**操作系统实验指导书**

**（2019.09）**

目录

[实验0 UNIX/LINUX及其使用环境（课后自学完成） 3](#_Toc22039217)

[（一）UNIX常用命令简介 3](#_Toc22039218)

[（二）LINUX](#_Toc22039219)

[C语言使用、编译与调试实验 12](#_Toc22039219)

[实验一 进程管理进程通信 16](#_Toc22039220)

[（一）进程的创建实验 16](#_Toc22039221)

[（二）进程的控制实验 21](#_Toc22039223)

[（三）进程互斥实验 24](#_Toc22039224)

[（四）守护进程实验 26](#_Toc22039225)

[（五） 信号机制实验 27](#_Toc22039226)

[（六）进程的管道通信实验 32](#_Toc22039227)

[（七）消息的发送与接收实验 35](#_Toc22039228)

[（八）共享存储区通信 41](#_Toc22039229)

[实验二 银行家算法 46](#_Toc22039230)

[实验三 内存管理实验 48](#_Toc22039231)

[（一）常用页面置换算法模拟实验 48](#_Toc22039232)

[（二）Linux下的内存分配与回收的管理 50](#_Toc22039233)

[实验四 文件管理实验 52](#_Toc22039234)

[实验五 磁盘管理实验 53](#_Toc22039235)

实验0 UNIX/LINUX及其使用环境（课后自学完成）

# （一）UNIX常用命令简介

实验目的

1、了解UNIX的命令及使用格式。

2、熟悉UNIX/LINUX的常用基本命令。

实验内容

1、通过WINDOWS操作系统中的远程登录程序telnet.exe 登录UNIX。

2、熟悉UNIX/LINUX的常用基本命令如ls、who、w、pwd、ps、pstree、top等。

3、通过WINDOWS操作系统中的FTP程序，练习WINDOWS和UNIX之间的文件交换。

实验准备

预习附录一《UNIX/LINUX简介》

实验指导

一、UNIX的登录与退出

1、登录

在DOS环境下用MS提供的telnet程序（也可使用WINDOWS 自带的telnet图形界面程序或多功能的S-Term终端程序），可使PC作为终端（terminal）登录（login）UNIX服务器（UNIX Server）。

（1）执行格式：

telnet hostname(主机名)

或 telnet 主机的IP地址

例： telnet www.yahoo.com

telnet 140.122.77.120

（2）步骤

login： （输入username）

password： （输入密码）

2、退出

在UNIX系统提示符$下，输入logout、exit或shutdown 。

例：$ logout

二、UNIX命令格式

命令 [选项] [处理对象]

例：ls -la mydir

注意：（1）命令一般是小写字串。注意大小写有别

（2）选项通常以减号（-）再加上一个或数个字符表示，用来选择一个命令的不同操作

（3）同一行可有数个命令，命令间应以分号隔开

（4）命令后加上&可使该命令后台（background）执行

三、常用命令

1、目录操作

和DOS相似，UNIX采用树型目录管理结构，由根目录（/）开始一层层将子目录建下去，各子目录以 / 隔开。用户login后，工作目录的位置称为 home directory，由系统管理员设定。‘~’符号代表自己的home directory，例如 ~/myfile 是指自己home目录下myfile这个文件。

UNIX的通配符有三种：’\*’ 和 ’?’ 用法与DOS相同， ‘-‘ 代表区间内的任一字符，如test[0-5]即代表test0，test1，……，test5的集合。

（1）显示目录文件 ls

执行格式： ls [-atFlgR] [name] (name可为文件或目录名称)

例： ls 显示出当前目录下的文件

ls -a 显示出包含隐藏文件的所有文件

ls -t 按照文件最后修改时间显示文件

ls -F 显示出当前目录下的文件及其类型

ls -l 显示目录下所有文件的许可权、拥有者、文件大小、修改时间及名称

ls -lg 同上

ls -R 显示出该目录及其子目录下的文件

注:ls与其它命令搭配使用可以生出很多技巧(最简单的如"ls -l | more")，更多用法请输入ls --help查看，其它命令的更多用法请输入 命令名 --help 查看.

（2）建新目录 mkdir

执行格式： mkdir directory-name

例： mkdir dir1　　　(新建一名为dir1的目录)

（3）删除目录　　rmdir

执行格式： rmdir directory-name 或 rm directory-name

例：rmdir dir1 删除目录dir1，但它必须是空目录，否则无法删除

rm -r dir1 删除目录dir1及其下所有文件及子目录

rm -rf dir1 不管是否空目录，统统删除，而且不给出提示,使用时要小心

（4） 改变工作目录位置 cd

执行格式： cd [name]

例： cd 改变目录位置至用户login时的working directory

cd dir1 改变目录位置，至dir1目录

cd ~user 改变目录位置，至用户的working directory

cd .. 改变目录位置，至当前目录的上层目录

cd ../user 改变目录位置，至上一级目录下的user目录

cd /dir-name1/dir-name2 改变目录位置，至绝对路径（Full path）

cd - 回到进入当前目录前的上一个目录

（5）显示当前所在目录pwd

执行格式： pwd

（6）查看目录大小du

执行格式： du [-s] directory

例： du dir1 显示目录dir1及其子目录容量（以kb为单位）

du -s dir1 显示目录dir1的总容量

（7）显示环境变量

echo $HOME 显示家目录

echo $PATH 显示可执行文件搜索路径

env 显示所有环境变量(可能很多,最好用"env | more","env | grep PATH"等)

（8）修改环境变量，在bash下用export,如：

export PATH=$PATH:/usr/local/bin

想知道export的具体用法，可以用shell的help命令：help export

2、文件操作

（1）查看文件(可以是二进制的)内容 cat

执行格式：cat filename或more filename 或cat filename|more

例： cat file1 以连续显示方式，查看文件file1的内容

more file1

或 cat file1|more 以分页方式查看文件的内容

（2）删除文件 rm

执行格式： rm filename

例： rm file?

rm f\*

（3）复制文件 cp

执行格式： cp [-r] source destination

例： cp file1 file2 将file1复制成file2

cp file1 dir1 将file1复制到目录dir1

cp /tmp/file1 将file1复制到当前目录

cp /tmp/file1 file2 将file1 复制到当前目录名为file2

cp –r dir1 dir2 (recursive copy)复制整个目录。

（4）移动或更改文件、目录名称mv

执行格式： mv source destination

例： mv file1 file2 将文件file1，更名为file2

mv file1 dir1 将文件file1，移到目录dir1下

mv dir1 dir2

（5）比较文件(可以是二进制的)或目录的内容 diff

执行格式： diff [-r] name1 name2 (name1、name2同为文件或目录)

例： diff file1 file2 比较file1与file2的不同处

diff -r dir1 dir2 比较dir1与dir2的不同处

（6）文件中字符串的查找 grep

执行格式： grep string file

例： grep abc file1 查找并列出串abc所在的整行文字

（7）文件或命令的路径寻找

执行格式一：whereis command 显示命令的路径

执行格式二：which command 显示路径及使用者所定义的别名

执行格式三：whatis command 显示命令的功能摘要

执行格式四：find search -path -name filename -print

搜寻指定路径下某文件的路径

执行格式五：locate filename

根据系统预先生成的文件/目录数据库(/var/lib/slocate/slocate.db)查找匹配的文件/目录,查找速度很快,如果有刚进行的文件改变而系统未到执行定时更新数据库的时间,可以打入updatedb命令手动更新.

