能时才发生变化；一般奇数表示测试版，偶数表示稳定版。

（3）patch：表示对次版本的修订次数或补丁包数。

Linux发行版指的是通常所说的“Linux操作系统”，它是各个公司或组织推出的版本。一个典型的Linux发行版包括：Linux内核，一些GNU库和工具，命令行shell，图形界面的X窗口系统和相应的桌面环境（如KDE或GNOME），并包含数千种从办公包、编译器、文本编辑器到科学工具的应用软件。发行版一般为不同的目的而制作，包括对不同计算机结构的支持，对一个具体区域或语言的本地化，甚至针对一些实时应用和嵌入式系统。

Linux主流的发行版本有Ubuntu、Fedora、openSUSE、Linux Mint、CentOS、RedHat，本实验主要在RedHat上进行。

RedHat（www.redhat.com）是全球最大的开源技术厂家，专注于服务器版的开发，其产品Red Hat Enterprise Linux也是全世界应用最广泛的Linux。RedHat公司总部位于美国北卡罗来纳州，在全球拥有22个分部。RedHat Linux因其易于安装而闻名，在很大程度上减轻了用户安装程序的负担，其中RedHat提供的图形界面安装方式非常类似Windows系统的软件安装。这对于某些Windows用户而言，几乎可以像安装Windows系统一样轻松安装RedHat发行套件。2004年，Red Hat公司正式停止对Red Hat9.0版本的支持，标志着Red Hat Linux的正式完结，而桌面版的Red Hat Linux发行包则与来自民间的Fedora计划合并，Red Hat公司不再开发桌面版的Linux发行包，而将全部力量集中在服务器版的开发上，也就是Red Hat Enterprise Linux版。2011年11月10日，RHEL6正式版发布。

**2、Linux目录结构**

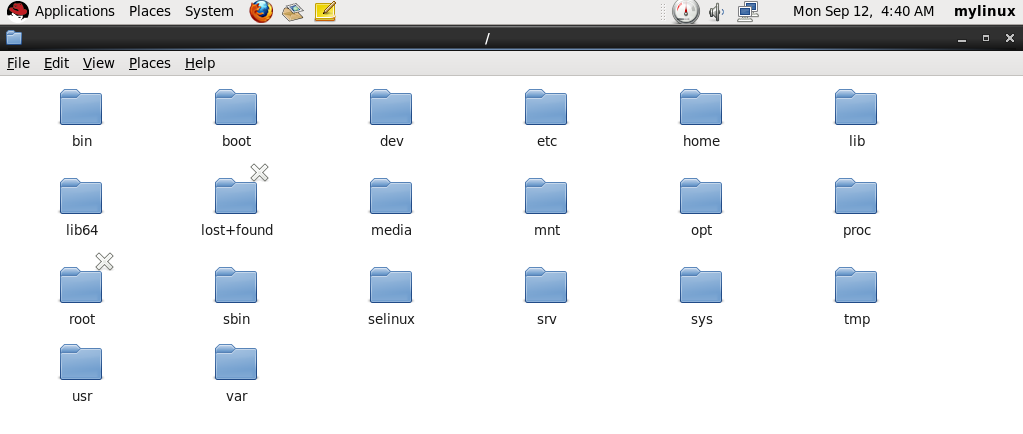


图4-1 RHEL6目录示意图

（1）/bin： 也称二进制（binary）目录，包含了那些供系统管理员和普通用户使用的重要的Linux命令的可执行文件。

（2）/sbin、/usr/sbin、/usr/root/sbin: 存放了系统管理的工具、应用软件和通用的root用户权限的命令，只有系统管理员可以使用这些目录下的程序。

（3）/boot: 存放系统启动时要用到的程序，包括Linux内核的二进制映像。

（4）/dev：设备（device）的英文缩写。在这个目录下存放了所有Linux系统中使用的外部设备，每个设备都是以文件的形式存在，包括键盘和鼠标，但是这里没有存放外部设备的驱动程序。

（5）/etc：Linux系统中最重要的目录之一。这个目录下存放了系统管理时要用到的各种配置文件和子目录。

1）/etc/init.d: 用来存放系统或服务器以System V模式启动的脚本，还在以System V模式启动或初始化的系统中常见，如Fedora/RedHat。

2）/etc/xinit.d:如果服务器是通过xinetd模式运行的，它的脚本要放在这个目录下。有些系统没有这个目录，如Slackware，有些老的版本也没有。在RedHat/Fedora中比较新的版本中存在。

3）/etc/X11是X-Window相关的配置文件存放地。

（6）/home:存放用户的主目录。

（7）/lib:用来存放系统动态连接共享库。

（8) /mnt：主要用来临时挂载文件系统的目录，系统管理员运行mount命令可以完成不同存储设备的挂载工作。

（9）/opt：用来安放临时测试的软件包。

（10）/proc：一个虚拟的目录，是系统内存的映射，可以通过直接访问这个目录来获取系统信息，即这个目录在内存里而不在硬盘上。

（11）/root：根用户的主目录。

（12）/tmp：用来存放不同程序执行时产生的临时文件。

（13）/usr：Linux文件系统中最大的目录之一，系统中要用到的应用程序和文件几乎都在这里。

1）/usr/local：用来存放用户自编译安装软件的存放目录。

2）/usr/lib：是库文件的存储目录。

3）/usr/share：系统共用的文件存放目录。

4）/usr/src：内核源码存放的目录。

（14）/lost+found：一般是空的，当系统不正常关闭后，目录中就会出现没有关联的文件，这些文件可以用Linux工具fsck进行检查。

（15）/var：用来存放易变的数据，这些数据在系统运行过程中不断变化。

（16）/media：作为移动存储设备默认挂载点，如光盘。

（17）/sys：被建立在内存中的虚拟文件系统。

**三、 实验内容**

**1、 安装VMware创建虚拟机**

（1）运行VMware workstation 12.0，选择左上角菜单文件-新建虚拟机，或者直接按Ctrl+N快捷键进入创建虚拟机向导。

（2）在弹出的窗口中选择自定义（高级）安装，点击下一步。

（3）默认，继续点击下一步，选择稍后安装操作系统，点击下一步。

（4）客户机操作系统选择Linuux，版本选择centos 64位，点击下一步，命名名称和选择位置，点击下一步。

（5）默认，点击下一步。

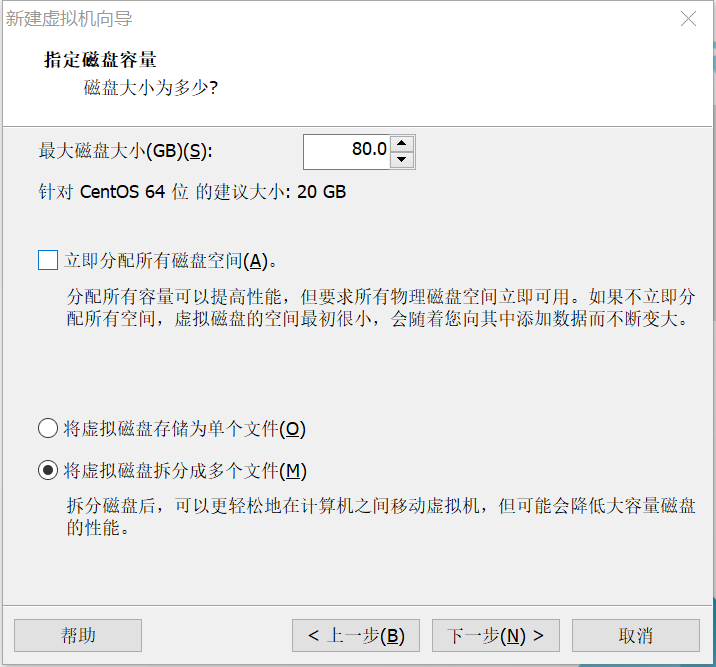
（6）把虚拟机内存大小改为2048MB，建议使用内存不超过最大建议使用内存，点击下一步。

（7）选择“使用桥接网络”，点击下一步。

（8）后面两步默认选择，点击下一步。

（9）选择“创建新虚拟磁盘”，点击下一步。

（10）把最大磁盘大小改为80GB（默认20G一般不够使用，建议设置略大一些），选择“将虚拟磁盘拆分为多个文件”，点击下一步。

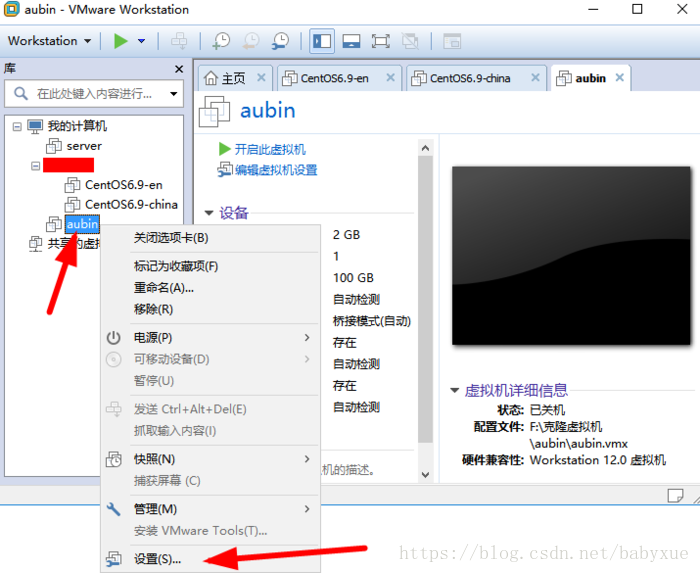


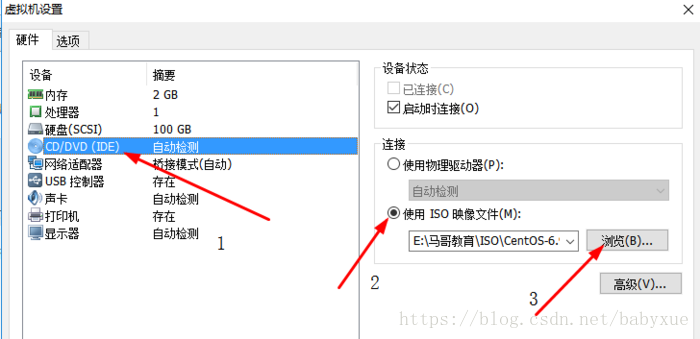
（11）默认，点击下一步

（12）默认，点击完成。

**2、 安装centos**

（1） 打开步骤1中创建的虚拟机软件，在其上安装centos。



右击刚创建的虚拟机，选择设置。先选择CD/DVD，再选择使用ISO映像文件，最后选择下载好的镜像文件，点击确定。

（2）开启虚拟机，选择第一项“Install CentOS 7 ”，安装直接CentOS 7，回车，选择安装语言为简体中文，点击继续，

（3）设置时间，默认上海，点击“软件选择”，选择“Gnome 桌面”，点击完成

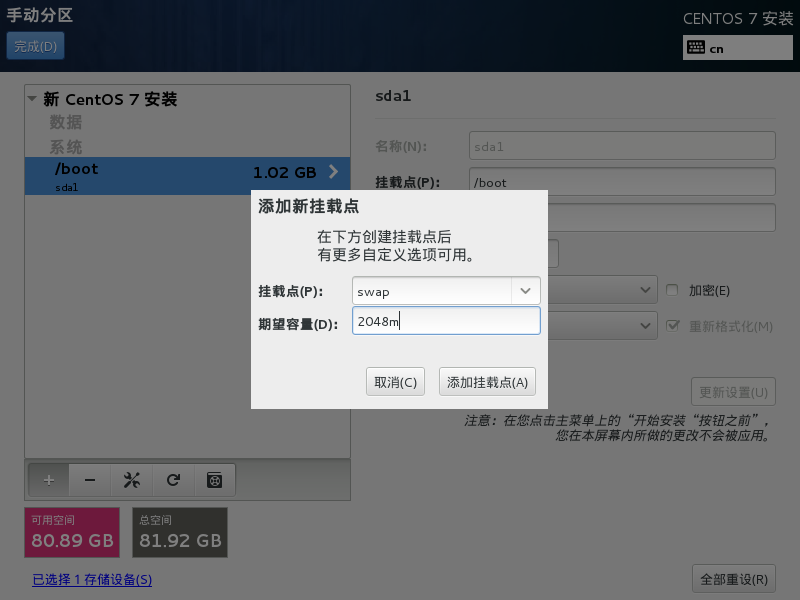


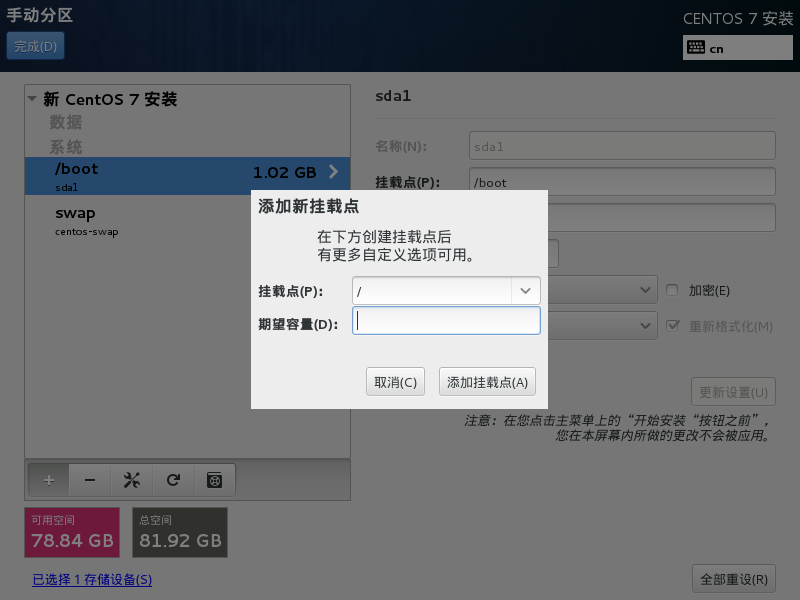
选择“安装位置”，选择“我要配置分区”，点击完成

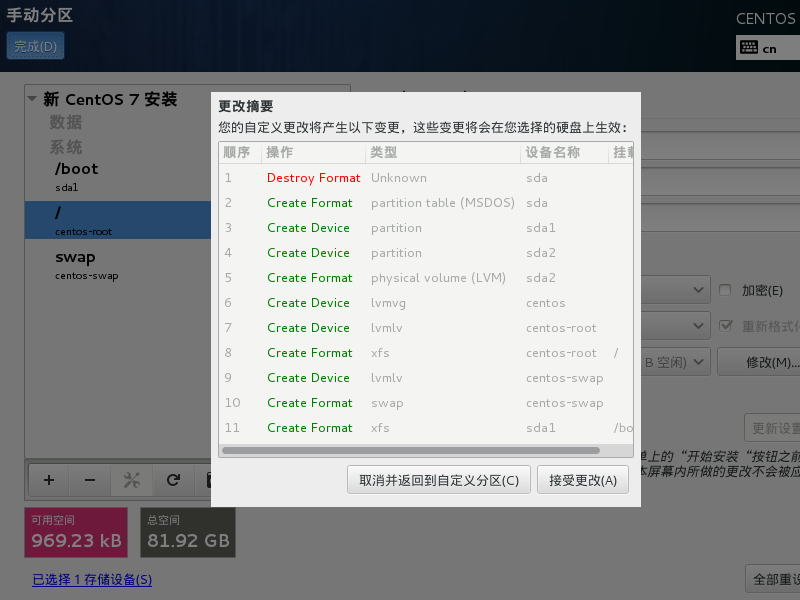


点击左下“+”号，选择/boot，给boot分区分1024M。最后点击“添加挂载点”

然后以同样的办法给其他两个区分配好空间后点击完成，其中swap分2048m，/分剩余所有空间（无需输入期望容量），之后再弹出窗口点击“接受更改”。







（4）点击“网络和主机名”，可设置主机名，点击完成，然后点击“开始安装”。

（5）设置root密码，继续创建用户（如果密码过于简单，需要点击两次完成），等待系统安装完毕。

**注**：关于图形界面和命令行界面之间切换的问题如下：

从Linux的命令界面切换到[图形界面](https://www.baidu.com/s?wd=%E5%9B%BE%E5%BD%A2%E7%95%8C%E9%9D%A2&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPAnznyfkPjnsrAFWuj0L0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0sPjnYrH6zrj0knWf4rHcY)可以按Ctrl+Alt+F1进行切换；从图形界面切换到命令界面可以按Ctrl+Alt+F2进行切换。

**3、安装****VMware Tools for Linux**

（1）在VMware中启动linux虚拟机。找菜单虚拟机/设置/添加CD光驱，将C:\programfiles(X86)\VMware\linux.iso（该文件在VMware的安装目录下查找）选中后确定。则在linux桌面上出现了一个装载了VMware Tools的虚拟光驱（事先用root登录）。