（8）建立文件或目录的链接 ln

例: ln source target1 建立source文件（已存在）的硬链接,命名为target1

ln -s source target2 建立source文件的符号链接,命名为target2

3、系统询问与权限口令

（1）查看系统中的使用者

执行格式： who

（2）查看username

执行格式： who am I 查看自己的username

（3）改变自己的username的帐号与口令 su

执行格式： su username

例： su username 输入帐号

password 输入密码

（4）文件属性的设置 chmod

改变文件或目录的读、写、执行的允许权

执行格式： chmod [-R] mode name

其中：[-R]为递归处理,将指定目录下所有文件及子目录一并处理

mode为3-8位数字，是文件/目录读、写、执行允许权的缩写(r:read,数字代号为"4" w:write,数字代号为"2" x:execute,数字代号为"1")

mode： rwx rwx rwx

user group other

缩写： (u) (g) (o)

例：chmod 755 dir1 将目录dir1设定成任何人皆有读取及执行的权利，但只有拥有者可作写修改。其中7=4+2+1,5=4+1

chmod 700 file1 将file1设为拥有者可以读、写和执行

chmod o+x file2 将file2，增加拥有者可执行的权利

chmod g+x file3 将file3，增加组使用者可执行的权利

chmod o-r file4 将file4，除去其它使用者可读取的权利

（5）改变文件或目录所有权 chown

执行格式： chown [-R] username name

例： chown user file1 将文件file1改为user所有

chown .fox file1 将文件file1改为fox组所有

chown user.fox file1 将文件file1改为fox组的user所有

chown -R user dir1 将目录dir1及其下所有文件和子目录，改为user 所有

（6）检查用户所在组名称 groups

执行格式： groups

（7）改变文件或目录的组拥有权 chgrp

执行格式： chgrp [-R] groupname name

例： chgrp vlsi file1 将文件file1改为vlsi组所有

chgrp -R image dir1 将目录dir1及其下所有文件和子目录，改为image群组

（8）改变文件或目录的最后修改时间 touch

执行格式： touch name

4、进程操作

（1）查看系统目前的进程 ps

执行格式： ps [-aux]

例： ps 或ps -x 查看系统中属于自己的process

ps -au 查看系统中所有使用者的process

ps -aux 查看系统中包含系统内部及所有使用者的process

ps -aux|grep apache 找出系统中运行的所有名称中带有"apache"串的process

（2）查看正在background中执行的process

执行格式： jobs

（3）结束或终止进程 kill

执行格式： kill [-9] PID （PID为利用ps命令所查出的process ID）

例： kill 456

或 kill -9 456 终止process ID 为456的process

（4）后台（background）执行process command的命令

执行格式： command & （在命令后加上 &）

例： gcc file1 & 在后台编译file1.c

注意：按下^Z，暂停正在执行的process。键入”bg”，将所暂停的process置入background中继续执行。

例： gcc file1 &

^Z

stopped

bg

（5）结束或终止在background中的进程 kill

执行格式： kill %n

例： kill %1 终止在background中的第一个job

kill %2 终止在background中的第二个job

（6）显示系统中程序的执行状态

例: top -q 不断地更新、显示系统程序的执行状态

第一行显示的项目依次为当前时间、系统启动时间、当前系统登录用户数目、平均负载。

第二行为进程情况，依次为进程总数、休眠进程数、运行进程数、僵死进程数、终止进程数。

第三行为CPU状态，依次为用户占用、系统占用、优先进程占用、闲置进程占用。

第四行为内存状态，依次为平均可用内存、已用内存、空闲内存、共享内存、缓存使用内存。

第五行为交换状态，依次为平均可用交换容量、已用容量、闲置容量、高速缓存容量。

PID 每个进程的ID。

PPID 每个进程的父进程ID。

UID 每个进程所有者的UID 。

USER 每个进程所有者的用户名。

PRI 每个进程的优先级别。

NI 该进程的优先级值。

SIZE 该进程的代码大小加上数据大小再加上堆栈空间大小的总数。单位是KB。

TSIZE 该进程的代码大小。对于内核进程这是一个很奇怪的值。

DSIZE 数据和堆栈的大小。

TRS 文本驻留大小。

D 被标记为“不干净”的页项目。

LIB 使用的库页的大小。对于ELF进程没有作用。

RSS 该进程占用的物理内存的总数量，单位是KB。

SHARE 该进程使用共享内存的数量。

STAT 该进程的状态。其中S代表休眠状态；D代表不可中断的休眠状态；R代表运行状态；Z代表僵死状态；T代表停止或跟踪状态。

TIME 该进程自启动以来所占用的总CPU时间。如果进入的是累计模式，那么该时间还包括这个进程子进程所占用的时间。且标题会变成CTIME。

%CPU 该进程自最近一次刷新以来所占用的CPU时间和总时间的百分比。

%MEM 该进程占用的物理内存占总内存的百分比。

COMMAND 该进程的命令名称，如果一行显示不下，则会进行截取。内存中的进程会有一个完整的命令行

按"ctrl+c"停止查看

（7）以树状图显示执行的程序 pstree

例: pstree -h 列出进程树并高亮标出当前执行的程序

（8）监视虚拟内存 vmstat

vmstat对系统的虚拟内存、进程、CPU活动进行监视，同时它也对磁盘和forks和vforks操作的个数进行汇总。

不足是：vmstat不能对某个进程进行深入分析，它仅是一对系统的整体情况进行分析。

例如：[angel@home /angel]# vmstat

procs memory swap io system cpu

r b w swpd free buff cache si so bi bo in cs us sy id

0 0 0 7180 1852 56092 48400 0 0 6 5 24 8 0 0 18

其中：

Procs

r: 等待运行的进程数 b: 处在非中断睡眠状态的进程数 w: 被交换出去的可运行的进程数。

Memory

swpd: 虚拟内存使用情况，单位：KB free: 空闲的内存，单位KB

buff: 被用来做为缓存的内存数，单位：KB

Swap

si: 从磁盘交换到内存的交换页数量，单位：KB/秒 so: 从内存交换到磁盘的交换页数量，单位：KB/秒

IO

bi: 发送到块设备的块数，单位：块/秒 bo: 从块设备接收到的块数，单位：块/秒

System

in: 每秒的中断数，包括时钟中断 cs: 每秒的环境（上下文）切换次数

CPU 按 CPU 的总使用百分比来显示

us: CPU 使用时间 sy: CPU 系统使用时间 id: 闲置时间

（9）分析共享内存、信号量和消息队列 ipcs(相关命令ipcrm：用于给有权限的用户清除这些量，注意不要乱清除，除非该量确实失效了)

例如：[angel@home /angel]# ipcs

------ Shared Memory Segments --------

key shmid owner perms bytes nattch status

0x00280267 0 root 644 1048576 1

0x61715f01 1 root 666 32000 33

0x00000000 2 nobody 600 92164 11 dest

------ Semaphore Arrays --------

key semid owner perms nsems status

0x00280269 0 root 666 14

0x61715f02 257 root 777 1

------ Message Queues --------

key msqid owner perms used-bytes messages

（10）监视用户空间程序发出的全部系统调用 strace

strace 还能显示调用的参数，以及用符号方式表示的返回值。

strace 从内核中接收信息，所以一个程序无论是否按调试方式编译(gcc -g)或是否被去掉了调试信息，都可以被跟踪。

执行格式： strace [-tTeo] executable-program-name

-t ： 用来显示调用发生的时间

-T ： 显示调用花费的时间

-e ： 限定被跟踪的调用类型

-o ： 将输出重定向到一个文件中

类似命令：ltrace [-fiS] executable-program-name

5、通信类

（1）本地工作站与UNIX服务器间的文件传输 ftp

执行格式： ftp 主机名

或 ftp 主机的IP地址

后续执行步骤：

|  |  |
| --- | --- |
| name： | 输入帐号 |
| password： | 输入密码 |
| ftp>help | 显示ftp可使用的所有命令 |
| ftp>lcd dir1 | 改变本地机当前目录为dir1 |
| ftp>get file1 | 将UNIX服务器文件file1拷到本地机 |
| ftp>put file2 | 将本地文件file2，拷到UNIX服务器 |
| ftp>!ls | 显示本地机当前目录下所有文件 |
| ftp>!pwd | 显示本地机当前所在目录下所有文件 |
| ftp>ls | 显示UNIX服务器当前目录下所有文件 |
| ftp>dir | 显示服务器当前目录下所有文件（略同于UNIX的 ls -l指令） |
| ftp>pwd | 显示UNIX服务器当前所有目录位置 |
| ftp>cd dir1 | 更改UNIX服务器的目录至dir1下 |
| ftp>mget \*.c | 将服务器中 .c 文件拷到本地机中 |
| ftp>mput \*.txt | 将所有 .txt 文件拷贝到服务器 |
| ftp>quit | 结束ftp工作 |
| ftp>bye | 结束ftp工作 |

（2）检查与UNIX服务器连接是否正常 ping

执行格式： ping hostname

或 ping IP-Address

例： ping 127.1.1.1

（3）将文件当做E-mail的内容送出 mail

执行格式：mail -s “Subjict-string” username@address<filename

例： mail -s “program” user <file.c

功能：将file.c当做mail的内容，送至user，subject name为program

（4）传送E-mail给本地UNIX服务器上的用户 mail

执行格式： mail username

（5）读取信件 mail

执行格式： mail

（6）列出套接字使用情况 socklist

（7）查看网络连接 netstat

6、I/O命令

（1）管道（pipe-line）的使用

执行格式： command1|command2

功能：将command1的执行结果送到command2 作为输入

例： ls -R1|more 以分页方式列出当前目录文件及子目录名称

cat file1|more 以分页方式，列出file1的内容

（2）标准输入控制

执行格式： command-line<file 将file作为command-line的输入

例： mail -s “mail test” user@iis.sinica.edu.tw<file1

功能：将文件file1当作信件的内容，subject 名称为mail test　送给收信人

（3）标准输出控制

执行格式一： command>filename

功能：将command的执行结果送至指定的filename中

例： ls -l >list 将执行”ls -l” 的结果写入文件list中

执行格式**二**： command>!filename

功能：同上，若filename文件已存在，则强迫重写

执行格式三： command>&filename

功能：将command执行所产生的任何信息写入filename

执行格式四：command>>filename

功能：将command 的执行结果，附加（append）到filename

执行格式五：command>>&filename

????????????功能：将command执行所产生的任何信息附加于filename中

7、其它常用命令

（1）命令在线帮助 man

执行格式： man command

例： man ls 查询ls这个指令的用法

（2）设定命令记录表长度 history

执行格式一： set history=n

例： set history=40

功能：设定命令记录表长度为40（可记载执行过的前面40个命令）

执行格式二： history 查看命令记录表的内容

（3）显示说明 info

执行格式: info command-name

例: info gcc

功能: 查看gcc的说明,按上下箭头选定菜单,回车进入,"u"键返回上级菜单.

info不加参数则进入最上一级菜单.

四、用cat 命令查看 /proc 动态文件系统目录下的文件,辨识其中的系统信息.

例如: cat interrupts 列出当前中断占用情况

cat ioports 列出设备的硬件IO占用情况

cat pci 列出pci设备的情况

# （二）LINUX 下C语言使用、编译与调试实验

实验目的

1、复习C语言程序基本知识

2、练习并掌握UNIX提供的vi编辑器来编译C程序

3、学会利用gcc、gdb编译、调试C程序

实验内容

1、用vi编写一个简单的、显示"Hello,World!"的C程序，用gcc编译并观察编译后的结果

2、利用gdb调试该程序

3、运行生成的可执行文件。

实验指导

一、C语言使用简介

LINUX中包含了很多软件开发工具。它们中的很多是用于C和C++应用程序开发的。

C是一种能在UNIX的早期就被广泛使用的通用编程语言。它最早是由Bell实验室的Dennis Ritchie为了UNIX的辅助开发而写的，从此C就成为世界上使用最广泛的计算机语言。

C能在编程领域里得到如此广泛支持的原因有：

（1）它是一种非常通用的语言，并且它的语法和函数库在不同的平台上都是统一的，对开发者非常有吸引力；

（2）用C写的程序执行速度很快；

（3）C是所有版本UNIX上的系统语言；

二、文件编辑器vi

vi是在UNIX 上被广泛使用的中英文编辑软件。vi是visual editor的缩写，是UNIX提供给用户的一个窗口化编辑环境。

进入vi，直接执行vi编辑程序即可。

例：$vi test.c

显示器出现vi的编辑窗口，同时vi会将文件复制一份至缓冲区（buffer）。vi先对缓冲区的文件进行编辑，保留在磁盘中的文件则不变。编辑完成后，使用者可决定是否要取代原来旧有的文件。

1、vi的工作模式

vi提供二种工作模式：输入模式（insert mode）和命令模式（command mode）。使用者进入vi后，即处在命令模式下，此刻键入的任何字符皆被视为命令，可进行删除、修改、存盘等操作。要输入信息，应转换到输入模式。

（1）命令模式

在输入模式下，按ESC可切换到命令模式。命令模式下，可选用下列指令离开vi：

|  |  |
| --- | --- |
| ：q! | 离开vi，并放弃刚在缓冲区内编辑的内容 |
| ：wq | 将缓冲区内的资料写入磁盘中，并离开vi |
| ：ZZ | 同wq |
| ：x | 同wq |
| ：w | 将缓冲区内的资料写入磁盘中，但并不离开vi |
| ：q | 离开vi，若文件被修改过，则要被要求确认是否放弃修改的内容，此指令可与：w配合使用 |

（2）命令模式下光标的移动

|  |  |
| --- | --- |
| H | 左移一个字符 |
| J | 下移一个字符 |
| K | 上移一个字符 |
| L | 右移一个字符 |
| 0 | 移至该行的首 |
| $ | 移至该行的末 |
| ^ | 移至该行的第一个字符处 |
| H | 移至窗口的第一列 |
| M | 移至窗口中间那一列 |
| L | 移至窗口的最后一列 |
| G | 移至该文件的最后一列 |
| W, W | 下一个单词 (W 忽略标点) |
| B, B | 上一个单词 (B 忽略标点) |
| + | 移至下一列的第一个字符处 |
| - | 移至上一列的第一个字符处 |
| ( | 移至该句首 |
| ) | 移至该句末 |
| { | 移至该段首 |
| } | 移至该段末 |
| NG | 移至该文件的第n列 |
| N+ | 移至光标所在位置之后第n列 |
| n- | 移至光标所在位置之前第n列 |

（3）输入模式

输入以下命令即可进入vi输入模式：

|  |  |
| --- | --- |
| a(append) | 在光标之后加入资料 |
| A | 在该行之末加入资料 |
| i(insert) | 在光标之前加入资料 |
| I | 在该行之首加入资料 |
| o(open) | 新增一行于该行之下，供输入资料用 |
| O | 新增一行于该行之上，供输入资料用 |
| Dd | 删除当前光标所在行 |
| X | 删除当前光标字符 |
| X | 删除当前光标之前字符 |
| U | 撤消 |
| · | 重做 |
| F | 查找 |
| s | 替换,例如:将文件中的所有"FOX"换成"duck",用":%s/FOX/duck/g" |
| ESC | 离开输入模式 |

更多用法见 info vi

三、GNU C编译器

LINUX上可用的C编译器是GNU C编译器，它建立在自由软件基金会编程许可证的基础上，因此可以自由发布。

LINUX 上的GNU C编译器（GCC）是一个全功能的ANCI C兼容编译器，而一般UNIX（如SCO UNIX）用的编译器是CC。下面介绍GCC和一些GCC编译器最常用的选项。

1、使用GCC

通常后跟一些选项和文件名来使用GCC编译器。GCC命令的基本用法如下：

gcc [options] [filenames]

命令行选项指定的编译过程中的具体操作

2、GCC常用选项

GCC有超过100个的编译选项可用，这些选项中的许多可能永远都不会用到，但一些主要的选项将会频繁使用。很多的GCC选项包括一个以上的字符，因此必须为每个选项指定各自的连字符，并且就像大多数LINUX 命令一样不能在一个单独的连字符后跟一组选项。例如，下面的命令是不同的：

gcc -p-g test.c

gcc -pg test.c

第一条命令告诉GCC编译test.c时为prof命令建立剖析（profile）信息并且把调试信息加入到可执行文件里。第二条命令告诉GCC只为gprof命令建立剖析信息。

当不用任何选项编译一个程序时，GCC将建立（假定编译成功）一个名为a.out的可执行文件。例如，

gcc test.c

编译成功后，当前目录下就产生了一个a.out文件。

也可用-o选项来为即将产生的可执行文件指定一个文件名来代替a.out。例如：

gcc –o count count.c

此时得到的可执行文件就不再是a.out，而是count。

GCC也可以指定编译器处理步骤多少。-c选项告诉GCC仅把源代码编译为目标代码而跳过汇编和连接步骤。这个选项使用得非常频繁因为它编译多个C程序时速度更快且更易于管理。默认时GCC建立的目标代码文件有一个.o的扩展名。

3、执行文件

格式： ./可执行文件名

例：./a.out

./count

三、gdb调试工具

LINUX包含了一个叫gdb的GNU调试程序。gdb是一个用来调试C和C++程序的强有力调试器。它使你能在程序运行时观察程序的内部结构和内存的使用情况。它具有以下一些功能：

·监视程序中变量的值；

·设置断点以使程序在指定的代码行上停止执行；

·一行行的执行代码。

以下是利用gdb进行调试的步骤：

1、调试编译代码

为了使gdb正常工作，必须使你的程序在编译时包含调试信息。调试信息里包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb利用这些信息使源代码和机器码相关联。

在编译时用 –g 选项打开调试选项。

2、gdb基本命令

|  |  |
| --- | --- |
| 命 令 | 描 述 |
| file | 装入欲调试的可执行文件 |
| kill | 终止正在调试的程序 |
| list | 列出产生执行文件的源代码部分 |
| next | 执行一行源代码但不进入函数内部 |
| step | 执行一行源代码并进入函数内部 |
| run | 执行当前被调试的程序 |
| quit | 终止gdb |
| watch | 监视一个变量的值而不管它何时被改变 |
| break | 在代码里设置断点，使程序执行到这里时被挂起 |
| make | 不退出gdb就可以重新产生可执行文件 |
| shell | 不离开gdb就执行UNIX shell 命令 |