（2）选中该光盘，直接打开光盘文件，选中VMwareTools-9.6.0-1294478.tar.gz 右键选择Extract To..（提取到），在弹出的目录列表中选择File System（文件系统）下面的tmp，解压到tmp文件下。

（3）再输入命令cd /tmp/vmware-tools-distrib

（4）继续输入下面命令，按Enter键，选择默认值安装即可：

./vmware-install.pl，之后一路回车，完成后重启。（在终端用命令安装）

在实验室安装好的linux系统登录时，用户名：centos

密码：centos

实验五 Linux的常用命令操作

**一、 实验目的**

了解linux命令格式，掌握文件管理的常用命令。

**二、预备知识**

**Linux命令简介**

命令行界面始终是Linux与UINX强大的操控基础，这与Windows OS的操控原理有很大的不同。Windows在Windows 2000以后就彻底成为一个完全图形化的操作系统。虽然在Linux文件中存在GNOME和KDE桌面环境，但它们只能算作是Linux上的一个应用，而不可能与Linux内核高度整合。因此，在Linux的学习中，Linux

命令的学习尤为关键。而命令的执行实际上是通过Shell来完成的，Shell是用户和操作系统内核之间的接口，在图形化界面下Shell界面一般通过“终端”来体现。

Linux命令行的语法结构如下所示。

$commad[[-]option(s)][option argument(s)][command argument(s)]

语法结构含义如下。

（1）$：Linux系统普通用户的命令提示符，root根用户的命令提示符为‘#’。

（2）Command：Linux命令的名字。

（3）[[-]option(s)]：改变命令行为的一个或多个修饰符，即选项。

（4）[option argument(s)]：选项的参数。

（5）[command argument(s)]：命令的参数。

一般来说，Linux命令主要包括以下9类。

（1）文件及目录管理命令，如ls、cd、cat、grep、touch、pwd、od等。

（2）磁盘及设备管理命令，如mount、umount、df、du、fsck、dd等。

（3）进程管理命令，如ps、top、kill、bg、fg等。

（4）用户管理及权限管理命令，如chmod、su、useradd、userdel等。

（5）系统管理及维护命令，如date、cal、shutdown、clear、echo等。

（6）网络通信命令，如netstat、ifconfig、ping、telnet等。

（7）数据备份命令，如bzip2、bunzip2、gzip、gunzip、zip、unzip、tar等。

（8）文档操作命令，如csplit、sort、wc等。

（9）文件传输打印命令：传输打印命令，如lpr、bye、ftp等。

Linux命令十分丰富，只有在具体的操:作和使用过程中才能体会命令的功能与强大。

1. **实验内容**

**1、帮助命令 man、help、info**

方法1：

（1）输入命令：man□ls，屏幕显示出手册页中ls命令相关帮助信息的第一页，介绍ls命令的含义。语法结构及-a、-A、-b和-B等选项的意义。

（2）使用PageDown键、PageUp键及上、下方向键找到-s选项的说明信息。

（3）由此可知，ls命令的-s选项等同于-size选项，以文件块为单位显示文件和目录的大小。

（4）在屏幕上的：后输入q，退出ls命令的手册页帮助信息。

方法2：

输入命令：ls□--help，□**代表空格**。屏幕显示中文的帮助信息，由此可知ls命令的-s选项等同于--size选项，以文件块为单位列出所有文件的大小。

方法3：

输入命令：info□ls，屏幕显示中文的帮助信息，由此可知ls命令的-s选项等同于--size选项，以文件块为单位列出所有文件的大小。

以长格式显示当前目录下的内容，包括隐藏文件。

输入命令：ls□-la，屏幕显示当前目录下的各种文件的详细信息。

**3、显示路径命令pwd、改变工作目录命令cd。**□代表空格，后同。

（1）输入命令：pwd看当前目录的绝对路径。

（2）输入命令：cd□/usr/src/kernels，査看目录下的文件。

（3）再次输入命令：pwd看当前目录位置，pwd命令显示的是绝对路径。

（4）默认的用户目录为～，输入命令cd□～，回到用户的工作目录下。回到上—层目录,输入命令“cd□..”。工作目录用“.”表示，其父目录用“..”表示。

（5）Linux提供了命令帮助输入方式，只需输入文件或目录名的前几个字符，然后按Tab键，如无相重的，完整的文件名立即自动在命令行出现；如有相重的，再按一下Tab键，系统会列出当前目录下所有以这几个字符开头的文件名。

（1）使用pwd命令确认回到用户主目录，在主目录中同时创建子目录newdirl，newdir2。依次输入命令mkdir newdir1， mkdir newdir2，或者mkdir newdir1|mkdir newdir2,再用命令ls查看新目录创建是否成功。

（2）在主目录中创建dirl/dir2/dir3 分层的目录结构，输入命令

mkdir□-p□dir1/dir2/dir3，如果把参数-p去了以后，尝试一下会出现什么情况。

（3）在主目录中将创建的目录newdir1、newdir2和dir1/dir2/dir3全部删除，使用命令：

rmdir□newdir1

rmdir□newdir2

通过帮助（rmdir□--help）查看如何利用rmdir命令一次性强制删除多层目录dir1/dir2/dir3。

（1）在当前用户的主目录下使用vi编辑器建立文件filel.txt、file2.txt，或者进入桌面用户的主文件夹，单击右键，新建空文档，建立文件file1.txt、file2.txt。或使用命令cat□>file1.txt和catcat□>file2.txt新建两个文件。

（2）进入“终端”，在主目录中建立子目录path，将文件filel.txt复制到目录path下输入下述命令：

mkdir□path /\* 创建子目录path\* /

cp□file1.txt path /\*将文件file1.txt拷贝到目录path \*/

（3）将主目录中的文件file2.txt移动到子目录path并更名为file3.txt，输入下述命令

mv□file2. txt path/file3. txt

（4）在主目录下交互式地删除子目录path，包含其下的所有文件和子目录，输入下面的命令：

rm□path/file3.txt

rm□path/file1.txt

rmdir□path

（1）使用cat命令显示文件内容。

输入命令：cat□>f1，创建文件f1。

输入命令：cat□f1，屏幕上输入点光标闪烁。

依次输入一些文字内容。使用cat命令进行输入时，只能用退格键（Back Space）来删除光标前一位置的字符。并且一旦按Enter键，该行输入的字符就不可修改。

文字内容输入后，按Enter键，让光标处于输入内容的下一行，按Ctrl+D键结束输入。

要查看文件是否生成，输入命令ls即可。

输入命令：cat□f1，查看f1文件的内容。

再输入命令：cat>>f1，屏幕上输入点光标闪烁。

输入“Hello□World!”后，按Enter键，让光标处于输入内容的下一行，按Ct1+D键结束输入。

输入命令：cat☑fl，查看fl文件的内容，会发现fl文件中有了刚才输入的内容。

（2）分页显示/etc目录中所有文件和子目录的信息。

输入命令：ls/etc | more，屏幕显示出ls命令输出结果的第1页，屏幕的最后一行上还出现-More-字样。按“空格”键可查看下一页信息，按Enter键可查看下一行信息。其中|是管道符，主要用来将左侧ls的显示结果重定向到右侧作为输入。

浏览过程中按q键，可结束分页显示。

（3）仅显示/etc目录中前/后15个文件和子目录。

输入命令：ls□/etc | head –n 15，屏幕显示出ls命令输出结果的前面15行。

输入命令：ls□/etc | tail –n 15，屏幕显示出ls命令输出结果的后面15行。

（4）用上述命令查看/etc下配置文件services的内容。

查看文件内容。输入命令：cat□/etc/services。

分屏查看该文件内容。输入命令：cat /etc/services | more。

只查看前面20行的内容。输入命令：head□–n 20 /etc/services。

只查看后面20行的内容。输入命令：tail□–n 20 /etc/services。

**7、清除屏幕内容命令clear。**

输入命令clear，则屏幕内容完全被清除。

**8、通配符“\*”、“?”的使用。**

（1）显示/bin目录中所有以c为首字母的文件和目录。

输入命令ls□/bin/c\*，屏幕将显示/bin目录中以c开头的所有文件和目录。

（2）显示/bin目录中所有以c为首字母，文件名只有三个字符的文件和目录。

输入命令ls /bin/c??，按Enter键，屏幕显示/bin目录中以c为首字母，文件名只有三个字符的文件和目录。

（3）显示/bin目录中所有的首字母为c或s或h的文件和目录。

输入命令：ls /bin/[c,s,h]\*，屏幕显示/bin目录中首字母为c或s或h的文件和目录。

（4）显示/bin目录中所有的首字母不是a、b、c、d、e的文件和目录。

输入命令：ls□/bin/[!a-e]\*，屏幕显示/bin目录中首字母不是a、b、c、d、e的文件和目录。

重复上一步操作，输入命令：!!，自动执行上一步操作中使用过的ls /bin/[!a-e] \*命令。

**9、文件归档备份命令tar**

tar是一个归档程序，就是说tar可以把许多文件打包成为一个归档文件或者把它们写入备份设备，如一个磁带驱动器。tar是一个基于文件的命令，它本质上是连续地、首尾相连地堆放文件。使用tar可以打包整个目录树，这使得它特别适合用于备份，通常Linux下保存文件都是先用tar命令将目录或者文件打包成tar归档文件（也称为tar包），然后用gzip或bzip2压缩。正因为如此，Linux下已压缩文件的常见扩展名为tar.gz、tar.bz2，以及tgz和tbz等。此外，tar是一个命令行的工具，没有图形界面。

作为系统的备份命令，对于管理员来讲，需要常对以下目录进行备份：

/etc包含所有核心配置文件，其中包括网络配置、系统名称、防火墙规则、用户、组以及其他全局系统项。

var包含系统守护进程（服务）所使用的信息，包括DNS配置、DHCP租期、邮件缓冲文件、HTTP服务器文件、dB2实例配置等。

/home包含所有默认用户的主目录，包括个人设置、己下载的文件和用户不希望失去的其他信息。

/root根（root）用户的主目录。

/opt是安装许多非系统文件的地方。Openoffice、JDK和其他软件在默认情况下也安装在这里。

创建归档可以使用-cf参数，如果需要显示日志，则可以使用-cvf参数。查看归档可以使用-tf参数。

（1）将整个/etc 录下的文件全部打包成为/tmp/etc.ta，输入以下命令。

# tar□–cvf□/tmp/etc.tar□/etc /\*仅对文件夹打包，不进行压缩\*/

# tar□–cvzf□/tmp/etc.tar.gz□/etc /\*文件打包后，并且以gzip压缩\*/

# tar□–cvjf□/tmp/etc.tar.bz2□/etc /\*文件打包后，并且以bzip压缩\*/

需要注意，在参数f之后的文件名是自己取的，习惯上都用.tar来作为辨识；参数z 表示以 tar.gz或.tgz来代表gzip压缩过的tar文件；参数j表示以.tar.bz2来作为tar文件名。

（2）查阅/tmp/etc.tar.gz中的归档文件，使用参数t。输入以下命令。

tar□-tvzf□/tmp/etc.tar.gz

（3）将/tmp/etc.tar.gz文件解压缩在/usr/local/src下，输入以下命令。

# cd□/usr/local/src

# tar□-zxvf□/tmp/etc.tar.gz

（4）在/home当中，比2016/09/01新的文件才备份。输入以下命令。

tar□- N□"2016/09/01"□-zcvf□home. tar. gz□/home

（5）备份/home, /etc，但不要/home/dmtsai。输入以下命令。

tar□--exclude□/home/dmtsai□–zcvf□myfile.tar.gz□/home/\*□/etc

**10、文档压缩命令gzip、bzip2**

文档压缩有两个明显的好处，是可以减小存储空间，二是通过网络传输时，可以减少传输的时间。gzip，bzip是在Linux系统中经常使用的一个对单个文件进行压缩和解压缩的命令、如果要对多个文件同时压缩，那么就要先用tar打包，然后再压缩。

（1）把用户目录下的每个文件压缩成.gz文件。

回到用户主目录，输入命令cd□~

输入命令：gzip□\*

显示压缩文件结果，注意原文件不存在了。

（2）把上述每个压缩的文件解压，并列出详细的信息，使用解压参数d。

输入命令：gzip□-dv□\*

显示是否将原文件解压。

3）详细显示例1中每个压缩的文件的信息，并不解压。

输入命令：gzip -l□\*

实验六 Linux的Vi操作

**一、 实验目的**

学习和掌握利用vi编辑器完成文件的输入和编辑。

**二、预备知识**

**vi编辑器的使用**

vi编辑器是Linux和UNIX上最基本的文本编辑器，工作在字符模式下。该编辑器最初由加州大学伯克利分校为BSD UNIX开发，后来作为标准包含在Linux所有版本中。由于不需要图形界面，使它成了效率很高的文本编辑器。

vi编辑器是Visual interface的简称，通常称之为vi。它在Linux上的地位就像Edit程序在DOS上一样。它可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。vi编辑器并不是一个排版程序，它不像Word或WPS那样对字体格、格式、段落等其他属性进行编排，它只是一个文本编辑程序。

vim是vi的加强版，比vi更容易使用。vi的全部命令都可以在vim上使用。

要在Linux下编写脚本、系统配置文件或者C、C++语言程序，尤其对于系统管理员来讲，vi编辑器是最佳的选择。使用它的好处是几乎每一个版本的Linux都有它的存在，并且它在操作中不需要鼠标的支持，能够胜任在各种环境下使用。

vi有三种基本工作模式：命令行模式、文本输入模式和末行模式。

命令行模式：当执行vi后，首先进入命令模式用于控制屏幕光标的移动，文本字符、字或行的删除、移动、复制某区段及进入插入模式或者进入末行模式。

插入模式：在命令行模式输入相应的命令进入该模式。只有在插入模式下，才可以进行文本输入，按Esc键可回到命令行模式。

末行模式：在命令模式下输入“：”，可进入末行命令模式。在该模式下可保存文件或退出vi，也可以设置编辑环境，如寻找字符串、列出行号等。

**1、进入、退出vi和切换命令**

在系统提示符号输入vi及文件名称后，就进入vi全屏幕编辑界面： $ vi□文件名

$ i 进入插入模式，可以输入文字。按“Esc”键可退回到命令模式。

退出vi时要在命令行模式，按“:”冒号进入末行模式，输入下述命令即可退出。

：w文件名（vi将以指定的文件名保存）

：w （保存文件，但不退出vi）

：wq （保存文件，退出vi）

：q！ （放弃编辑内容，退出vi）

**2、插入命令模式**

按i键进入插入模式后，是从光标所在位置前面开始输入文字。

按I键进入插入模式后，是在光标所在行的第一个非空白字符前输入文字。

按a键进入插入模式后，是从目前光标所在位置的下一个位置开始输入文字。

按A键进入插入模式后，是从目前光标所在行尾输入文字。

按o键进入插入模式后，是在光标所在行的下一行插入新行，从行首输入文字。

按O键进入插入模式后，是在光标所在行的上一行插入新行，从行首输入文字。

按Esc键切换为命令模式。

**3、末行模式命令**

在命令行模式下，按：、/、?键进入末行模式。

输入：w>>文件名，是将内容写到文件原有内容的后面。

输入：x，是对修改后信息存盘，退出vi。

输入：r 文件名，是将文件调入到编辑缓冲区。

输入：e！，是另行编辑文件，并放弃编辑缓冲区内容。

输入：s/old/new/g，是将当前行中首次出现的字符串old改为new。

输入：%s/old/new/g，是将所有行出现的字符串old改为new。

输入：set nu，是将编辑器显示符号。

输入：set nonu，是不让编辑器显示行号。

输入：set all，是显示环境变量。

输入/exp,是从光标往后查找字符串exp，如果第一次找到的字符串不是想要的，可以一直按n键往后查找，直到找到想要的字符串为止。

输入/exp，是从光标往前查找字符串exp，如果第一次找到的字符串不是想要的，可以一直按n键往前寻找，直到找到想要的字符串为止。

**三、实验内容**

**1、vi新建文件f2，输入下面内容**

How to Read Faster

When I was a schoolboy I must have read every comic book ever published, But as I got older, my eyeballs must have slowed down or something I mean, comic books started to pile up faster then I could read them!

+本人学号姓名（另起一行）

注：在文件中需输入本人学号、姓名。

**操作步骤：**

（1）启动OS后，以普通用户身份登录字符界面。

（2）在shell命令提示符后面输入命令：vi，启动vi命令编辑器，进入命令模式。

（3）按i键，从命令模式转换为文本编辑模式，此时屏幕上的最底边出现--INSERT----字样.