3、应用举例

(1)设有一源程序greet.c

(2)编译，gcc -ggdb –o greet greet.c，出错

(3)gdb greet ，出现提示符(gdb)，此时可在提示符下输入gdb的命令了，如：

(gdb)run

(gdb)list

(4)退出调试状态，返回系统提示符下， (gdb)quit

四、参考程序

main( )

{

printf("Hello,world!\n");

}

实验一 进程管理进程通信

# （一）进程的创建实验

实验目的

1、掌握进程的概念，明确进程的含义

2、认识并了解并发执行的实质

实验内容

1、编写一段程序，使用系统调用fork( )创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示'a'，子进程分别显示字符'b'和字符'c'。试观察记录屏幕上的显示结果，并分析原因。

2、修改上述程序，每一个进程循环显示一句话。子进程显示'daughter …'及'son ……'，父进程显示 'parent ……'，观察结果，分析原因。

实验准备

（1）阅读LINUX的fork.c源码文件（见附录二），分析进程的创建过程。

（2）阅读LINUX的sched.c源码文件(见附录三)，加深对进程管理概念的认识。

实验指导

一、进程

UNIX中，进程既是一个独立拥有资源的基本单位，又是一个独立调度的基本单位。一个进程实体由若干个区（段）组成，包括程序区、数据区、栈区、共享存储区等。每个区又分为若干页，每个进程配置有唯一的进程控制块PCB，用于控制和管理进程。

PCB的数据结构如下：

1、进程表项（Process Table Entry）。包括一些最常用的核心数据：

进程标识符PID、用户标识符UID、进程状态、事件描述符、进程和U区在内存或外存的地址、软中断信号、计时域、进程的大小、偏置值nice、指向就绪队列中下一个PCB的指针P\_Link、指向U区进程正文、数据及栈在内存区域的指针。

2、U区（U Area）。用于存放进程表项的一些扩充信息。

每一个进程都有一个私用的U区，其中含有：进程表项指针、真正用户标识符u-ruid(read user ID)、有效用户标识符u-euid(effective user ID)、用户文件描述符表、计时器、内部I/O参数、限制字段、差错字段、返回值、信号处理数组。

由于UNIX系统采用段页式存储管理，为了把段的起始虚地址变换为段在系统中的物理地址，便于实现区的共享，所以还有：

3、系统区表项。以存放各个段在物理存储器中的位置等信息。

系统把一个进程的虚地址空间划分为若干个连续的逻辑区，有正文区、数据区、栈区等。这些区是可被共享和保护的独立实体，多个进程可共享一个区。为了对区进行管理，核心中设置一个系统区表，各表项中记录了以下有关描述活动区的信息：

区的类型和大小、区的状态、区在物理存储器中的位置、引用计数、指向文件索引结点的指针。

4、进程区表

系统为每个进程配置了一张进程区表。表中，每一项记录一个区的起始虚地址及指向系统区表中对应的区表项。核心通过查找进程区表和系统区表，便可将区的逻辑地址变换为物理地址。

二、进程映像

UNIX系统中，进程是进程映像的执行过程，也就是正在执行的进程实体。它由三部分组成：

1、用户级上、下文。主要成分是用户程序；

2、寄存器上、下文。由CPU中的一些寄存器的内容组成，如PC，PSW，SP及通用寄存器等；

3、系统级上、下文。包括OS为管理进程所用的信息，有静态和动态之分。

三、所涉及的系统调用

**1、fork( )**

创建一个新进程。

系统调用格式：

pid=fork( )

参数定义：

int fork( )

fork( )返回值意义如下：

0：在子进程中，pid变量保存的fork( )返回值为0，表示当前进程是子进程。

>0：在父进程中，pid变量保存的fork( )返回值为子进程的id值（进程唯一标识符）。

-1：创建失败。

如果fork( )调用成功，它向父进程返回子进程的PID，并向子进程返回0，即fork( )被调用了一次，但返回了两次。此时OS在内存中建立一个新进程，所建的新进程是调用fork( )父进程（parent process）的副本，称为子进程（child process）。子进程继承了父进程的许多特性，并具有与父进程完全相同的用户级上下文。父进程与子进程并发执行。

核心为fork( )完成以下操作：

（1）为新进程分配一进程表项和进程标识符

进入fork( )后，核心检查系统是否有足够的资源来建立一个新进程。若资源不足，则fork( )系统调用失败；否则，核心为新进程分配一进程表项和唯一的进程标识符。

（2）检查同时运行的进程数目

超过预先规定的最大数目时，fork( )系统调用失败。

（3）拷贝进程表项中的数据

将父进程的当前目录和所有已打开的数据拷贝到子进程表项中，并置进程的状态为“创建”状态。

（4）子进程继承父进程的所有文件

对父进程当前目录和所有已打开的文件表项中的引用计数加1。

（5）为子进程创建进程上、下文

进程创建结束，设子进程状态为“内存中就绪”并返回子进程的标识符。

（6）子进程执行

虽然父进程与子进程程序完全相同，但每个进程都有自己的程序计数器**PC(注意子进程的PC开始位置)**，然后根据pid变量保存的fork( )返回值的不同，执行了不同的分支语句。

例：

|  |
| --- |
| …..  pid=fork( );  if (! pid)  printf("I'm the child process!\n");  else if (pid>0)  printf("I'm the parent process! \n");  else  printf("Fork fail!\n");  …… |

PC

fork( )调用前

fork( )调用后

PC

|  |  |  |
| --- | --- | --- |
| ….. PC pid=fork( );  if (! pid)  printf("I'm the child process!\n");  else if (pid>0)  printf("I'm the parent process!\n ");  else  printf("Fork fail!\n");  …… |  | …..  pid=fork( );  if (! pid)  printf("I'm the child process!\n");  else if (pid>0)  printf("I'm the parent process!\n ");  else  printf("Fork fail!\n");  …… |

四、参考程序

1、

#include <stdio.h>

main( )

{

int p1,p2;

while((p1=fork( ))= = -1); /\*创建子进程p1\*/

if (p1= =0) putchar('b');

else

{

while((p2=fork( ))= = -1); /\*创建子进程p2\*/

if(p2= =0) putchar('c');

else putchar('a');

}

}

2、

#include <stdio.h> //pid\_t p1,p2; p1=fork()

main( )

{

int p1,p2,i;

while((p1=fork( ))= = -1); /\*创建子进程p1\*/

if (p1= =0)

for(i=0;i<10;i++)

printf("daughter %d\n",i);

else

{

while((p2=fork( ))= = -1); /\*创建子进程p2\*/

if(p2= =0)

for(i=0;i<10;i++)

printf("son %d\n",i);

else

for(i=0;i<10;i++)

printf("parent %d\n",i);

}

}

五、运行结果

1、bca，bac, abc ,……都有可能。

2、parent…

son…

daughter..

daughter..

或 parent…

son…

parent…

daughter…等

六、分析原因

除strace 外，也可用ltrace -f -i -S ./executable-file-name查看以上程序执行过程。

1、从进程并发执行来看，各种情况都有可能。上面的三个进程没有同步措施，所以父进程与子进程的输出内容会叠加在一起。输出次序带有随机性。

2、由于函数printf( )在输出字符串时不会被中断，因此，字符串内部字符顺序输出不变。但由于进程并发执行的调度顺序和父子进程抢占处理机问题，输出字符串的顺序和先后随着执行的不同而发生变化。这与打印单字符的结果相同。

补充：进程树

在UNIX系统中，只有0进程是在系统引导时被创建的，在系统初启时由0进程创建1进程，以后0进程变成对换进程，1进程成为系统中的始祖进程。UNIX利用fork( )为每个终端创建一个子进程为用户服务，如等待用户登录、执行SHELL命令解释程序等，每个终端进程又可利用fork( )来创建其子进程，从而形成一棵进程树。可以说，系统中除0进程外的所有进程都是用fork( )创建的。

七、思考题

（1）系统是怎样创建进程的？

系统调用格式为pid=fork( )，参数定义为int fork( )，fork( )返回值意义如下：

0：在子进程中，pid变量保存的fork( )返回值为0，表示当前进程是子进程。

>0：在父进程中，pid变量保存的fork( )返回值为子进程的id值（进程唯一标识符）。

-1：创建失败。

如果fork( )调用成功，它向父进程返回子进程的PID，并向子进程返回0，即fork( )被调用了一次，但返回了两次。此时OS在内存中建立一个新进程，所建的新进程是调用fork( )父进程（parent process）的副本，称为子进程（child process）。子进程继承了父进程的许多特性，并具有与父进程完全相同的用户级上下文。父进程与子进程并发执行