（4）输入上述文本内容。如果输入出错，可使用退格键或Delete键删除错误的字符。

（5）按Esc键返回命令行模式。

（6）按：键进入最后行模式，输入w f2模式，就可以将正在编辑的内容保存为f2文件，屏幕底部显示“f2”[new] 3L，\*C written字样（其中\*表示某整数），表示此文件有三行，\*（\*表示某整数）个字符。注意：vi中行的概念与平时所说的行有所区别，再输入文字的过程中由于字符串长度超过屏幕宽度而发生的自动换行，vi并不认为是一行。只有在vi中按一次enter键，另起一行的一行才是新的一行。

（7）按：键后输入q，退出vi。

**2、打开文件f2并显示行号。**

（1）输入命令：vi f2，启动vi文本编辑器并打开f2文件，

（2）按：键切换到最后行模式，输入命令：set nu，每一行前出现行号。

（3）vi自动返回到命令模式连续两次输入Z，就退出vi。

**3、在f2文件的第一行后插入如下一行内容**：

With the development of society, the ability of reading becomes more and more important. 并在最后添加一行内容。

内容为：“We must know some methods to read-faster”。

（1）再次输入命令：vi□f2，启动vi文本编辑器并打开f2文件，显示符号。

（2）按：键进入末行模式输入数字1，光标指示第一行。

（3） 按a或i键，进入文件编辑模式，屏幕上底部出现---INSERT---字样。利用方向键移动光标到第一行行尾后，按Enter键，另起一行。或者按A键进入文本编辑模式，光标直接定位到行尾，按Enter键，另起一行。或者按O键直接另起一行开始输入。输入With the development of society, the ability of reading becomes more and more important.

（4）按：键进入末行模式，输入末行行尾数字，利用Control+D键翻屏，光标指示最后一行，将光标移动到最后一行的末尾，按enter键移动到最后一行末尾，按Enter键另起。输入We must know some methods to read faster.

**4、将文本中所有的eyeballs用eye-balls替换。**

按Esc键后输入“：”进入末行模式。因为当前f2文件中共有5行，所以输入命令：1,5 s/eyeballs/eye-balls/g，或者输入命令：%s/eyeballs/eye-balls/g，并按Enter键，

将文件中所有的eyeballs替换为eye-balls。

**5、把第二行移动到文件的最后，删除第一行和第二行并恢复删除，并不保存修改。**

（1）按：键，再次进入末行模式，输入：2,2 m 5。

（2）按：键，输入1,2 d，删除第一行和第二行。

（3）按u键恢复被删除的部分。

（4）按：键，进入末行模式，输入：q！，退出时不保存对文件的修改。

**6、复制第二行，并添加到文件的最后，删除第二行，保存修改后退出vi。**

（1）再次输入命令：vi f2，启动vi文本编辑器并打开f2文件。

（2）按：键，进入末行模式，输入：2, 2 co 5，将第二行内容复制到第五行的后面。

（3）移动光标到第二行，输入：2d，原来的第二行消失。

（4）按：键，输入wq，存盘并退出vi。

实验七 Linux下的C语言编程

**一、 实验目的**

学习和掌握在linux下用vi编辑器编写程序，利用gcc软件编译和执行C语言编写的程序。

**二、预备知识**

**Linux下C语言编译与调试**

用户在Linux下编程，建议使用C和C++语言。而要使用C和C++语言编程，就必须要知道它们的编译和开发环境GCC。GCC是GNU推出的功能强大、性能优越的多平台编译器，是GNU的代表作品之一。GCC不仅可以编译C、C++程序，而且做了很多扩展，也可以处理Fortran、Pascal、Objective-C、Java，以及Ada等语言程序。目前GCC的官方网址是http:gcc.gnu.org/。

使用GCC由C语言源代码文件生成可执行文件的过程要经历四个相互关联的步骤：预处理（也称预编译，Preprocessing）、编译（Compilation）、汇编（Assembly）、和链接（Linking）。因此，其最终编译完成的目标代码的执行效率比一般的编译器相比平均要高20%-30%。在编译的各个阶段，GCC默认的文件格式是不同的。具体来讲，.c为扩展的文件是C语言源代码文件；.a为扩展的文件是由目标的文件构成的库文件；C、.cc或.cxx为扩展名的文件是C++源代码文件；.m为扩展的文件是Object-C源代码文件；.o为扩展名的文件是是编译后的目标文件；.s为扩展的文件是经过预编译的汇编语言源代码文件。

下面通过HelloWord程序开始，简单介绍GCC编译过程。

（1）编写源代码。

下面是hello.c的源代码，可以使用vi编辑器来编写源代码。

#include<stdio.h>

int main( )

{

printf("Hello World!\n");

}

（2）分步编译源程序。

GCC编译器是在命令行模式下工作，在桌面环境中，使用GCC编译器要在“终端”中使用。在使用GCC编译器的时候，用户必须给出一系列必要的调用参数和文件名称。

gcc -E hello.c -o hello.i /\*第一步，预编译过程：处理宏定义和include,并做语法检查\*/

gcc -S hello.i -o hello.s /\*第二步，编译过程：编译过程： 生成汇编代码\*/

gcc -c hello.s -o hello.o /\*第三步，编译过程：编译过程：生成目标代码\*/

gcc hello.o -o hello /\*第四步，连接过程：生成可执行代码。链接分为两种，一种是静态链接，另一种是动态链接。使用静态链接的好处是，依赖的动态链接库较少，对动态链接库的版本不会很敏感，具有较好的兼容性；缺点是生成的程序比较大。使用动态连接的好处是，生成的程序比较小，占用较少的内存\*/

./hello /\*执行hello文件，在执行hello文件时.在hello前添加“./”，shell会在PATH环境变量设置的目录中找可执行文件，这些目录中通常不包括当前目录，也无法找到hello文件\*/

Hello.world! /\*输出运行结果\*/

其实，上述GCC四个编译过程，只需要一条命令就可以完成。之所以详细列出GCC在编译时候的每一个步骤，是希望能够清楚地理解GCC的编译过程。

gcc hello.c -o hello /\*源代码直接变异成可执行文件\*/

编写小程序的时候，都可以简单的编译源程序，以重建目标程序。但对于开发大型程序来说，由于某个程序的变动往往影响到其他程序的使用，并且程序间的依赖关系非常复杂，如果不能清晰的描述程序间的关系，则开发大程序是非常痛苦的事情，这就要使用到make工具，它会在必要时重新编译所有受改动影响的源文件。使用make工具时，程序员需要提供一个能够说明源文件之间的依赖关系和构建原则的文件，即makefile文件。make命令会读取makefile文件的内容，它先确定要创建的目标文件，然后比较该目标所依赖的源文件的日期和时间，以决定采用哪条规则来创建目标。在Linux内核源代码中，就采用makefile文件对内核进行编译，每个目录中都存在一个makefile文件，任务是根据上级目录makefile的命令启动编译。

关于如何在Linux下进行软件开发已经超出了本实验的范围，感兴趣的读者可以查阅相关资料。一个强大的编译器更需要功能强大的调试器配合，才能算是完美。在Unix和Linux系统下一般使用GDB调试器。GDB是GNU发布的一个强大的Linux下的程序调试工具。目前有一些针对GDB的“前端”程序，它们提供非常友好的用户界面，xxgdb、tgdb和ddd都是这样的程序。虽然不如Visual stdio、BCB等可视化IDE的调试方便，但是GDB功能更加强大。

一般来说，GDB主要调试的是C/C++的程序。要调试C/C++的程序，首先在编译时，要求必须把调试信息加到可执行文件中。使用编译器GCC的-g参数可以做到这一点。例如

gcc -g hello.c -o hello

如果没有-g，用GDB调试时将看不见函数的函数名、变量名，所代替的全是运行时的内存地址。当用-g把调试信息加入，并成功编译目标代码后，就可以使用GDB来调试。

启动GDB的方法有以下几种：

（1）gdb□program /\*program也就是执行文件\*/

（2）gdb□program□core /\*用GDB同时调试一个运行程序和

core文件，core是程序非法执行后，core dump后产生的文件\*/

（3）gdb□program□PID /\*如果程序是一个服务程序，那么可

以指定这个服务程序运行时的进程ID.GDB会自

动attach上去，并调试它。program应该应该在PATH

环境变量中搜索得到\*/

GDB的命令可以使用help命令来查看。启动GDB后，进入GDB的调试环境，可以使用GDB的命令调试程序。进入GDB后，若想退出可以在（gdb）后输入quit。

注：若安装VMware12.0和CentOS7，gcc和gdb将自动安装。若安装的是RedHat6.2，gcc已自动安装，而gdb需自行安装。先下载安装termcap，再安装gdb。

1. **实验内容**

**实验基础：**

先创建一个hello.c文件，然后编译并执行，查看结果。

#include<stdio.h>

main()

{

printf(“Hello !”);

}

**Linux下C语言编程调试：**

本实验的目的是调试C语言程序greeting，该程序的功能是显示一个简单问候语，再将该字符串反序列出。例如，输入Linux，反序为xunil。

实验步骤如下。

**1、用vi编辑器创建一个greeting.c文件。**

/\*\*\*\*\*\*\*\*greeting.c\*\*\*\*\*\*\*/

#include<stdio.h>

#include <string.h>

main()

{

char my\_string[]="hello there";

my\_print(my\_string);

my\_print2(my\_string);

}

void my\_print(char \*string)

{

printf("the string is %s\n",string);

}

void my\_print2(char \*string)

{

char \*string2;

int size,i;

size=strlen(string);

string2=(char \*)malloc(size+1);

for(i=0;i<size;i++) /\*第19行\* /

string2[size-i]=string[i];

string2[size+1]='\0';

printf("The string printed backward is %s\n",string2);

}

**2、gcc命令编辑这个程序**：gcc -o greeting greeting.c，执行程序（./greeting）。显示结果如下：

The string is hello there

The string printed backward is

输出的第一行是正确的，但是第二行并不是所期望的，所设想的输出应该是The string printed backward is ereht olleh

显然，由于某种原因，my\_print2函数没有正常工作，可以用GDB对程序进行调试。

**3、重新编译程序**

为了能够使用GDB进行调试，需要在程序编译的时候加入调试参数-g，重新编译程序，输入命令：gcc -o greeting -g greeting.c

进入调试器，输入命令gdb greeting 如果在输入命令时忘了把要调试的程序作为参数传给gdb，则可以在gdb提示符下用file命令来载入它。

(gdb)file greeting /\*该命令载入greeting可执行文件\*/

(gdb) run /\*该命令用来运行greeting\*/

查看在调试器中的输出和在gdb外面运行的结果是否一样。

**4、查找问题**

为了查找问题的所在，可以在my\_print2函数的for语句后设置一个断点。具体做法是，在gdb提示符下输入list命令2次，列出源代码，查看行号。

(gdb) list /\*该命令只能列出文件的部分，可以使用多次\*/

(gdb) list

(gdb) list

假设断点的地方在第20行，

(gdb) break 20 /\*输入如下命令作为断电\*/

**5、**再次输入run命令，**程序挂起在输入断点的位置。**

**6、设置一个观察string2[size-i]变量的值的观察点**来看错误是怎样产生的，输入命令:

(gdb) watch string2[size-i] /\*该命令设置变量观察点\*/

**7、用next命令一步步执行for循环。**

(gdb) next

记录每次循环后string2[size-i]的值是多少，是否和期望值相同。确定程序出错位置，并分析其原因。如果next命令无法看出观察变量的变化，试着用c来查看。

**8、修正程序错误并重新编译**

找出了问题出在哪里后，退出gdb，使用(gdb)quit命令，修正程序错误并重新编译。下面是修改后的代码，仅供参考：

#include <stdio.h>

#include <string.h>

main()

{

char my\_string[]="hello there";

my\_print(my\_string);

my\_print2(my\_string);

}

void my\_print(char \*string)

{

printf("the string is %s\n",string);

}

void my\_print2(char \*string)

{

char \*string2;

int size,i;

size=strlen(string);

string2=(char\*)malloc(size+1);

for(i=0;i<size;i++)

string2[size-1-i]=string[i];

string2[size]='\0';

printf("The string printed backward is %s\n",string2);

}

实验八 Linux的Shell编程

**一、 实验目的**

学会编写简单的Shell脚本程序，学会运行命令文件。

**二、预备知识**

Shell本身是一个用C语言编写的程序，它是用户使用Linux的桥梁。Shell既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式的解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。它虽然不是Linux系统核心的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，Shell是最重要的实用程序，深入了解和熟练掌握Shell的特性及其使用方法，是用好Linux系统的关键。可以说，Shell使用的熟练程度反映了用户对Linux使用的熟练程度。

bash。bash是Linux系统默认使用的Shell，它是Brian Fox和Chet Ramey共同完成，是Bourne Again Shell的缩写，内部命令一共有40个。Linux使用它作为默认的Shell是因为它有以下的特色：I.可以使用类似DOS下面的doskey的功能，用方向键查阅和快速输入并修改命令；II.自动通过查找匹配的方式给出以某字符串开头的命令；III.包含了自身的帮助功能，只要在提示符下面输入help就可以得到相关的帮助。

**Shell功能与变量参数：**

作为命令语言交互式地解释和执行用户输入的命令只是Shell功能的一个方面，Shell还可以用来进行程序设计，它提供了定义变量和参数的手段及丰富的程序控制结构。使用Shell编程类似于Windows中的批处理文件，称为Shell script，又叫Shell程序或Shell脚本文件。

1、Shell基本功能

Shell的基本功能是如何输入命令运行程序及如何在程序之间通过Shell的一些参数提供便利手段来进行通信。

（1）输入/输出重定向

在Linux中，每一个进程都有三个特殊的文件描述指针：标准输入（standard input，文件描述指针为0）、标准输出（standard output，文件描述指针为1）、标准错误输出（standard error，文件描述指针为2）。这三个特殊的文件描述指针使进程在一般情况下接收标准输入终端的输入，同时由标准终端来显示输出，Linux同时也向使用者提供可以使用普通的文件或管道来取代这些标准输入/输出设备。在Shell中，使用者可以利用>和<来进行输入/输出重定向。例如：

command >file /\*将命令的输入结果重定向到一个文件\*/

command >&file /\*将命令的标准错误输出一起重定向到一个文件\*/

command >>file /\*将标准输出的结果追加到文件中\*/

command>>&file /\*将标准输出和标准错误输出的结构都追加到文件中\*/

（2）Shell管道

管道同样可以在标准输入/输出和标准错误输出间做代替工作。这样一来，可以将某一个程序的输出送到另一个程序的输入，其语法如下。

command1| command2 [| command3…]

也可以连同标准错误输出一起送入管道：

command1| &command2 [| &command3…]

（3）前台和后台

在Shell下面，一个新产生的进程可以通过用命令后面的符号；和&来分别以前台和后台的方式来执行，语法如下。

command; /\*产生一个前台的进程下一个命令须等该命令运行结束后才能输入\*/

command & /\*产生一个后台的进程，此进程在后台运行的同时，可以输入其他的命令\*/

2、Shell程序的变量和参数

对Shell来讲，所有变量的取值都是一个字符串，Shell程序采用$var的形式来引用名为var的变量的值。

Shell有以下几种基本类型的变量。

（1）Shell定义的环境变量

Shell在开始执行时就已经定义了一些和系统的工作环境有关的变量，这些变量用户还可以重新定义，常用的Shell环境变量如下。

HOME：用于保存注册目录的完全路径名。

PATH：用于保存用冒号分隔的目录路径名，Shell将按PATH变量中给出的顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行。

TERM：终端的类型。

UID：当前用户的标识符，取值是由数字构成的字符串。

PWD：当前工作目录的绝对路径名，该变量的取值随cd命令的使用而变化。

PS1：主提示符，在特权用户下，默认的主提示符是#；在普通用户下，默认的主提示符是$。

PS2：在Shell接收用户输入命令的过程中，如果用户在输入行的末尾输入“\”，然后按Enter键，或者当用户按Enter键时Shell判断出用户输入的命令没有结束，显示这个辅助提示符，提示用户继续输入命令的其余部分。默认的辅助提示符是>。

（2）用户定义的变量

用户可以按照下面的语法规则定义自己的变量。

变量名=变量值

要注意的一点是，在定义变量时，变量名前不应加符号$，在引用变量的内容时则应在变量名前加$；在给变量赋值时，等号两边一定不能留空格，若变量中本身就包含了空格，则整个字符串都要用双引号括起来。

在编写Shell程序时，为了使变量名和命令名相区别，建议所有的变量名都用大写字母来表示。

在任何时候，建立的变量都只是当前Shell的局部变量，所以不能被Shell运行的其他命令或Shell程序所利用，export命令可以将一局部变量提供给Shell执行的其他命令使用，其格式为

export 变量名

也可以在给变量赋值的同时使用export命令：

export 变量名=变量值

使用export说明的变量，在Shell以后运行的所有命令或程序中都可以访问到。

（3）位置参数

位置参数是一种在调用Shell程序的命令行中按照各自的位置决定的变量，是在程序名之后输入的参数。位置参数之间用空格分隔，Shell取第一个位置参数替换程序文件中的$1，第二个替换$2，以此类推。$0是一个特殊的变量，它的内容是当前这个Shell程序的文件名，所以$0不是一个位置参数，在显示当前所有的位置参数时是不包括$0的。