1,申请空白PCB（进程控制块）；  
2,为新进程分派资源；  
3,初始化PCB；  
4,将新进程插入就绪队列；

（2）当首次调用新创建进程时，其入口在哪里？

fork系统调用创建的子进程继承了原进程的context，也就是说fork调用成功后，子进程与父进程并发执行相同的代码。但由于子进程也继承了父进程的程序指针，所以子进程是从fork（）后的语句开始执行(也就是新进程调用的入口)。另外fork在子进程和父进程中的返回值是不同的。在父进程中返回子进程的PID，而在子进程中返回0。所以可以在程序中检查PID的值，使父进程和子进程执行不同的分支

# （二）进程的控制实验

实验目的

1、掌握进程另外的创建方法

2、熟悉进程的睡眠、同步、撤消等进程控制方法

实验内容

1、用fork( )创建一个进程，再调用exec( )用新的程序替换该子进程的内容

2、利用wait( )来控制进程执行顺序

实验指导

一、所涉及的系统调用

在UNIX/LINUX中fork( )是一个非常有用的系统调用，但在UNIX/LINUX中建立进程除了fork( )之外，也可用与fork( ) 配合使用的exec( )。

**1、exec( )系列**

系统调用exec( )系列，也可用于新程序的运行。fork( )只是将父进程的用户级上下文拷贝到新进程中，而exec( )系列可以将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上，以更改新进程的用户级上下文。exec( )系列中的系统调用都完成相同的功能，它们把一个新程序装入内存，来改变调用进程的执行代码，从而形成新进程。如果exec( )调用成功，调用进程将被**覆盖，**然后从新程序的入口开始执行，这样就产生了一个新进程，新进程的进程标识符id 与调用进程相同。

exec( )没有建立一个与调用进程并发的子进程，而是用新进程取代了原来进程。所以exec( )调用成功后，没有任何数据返回，这与fork( )不同。exec( )系列系统调用在UNIX系统库unistd.h中，共有execl、execlp、execle、execv、execvp五个，其基本功能相同，只是以不同的方式来给出参数。

一种是直接给出参数的指针，如：

int execl(path,arg0[,arg1,...argn],0);

char \*path,\*arg0,\*arg1,...,\*argn;

另一种是给出指向参数表的指针，如：

int execv(path,argv);

char \*path,\*argv[ ];

具体使用可参考有关书。

**2、exec( )和fork( )联合使用**

系统调用exec和fork( )联合使用能为程序开发提供有力支持。用fork( )建立子进程，然后在子进程中使用exec( )，这样就实现了父进程与一个与它完全不同子进程的并发执行。

一般，wait、exec联合使用的模型为：

int status;

............

if (fork( )= =0)

{

...........;

execl(...);

...........;

}

wait(&status);

**3、wait（ ）**

等待子进程运行结束。如果子进程没有完成，父进程一直等待。wait( )将调用进程挂起，直至其子进程因暂停或终止而发来软中断信号为止。如果在wait( )前已有子进程暂停或终止，则调用进程做适当处理后便返回。

系统调用格式：

int wait(status)

int \*status;

其中，status是用户空间的地址。它的低8位反应子进程状态，为0表示子进程正常结束，非0则表示出现了各种各样的问题；高8位则带回了exit( )的返回值。exit( )返回值由系统给出。

核心对wait( )作以下处理：

（1）首先查找调用进程是否有子进程，若无，则返回出错码；

（2）若找到一处于“僵死状态”的子进程，则将子进程的执行时间加到父进程的执行时间上，并释放子进程的进程表项；

（3）若未找到处于“僵死状态”的子进程，则调用进程便在可被中断的优先级上睡眠，等待其子进程发来软中断信号时被唤醒。

**4、exit（ ）**

终止进程的执行。

系统调用格式：

　　 　void exit(status)

　　 int status;

其中，status是返回给父进程的一个整数，以备查考。

为了及时回收进程所占用的资源并减少父进程的干预，UNIX/LINUX利用exit( )来实现进程的自我终止，通常父进程在创建子进程时，应在进程的末尾安排一条exit( )，使子进程自我终止。exit(0)表示进程正常终止，exit(1)表示进程运行有错，异常终止。

如果调用进程在执行exit( )时，其父进程正在等待它的终止，则父进程可立即得到其返回的整数。核心须为exit( )完成以下操作：

（1）关闭软中断

（2）回收资源

（3）写记帐信息

（4）置进程为“僵死状态”

二、参考程序

#include<stdio.h>

#include<unistd.h>

main( )

{

int pid;

pid=fork( ); /\*创建子进程\*/

switch(pid)

{

case -1: /\*创建失败\*/

printf("fork fail!\n");

exit(1);

case 0: /\*子进程\*/

execl("/bin/ls","ls","-1","-color",NULL);

printf("exec fail!\n");

exit(1);

default: /\*父进程\*/

wait(NULL); /\*同步\*/

printf("ls completed !\n");

exit(0);

}

}

三、运行结果

执行命令ls -l -color ，（按倒序）列出当前目录下所有文件和子目录；

ls completed!

四、分析原因

程序在调用fork( )建立一个子进程后，马上调用wait( )，使父进程在子进程结束之前，一直处于睡眠状态。子进程用exec( )装入命令ls ，exec( )后，子进程的代码被ls的代码取代，这时子进程的PC指向ls的第1条语句，开始执行ls的命令代码。

注意在这里wait( )给我们提供了一种实现进程同步的简单方法。

五、思考

（1）编写一hello程序，实现输出“hello,姓名”。将hello替代上述的ls命令。

（2）什么是进程同步？wait( )是如何实现进程同步的？

# （三）进程互斥实验

实验目的

1、进一步认识并发执行的实质

2、分析进程竞争资源的现象，学习解决进程互斥的方法

实验内容

1、修改实验（一）中的程序2，用lockf( )来给每一个进程加锁，以实现进程之间的互斥

2、观察并分析出现的现象

实验指导

一、所涉及的系统调用

**lockf(files,function,size)**

用作锁定文件的某些段或者整个文件。

本函数的头文件为

#include "unistd.h"

参数定义：

int lockf(files,function,size)

int files,function;

long size;

其中：files是文件描述符；function是锁定和解锁：1表示锁定，0表示解锁。size是锁定或解锁的字节数，为0，表示从文件的当前位置到文件尾。

二、参考程序

#include <stdio.h>

#include <unistd.h>

main(　)

{

int p1,p2,i;

while((p1=fork( ))= = -1); /\*创建子进程p1\*/

if (p1= =0)

{

lockf(1,1,0); /\*加锁，这里第一个参数为stdout（标准输出设备的描述符）\*/

for(i=0;i<10;i++)

printf("daughter %d\n",i);

lockf(1,0,0); /\*解锁\*/

}

else

{

while((p2=fork( ))= =-1); /\*创建子进程p2\*/

if (p2= =0)

{

lockf(1,1,0); /\*加锁\*/

for(i=0;i<10;i++)

printf("son %d\n",i);

lockf(1,0,0); /\*解锁\*/

}

else

{

lockf(1,1,0); /\*加锁\*/

for(i=0;i<10;i++)

printf(" parent %d\n",i);

lockf(1,0,0); /\*解锁\*/

}

}

}

三、运行结果

观察上锁和实验1中未上锁的区别。

parent…

son…

daughter..

daughter..

或parent…

son…

parent…

daughter…

大致与未上锁的输出结果相同，也是随着执行时间不同，输出结果的顺序有所不同。

四、分析原因

上述程序执行时，不同进程之间不存在共享临界资源（其中打印机的互斥性已由操作系统保证）问题，所以加锁与不加锁效果相同。

# （四）守护进程实验

写一个使用守护进程(daemon)的程序，来实现：

1, 创建一个日志文件/var/log/Mydaemon.log

2，每分钟都向其中写入一个时间戳(使用time\_t的格式)

注意：要root权限才能在/var/log创建文件。

守护神程序：

main(){

time\_t t; //建立time\_t格式变量

FILE \*fp; //建立文件

fp=fopen("/var/log/Mydaemon.log","a");//打开文件

**setlinebuf(fp); //设置行缓冲**

pid\_t pid; //守护神

pid=fork();

if(pid>0){

printf("Daemon on duty!\n");

exit(0);

}

else if(pid<0){

printf("Can't fork!\n");

exit(-1);

}

while(1){

if(fp>=0){

sleep(60); //等待一分钟再往文件中写入时间戳

printf("Daemon on duty!\n");

t=time(0);

fprintf(fp,"The current time is %s\n",asctime(localtime(&t)));

}

}

fclose(fp);//关闭文件

}

思考：

1、 以下语句“ setlinebuf(fp); //设置行缓冲”起到什么作用？

# （五） 信号机制实验

UNIX/LINUX系统的进程间通信机构（IPC）允许在任意进程间大批量地交换数据。本实验的目的是了解和熟悉LINUX支持的信号量机制、管道机制、消息通信机制及共享存储区机制。

实验目的

1、了解什么是信号

2、熟悉LINUX系统中进程之间软中断通信的基本原理

实验内容

1、编写程序：用fork( )创建两个子进程，再用系统调用signal( )让父进程捕捉键盘上来的中断信号（即按^c键）；捕捉到中断信号后，父进程用系统调用kill( )向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child process1 is killed by parent!

Child process2 is killed by parent!

父进程等待两个子进程终止后，输出如下的信息后终止：

Parent process is killed!

2、分析利用软中断通信实现进程同步的机理

实验指导

一、信号

1、信号的基本概念

每个信号都对应一个正整数常量(称为signal number,即信号编号。定义在系统头文件<signal.h>中)，代表同一用户的诸进程之间传送事先约定的信息的类型，用于通知某进程发生了某异常事件。每个进程在运行时，都要通过信号机制来检查是否有信号到达。若有，便中断正在执行的程序，转向与该信号相对应的处理程序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。实质上，信号机制是对中断机制的一种模拟，故在早期的UNIX版本中又把它称为软中断。

信号与中断的相似点：

（1）采用了相同的异步通信方式；

（2）当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；

（3）都在处理完毕后返回到原来的断点；

（4）对信号或中断都可进行屏蔽。

信号与中断的区别：

（1）中断有优先级，而信号没有优先级，所有的信号都是平等的；

（2）信号处理程序是在用户态下运行的，而中断处理程序是在核心态下运行；

（3）中断响应是及时的，而信号响应通常都有较大的时间延迟。

信号机制具有以下三方面的功能：

（1）发送信号。发送信号的程序用系统调用kill( )实现；

（2）预置对信号的处理方式。接收信号的程序用signal( )来实现对处理方式的预置；

（3）收受信号的进程按事先的规定完成对相应事件的处理。

2、信号的发送

信号的发送，是指由发送进程把信号送到指定进程的信号域的某一位上。如果目标进程正在一个可被中断的优先级上睡眠，核心便将它唤醒，发送进程就此结束。一个进程可能在其信号域中有多个位被置位，代表有多种类型的信号到达，但对于一类信号，进程却只能记住其中的某一个。

进程用kill( )向一个进程或一组进程发送一个信号。

3、对信号的处理

当一个进程要进入或退出一个低优先级睡眠状态时，或一个进程即将从核心态返回用户态时，核心都要检查该进程是否已收到软中断。当进程处于核心态时，即使收到软中断也不予理睬；只有当它返回到用户态后，才处理软中断信号。对软中断信号的处理分三种情况进行：

（1）如果进程收到的软中断是一个已决定要忽略的信号（function=1），进程不做任何处理便立即返回；

（2）进程收到软中断后便退出（function=0）；

（3）执行用户设置的软中断处理程序。

二、所涉及的中断调用

**1、kill( )**

系统调用格式

int kill(pid,sig)

参数定义

int pid,sig;

其中，pid是一个或一组进程的标识符，参数sig是要发送的软中断信号。

（1）pid>0时，核心将信号发送给进程pid。

（2）pid=0时，核心将信号发送给与发送进程同组的所有进程。

（3）pid=-1时，核心将信号发送给所有用户标识符真正等于发送进程的有效用户标识号的进程。

**2、signal( )**

预置对信号的处理方式，允许调用进程控制软中断信号。

系统调用格式

signal(sig,function)

头文件为

　 #include <signal.h>

参数定义

signal(sig,function)

int sig;

void (\*func) ( )

其中sig用于指定信号的类型，sig为0则表示没有收到任何信号，余者如下表：

|  |  |  |
| --- | --- | --- |
| 值 | 名 字 | 说 明 |
| 01 | SIGHUP | 挂起（hangup） |
| 02 | SIGINT | 中断，当用户从键盘按^c键或^break键时 |
| 03 | SIGQUIT | 退出，当用户从键盘按quit键时 |
| 04 | SIGILL | 非法指令 |
| 05 | SIGTRAP | 跟踪陷阱（trace trap），启动进程，跟踪代码的执行 |
| 06 | SIGIOT | IOT指令 |
| 07 | SIGEMT | EMT指令 |
| 08 | SIGFPE | 浮点运算溢出 |
| 09 | SIGKILL | 杀死、终止进程 |
| 10 | SIGBUS | 总线错误 |
| 11 | SIGSEGV | 段违例（segmentation violation），进程试图去访问其虚地址空间以外的位置 |
| 12 | SIGSYS | 系统调用中参数错，如系统调用号非法 |
| 13 | SIGPIPE | 向某个非读管道中写入数据 |
| 14 | SIGALRM | 闹钟。当某进程希望在某时间后接收信号时发此信号 |
| 15 | SIGTERM | 软件终止（software termination） |
| 16 | SIGUSR1 | 用户自定义信号1 |
| 17 | SIGUSR2 | 用户自定义信号2 |
| 18 | SIGCLD | 某个子进程死 |
| 19 | SIGPWR | 电源故障 |

function：在该进程中的一个函数地址，在核心返回用户态时，它以软中断信号的序号作为参数调用该函数，对除了信号SIGKILL，SIGTRAP和SIGPWR以外的信号，核心自动地重新设置软中断信号处理程序的值为SIG\_DFL，一个进程不能捕获SIGKILL信号。

function 的解释如下：

（1）function=1时，进程对sig类信号不予理睬，亦即屏蔽了该类信号；

（2）function=0时，缺省值，进程在收到sig信号后应终止自己；

（3）function为非0，非1类整数时，function的值即作为信号处理程序的指针。

三、参考程序

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

void waiting( ),stop( );

int wait\_mark;

main( )

{

int p1,p2,stdout;

while((p1=fork( ))= =-1); /\*创建子进程p1\*/

if (p1>0)

{

while((p2=fork( ))= =-1); /\*创建子进程p2\*/

if(p2>0)

{

wait\_mark=1;

signal(SIGINT,stop); /\*接收到^c信号，转stop\*/

waiting( );

kill(p1,16); /\*向p1发软中断信号16\*/

kill(p2,17); /\*向p2发软中断信号17\*/

wait(0); /\*同步\*/

wait(0);

printf("Parent process is killed!\n");

exit(0);

}

else

{

wait\_mark=1;

signal(17,stop); /\*接收到软中断信号17，转stop\*/

waiting( );

lockf(stdout,1,0);

printf("Child process 2 is killed by parent!\n");

lockf(stdout,0,0);

exit(0);

}

}

else

{

wait\_mark=1;

signal(16,stop); /\*接收到软中断信号16，转stop\*/

waiting( );

lockf(stdout,1,0);

printf("Child process 1 is killed by parent!\n");

lockf(stdout,0,0);

exit(0);

}  
 }

void waiting( )

{

while(wait\_mark!=0);

}

void stop( )

{

wait\_mark=0;

}

四、运行结果

屏幕上无反应，按下^C后，显示 Parent process is killed!

五、分析原因

上述程序中，signal( )都放在一段程序的前面部位，而不是在其他接收信号处。这是因为signal( )的执行只是为进程指定信号值16或17的作用，以及分配相应的与stop( )过程链接的指针。因而，signal( )函数必须在程序前面部分执行。

本方法通信效率低，当通信数据量较大时一般不用此法。

六、思考

1、为何预期的结果并未显示出？程序该如何修改才能得到正确结果？

2、该程序段前面部分用了两个wait(0)，它们起什么作用？

# （六）进程的管道通信实验

实验目的

1、了解什么是管道

2、熟悉UNIX/LINUX支持的管道通信方式

实验内容

编写程序实现进程的管道通信。用系统调用pipe( )建立一管道，二个子进程P1和P2分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收P1，后P2）。

实验指导

一、什么是管道

UNIX系统在OS的发展上，最重要的贡献之一便是该系统首创了管道（pipe）。这也是UNIX系统的一大特色。

所谓管道，是指能够连接一个写进程和一个读进程的、并允许它们以生产者—消费者方式进行通信的一个共享文件，又称为pipe文件。由写进程从管道的写入端（句柄1）将数据写入管道，而读进程则从管道的读出端（句柄0）读出数据。

|  |  |  |
| --- | --- | --- |
| 句柄fd[0]  句柄fd[1] |  | 读出端  写入端 |

二、管道的类型：

1、有名管道

一个可以在文件系统中长期存在的、具有路径名的文件。用系统调用mknod( )建立。它克服无名管道使用上的局限性，可让更多的进程也能利用管道进行通信。因而其它进程可以知道它的存在，并能利用路径名来访问该文件。对有名管道的访问方式与访问其他文件一样，需先用open( )打开。