（4）内部命令

bash命令解释程序包含了一些内部命令。内部命令在目录列表时是看不见的，它们由Shell本身提供。常用的内部命令有echo、eval、exec、export、readonly、read、shift、wait、和点（.）。

echo。命令格式：echo arg；功能：在屏幕上打印出由arg指定的字符串。

eval。命令格式：eval args。功能：当Shell程序执行到eval语句时，Shell读入参数args，并将它们组合成一个新的命令，然后执行。

exec。命令格式：exec命令 命令参数。功能：当Shell执行到exec语句时，不会去创建新的子进程，而是转去执行指定的命令，当指定的命令执行完时，该进程，也就是最初的Shell就终止了，所以Shell程序中exec后面的语句将不再被执行。

export。命令格式：export 变量名 或 export 变量名=变量值。功能：Shell可以用export把它的变量向下带入子Shell从而让子进程继承父进程中的环境变量；但子Shell不能用export把它的变量向上带入父Shell。

注意：不带任何变量名的export语句将显示出当前所有的export变量。

readonly。命令格式：readonly 变量名。功能：将一个用户定义的Shell变量标识为不可变的；不带任何参数的readonly命令将显示出所有只读的Shell变量。

read。命令格式：read变量名表。功能：从标准输入设备读入一行，分解成若干字，赋值给Shell程序内部定义的变量。

shift语句。命令格式：Shift[n]功能：shift语句按如下方式重新命名所有的位置参数变量：$2成为$1，$3成为$2…在程序中每使用一次shift语句，都使所有的位置参数依次向左移动一个位置，并使位置参数“$#”减一，直到减到0。

wait。命令格式：wait[Process ID] 功能：Shell等待在后台启动的所有子进程结束。wait的返回值总是真。

exit。命令格式：exit[状态值] 功能：退出Shell程序。在exit之后可有选择地指定一个数字作为返回状态。

“.”（点）。命令格式：.Shell程序文件名。功能：使Shell读入指定的Shell程序文件并依次执行文件中的所有语句。

（5）预定义变量

预定义变量和环境变量相类似，也是在Shell一开始时就定义了的变量。所不同的是，用户只能根据Shell的定义来使用这些变量，而不能重定义它。所有预定义变量都是由$符和另一个符号组成的，常用的Shell预定义变量如下。

$#：位置参数的数量。

$\*：所有位置参数的内容。

$?：命令执行后返回的状态。

$$：当前进程的进程号。

$!：后台运行的最后一个进程号。

$0：当前执行的进程名。

其中，$?用于检查上一个命令执行是否正确（在Linux中，命令退出状态为0表示该命令正确执行，任何非0值表示命令出错）。

$$变量最常见的用途是用做临时文件的名字以保证临时文件不会重复。

（6）参数置换的变量

Shell提供了参数置换能力以便用户可以根据不同的条件来给变量赋不同的值。参数置换的变量有四种，这些变量通常与某一个位置参数相联系，根据指定的位置参数是否已经设置来决定变量的取值，它们的语法和功能分别如下。

变量=${参数-word}：如果设置了参数，则用参数的值置换变量的值，否则用word置换。即这种变量的值等于某一个参数的值，如果该参数没有设置，则变量就等于word的值。

变量=${参数=word}：如果设置了参数，则用参数的值置换变量的值，否则把变量设置成word然后再用word替换参数的值。注意，位置参数不能用于这种方式，因为在Shell程序中不能为位置参数赋值。

变量=${参数?word}：如果设置了参数，则用参数的值置换变量的值，否则就显示word并从Shell中退出，如果省略了word，则显示标准信息。这种变量要求一定等于某一个参数的值，如果该参数没有设置，就显示一个信息，然后退出，因此这种方式常用于出错指示。

变量=${参数+word}：如果设置了参数，则用word置换变量，否则不进行置换。

所有这四种形式中的“参数”既可以是位置参数，也可以是另一个变量，只是用位置参数的情况比较多。

**Shell程序的流程控制：**

Shell提供了用来控制程序执行流程的命令，包括条件、分支和循环条件，用户可以用这些命令建立非常复杂的程序。

与传统的语言不同的是，Shell用于指定条件值的不是布尔表达式而是命令和字符串。

1、测试语句

测试语句test作为控制条件，通过测试expression来实现一个条件测试，测试语句的计算表达式的值返回真（0）或假（1）。test在Shell脚本中常使用[ expression ]代替test命令，但是要注意在“[”之后和“]”之前保留空格。test测试可以进行数值、字符和文件三个方面的测试，其测试符和相应的功能分别如下。

（1）整数测试

[ n1-eq n2 ]表示若n1等于n2，则测试条件为真。

[ n1-ne n2 ]表示若n1不等于n2，则测试条件为真。

[ n1-gt n2 ]表示若n1大于n2，则测试条件为真。

[ n1-ge n2 ]表示若n1大于等于n2，则测试条件为真。

[ n1-lt n2 ]表示若n1小于n2，则测试条件为真。

[n1-le n2]表示若n1小于等于n2，则测试条件为真。

（2）字符串测试

[ -z s ]表示若字符串s1长度为0，则测试条件为真。

[ -n s ]表示若字符串s1长度大于0，则测试条件为真。

[ s1 ]表示若字符串s1不为空，则测试条件为真。

[ s1=s2 ]表示若两个字符串相等，则测试条件为真。

[ s1!=s2 ]表示若两个字符串不相等，则测试条件为真。

（3）文件测试

[ -e file ]表示若文件存在，则测试条件为真。

[ -r file ]表示若文件存在且为用户可读，则测试条件为真。

[ -w file ]表示若文件存在且为用户可写，则测试条件为真。

[ -x file ]表示若文件存在且为用户可执行，则测试条件为真。

[ -b file ]表示若文件存在且为块设备，则测试条件为真。

[ -c file ]表示若文件存在且为字符设备，则测试条件为真。

[ -d file ]表示若文件存在且为目录文件，则测试条件为真。

[ -f file ]表示若文件存在且为普通文件，则测试条件为真。

[ -p file ]表示若文件存在且为FIFO文件，则测试条件为真。

[ -s file ]表示若文件存在且文件长度>0，则测试条件为真。

[ -t file ]表示若文件描述符与终端相关，则测试条件为真。

2、条件语句

Shell程序中的条件判断是通过if条件语句来实现的，其一般格式为：

if 条件命令串 then 条件为真时的命令串 else 条件为假时的命令串 fi

3、循环语句

循环语句分三种：for、while和until。根据测试条件执行相应命令。

for循环对一个变量的可能的值都执行一个命令序列。赋给变量的几个数值既可以在程序内以数值列表的形式提供，也可以在程序以外以位置参数的形式提供。for循环的一般格式为：

for 变量名 [in 数值列表] do 若干个命令行 done

变量名可以使用户选择的任何字符串，如果变量名是var，则在in之后给出的数值将顺序替换循环命令列表中的$var。如果省略了in，则变量var的取值将是位置参数。对变量的每一个可能的赋值都将执行do和done之间的命令列表。

while和until命令都是用命令的返回状态值来控制循环的。while循环的一般格式为：

while 若干个命令行1 do 若干个命令行2 done

只要while的“若干个命令行1”中最后一个命令的返回状态为真，while循环就继续执行，do...done之间的“若干个命令行2”。

until循环的格式为：

until 若干个命令行1 do 若干个命令行2 done

until循环和while循环的区别在于：while循环在条件为真时继续执行循环，为until是在条件为假时继续执行循环。

Shell还提供了true和false两条命令用于建立无限循环结构的需要，它们的返回状态分别是总为0或总为非0。

4、开关语句

case条件选择为用户提供了根据字符串或变量的值从多个选项中选择一项的方法，其格式如下：

case 字符串 in

模式字符串1)

若干个命令行1

;;

模式字符串2)

若干个命令行2

;;

...

其他命令行

esac

Shell通过计算字符串string的值，将其结果依次和各模式字符串匹配，直到找到一个匹配的表达式为止。如果找到了匹配项则执行它下面的命令，直到遇到一对分号（;;）为止。

在case表达式中也可以使用Shell的通配符（\*、?、[]）。通常用\*作为case命令的最后表达式，以便使在前面找不到任何相应的匹配项时执行“其他命令行”的命令。

5、退出控制语句

break用于立即终止当前循环的执行，而continue用于不执行循环中后面的语句而立即开始下一个循环的执行。这两个语句只有放在do和done之间才有效。

**Shell程序的运行与调试：**

1、Shell程序的运行方法

用户可以用任何编辑程序来编写Shell程序。因为Shell程序是解释执行的，所以不需要编译装配成目标程序。按照Shell编程的惯例，以bash为例，程序的第一行一般为#!/bin/bash，其中#表示该行是注释，叹号!告诉Shell运行叹号之后的命令并用文件的其余部分作为输入，也就是运行/bin/bash并让/bin/bash去执行Shell程序的内容。

（1）sh Shell程序文件名

格式为 bash Shell程序文件名

这实际上是调用一个新的bash命令解释程序，而把Shell程序文件名作为参数传递给它。新启动的Shell将去读指定的文件，执行文件中列出的命令，当所有的命令都执行完结束。该方法的优点是可以利用Shell调用功能。

（2）sh<Shell程序文件名

格式为 bash<Shell 程序文件名

这种方式就是利用输入重定向，使Shell命令解释程序的输入取自指定的程序文件。

（3）用chmod命令使Shell程序成为可执行的

一个文件能否运行取决于该文件的内容本身可执行且该文件具有执行权。对于Shell程序，当用编辑器生成一个文件时，系统赋予的许可权限都是644（rw-r-r--），因此，当用户需要运行这个文件时，只需要直接输入文件名即可。

在这三种运行Shell程序的方法中，最好按下面的方式选择：当刚建立一个Shell程序，对它的正确性还没有把握时，应当使用第一种方式进行调试。当一个Shell程序已经调试好时，应使用第三种方式把它固定下来，以后只要输入相应的文件名即可，并可被另一个程序所调用。

2、bash程序的调试

在Shell编程过程中难免会出错，有的时候，调试程序比编写程序花费的时间还要多。

Shell程序的调试主要是利用bash命令解释程序的选择项。调用bash的形式是

bash -选择项 Shell程序文件名

几个常用的选择项如下。

-e：如果一个命令失败就立即退出。

-n：读入命令但是不执行它们。

-u：置换时把未设置的变量看做出错。

-v：当读入Shell输入行时把它们显示出来。

-x：执行命令时把命令和它们的参数显示出来。

上面的所有选项也可以在Shell程序内部用set -选择项的形式引用，而set +选择项则将禁止该选择项起作用。如果只想对程序的某一部分使用某些选择项时，则可以将该部分用上面两个语句包围起来。

（1）未置变量退出和立即退出

未置变量退出特性允许用户对所有变量进行检查，如果引用了一个未赋值的变量就终止Shell程序的执行。Shell通常允许未置变量的使用，在这种情况下，变量的值为空。如果设置了未置变量退出选择项，则一旦使用了未置变量就显示错误信息，并终止程序的运行。未置变量退出选择项为-u。

当Shell运行时，若遇到不存在或不可执行的命令、重定向失败或命令非正常结束的等情况时，如果未经重新定向，该出错信息会打印在终端屏幕上，而Shell程序仍将继续执行。要想在错误发生时迫使Shell程序立即结束，可以使用-e选项将Shell程序的执行立即终止。

（2）Shell程序的跟踪调试Shell程序的主要方法是利用Shell命令解释程序的-v或-x选项来跟踪程序的执行

-v选择项使Shell在执行程序的过程中，把它读入的每一个命令行都显示出来，而-x选择项使Shell在执行程序的过程中把它执行的每一个命令在行首用一个“+”加上命令名显示出来。并把每一个变量和该变量所取的值也显示出来，因此，它们的主要区别在于：在执行命令行之前无-v则打印出命令行的原始内容，而有-v则打印出经过替换后的命令行的内容。

**三、实验内容**

本实验要求熟读预备知识中关于Shell的原理及使用方法，将可执行的命令在Shell“终端”上进行测试执行，体会Shell的功能。然后将下述Shell脚本分别生成单独文件并执行，请描述各脚本的功能和输出结果。需要说明的是，在下述脚本中，会使用到expr的命令。

expr命令一般用于整数值计算，但也可用于字符串操作。格式如下。

expr□argument□operator□argument

举例如下。

$expr□10□+□10

1、描述以下各脚本的功能和输出结果，写入实验报告。

（1）输出。

#!/bin/bash

echo□“Please type your number:”

read a

6

for ((i=0;i<=a;i++))

do

for ((p=1;p<=i;p++))

do

echo -n “$p”

done

echo

done

（2）计算器。

#!/bin/bash

s=0

while true

echo “...................+”

echo “...................-”

echo “...................\*”

echo “.................../”

echo “...................q”

echo “Please type your word:(e.g.1 + 2)”

read a b c

do

case $b in

+)

let s=a+c

echo “ $a + $c =” $s;;

-)

let s=a-c

echo “ $a - $c =” $s;;

\*)

let s=a\*c

echo “ $a \* $c =” $s;;

/)

let s=a/c

echo “ $a / $c =” $s;;

esac

case $a in

q) break ;;

esac

done

（3）输出当前目录下所有文件，并输出文件总数和目录总数。

#!/bin/bash

ls -al

filenum=0

dirnum=0

for□q□in□`ls –a` /\*注意是斜撇\*/

do

if [□-d□$q ]

then

dirnum=`expr $dirnum + 1`

fi

if [□-f□$9□]

then

filenum=`expr $filenum + 1`

fi

done

echo “The number of dirctory is $dirnum”

echo “The number of file is $filenum”

（4）带参数的输入。先将shell命令写入一个文件，然后这样执行：

chmod□777 □文件名

文件名 a b c d e f

#!/bin/bash

for (( b=$#; b>0;b-- ))

do

echo $\*;

shift;

done

（5）用一个文件记录系统中所有rpm包名，如果文件不存在则创建，如果存在则显示文件内容。

#!/bin/bash

a=/root/file

if [ -f $a ]

then

cat $a

else

rpm -qa > $a

fi

2、编写一个Shell脚本，完成以下功能：创建四个文件test1、test2、test3、test4，实现自动创建dir1、dir2、dir3、dir4四个目录，并将test1、test2、test3、test4四个文件复制到dir1、dir2、dir3、dir4相应的目录下。请将Shell脚本和实验结果截图写入实验报告。

实验九 Linux的进程管理

**一、 实验目的**

熟悉和理解进程和进程树的概念，掌握Linux进程的管理机制；掌握使用Linux命令管理和操作进程的方法；理解进程的运行机制。理解线程与进程的关系。

**二、预备知识**

Linux的进程结构

进程是现代操作系统中程序运行的基本单位，多应用任务的并发就得依靠进程来实现。

在操作系统中，进程的相关描述有三个部分组成：进程控制块（Process Control Block，PCB）、有关程序段和该程序段对其进行操作的数据结构集。进程控制块是一个内核数据结构，每个进程对应一个，用来存储进程的描述信息、控制信息和资源信息；程序段是该进程执行的指令代码，部分程序可以与其他进程共享；数据结构集则是进程在执行时必不可少的工作区和操作对象。

Linux中的进程具体实现由进程控制块、进程程序块、进程内核栈及进程数据块组成。每个进程用数据结构task\_struct描述，最初的进程在系统初始化时生成，而以后的进程均由已有的进程通过进程创建函数fork()来创建。当一个进程在创建时，系统为该进程建立一个task\_struct结构。当进程运行结束时，系统就会撤销该进程的task\_struct结构。Linux在操作系统的内核空间专门设置一个task指针数组，该数组的每个元素是一个指向任务结构的指针。该task指针数组结构的定义如下。

struct task\_struct \*task[NR\_TASKS]={&init\_task};