2、无名管道

一个临时文件。利用pipe( )建立起来的无名文件（无路径名）。只用该系统调用所返回的文件描述符来标识该文件，故只有调用pipe( )的进程及其子孙进程才能识别此文件描述符，才能利用该文件（管道）进行通信。当这些进程不再使用此管道时，核心收回其索引结点。

二种管道的读写方式是相同的，本文只讲无名管道。

3、pipe文件的建立

分配磁盘和内存索引结点、为读进程分配文件表项、为写进程分配文件表项、分配用户文件描述符

4、读/写进程互斥

内核为地址设置一个读指针和一个写指针，按先进先出顺序读、写。

为使读、写进程互斥地访问pipe文件，需使各进程互斥地访问pipe文件索引结点中的直接地址项。因此，每次进程在访问pipe文件前，都需检查该索引文件是否已被上锁。若是，进程便睡眠等待，否则，将其上锁，进行读/写。操作结束后解锁，并唤醒因该索引结点上锁而睡眠的进程。

三、所涉及的系统调用

**1、pipe( )**

建立一无名管道。

系统调用格式

pipe(filedes)

参数定义

int pipe(filedes);

int filedes[2];

其中，filedes[1]是写入端，filedes[0]是读出端。

该函数使用头文件如下：

#include <unistd.h>

#inlcude <signal.h>

#include <stdio.h>

**2、read( )**

系统调用格式

read(fd,buf,nbyte)

功能：从fd所指示的文件中读出nbyte个字节的数据，并将它们送至由指针buf所指示的缓冲区中。如该文件被加锁，等待，直到锁打开为止。

参数定义

int read(fd,buf,nbyte);

int fd;

char \*buf;

unsigned nbyte;

**3、write( )**

系统调用格式

read(fd,buf,nbyte)

功能：把nbyte 个字节的数据，从buf所指向的缓冲区写到由fd所指向的文件中。如文件加锁，暂停写入，直至开锁。

参数定义同read( )。

四、参考程序

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

int pid1,pid2;

main( )

{

int fd[2];

char outpipe[100],inpipe[100];

pipe(fd); /\*创建一个管道\*/

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

if(pid1= =0)

{

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

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

/\*把串放入数组outpipe中\*/

write(fd[1],outpipe,50); /\*向管道写长为50字节的串\*/

sleep(5); /\*自我阻塞5秒\*/

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

exit(0);

}

else

{

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

if(pid2= =0)

{ lockf(fd[1],1,0); /\*互斥\*/

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); /\*同步\*/

read(fd[0],inpipe,50); /\*从管道中读长为50字节的串\*/

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

wait(0);

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

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

exit(0);

}

}

}

五、运行结果

延迟5秒后显示

child 1 process is sending message!

再延迟5秒

child 2 process is sending message!

六、思考题

1、程序中的sleep(5)起什么作用？

2、怎样保证先child1进程，再child2进程？

3、子进程1和2为什么也能对管道进行操作？

# （七）消息的发送与接收实验

实验目的

1、了解什么是消息

2、熟悉消息传送的机理

实验内容

消息的创建、发送和接收。使用系统调用msgget( ),msgsnd( ),msgrev( ),及msgctl( )编制一长度为１k的消息发送和接收的程序。

实验指导

一、什么是消息

消息（message）是一个格式化的可变长的信息单元。消息机制允许由一个进程给其它任意的进程发送一个消息。当一个进程收到多个消息时，可将它们排成一个消息队列。消息使用二种重要的数据结构：一是消息首部，其中记录了一些与消息有关的信息，如消息数据的字节数；二个消息队列头表，其每一表项是作为一个消息队列的消息头，记录了消息队列的有关信息。

1、消息机制的数据结构

（1）消息首部

记录一些与消息有关的信息，如消息的类型、大小、指向消息数据区的指针、消息队列的链接指针等。

（2）消息队列头表

其每一项作为一个消息队列的消息头，记录了消息队列的有关信息如指向消息队列中第一个消息和指向最后一个消息的指针、队列中消息的数目、队列中消息数据的总字节数、队列所允许消息数据的最大字节总数，还有最近一次执行发送操作的进程标识符和时间、最近一次执行接收操作的进程标识符和时间等。

2、消息队列的描述符

UNIX中，每一个消息队列都有一个称为关键字（key）的名字，是由用户指定的；消息队列有一消息队列描述符，其作用与用户文件描述符一样，也是为了方便用户和系统对消息队列的访问。

二、涉及的系统调用

**1. msgget( )**

创建一个消息，获得一个消息的描述符。核心将搜索消息队列头表，确定是否有指定名字的消息队列。若无，核心将分配一新的消息队列头，并对它进行初始化，然后给用户返回一个消息队列描述符，否则它只是检查消息队列的许可权便返回。

系统调用格式：

msgqid=msgget(key,flag)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

参数定义

int msgget(key,flag)

key\_t key;

int flag;

其中：

key是用户指定的消息队列的名字；flag是用户设置的标志和访问方式。如 IPC\_CREAT |0400 是否该队列已被创建。无则创建，是则打开；

IPC\_EXCL |0400 是否该队列的创建应是互斥的。

msgqid 是该系统调用返回的描述符，失败则返回-1。

**2. msgsnd（）**

发送一消息。向指定的消息队列发送一个消息，并将该消息链接到该消息队列的尾部。

系统调用格式：

msgsnd(msgqid,msgp,size,flag)

该函数使用头文件如下：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

参数定义：

int msgsnd(msgqid,msgp,size,flag)

I int msgqid,size,flag;

struct msgbuf \* msgp;

其中msgqid是返回消息队列的描述符；msgp是指向用户消息缓冲区的一个结构体指针。缓冲区中包括消息类型和消息正文，即

{

long mtype; /\*消息类型\*/

char mtext[ ]; /\*消息的文本\*/

}

size指示由msgp指向的数据结构中字符数组的长度；即消息的长度。这个数组的最大值由MSG-MAX( )系统可调用参数来确定。flag规定当核心用尽内部缓冲空间时应执行的动作:进程是等待，还是立即返回。若在标志flag中未设置IPC\_NOWAIT位，则当该消息队列中的字节数超过最大值时，或系统范围的消息数超过某一最大值时，调用msgsnd进程睡眠。若是设置IPC\_NOWAIT，则在此情况下，msgsnd立即返回。

对于msgsnd( )，核心须完成以下工作：

（1）对消息队列的描述符和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；

（2）核心为消息分配消息数据区。将用户消息缓冲区中的消息正文，拷贝到消息数据区；

（3）分配消息首部，并将它链入消息队列的末尾。在消息首部中须填写消息类型、消息大小和指向消息数据区的指针等数据；

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

**3. msgrcv( )**

接受一消息。从指定的消息队列中接收指定类型的消息。

系统调用格式：

msgrcv(msgqid,msgp,size,type,flag)

本函数使用的头文件如下：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

参数定义：

int msgrcv(msgqid,msgp,size,type,flag)

int msgqid,size,flag;

struct msgbuf \*msgp;

long type;

其中，msgqid,msgp,size,flag与msgsnd中的对应参数相似，type是规定要读的消息类型，flag规定倘若该队列无消息，核心应做的操作。如此时设置了IPC\_NOWAIT标志，则立即返回，若在flag中设置了MS\_NOERROR，且所接收的消息大于size，则核心截断所接收的消息。

对于msgrcv系统调用，核心须完成下述工作：

（1）对消息队列的描述符和许可权等进行检查。若合法，就往下执行；否则返回；

（2）根据type的不同分成三种情况处理：

type=0，接收该队列的第一个消息，并将它返回给调用者；

type为正整数，接收类型type的第一个消息；

type为负整数，接收小于等于type绝对值的最低类型的第一个消息。

（3）当所返回消息大小等于或小于用户的请求时，核心便将消息正文拷贝到用户区，并从消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的大时，则做出错返回。

**4. msgctl( )**

消息队列的操纵。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。

系统调用格式：

msgctl(msgqid,cmd,buf);

本函数使用的头文件如下：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

参数定义：

int msgctl(msgqid,cmd,buf);

int msgqid,cmd;

struct msgqid\_ds \*buf;