#define NR\_TASKS 512

可以看到，task数组的大小由全局变量NR\_TASKS决定，默认可容纳的进程数是512，表明Linux系统中最多能并发运行512个进程。下面是内核2.4中task\_struct结构的主要数据结构的描述，其详细定义在include\linux\sched.h文件中，可以通过进入/usr/include/linux目录下，使用cat sched.h查看数据结构，其主要数据成员说明如下。

struct task\_struct{

volatile long state; /\*进程当前状态，进程有运行状态（0）、 不可运行状态（-1）和暂停状态（>0）\*/

unsigned long flags; /\*进程标识描述进程管理信息或过渡状态， 如进程正在工作内容、开始关闭、被信号终止等，

有以下定义：

#define PF\_EXITING 0x00000004 进程关闭 #define PF\_FORKNOEXEC 0x00000040 进程刚创建还没运行

#define PF\_SUPERPRIV 0x00000100

使用超级用户特权

#define PF\_SIGNALED 0x00000400 进程被信号终止

#define PF\_MEMALLOC 0x00000800

进程正在分配主存

#define PF\_FSTRANS 0x00020000

进程正在文件传输

#define PF\_SWAPWRITE 0x00800000 进程运行写交换区

#define PF\_KTHREAD 0x00200000

标志为核心线程 ...... \*/

unsigned long ptrace; /\*跟踪标志\*/

int prio,static\_prio,normal\_prio; /\*进程优先级。prio是进程动态 优先级。在进程运行过程中动态计算,主要影响因素

为进程等待时间sleep\_avg。创建时子进程继承父进

程的优先级、唤醒等待进程时对它进行优先级修正、

时钟中断中重新计算进程优先级并进入相应队列等

都有可能改变进程动态优先级。 static\_prio是非实时进程静态优先级。

由nice值转换而来。 nice值为-20~19,公式为： static\_prio=MAX\_RT\_PRIO+nice-20,

MAX\_RT\_PRIO定义为100,故其范围为100~319。

normal\_prio是正常优先级,调度根据其判断是否为

实时进程 \*/

unsigned long policy; /\*该进程的调度策略,可以通过系统调用 sys\_sched\_setscheduler()更改。

定义值如下：#define SCHED\_OTHER 0 用于非实时

进程,基于优先权的轮转法(round robin)。

#define SCHED\_FIFO 1 实时进程的先进先出算法。

#define SCHED\_RR 2 实时进程的基于优先权的 轮转法\*/

long counter; /\*在轮转法调度时表示进程当前还可运 行多久。在进程开始运行时被赋为priority的值,

以后每隔一个tick(时钟中断)递减1,减到0时引起新

一轮调度。重新调度将从run\_queue队列选出counter

值最大的就绪进程并给予CPU使用权,因此counter

起到了进程的动态优先级的作用 (priority则是

静态优先级) \*/

long priority; /\*priority是静态优先级\*/

int lock\_depth; /\*锁的深度\*/

struct mm\_struct \*mm; /\*进程内存管理信息\*/

int processor; /\*进程所使用的处理器\*/

unsigned long cpus\_runnable,cpus\_allowed;/\*若进程不在任何CPU 上运行，cpus\_runnable 的值是0,否则是1。

这个值在运行队列被锁时更新\*/

struct list\_head run\_list; /\*指向进程运行队列的指针\*/

unsigned long sleep\_time; /\*进程的睡眠时间\*/

struct task\_struct \*next\_task,\*prev\_task; /\*用于将系统中所有的 进程连成一个双向循环链表,其根是init\_task\*/

struct mm\_struct \*active\_mm /\*进程虚拟主存信息;内核线程借 用的地址空间\*/

struct list\_head local\_pages; /\*指向本地页面\*/

struct linux\_binfmt \*binfmt; /\*进程所运行的可执行文件的格式\*/

int exit\_code,exit\_signal; /\*退出信号,系统强行退出时发出的信 号\*/

int pdeath\_signal; /\*父进程终止时向子进程发送的信号\*/

pid\_t pid; /\*进程标识符,用来代表一个进程\*/

pid\_t pgrp; /\*进程组标识,表示进程所属的进程组\*/

pid\_t tty\_old\_pgrp; /\*进程控制终端所在的组标识\*/

pid\_t session; /\*进程的会话标识\*/

struct task\_struct \*p\_opptr,\*p\_pptr,\*p\_cptr,\*pysptr,\*p\_osptr;

/\*指向原始父进程(original parent)、父进程 (parent)、子进程(youngest child)及新老兄弟 进程(younger sibling,older sibling)的指针\*/

struct list\_head thread\_group; /\*线程链表\*/

struct task\_struct \*pidhash\_next; /\*用于将进程链入HASH表\*/

struct task\_struct \*\*pidhash\_pprev;

wait\_queue\_head\_t wait\_chldexit; /\*供wait4()使用\*/

struct completion \*vfork\_done; /\*供vfork()使用\*/

unsigned int rt\_priority; /\*实时进程的优先级。其范围为0~99\*/

unsigned long it\_real\_value,it\_prof\_value,it\_virt\_value;

unsigned long it\_real\_incr,it\_prof\_incr,it\_virt\_value;

struct timer\_list real\_timer; /\*指向实时定时器的指针\*/

struct tms times; /\*记录进程消耗的时间\*/

long per\_cpu\_utime[NR\_CPUS],per\_cpu\_stime[NR\_CPUS];

/\*记录进程在每个CPU上所消耗的用 户态时间和核心态时间\*/

unsigned long min\_flt,maj\_flt,mswap,cmin\_flt,cmaj\_flt,cnswap;

/\*min\_flt,maj\_flt累计进程的次缺页数(Copy on Write页 和匿名页)和主缺页数(从映射文件或交换设备读入的页 面数);nswap记录进程累计换出的页面数,即写到交换设 备上的页面数。cmin\_flt,cmaj\_flt,cnswap记录本进程为 祖先的所有子孙进程的累积次缺页数,主缺页数和换出 页面数\*/

uid\_t uid,euid,suid,fsuid; /\*用户的标识符、有效标识符、备份 标识符、网络环境下文件标识符\*/

gid\_t gid,egid,sgid,fsgid; /\*组的标识符、有效标识符、备份标 识符、网络环境下文件标识符\*/

int ngroups; /\*记录进程在多少个用户组中\*/

gid\_t groups[NGROUPS]; /\*记录进程所在的组\*/

struct user\_struct \*user; /\*用来唯一地标示一个用户,该进程所 属的用户\*/

kernel\_cap\_t cap\_effective,cap\_inheritable,cap\_permitted;

/\*进程的权能,分别是有效位集合、继承位集合、允许位集合\*/

struct rlimit rlim[RLIM\_NLIMITS]; /\*与进程相关的资源限制 信息\*/

unsigned short used\_math; /\*是否使用FPU\*/

char comm[16]; /\*进程正在运行的可执行文件名\*/

struct sem\_undo \*semundo; /\*进程在信号量上的所有undo操作\*/

struct sem\_queue \*semsleeping; /\*当进程因为信号量操作而挂 起时，它在该队列中记录等待的操作\*/

struct fs\_struct \*fs; /\*文件系统信息\*/

struct files\_struct \*files; /\*打开文件信息\*/

/\*信号处理函数\*/

spinlock\_t sigmask\_lock;

struct signal\_struct \*sig; /\*信号处理函数\*/

sigset\_t blocked; /\*进程当前要阻塞的信号,每个信号对应一位\*/

struct sigpending pending; /\*进程上是否有待处理的信号\*/

unsigned long sas\_ss\_sp;

size\_t sas\_ss\_size;

int (\*notifier)(void \*priv);

void \*notifier\_data;

sigset\_t \*notifier\_mask;

u32 parent\_exec\_id; /\*父进程的执行域\*/

u32 self\_exec\_id; /\*本进程的执行域\*/

spinlock\_t alloc\_lock; /\*分配mm、files、fs、tty等的自旋保 护锁\*/

void \*journal\_info;

...

};

需要指出的是，由于Linux内核版本的升级，task\_struct的成员结构也是在不断地变化中。高版本内核的task\_struct结构将部分的定义放在了其他文件中，但是其实现的功能依然遵循进程的基本功能要求。这也希望读者在阅读各个内核版本的源代码时体会其中的变化。

**Linux进程创建剖析**

在Linux系统中，进程的创建是由fork( )函数而创建，通过schedule( )函数调用执行，由wait( )函数而等待，最后通过给进程发送信号强行终止运行进程，或者在完成任务后进程执行exit( )函数自动退出而消亡。

Linux中在下列服务请求情况下，需要创建新进程。

（1）从后备批处理作业中选择新作业时，将创建一个新进程处理该作业。

（2）终端用户登录到系统时，将为其创建一个新进程。

（3）用户请求系统服务时，如打印文件，将为其创建一个系统进程。

（4）一个进程显示要求创建另一个进程，已完成并发操作。

系统允许一个进程创建新进程，新进程即为子进程，而生成子进程的进程为父进程。子进程还可以创建新的子进程，形成进程树结构模型。树根是系统自动构造的在内核态下执行的0号进程，它是所有进程的祖先。由0号进程创建的在内核态执行的1号进程，负责执行内核的部分初始化工作及进行系统配置，并创建若干个用于高速缓存和虚拟主存管理的内核线程。随后，内核态1号进程调用execve()运行可执行程序init，并演变成用户态1号进程，即init进程。它按照配置文件/etc/inittab的要求，完成系统启动工作，创建若干个终端注册进程getty。每个终端响应终端连接信号，启动注册程序login。如果用户启动成功，则要启动shell进程来取代getty进程。所有其他用户进程则由shell进程在用户空间直接或间接生成。

Linux系统创建进程主要使用fork( )函数，此外还使用vfork( )和clone( )来创建进程。这三个函数都调用同一个内核函数do\_fork( )实现具体的进程创建过程。

在创建新进程调用fork( )函数时，fork( )将父进程的所有资源通过数据结构的复制“传”给子进程。系统让新的进程与旧的进程使用同一个代码段，并且对于数据段和堆栈段，系统则复制一份给新的进程。这样，父进程的所有数据都可以留给子进程。但是，子进程一旦开始运行，虽然它继承了父进程的一切数据，但实际上地址空间已经分开，相互之间不再有影响了。也就是说，它们之间不再共享任何数据了。

fork( )不仅创建出与父进程代码相同的子进程，而且父进程在fork执行点的所有上下文也被自动复制到子进程中，包括以下内容。

1. 全局和局部变量。
2. 打开的文件句柄。
3. 共享内存、消息等同步对象。

由此可见，用fork( )创建进程时，子进程只是完全复制父进程的资源，子进程的执行独立于父进程，因而具有良好的并发性。但两个进程要共享数据，则需要通过专门的进程通信机制。

**下面通过代码fork\_test.c给出使用fork( )函数的基本过程。**

程序文件如下。

/\*\*\* fork\_test.c \*\*\*/

#include <unistd.h>

#include <stdio.h>

int main()

{

pid\_t fpid;

int count=0;

fpid=fork();

if(fpid < 0)

printf(“error in fork!”);

else if(fpid==0){

printf(“i am the child process,my process id is %d\n”,

getpid());

count++; }

else {

printf(“i am the parent process,my process id is %d\n”,

getpid());

count++; }

printf(“The counter result is：%d\n”,count);

return 0;

}

程序运行结果如下（注：进程ID号每次的执行是不一样的）。

i am the child process,my process id is 26213

The counter result is：1

i am the parent process,my process id is 26212

The counter result is：1

代码fork\_test.c编译执行后实际上会产生两个进程，这在程序运行结果中可以看出来。对于父进程，fork函数返回了子进程的进程号；而对于子进程，fork函数则返回零。这样，对于程序，只要判断fork函数的返回值，就知道自己是处于父进程还是子进程中。

在语句fpid=fork()之前，只有一个进程在执行这段代码。但在这条语句之后，就变成两个进程在执行了，这两个进程几乎完全相同，将要执行的下一条语句都是if(fpid<0)。

为什么两个进程的fpid不同呢，这与fork函数的特性有关。fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有以下三种不同的返回值。

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。

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程时子进程还是父进程。

fpid的值为什么在父子进程中不同呢？其实就相当于链表，进程形成了链表。父进程的fpid（p意为point）指向子进程的进程id，因为子进程没有子进程，所以其fpid为0。

fork出错可能有以下两种原因。

1. 当前的进程数已经达到了系统规定的上限，这时errno的值被设置为EAGAIN。
2. 系统内存不足，这时errno的值被设置为ENOMEM。

创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。

每个进程都有一个独特（互不相同）的进程标识符（process ID），可以通过getpid()函数获得，还有一个记录父进程pid的变量，可以通过getpid()函数获得变量的值。

fork执行完毕后，出现两个进程，如图9-1所示。

if(fpid<0){

...}

else if(fpid==0){

...}

else{

...}

printf(“The counter result is:%d\n”,count);

父进程代码

相同

父进程

if(fpid<0){

...}

else if(fpid==0){

...}

else{

...}

printf(“The counter result is:%d\n”,count);

子进程代码

图9-1 fork执行原理示意图

既然两个进程的内容完全一样，怎么输出的结果不一样？那是因为判断条件的原因。上面列举的只是进程的代码和命令，还有变量。

执行完fork后，进程1的变量为count=0，fpid!=0（父进程）。进程2的变量为count=0，fpid=0（子进程）。这两个进程的变量都是独立的，存在不同的地址中，不是共用的。可以说通过fpid可以识别和操作父子进程。

可能还有读者疑惑为什么不是从#include处开始复制代码的。这是因为fork是把进程当前的情况拷贝一份，执行fork命令时，进程已经执行完了int count=0语句。fork只拷贝下一个要执行的代码到新的进程。

进程创建后，如何去执行一个新的程序，单依靠fork是无法实现的。对具体程序的引用，Linux实现是通过使用exec( )函数实现。进程一旦调用exec( )，将准备执行的程序文件读入，代替发出调用进程的原先程序的执行。此时，系统把代码段替换成新程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段和堆栈段，唯一留下的就是进程号。对系统而言，仍然是同一个进程，只不过运行另一个可执行程序。所以，Linux新进程产生模式就是使用fork( )/exec( )组合来完成，先用fork( )创建新进程，然后新进程通过调用exec( )类执行自己的任务。

**三、实验内容**

**1**、**按照以下步骤学习使用Linux进程管理命令**

本实验主要了解进程查看命令ps、pstree、top，进程终止命令kill，前后台运行和暂停进程命令fg、bg、jobs、Ctrl+z，理解各命令的使用方式。

Linux进程管理的命令功能描述如下。

ps：显示瞬间进程的状态，不是动态的。常用格式：ps -ef、ps -aux。f参数以长格式输出。

top：性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似Windows的任务管理器。

pstree：将所有进程以树状图显示，树状图将会以pid（如果有指定）或是以init这个基本进程为根（root），可以指定显示某个用户的进程。

kill：终止进程。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。常用格式：kill pid或kill -9 pid。

jobs：查看当前在后台运行的程序。

fg：将后台的命令调至前台继续运行。格式：fg %jobId，这里不是进程的pid，是用jobs查到的jobid。

bg：将一个在后台暂停的命令，变成继续执行。格式：bg %jobId。

Ctrl+z：这是组合键，将一个正在前台执行的命令放到后台，并暂停。

实验操作步骤如下。

（1）在终端中，列出每个与当前shell有关的进程基本信息。

1）输入命令 # ps

2）了解显示各项的含义

（2）显示系统中所有进程的全面信息。

1）输入命令 # ps□r-f

2）了解显示各项的含义

（3）显示所有终端上所有用户有关进程的所有信息。

1）输入命令 # ps -aux

2）了解显示各项的含义

（4）显示当前所有进程，以及它们的树状结构。

1）输入命令 # pstree，重点观察pstree命令的进程树结构，体会init进程是所有进程的父进程。查看命令pstree的级别，并记录其父进程名。

2）输入命令：# ps -e或者# ps -e|more，查看进程init的进程号并记录。

3）pstree命令后面也可以跟进程号和用户名，分别显示某进程下的树状结构或某用户名下的树状结构。输入命令 # pstree pid号或者输入命令 # pstree用户名。

（5）动态显示系统当前的进程和状态，每隔5s的时间刷新一次。

1）输入命令 # top d 5

2）了解显示各项的含义，注意每隔5s数据刷新一次。

3）多次按“空格”键，可以不断刷新数据。

4）按“h”键，了解帮助，然后按“q”键退出该命令。

（6）用vi新建os.txt文件，使用进程终止命令kill将其终止。

1）输入命令 #vi os.txt

2)新建一个终端，输入命令 # ps -ef查看vi的进程PID。

3）终止该进程，输入命令 # kill vi的PID。

4）输入命令 # ps -ef，查看vi进程是否存在。

（7）让进程在前后台运行，并且能暂停前台正在运行的进程。

1）用vi编辑某个文档如test1.txt，用Ctrl+z组合键将其暂停，暂停后会返回如下结果。

[1] + Stopped vi test1.txt /\*其中方括号中的数字为任务编号\*/

在用vi编辑文档如test2.txt，用Ctrl+z组合键将其暂停。

2）jobs命令的功能是查看后台运行的程序，输入命令 # jobs，查看任务编号。

3）将vi test1.txt由后台暂停变为后台工作，用bg命令加上该任务的编号1将其置于后台运行。输入命令：

# bg test1.txt的进程号

4) 将vi test2.txt置于前台工作，用fg命令加上相应的任务编号，输入命令：# fg test2.txt的进程号

**2、按照以下步骤学习使用进程创建**

本实验目的是学会使用Linux进程创建函数fork( )，理解fork函数的工作原理，了解父进程与子进程的并发关系，通过循环创建子进程，明白各个进程的执行结果。

实验步骤如下。

（1）编写程序fork\_test1.c，使用系统调用fork( )创建进程，编译执行该程序，观察程序的运行结果，并分析原因。

程序文件如下。

/\*\*\*\*\* fork\_test1.c \*\*\*\*\*/

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int i=0;

printf("I son/pa ppid pid fpid/n"); /\*ppid指当前进程的父进程 pid,pid指当前进程的pid,

fpid指fork返回给当前进 程的值\*/

for(i=0;i<2;i++){

pid\_t fpid=fork();

if(fpid==0)

printf("%d child %4d %4d %4d\n",i,getppid(),getpid(),fpid);

else

printf("%d parent %4d %4d %4d/n",i,getppid(),getpid(),fpid);

}

return 0;

}

（2）试通过各进程pid号，分析fork\_test1.c产生了几个子进程，并且考虑需要执行几次printf语句。

（3）试给出每次循环的工作流程，画出该程序的执行流程图。

**3、按照以下步骤学习使用Linux下线程与进程的数据共享**

本实验的主要目的是理解Linux下线程和进程的工作原理和关系，重点了解线程和进程之间的数据共享关系。

和进程相比，线程的最大优点之一是数据的共享性。各个线程共享父进程处沿袭的数据段，可以方便得获得、修改数据。

在Linux系统中，提供内核支持的用户线程，可通过函数clone( )及pthread\_create( )创建线程。这两个函数的原型定义如下。

#include<pthread.h>

int clone(int (\*fn)(void\* arg),void \*stack,int flags,void\* arg)

int pthread\_create(pthread\_t \*tidp,const pthread\_attr\_t\* attr,void\* (\*start\_rtn)(void),void \*restrict arg);

注意：pthread\_create( )函数在调用现成的地址空间内创建了一个新线程来运行。参数tidp为指向线程标识符的指针，参数pthread\_attr\_t用来设置线程属性，使用默认属性时将NULL赋予该参数，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。

实验首先创建一个线程，然后在线程中更改进程中的数据。具体步骤如下。

（1）编写程序thread\_pro\_share.c，程序代码如下。

/\*\*\*\* thread\_pro\_share.c \*\*\*\*\*/

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

static int sharedata = 4;

void \*create(void\* arg){

printf("new thread...\n");

printf("shared data = %d\n",sharedata);

sharedata = 8;

return (void\*)0;

}

int main(){

/\*用thread\_create()创建新进程\*/

pthread\_t tidp;

int error = pthread\_create(&tidp,NULL,create,NULL);

if(error !=0){

printf("pthread\_create is not create ...");

return -1;

}

sleep(1); /\*等待一段时间\*/

printf("pthread\_create is create...\n");

printf("shared data = %d\n",sharedata);

return 0;

}

1. 编译该文件，写出程序运行结果。在编译时注意加上-lpthread参数，以调用静态链接库。因为pthread并非Linux系统的默认库，使用下面命令：

# gcc -lpthread -o thread\_pro\_share thread\_pro\_share.c

1. 分析sharedata的值变化，体会线程共享进程的数据空间。

**实验要求：**

（1）将实验1使用到的各命令及其结果，写入实验报告。

（2）描述fork( )函数的工作原理，理解子进程与父进程资源共享，将fork\_test1.c的分析写入实验报告。

（3）画出Linux系统创建进程的流程图，写入实验报告。

（4）分析线程与进程的数据共享关系，将实验结果写入实验报告。

实验十 Linux的进程通信

**一、 实验目的**

理解信号和管道的概念及实现进程间通信的原理；掌握和使用信号量实现进程同步。

**二、预备知识**

Linux进程间通信主要有信号、管道、消息队列、共享主存、信号量及套接字等方式。下面分别介绍每种通信方式的具体实现机制。

1、信号

信号（signal）是一种简洁的通信方式，用于通知接收进程有某种事件发生。除了用于进程间通信外，进程还可以发送信号给进程本身。事实上，信号机制是早期UNIX为进程中断（软中断）处理而设置的。它只是一组预定义的值，因此不能用于信息交换，仅用于进程中断控制。例如，在发生浮点错、非法内存访问、执行无效指令、按某些键（如Ctrl+c、Del等）都会产生一个信号，操作系统就会调用有关的系统调用或用户定义的处理过程来处理。

Linux除了支持UNIX早期信号语义函数signal( )外，还支持语义符合POSIX.1标准的信号函数sigaction( )。实际上，该函数是基于UNIX BSD的。UNIX BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数。如果进程要处理某一信号，那么就要在进程中安装该信号。安装信号主要用来确定信号值及进程针对该信号值的动作之间的映射关系，即进程将要处理哪个信号。该信号被传递给进程时，将执行何种操作。signal( )函数用来安装某个信号的处理函数。该函数的声明格式如下。

void (\*signal(int signum, void (\*handler)) (int))(int);

其中，参数signum指出要设置函数的信号。参数handler是一个处理函数。

kill( )函数用来向进程或进程组发送一个信号。该函数的声明格式如下。

int kill(pid\_t pid , int sig);

使用这两个函数时，必须包括头文件#include <signal.h>。

2、管道及命令管道

管道（pipe）可用于具有亲缘关系进程间的通信。父子进程之间或兄弟进程之间的通信，可以通过函数建立单向管道。管道两端的进程将管道视为一个文件，一个进程向管道写入数据，另一个进程从管道中读出数据。数据传递遵循“先进先出”的规则。

管道的创建通过函数pipe( )实现，其声明格式如下。

#include <unistd.h>

int pipe(int fd[2])

管道的另一种形式是命名管道（named pipe）。命名管道克服了管道没有名字的限制，以FIFO文件的形式出现在文件系统中。因此，任何进程都可以使用其文件名来打开文件，然后进行读写。命名管道除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信，命名管道实际是管道的推广。

命名管道的创建通过mkfifo( )实现，该调用函数的声明格式如下。

#include <sys/types.h>

#include<sys/stat.h>

int mkfifo(const char \* pathname,mode\_t mode)

3、消息队列

消息（message）队列是消息的链接表，在进程之间以传递消息形式进行通信，包括POSIX消息队列和System V消息队列。拥有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流及缓冲区大小受限等缺点。

消息是一个格式化的可变长的信息单位。消息机制允许一个进程向任何其他进程发送一个消息。当一个进程收到多个消息时，可将它们排成一个消息队列。

消息队列基本操作包括打开或创建消息队列、对消息队列进行读写操作、获取或设置消息队列属性。与这些操作相关的函数包括msgget( )、msgsnd( )、msgrcv( )和msgctl( )等。这些函数包含在以下三个头文件中。

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

（1）打开或创建消息队列

msgget( )可以创建一个新消息队列或打开一个存在的消息队列，该函数的原型定义为

int msgget(ket\_t key, int msgflg );

其中，参数key创建/打开队列的一个键值，由ftok()函数获得；msgflg参数指定创建/打开方式，可以是IPC\_CREAT、IPC\_EXCL、IPC\_NOWAIT或三者的或结果，通常是msgflg=IPC\_CREAT|IPC\_EXCL|0666

CL|0666。意思是，若不存在key值的队列，则创建；否则，如果存在则打开队列。0666表示与一般文件权限一样，指本用户、同组用户、其他用户的读写权限。

在以下两种情况下，该系统调用函数将创建一个新的消息队列。①如果没有消息队列与键值key相对应，并且msgflg中包含了IPC\_CREAT标志位；②key参数为IPC\_PRIVATE。

需要注意，IPC\_CREAT标志位表示创建，一般由服务器创建消息队列时使用。如果是客户程序，必须打开现有的消息队列，且不使用IPC\_CREAT。

调用成功返回消息队列描述字，否则返回-1。

（2）读写操作

消息队列的读写比较简单，每个消息都有类似如下数据结构。

struct msgbuf{

long mtype; /\*消息类型\*/

char mtext[]; /\*消息文本\*/

};

读取消息的函数原型定义如下。

int msgrcv(int msgid, struct msgbuf \*msgp, int msgsz, long msgtyp, int msgflg);

该系统调用从msgid代表的消息队列中读取一个消息，并把消息存储在msgp指向的msgbuf结构中。msgid为消息队列描述字；消息返回后存储在msgp指向的地址，msgsz指定msgbuf的mtext成员的长度（即消息内容的长度），msgtyp为请求读取的消息类型。读消息标志msgflg可以为以下几个常值的逻辑或。

IPC\_NOWAIT如果没有满足条件的消息，调用立即返回。此时，errno=ENOMSG。

IPC\_EXCEPT与msgid>0配合使用，返回队列中第一个类型不为msgtyp的消息。

IPC\_NOERROR如果队列中满足条件的消息内容大于所请求的msgsz字节，则把该消息截断，截断部分将丢失。

对于msgrcv( )，内核将完成下述工作。

·对消息队列的描述符和许可权进行检查。若合法，继续执行；否则返回。

·根据type值的不同分为三种情况处理。

type=0：接收该队列的第一条消息，并将它返回给调用者；

type>0：接收类型type的第一条消息。

type<0：接受小于等于type绝对值的最低类型的第一条消息。

·当所返回消息大小等于或小于用户的请求时，内核便将消息正 文复制到用户区，并从信息队列中删除此消息，然后唤醒睡眠 的发送进程。但如果消息长度比用户要求大时，则返回出错， 值为-1。

发送消息的函数原型定义如下。

int msgsnd(int msgid, struct msgbuf \*msgp, int msgsz, int msgflg);

参数说明与msgrcv( )相似，对发送消息来说，有意义的msgflg标志为IPC\_NOWAIT，指明在消息队列没有足够空间容纳要发送的消息时，msgsnd是否等待。造成msgsnd()等待的条件有两种：

·当前消息的大小与当前消息队列中的字节数之和超过了消息队 列的总容量。

·当前消息队列的消息数（单位“个”）不小于消息队列的总容 量（单位“字节数”）。此时，虽然消息队列中的消息数目很多， 但基本上都只有一个字节。

对于msgsnd( )函数，内核必须完成以下工作。

·对消息队列的描述符、许可权及消息长度等进行检查。若合法 才继续执行，否则返回。

·内核为消息分配消息数据区，将用户消息缓冲区中的消息正文 复制到消息数据区。

·分配消息首部，并将它链入消息队列队尾。在消息首部中必须 填写消息类型、消息大小和指向消息数据区的指针等数据。

·修改消息队列头中的数据，如队列中的消息数和字节总数等。 最后，唤醒等待消息的进程。

（3）消息队列属性操作

消息队列中的控制操作指读取消息队列的状态信息并进行修改，如查询消息队列描述符、修改它的许可权及删除该队列等。其函数原型定义如下。

int msgctl(int msgid, int cmd, struct msgid\_ds \*buf);

其中，cmd是规定的命令，buf是用户缓冲区地址。

cmd命令规定分为三类：PC\_STAT、IPC\_SET、IPC\_RMID。各命令的意义如下。

IPC\_STAT：该命令用来获取消息队列信息，返回的信息存储在buf指向的msgid结构中。

IPC\_SET：该命令用来设置消息队列的属性，要设置的属性存储在buf指向的msgid结构中；可设置属性包括msg\_perm.uid、msg\_perm.gid、msg\_perm.mode及msg\_qbytes，同时，也影响msg\_ctime成员。

IPC\_RMID：删除msgid标识的消息队列。

调用成功返回0，否则返回-1。

4、共享主存

共享主存（shared memory）是一种简单而高效的进程通信方法，它使得多个进程可以访问同一块内存空间。共享主存方式通常与其他通信机制，如信号量结合使用，来达到进程间的同步及互斥。Linux的2.6内核支持多种共享主存方式，如mmap()系统调用和System V共享主存。System V 共享主存指的是把所有共享数据放在共享主存区域（IPC sharedmemory region），任何想要访问该数据的进程都必须在本进程的地址空间新增一块内存区域，用来映射存放共享数据的物理内存页面。System V共享主存通过shmget()获得（或创建）一个IPC共享主存区域，并返回相应的标识符。内核在保证shmget获得或创建一个共享主存区并初始化该共享主存区相应的shmid\_kernel结构同时，还将在特殊文件系统shm中，创建并打开一个同名文件，并在内存中建立起该文件的相应dentry及inode结构。需要注意的是新打开的文件不属于任何一个进程，任何进程都可以访问该共享主存区。所有这一切都是系统调用shmget完成的。

每一个共享主存区都有一个控制结构struct shmid\_kernel。shmid\_kernel是共享主存区域中非常重要的一个数据结构，它是存储管理和文件系统结合起来的桥梁，定义如下。

struct shmid\_kernel{ /\* private to the kernel\*/

struct kern\_ipc\_perm shm\_perm;

struct file \* shm\_file;

int id;

unsigned long shm\_nattch;

unsigned long shm\_segsz; /\*共享主存的大小\*/

time\_t shm\_atim; /\*最后一次attach此共享主存的时间\*/

time\_t shm\_dtim; /\*最后一次detach此共享主存的时间\*/

time\_t shm\_ctim; /\*最后一次更改此共享主存结构的时间\*/

pid\_t shm\_cprid; /\*建立此共享主存的进程识别码\*/

pid\_t shm\_lprid; /\*最后一次操作此共享主存的进程码\*/

};

对于System V 共享主存涉及的函数主要有以下几个系统调用函数：shmget( )、shmat( )、shmdt( )及shmctl( )，这些函数包括在以下两个头文件中。

# include <sys/ipc.h>

# include <sys/shm.h>

shmget( )用来获得共享主存区域的ID，如果不存在指定的共享区域就创建相应的区域。shmat( )把共享主存区域映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区域进行访问操作。shmdt( )调用用来解除进程对共享主存区域的映射。shmctl实现对共享主存区域的控制操作。

需要注意的是，shmget的内部实现包含了许多重要的System V共享主存机制；shmat在把共享主存区域映射到进程空间时，并不真正改变进程的页表。当进程第一次访问内存映射区域访问时，会因为没有物理页表的分配而导致一个缺页异常，然后内核再根据相应的存储管理机制为共享主存映射区域分配相应的页表。

各函数的原型定义如下。

函数shmget( )可以创建或打开一块共享内存区。函数原型如下。

int shmget( key\_t key, size\_t size, int flag);

其中，函数中参数key用来变换成一个标识符，而且每一个IPC对象与一个key相对应。当新建一个共享内存段时，参数size为要请求的内存长度（以字节为单位）。

函数shmctl( )可以对共享内存段进行多种操作，其函数原型如下。

int shmctl( int shm\_id, int cmd, struct shmid\_ds \*buf );

其中，函数中参数sh\_mid为所要操作的共享内存段的标识符，struct shmid\_ds型指针参数buf的作用与参数cmd的值相关，参数cmd指明了所要进行的操作。

函数shmat( )将一个存在的共享内存段连接到本进程空间，其函数原型如下。

void \*shmat( int shm\_id, const void \*addr, int flag);

其中，参数shm\_id指定要引入的共享内存的标识符。addr是指针变量，一般情况下为NULL，这是系统用进程中的首个适用地址来映射共享主存，并把地址赋值给addr。除非特殊情况，一般不会给addr赋初值。flag规定共享主存区的读、写权限，以及系统是否应对用户规定的地址做舍入操作。其值为SHM\_RDONLY时，表示只能读；其值为0时，表示可读、可写；其值为SHM\_RND（取整）时，表示内核在必要时要舍去这个地址。

函数shmdt( )是当对共享内存段操作结束时，调用此函数，作用是将指定的共享内存段从当前进程空间中脱离出去。函数原型如下。

int shmdt( void \*addr) ;

其中，参数addr是调用shmat函数的返回值，函数执行成功返回0，并将该共享主存的shmid\_ds结构的shm\_nattch计数器减1，失败返回-1。

5、信号量

信号量（semaphore）主要用做用户空间中进程之间及同一进程不同线程之间的同步手段，是内核信号量机制的一种推广。

信号量机制与其他进程间通信方式不大相同，它主要提供对进程间共享资源访问控制机制。信号量相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，同时，进程也可以修改该标志。除了用于访问控制外，还可用于进程同步。信号量有以下两种类型：二值信号量（取0或1）和一般信号量（取任意非负整数值）。

信号量的基本操作包括打开或创建信号量、信号量值操作、获取或设置信号量属性。相关的函数包括semget()、semop()和semctl()，它们被包含在如下三个头文件中。

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

（1）打开或创建信号量

semget( )可以创建一个新信号量或打开一个现有的信号量。该函数的原型定义如下。

int semget( key\_t key, int nsems, int semflg );

其中，参数key是一个键值，由ftok获得，唯一标识一个信号量集，用法与msgget()中的key相同。参数nsems指定打开或者新创建的信号量集中将包含信号量的数目。semflg参数是一些标志位。参数key和semflg的取值，以及何时打开已有信号量集或者创建一个新的信号量集与msgget()中的对应部分相同。

调用返回：成功返回信号量集描述字，否则返回-1。

（2）信号量值操作

进程可以增加或减少信号量值，相对于对共享资源的释放和占有，操作函数的原定义如下。

int semop( int semid, struct sembuf \*sops, unsigned nsops) ;

其中，semid是信号量集ID，sops指向数组的每一个sembuf结构都刻画一个在特定信号量上的操作。nsops为sops指向数组的大小。

sops在/include/linux/sem.h中表明，是主存中的一个数据结构，描述如下。

sembuf结构如下。

struct sembuf {

unsigned short sem\_num; /\*在数组中信号量的索引值\*/

short sem\_op; /\*信息量操作值\*/

short sem\_flg; /\*操作标志，为IPC\_NOWAIT 或SEM\_UNDO\*/

};

其中，sem\_num对一个信号量集中的信号量，0对应第一个信号量。sem\_flg可取PC\_NOWAIT及SEM\_UNDO两个标志。如果设置了SEM\_UNDO标志，那么在进程结束时，相应的操作将被取消，这是比较重要的一个标志位。如果设置了该标志位，那么在进程没有释放共享资源就退出时，内核将代为释放。如果为一个信号量设置了该标志，内核都要分配一个sem\_undo结构来记录它，为的是确保以后资源能够安全释放。事实上，如果进程退出了，那么它所占用的资源就释放了，但信号量值却没有改变。此时，信号量值反映的已经不是资源占有的实际情况。在这种情况下，问题的解决就靠内核来完成。这有点像僵尸进程，进程虽然退出了，资源也都释放了，但内核进程表中仍然有它的记录，此时就需要父进程调用waitpid来解决问题。

根据sem\_op的值大于0、等于0及小于0确定了对sem\_num指定的信号量进行的三种操作。

这里需要强调的是semop同时操作多个信号量，在实际应用中，对应多种资源的申请或释放。semop保证操作的原子性，这一点尤为重要。尤其对于多种资源的申请来说，要么一次性获得所有资源，要么放弃申请，要么在不占有任何资源情况下继续等待。这样，一方面避免了资源的浪费；另一方面，避免了进程之间由于申请共享资源造成死锁。

信号量的当前值记录相应资源目前可用数目；sem\_op>0对应相应进程要释放sem\_op数目的共享资源；sem\_op=0可以用于对共享资源是否已用完的测试；sem\_op<0相当于进程要申请-sem\_op个共享资源。再联想操作的原子性，更不难理解该系统调用何时正常返回、何时睡眠等待。

调用成功返回0，否则返回-1。

（3）信号量属性操作

系统中的每个信号量集都对应一个struct sem\_array结构，该结构记录信号量集的各种信息，存在于系统空间。为了设置、获得该信号量集的各种信息及属性，在用户控件有一个重要的联合结构与之对应，即 union senum。这个特殊的联合结构定义在include/linux/sem.h中声明。而信号量属性操作函数就负责两者之间各种控制操作。该函数原型定义如下。

int semctl(int semid, int semnum,int cmd,union semun arg);

该系统调用实现对信号量的各种控制操作，参数semid指定信号量集，参数cmd指定具体的操作类型；参数semnum指定对哪个信号量操作，只对几个特殊的cmd操作有意义；arg用于设置或返回信号量信息。

6、套接口

套接口（socket）为更为一般的进程间通信机制，主要用于计算机网络中不同机器之间的进程间通信。起初是有UNIX系统的BSD分支开发出来的，但现在经过该移植，Linux内核同样也支持套接字的通信方式。

**三、实验内容**

**1、按照以下步骤学习使用进程的管道通信**

本实验要求父进程与两个子进程通过管道函数pipe( )建立一个管道线。子进程P1和P2分别向管道写入一句话：

Child 1 is sending a message!

Child 2 is sending a message!

而父进程则从管道中读出两个子进程的消息，显示在屏幕上，要求父进程先接收子进程P2发来的消息，然后再接收子进程P1发来的消息。

实验具体步骤如下：

（1）编写程序pipe\_test.c，使用系统调用pipe( )实现实验要求，编译执行该程序，观察程序的运行结果。

程序参考代码如下：

/\*\*\*\*\* pipe\_test.c \*\*\*\*\*/

#include<unistd.h>

#include<stdlib.h>

#include<signal.h>

#include<stdio.h>

int pid1,pid2;

int lockf(int files,int function,long size);

main()

{

int fd[2];

char OutPipe[100],InPipe[100];

pipe(fd);

while((pid1 = fork()) == -1);

if(pid1 == 0)

{

lockf(fd[1],1,0);

sprintf(OutPipe,"child 1 process is sending message!");

write(fd[1],OutPipe,50);

sleep(5);

lockf(fd[1],0,0);

exit(0);

}

else

{

while((pid2 = fork()) == -1);

if(pid2 == 0)

{

lockf(fd[1],1,0);

sprintf(OutPipe,"child 2 process is sending message!");

write(fd[1],OutPipe,50);

sleep(5);

lockf(fd[1],0,0);

exit(0);

}

else

{

wait(0);

read(fd[0],InPipe,50);

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

wait(0);

read(fd[0],InPipe,50);

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

exit(0);

}

}

}

（2）根据参考代码，思考下述问题

1）给出父子进程的使用管道的先后顺序。是子进程依次写入，父进程再依次读出呢，还是子进程写入，父进程就读出，然后另一进程写入，父进程再读出。通过插入提示语句验证之。

2）分析父进程wait(0)的作用。

3）分析在实验过程遇到的自己感兴趣的问题。

**2、按照以下步骤学习使用消息队列实现进程间的通信**

本实验利用消息队列解决客户及服务进程之间的通信问题。利用消息队列的系统调用函数编写一个长度为1KB的消息发送和接收程序。建立两个子进程，SERVER和CLIENT进行通信，SERVER端建立一个KEY为75的消息队列，等待其他进程发来的消息。当遇到类型为1的消息，则作为结束信号，取消该队列，并退出SERVER。SERVER每接收到一个消息后显示一句“（SERVER）RECEIVED”。CLIENT端使用KEY为75的消息队列，先后发送类型从10到1的消息，然后退出。最后的一个消息，即是SERVER端需要的结束信号。CLIENT每发送一条消息后会显示一句“（CLIENT）SENT”。父进程在SERVER和CLIENT均退出后结束。

实验具体步骤如下：

（1）编写程序msg\_test.c，使用消息队列系统调用函数实现本实验要求，编译执行程序，观察程序的运行结果。

程序参考代码如下。

/\*\*\*\*\*msg\_test.c \*\*\*\*\*/

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/msg.h>

#include<sys/ipc.h>

#define MSGKEY 75

struct msgform

{

long mtype;

char mtext[1024];

}msg;

int msgqid,i;

void CLIENT( )

{

int i;

msgqid = msgget(MSGKEY,0777);

for(i = 10; i >= 1; i--)

{

msg.mtype = i;

printf("(client)sent\n");

msgsnd(msgqid,&msg,1024,0);

}

exit(0);

}

void SERVER( )

{

msgqid = msgget(MSGKEY,0777|IPC\_CREAT);

do

{

msgrcv(msgqid,&msg,1030,0,0);

printf("(server)received\n");

}

while(msg.mtype != 1);

msgctl(msgqid,IPC\_RMID,0);

exit(0);

}

main( )

{

while((i = fork( )) == -1);

if(!i) SERVER( );

while((i = fork( )) == -1);

if(!i) CLIENT( );

wait(0);

}

（2）根据参考代码，思考如下问题。

1）理解消息传送和接收的机制，理论上应该是当CLIENT发送一个消息后，SERVER接收该消息，CLIENT再发送一条。也就是说“(CLIENT)SENT”和“(SERVER)RECEIVED”的字样应该在屏幕上交替出现。写出你看到的实验结果情况，有什么办法可以出现交替发送和接收。

2）想办法改进程序，能够给将发送和接收的消息显示出来。

**3、按照以下步骤学习使用信号量实现进程同步**

本实验利用信号量解决生产者—消费者问题。生产者—消费者问题是典型的进程间同步问题，其本质是如何控制并发进程对有界共享主存区的访问。生产者进程生产产品，然后将产品放置在一个空缓冲区中供消费者进程消费。消费者进程从缓冲区中获得产品，然后释放缓冲区。假设有一个生产者和两个消费者进程，创建了fullid、emptyid和mutxid共三个信号量，供进程间同步访问临界区。同时建立四个共享存储区，其中array用于维护生产者、消费者进程之间的共享数据，sum保存当前的求和结果，而set和get分别记录当前生产者进程和消费者进程的读写次数。

实验具体步骤如下。

（1）编写程序sem\_test.c，使用信号量系统调用函数实现生产者和消费者进程，编译执行程序，观察程序的运行结果。

程序参考代码如下。

#include<sys/mman.h>

#include<stdlib.h>

#include<sys/types.h>

#include<linux/sem.h>

#include<fcntl.h>

#include<unistd.h>

#include<stdio.h>

#include<errno.h>

#include<time.h>

#define MAXSEM 5

/\* 声明三个信号量ID \*/

int fullid;

int emptyid;

int mutxid;

int main()

{

struct sembuf P,V;

union semun arg;

/\* 声明共享内存 \*/

int \*array;

int \*sum;

int \*set;

int \*get;

/\* 映射共享内存 \*/

array=(int\*)mmap(NULL,sizeof(int)\*5,PROT\_READ|PROT\_WRITE,MAP\_SHARED|MAP\_ANONYMOUS,-1,0);

sum=(int\*)mmap(NULL,sizeof(int),PROT\_READ|PROT\_WRITE,MAP\_SHARED|MAP\_ANONYMOUS,-1,0);

get=(int\*)mmap(NULL,sizeof(int),PROT\_READ|PROT\_WRITE,MAP\_SHARED|MAP\_ANONYMOUS,-1,0);

set=(int\*)mmap(NULL,sizeof(int),PROT\_READ|PROT\_WRITE,MAP\_SHARED|MAP\_ANONYMOUS,-1,0);

\*sum = 0;

\*get = 0;

\*set = 0;

/\* 创建信号量 \*/

fullid = semget(IPC\_PRIVATE,1,IPC\_CREAT|00666);

emptyid = semget(IPC\_PRIVATE,1,IPC\_CREAT|00666);

mutxid = semget(IPC\_PRIVATE,1,IPC\_CREAT|00666);

/\* 为信号量赋值 \*/

arg.val = 0;

if(semctl(fullid, 0,SETVAL,arg) == -1)

perror("semctl setval error");

arg.val = MAXSEM;

if(semctl(emptyid, 0,SETVAL,arg) == -1)

perror("semctl setval error");

arg.val = 1;

if(semctl(mutxid, 0,SETVAL,arg) == -1)

perror("semctl setval error");

/\* 初始化P,V操作 \*/

V.sem\_num = 0;

V.sem\_op = 1;

V.sem\_flg = SEM\_UNDO;

P.sem\_num = 0;

P.sem\_op = -1;

P.sem\_flg = SEM\_UNDO;

/\* 生产者进程 \*/

if(fork( ) == 0) {

int i = 0;

while(i < 100)

{

semop(emptyid,&P,1);

semop(mutxid,&P,1);

array[\*(set)%MAXSEM] = i + 1;

printf("Producer %d\n",array[\*(set)%MAXSEM]);

(\*set)++;

semop(mutxid,&V,1);

semop(fullid,&V,1);

i++;

}

sleep(10);

printf("Producer is over");

exit(0);

}else {

/\* ConsumerA 进程 \*/

if(fork( ) == 0) {

while(1) {

semop(fullid, &P, 1);

semop(mutxid, &P, 1);

if(\*get == 100)

break;

\*sum += array[(\*get)%MAXSEM];

printf("The ConsumerA Get Number %d\n",

array[(\*get)%MAXSEM]);

(\*get)++;

if( \*get == 100)

printf("The sum is %d\n",\*sum);

semop(mutxid,&V,1);

semop(emptyid,&V,1);

sleep(1);

}

printf("ConsumerA is over");

exit(0);

}else {

/\* Consumer B 进程 \*/

if(fork( ) == 0) {

while(1) {

semop(fullid, &P, 1);

semop(mutxid, &P, 1);

if(\*get == 100)

break;

\*sum += array[(\*get)%MAXSEM];

printf("The ConsumerA Get Number %d\n",

array[(\*get)%MAXSEM]);

(\*get)++;

if( \*get == 100)

printf("The sum is %d\n",\*sum);

semop(mutxid,&V,1);

semop(emptyid,&V,1);

sleep(1);

}

printf("ConsumerA is over");

exit(0);

}

}

}

/\* sleep(20) \*/

return 0;

}

（2）根据参考代码，思考下述问题。

1）关注程序中P、V操作的变化，体会信号量实现进程同步的实现机制。

2）掌握信号量机制的系统调用函数的使用方法。

3）描述程序的运行结果。

实验十一Linux的内存管理

**一、 实验目的**

掌握基本的内存管理知识，能够实时查看内存、回收内存；掌握请求页式存储管理的页面调度算法。

**二、预备知识**

1、Linux内存管理命令

Linux系统的内存管理命令主要使用free来监控内存的使用情况，通过命令ps查看进程的状态，使用命令kill关闭进程从而释放内存，此外还有用命令vmstat统计虚拟内存的使用情况。对于命令ps与kill出在第5章已经有了介绍，这里主要关注命令free与vmstat。（1）free命令

free命令会显示内存的使用情况，包括实体内存、虚拟的交换文件内存、共享内存区段，以及系统内核使用的缓冲区等。如图11-1所示是free命令执行后结果显示。

图11-1free命令执行结果显示

1）显示内容的第一部分为Mem行：total表示内存总数1999MB；used表示已经使用的内存数660MB；free表示空闲的内存数1339MB；shared表示共享内存，系统一般不会用到，值常为0；buffers表示系统分配但未被使用的buffers数量77MB；主要用来给块设备做缓冲；cached表示系统分配但未被使用的cache数量270MB，主要用来给文件做缓冲。显示内容有如下关系。

total(1999MB) =used (660MB) +free (1339MB)。

2）显示内容的第二部分为(-/+buffers/cache)行。

(-buffers/cache)used内存数：321MB（指的第一部分Mem行中的used-buffers-cached)

(+buffers/cache)free内存数：1686MB(指的第一部分Mem行中的free + buffers + cached)

可见-buffers/cache反映的是被程序实实在用掉的内存，而+buffers/cache反映的是可以挪用的内存总数。

3）显示内容的第三部分是指交换分区行。

第一部分（Mem）与第二部分（-/+buffers/cache）的结果中都有used和free项，这可以从两个方面来解释。

对操作系统来讲这两项是Mem的参数。buffers/cached都是属于被使用，所以它认为free 只有245MB的大小。

对应用程序来讲这两项是buffers/cached的使用情况。buffers/cached是等同可用的。因为buffers/cached是为了提高程序执行的性能，当程序使用内存时，buffer/cached会很快地被使用。

此外，Linux为了提高磁盘和内存存取效率，做了很多精心的设计，除了对dentry进行缓存（用于VFS，加速文件路径名到inode的转换），还采取了两种Cache方式：Buffer Cache和PageCache。前者针对磁盘块的读写，后者针对文件inode的读写。这些Cache能有效缩短I/O系统调用（如read、write、getdents）的时间。

（2）vmstat命令

vmstat命令是一个查看虚拟内存(Visual Memory)使用状况的工具，下面首先介绍Linux中关于虚拟内存相关内容。

在系统中运行的每个进程都需要使用到内存，但不是每个进程都需要每时每刻使用系统分配的内存空间。当系统运行所需内存超过实际的物理内存，内核会释放某些进程所占用但未使用的部分或所有物理内存，将这部分资料存储在磁盘上直到进程下一次调用，并将释放出的内存提供给有需要的进程使用。

在Linux内存管理中，主要是通过“调页（Paging）”和“交换（Swapping）”来完成上述的内存调度。调页算法是将内存中最近不常使用的页面换到磁盘上，把活动页面保留在内存中供进程使用。交换技术是将整个进程，而不是部分页面，全部交换到磁盘上。

分页（Page）写入磁盘的过程被称为Page-Out。分页从磁盘重新回到内存的过程被称为Page-In。当内核需要一个分页时，但发现此分页不在物理内存中（因为已经被Page-Out了），此时就发生了分页错误（Page Fault）。

当系统内核发现可运行内存变少时，就会通过Page-Out来释放一部分物理内存。尽管Page-Out不是经常发生，但是如果Page-Out频繁不断发生，直到当内核管理分页的时间超过运行程式的时间时，系统效能会急剧下降。这时的系统已经运行非常慢或进入暂停状态，这种状态亦被称为thrashing（抖动）。

下面是命令vmstat显示参数说明。

输入命令# vmstat 2，每2s输出一条结果，如图11-2所示。其中，procs为进程区信息：r表示运行队列中进程数量，b表示等待I/0的进程数量。

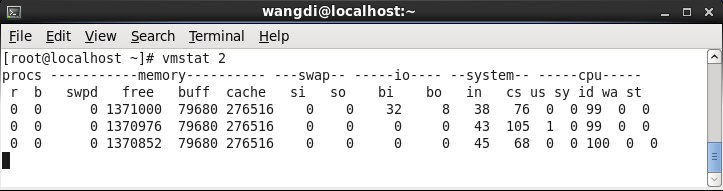


图11-2 vmstat命令每隔2s执行结果显示

memory为内存区信息：swpd表示使用虚拟内存大小，free可表示可用内存大小，buff表示用作缓冲的内存大小，cache表示用作缓存的内存大小。

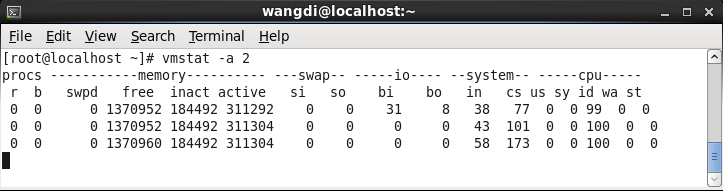
swap为交换区信息：si表示每秒从交换区写到内存的大小，so表示每秒写入交换区的内存大小。

io为输入输出区信息：bi表示每秒读取的块数，bo表示每秒写入的块数。

system为系统区信息：in表示每秒中断数，包括时钟中断，cs表示每秒上下文切换数。

cpu为CPU信息，以百分比表示：us表示用户进程执行时间（user time），sy为系统进程执行时间（system time），id表示cpu空闲时间，wa表示等待I/0时间。

vmstat还可以显示活跃和非活跃内存，输入命令# vmstat -a  2，结果如图11-3所示。

图11-3 vmstat命令每隔2秒执行结果显示

使用-a选项显示活跃和非活跃内存时，所显示的内容除增加inact和active 不显示buffer和cache外，其他显示内容与不用-a选项结果相同。

这里，inact表示非活跃内存大小，active表示活跃的内存大小，这两者信息均当使用-a 选项时显示。

最后需要说明的是，除了使用命令free与vmstat实现实时监控内存使用情况，另外也可以使用在Linux下提供的虚拟目录/proc进行内存信息的查看，输入命令# cat /proc/meminfo, 试着理解显示的结果信息。

**2、Linux内存页面置换算法**

在Linux中，为了提高内存利用率，提供了内、外存进程对换机制；内存空间的分配和回收均以页为单位进行；一个进程只需将其一部分（段或页）调入内存便可运行；还支持请求调页的存储管理方式。当进程在运行中需要访问某部分程序和数据时，发现其所在页面不  
在内存，就立即提出请求（向CPU发出缺页中断），由系统将其所需页面调入内存。这种页面调入方式叫请求调页。为实现请求调页，核心配置了几种数据结构：页表、页框号、访问位 、修改位、有效位、保护位等。

当CPU接收到缺页中断信号，中断处理程序先保存现场，分析中断原因，转入缺页中断处理程序。该程序通过査找页表，得到该页所在外存的物理块号。如果此时内存未满，能容纳新页，则启动磁盘I/0将所缺之页调入内存，然后修改页表。如果内存已满，则需按某种置换算法从内存中选出一页准备换出，是否重新写盘由页表的修改位决定，然后将缺页调入，修改页表。利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。

常用的页面置换算法如下。

（1）最佳置换算法(OPT，Optimal)。

（2）先进先出法(FIFO，First In First Out)。

（3）最近最久未使用(LRU，Least Recently Used)。

（4）最不经常使用法(LPU，Least Frequently Used)。

（5）最近未使用法(NUR，No Used Recently)。

**三、实验内容**

**1、按照以下步骤学习使用Linux内存管理命令使用**

本实验的目的是通过使用free与vmstat命令，体会上述命令的管理功能。

（1）用命令free监控内存使用情况。列出当前内存的使用情况。

1）输入命令# free -m。

2）了解显示各项的含义。

3）将显示的数据填入表11-1中，并通过计算来理解各参数之间的关系。

表11-1 free参数表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Total | used | free | shared | buffers | cached |
| Mem： |  |  |  |  |  |  |
| -/+buffers/cache |  |  |  |  |  |  |
| Swap: |  |  |  |  |  |  |

每隔3s列出当前内存的使用情况。输入命令# free -s3。

（2）用命令vmstat见识虚拟内存的使用情况。

1）输入命令# vmstat。

2）用man命令，了解显示各项的含义。

3）将vmstat显示的数据填入表11-2中。

表11-2 vmstat参数表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| pros | | memory | | | | swap | | io | | system | | cpu | | | | |
| r | b | swpd | free | buff | cache | si | so | bi | bo | in | cs | us | sy | id | wa | st |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**2、按照以下步骤学习使用模拟实现请求页式管理页面置换算法**

本实验目的要设计一个虚拟存储区和内存工作区，并使用下列页面置换算法计算访问命中率。

（1）先进先出的算法(FIFO)。

（2）最近最少使用的算法(LRU)。

（3）最佳淘汰算法(OPT)。

（4）最少访问页面算法(LFU)。

（5）最近最不经常使用算法(NUR)。

其中，命中率的计算公式为命中率=1-页面失效次数/页地址流长度。

首先用srand( )和rand( )函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。

（1）通过随机数产生一个指令序列，共320条指令。指令的地址按下述原则生成。

A:50%的指令是顺序执行的。

B:25%的指令是均匀分布在前地址部分。

C:25%的指令是均匀分布在后地址部分。

具体的实施方法如下。

A：在[0,319]的指令地址之间随机选取一起点m。

B：顺序执行一条指令，即执行地址为m+1的指令。

C：在前地址[0，m+1]中随机选取一条指令并执行，该指令的地址为m’。

D：顺序执行一条指令，其地址为m’+l。

E：在后地址[m’+2，319]中随机选取一条指令并执行。

F：重复步骤A-E，直到执行320次指令。

（2）将指令序列变换为页地址流。

设：页面大小为1KB。

用户内存容量4页～32页。

用户虚存容量为32KB。

在用户虚存中，按每KB存放10条指令排列虚存地址，即320条指令在虚存中的存放方式如下。

第0条～第9条指令为第0页（对应虚存地址为[0,9]）。

第10条～第19条指令为第1页（对应虚存地址为[10,19]）。

……

第310条～第319条指令为第31页（对应虚存地址为[310，319]）。

按以上方式，用户指令可组成32页。

**实验具体步骤如下：**

（1）给出实验所需的数据结构、函数及变量等定义。模拟实验首先用srand( )和rand( )函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。相关定义如下。

1）数据结构。

页面类型。

typedef struct{

int pn, pfn, counter, time ;

}pl-type；

其中pn为页号，pfn为面号，counter为一个周期内访问该页面的次数，time为访问时间。

页面控制结构。

pfc-struct {

int pn, pfn ；

struct pfc\_struct \*next；

}

typedef struct pfc\_struct pfc\_type;

pfc\_type pfc\_struct{total\_vp},\*freepf\_head,\*busypf\_head;

pfc\_type \*busypf\_tail;

其中，pfc[total\_vp]定义用户进程虚页控制结构。\*freepf\_head为空页面头的指针 。\*busypf\_head为忙页面头的指针。\*busypf\_tail为忙页面尾的指针。

2）函数定义。

void initialize( )：初始化函数，给每个相关的页面赋值。

voidFIF0( )：计算使用FIFO算法时的命中率。

voidLRU( )：计算使用LRU算法时的命中率。

void0PT( )：计算使用0PT算法时的命中率。

voidLFU( )：计算使用LFU算法时的命中率。

voidNUR( )：计算使用NUR算法时的命中率。

3）变量定义。

int a [total\_instruction]：指令流数据组。

int page[total\_instruction]:每条指令所属的页号。

int offset[total\_instruction]：每页装入10条指令后取模运算页号偏移值。

int total\_pf：用户进程的内存页面数。

int disaffect：页面失效次数。

（2）编写程序mm\_test.c，实现六种页面置换算法，编译执行程序，观察程序的运行结果。程序参考代码如下。

/ \* \* \* \* \* mm\_test. c\*\*\*\*\* /

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define total\_instruction 320 /\*指令流长\*/

#define total\_vp 32 /\*虚页长\*/

#define clear\_period 50 /\*清0周期\*/

typedef struct /\*页面结构\*/

{

int pn,pfn,counter,time;

}pl\_type;

pl\_type pl[total\_vp]; /\*页面结构数组\*/

struct pfc\_struct{ /\*页面控制结构\*/

int pn,pfn;

struct pfc\_struct \*next;

};

typedef struct pfc\_struct pfc\_type;

pfc\_type pfc[total\_vp],\*freepf\_head,\*busypf\_head,\*busypf\_tail;

int diseffect,a[total\_instruction];

int page[total\_instruction],offset[total\_instruction];

int initialize (int) ;

int FIFO(int);

int LRU(int);

int LFU(int);

int NUR(int);

int OPT(int);

int main()

{

int s,i,j;

srand(10\*getpid()); /\*由于每次运行时进程号不同，故可用来作为初始化随机数队列的“种子”\*/

s=(float)319\*rand()/32767/32767/2+1;

for(i=0;i<total\_instruction;i+=4) /\* 产生指令队列\*/

{

if(s<0||s>319)

{

printf("When i==%d,Error,s==%d\n",i,s);

exit (0);

}

a[i]=s; /\*任选一指令访问点m\*/

a[i+1]=a[i]+1; /\*顺序执行一条指令\*/

a[i+2]=(float)a[i]\*rand()/32767/32767/2; /\*执行前地址指 令m’\*/

a[i+3]=a[i+2]+1; /\*顺序执行一条指令\*/

s=(float)(318-a[i+2])\*rand()/32767/32767/2+a[i+2]+2;

if((a[i+2]>318)||(s>319))

printf("a[%d+2], a number which is : %d and s=%d\n",i,a[i+2],s);

}

for(i=0;i<total\_instruction;i++) /\*将指令序列变换成页地址流\*/

{

page[i]=a[i]/10;

offset[i]=a[i]%10;

}

for(i=4;i<=32;i++) /\*用户内存工作区从4个页面到32个 页面\*/

{

printf("---%2d page frames---",i);

FIFO(i);

LRU(i);

LFU(i);

NUR(i);

OPT(i);

}

return 0;

}

int initialize(total\_pf)

int total\_pf;

{

int i;

diseffect=0;

for(i=0;i<total\_vp;i++ )

{

pl[i].pn=i;

pl[i].pfn=INVALID; /\*置页面控制中的页号,页面为空\*/

pl[i].counter=0;

pl[i].time=-1; /\*页面控制结构中的访问次数为0， 时间为-1\*/

}

for(i=0;i<total\_pf-1;i++)

{

pfc[i].next=&pfc[i+1];

pfc[i].pfn=i;

} /\*建立pfc’[i-1]和pfc[i]之间的链接\*/

pfc[total\_pf-1].next=NULL;

pfc[total\_pf-1].pfn=total\_pf-1;

freepf\_head=&pfc[0]; /\*空页面队列的头指针为pfc[0]\*/

return 0;

}

int FIFO(total\_pf) /\*先进先出算法\*/

int total\_pf; /\*用户进程的内存页面数\*/

{

int i,j;

pfc\_type \*p;

initialize(total\_pf); /\*初始化相关页面控制用数据 结构\*/

busypf\_head=busypf\_tail=NULL; /\*忙页面队列头，队 列尾链接\*/

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) /\*页面失效\*/

{

diseffect+=1; /\*失效次数\*/

if(freepf\_head==NULL) /\*无空闲页面\*/

{

p=busypf\_head->next;

pl[busypf\_head->pn].pfn=INVALID;

freepf\_head=busypf\_head; /\*释放忙页面队 列的第一个页面\*/

freepf\_head->next=NULL;

busypf\_head=p;

}

p=freepf\_head->next; /\*按FIFO方式调新页 面入内存页面\*/

freepf\_head->next=NULL;

freepf\_head->pn=page[i];

pl[page[i]].pfn=freepf\_head->pfn;

if(busypf\_tail==NULL)

busypf\_head=busypf\_tail=freepf\_head;

else

{

busypf\_tail->next=freepf\_head; /\*free页面减 少一个\*/

busypf\_tail=freepf\_head;

}

freepf\_head=p;

}

}

printf("FIFO:%6.4f",1-(float)diseffect/320);

return 0;

}

int LRU(total\_pf) /\*最近最久未使用算法\*/

int total\_pf;

{

int min,minj,i,j,present\_time;

initialize(total\_pf);

present\_time=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) /\*页面失效\*/

{

diseffect++;

if(freepf\_head==NULL) /\*无空闲页面\*/

{

min=32767;

for(j=0;j<total\_vp;j++)

if(min>pl[j].time&&pl[j].pfn!=INVALID)

{

min=pl[j].time;

minj=j;

}

freepf\_head=&pfc[pl[minj].pfn];/\*腾出一个单元\*/

pl[minj].pfn=INVALID;

pl[minj].time=-1;

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn; /\*有空闲页面，改为 有效\*/

pl[page[i]].time=present\_time;

freepf\_head=freepf\_head->next;

}

else

pl[page[i]].time=present\_time;

present\_time++;

}

printf("LRU:%6.4f ",1-(float)diseffect/320);

return 0;

}

int NUR(total\_pf) /\*最近未使用算法\*/

int total\_pf;

{

int i,j,dp,cont\_flag,old\_dp;

pfc\_type \*t;

initialize(total\_pf);

dp=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) /\*页面失效\*/

{

diseffect++;

if(freepf\_head==NULL)

{

cont\_flag=TRUE;

old\_dp=dp;

while(cont\_flag)

if(pl[dp].counter==0&&pl[dp].pfn!=INVALID)

cont\_flag=FALSE;

else

{

dp++;

if(dp==total\_vp)

dp=0;

if(dp==old\_dp)

for(j=0;j<total\_vp;j++)

pl[j].counter=0;

}

freepf\_head=&pfc[pl[dp].pfn];

pl[dp].pfn=INVALID;

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn;

freepf\_head=freepf\_head->next;

}

else

pl[page[i]].counter=1;

if(i%clear\_period==0)

for(j=0;j<total\_vp;j++)

pl[j].counter=0;

}

printf("NUR:%6.4f",1-(float)diseffect/320);

return 0;

}

int OPT(total\_pf) /\*最佳置换算法\*/

int total\_pf;

{

int i,j,max,maxpage,d,dist[total\_vp];

pfc\_type \*t;

initialize(total\_pf);

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) /\*页面失效\*/

{

diseffect++;

if(freepf\_head==NULL)

{

for(j=0;j<total\_vp;j++) /\*无空闲页面\*/

if(pl[j].pfn!=INVALID)

dist[j]=32767; /\*最大“距离”\*/

else dist[j]=0;

d=1;

for(j=i+1;j<total\_instruction;j++)

{

if(pl[page[j]].pfn!=INVALID)

dist[page[j]]=d;

d++;

}

max=-1;

for(j=0;j<total\_vp;j++)

if(max<dist[j])

{

max=dist[j];

maxpage=j;

}

freepf\_head=&pfc[pl[maxpage].pfn];

freepf\_head->next=NULL;

pl[maxpage].pfn=INVALID;

}

pl[page[i]].pfn=freepf\_head->pfn;

freepf\_head=freepf\_head->next;

}

}

printf("OPT:6.4f\n",1-(float)diseffect/320);

return 0;

}

int LFU(total\_pf)

int total\_pf; /\*最不经常使用置换法\*/

{

int i,j,min,minpage;

pfc\_type \*t;

initialize(total\_pf);

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) /\*页面失效\*/

{

diseffect++;

if(freepf\_head==NULL) /\*无空闲页面\*/

{

min=32767;

for(j=0; j<total\_vp;j++)

{

if(min>pl[j].counter&&pl[j].pfn!=INVALID)

{

min=pl[j].counter;

minpage=j;

}

pl[j].counter=0;

}

freepf\_head=&pfc[pl[minpage].pfn];

pl[minpage].pfn=INVALID;

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn; /\*有空闲页面， 改为有效。\*/

pl[page[i]].counter++;

freepf\_head=freepf\_head->next; /\*减少一个free页面\*/

}

else

pl[page[i]].counter++;

}

printf("LFU:%6.4f",1-(float)diseffect/320);

return 0;

}

**根据参考代码，思考如下问题：**

（1）运行程序，査看实验结果，分析各页面置换算法在内存页面不同的情况下命中率的变化情况。

（2）修改程序，分析各置换算法在用户内存容量变化时命中率为什么达不到1。

（3）体会各页面置换算法的优缺点。

**实验要求：**

（1）将实验过程中涉及的内存管理命令运行结果写入实验报告。

（2）根据程序参考代码画出各页面置换算法的流程图。

（3）将不同的实验结果以贴图的形式写入实验报告，并给出具体分析。