其中，函数调用成功时返回0，不成功则返回-1。buf是用户缓冲区地址，供用户存放控制参数和查询结果；cmd是规定的命令。命令可分三类：

（1）IPC\_STAT。查询有关消息队列情况的命令。如查询队列中的消息数目、队列中的最大字节数、最后一个发送消息的进程标识符、发送时间等；

（2）IPC\_SET。按buf指向的结构中的值，设置和改变有关消息队列属性的命令。如改变消息队列的用户标识符、消息队列的许可权等；

（3）IPC\_RMID。消除消息队列的标识符。

msgqid\_ds 结构定义如下：

struct msgqid\_ds

{ struct ipc\_perm msg\_perm; /\*许可权结构\*/

short pad1[7]; /\*由系统使用\*/

ushort msg\_qnum; /\*队列上消息数\*/

ushort msg\_qbytes; /\*队列上最大字节数\*/

ushort msg\_lspid; /\*最后发送消息的PID\*/

ushort msg\_lrpid; /\*最后接收消息的PID\*/

time\_t msg\_stime; /\*最后发送消息的时间\*/

time\_t msg\_rtime; /\*最后接收消息的时间\*/

time\_t msg\_ctime; /\*最后更改时间\*/

};

struct ipc\_perm

{ ushort uid; /\*当前用户\*/

ushort gid; /\*当前进程组\*/

ushort cuid; /\*创建用户\*/

ushort cgid; /\*创建进程组\*/

ushort mode; /\*存取许可权\*/

{ short pid1; long pad2;} /\*由系统使用\*/

}

三、参考程序

1、client.c

#include <sys/types.h>

#include <sys/msg.h>

#include <sys/ipc.h>

#define MSGKEY 75

struct msgform

{ long mtype;

char mtext[1000];

}msg;

int msgqid;

void client()

{

int i;

msgqid=msgget(MSGKEY,0777); /\*打开75#消息队列\*/

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

{

msg.mtype=i;

printf(“(client)sent\n”);

msgsnd(msgqid,&msg,1024,0); /\*发送消息\*/

}

exit(0);

}

main( )

{

client( );

}

2、server.c

#include <sys/types.h>

#include <sys/msg.h>

#include <sys/ipc.h>

#define MSGKEY 75

struct msgform

{ long mtype;

char mtext[1000];

}msg;

int msgqid;

void server( )

{

msgqid=msgget(MSGKEY,0777|IPC\_CREAT); /\*创建75#消息队列\*/

do

{

msgrcv(msgqid,&msg,1030,0,0); /\*接收消息\*/

printf(“(server)received\n”);

}while(msg.mtype!=1);

msgctl(msgqid,IPC\_RMID,0); /\*删除消息队列，归还资源\*/

exit(0);

}

main( )

{

server( );

}

四、程序说明

1、为了便于操作和观察结果，编制二个程序client.c和server.c，分别用于消息的发送与接收。

2、server建立一个 Key 为75的消息队列，等待其它进程发来的消息。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。”

3、client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。最后一个消息，即是 server端需要的结束信号。client 每发送一条消息后显示一句 “(client)sent”。

4、注意： 二个程序分别编辑、编译为client与server。执行：

./server&

ipcs -q

./client。

五、运行结果

从理想的结果来说，应当是每当client发送一个消息后，server接收该消息，client再发送下一条。也就是说“(client)sent”和 “(server)received”的字样应该在屏幕上交替出现。实际的结果大多是，先由client发送了两条消息，然后server接收一条消息。此后client 、server交替发送和接收消息。最后server一次接收两条消息。client 和server 分别发送和接收了10条消息，与预期设想一致。

六、思考

1、实现“(client)sent”和 “(server)received”的交替输出。

注：message的传送和控制并不保证完全同步，当一个程序不在激活状态的时候，它完全可能继续睡眠，造成了上面的现象，在多次send message 后才recieve message。这一点有助于理解消息传送的实现机理。

# （八）共享存储区通信

实验目的

了解和熟悉共享存储机制

实验内容

编制一长度为1k的共享存储区发送和接收的程序。

实验指导

一、共享存储区

1、共享存储区机制的概念

共享存储区（Share Memory）是UNIX系统中通信速度最高的一种通信机制。该机制可使若干进程共享主存中的某一个区域，且使该区域出现（映射）在多个进程的虚地址空间中。另一方面，一个进程的虚地址空间中又可连接多个共享存储区，每个共享存储区都有自己的名字。当进程间欲利用共享存储区进行通信时，必须先在主存中建立一共享存储区，然后将它附接到自己的虚地址空间上。此后，进程对该区的访问操作，与对其虚地址空间的其它部分的操作完全相同。进程之间便可通过对共享存储区中数据的读、写来进行直接通信。图示列出二个进程通过共享一个共享存储区来进行通信的例子。其中，进程A将建立的共享存储区附接到自己的AA’区域，进程B将它附接到自己的BB’区域。

进程A的虚空间 内存空间 进程B的虚空间

正文

数据

栈

共享  
存储区

正文

数据

B

B’

栈

A

A’

应当指出，共享存储区机制只为进程提供了用于实现通信的共享存储区和对共享存储区进行操作的手段，然而并未提供对该区进行互斥访问及进程同步的措施。因而当用户需要使用该机制时，必须自己设置同步和互斥措施才能保证实现正确的通信。

二、涉及的系统调用

**1、shmget( )**

创建、获得一个共享存储区。

系统调用格式：

shmid=shmget(key,size,flag)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

参数定义

int shmget(key,size,flag);

key\_t key;

int size,flag;

其中，key是共享存储区的名字；size是其大小（以字节计）；flag是用户设置的标志，如IPC\_CREAT。IPC\_CREAT表示若系统中尚无指名的共享存储区，则由核心建立一个共享存储区；若系统中已有共享存储区，便忽略IPC\_CREAT。

附：

操作允许权 八进制数

用户可读 00400

用户可写 00200

小组可读 00040

小组可写 00020

其它可读 00004

其它可写 00002

控制命令 值

IPC\_CREAT 0001000

IPC\_EXCL 0002000

例：shmid=shmget(key,size,(IPC\_CREAT|0400))

创建一个关键字为key，长度为size的共享存储区

**2、shmat( )**

共享存储区的附接。从逻辑上将一个共享存储区附接到进程的虚拟地址空间上。

系统调用格式：

virtaddr=shmat(shmid,addr,flag)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

参数定义

char \*shmat(shmid,addr,flag);

int shmid,flag;

char \* addr;

其中，shmid是共享存储区的标识符；addr是用户给定的，将共享存储区附接到进程的虚地址空间；flag规定共享存储区的读、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，表示操作系统在必要时舍去这个地址。该系统调用的返回值是共享存储区所附接到的进程虚地址viraddr。

**3、shmdt( )**

把一个共享存储区从指定进程的虚地址空间断开。

系统调用格式：

shmdt(addr)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

参数定义

int shmdt(addr);

char addr;

其中，addr是要断开连接的虚地址，亦即以前由连接的系统调用shmat( )所返回的虚地址。调用成功时，返回0值，调用不成功，返回-1。

**4、shmctl( )**

共享存储区的控制，对其状态信息进行读取和修改。

系统调用格式：

shmctl(shmid,cmd,buf)

该函数使用头文件如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

参数定义

int shmctl(shmid,cmd,buf);

int shmid,cmd;

struct shmid\_ds \*buf;

其中，buf是用户缓冲区地址，cmd是操作命令。命令可分为多种类型：

（1）用于查询有关共享存储区的情况。如其长度、当前连接的进程数、共享区的创建者标识符等；

（2）用于设置或改变共享存储区的属性。如共享存储区的许可权、当前连接的进程计数等；

（3）对共享存储区的加锁和解锁命令；

（4）删除共享存储区标识符等。

上述的查询是将shmid所指示的数据结构中的有关成员，放入所指示的缓冲区中；而设置是用由buf所指示的缓冲区内容来设置由shmid所指示的数据结构中的相应成员。

三、参考程序

#include <sys/types.h>

#include <sys/shm.h>

#include <sys/ipc.h>

#define SHMKEY 75

int shmid,i; int \*addr;

void client( )

{ int i;

shmid=shmget(SHMKEY,1024,0777); /\*打开共享存储区\*/

addr=shmat(shmid,0,0); /\*获得共享存储区首地址\*/

for (i=9;i>=0;i--)

{ while (\*addr!=-1);

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

\*addr=i;

}

exit(0);

}

void server( )

{

shmid=shmget(SHMKEY,1024,0777|IPC\_CREAT); /\*创建共享存储区\*/

addr=shmat(shmid,0,0); /\*获取首地址\*/

do

{

\*addr=-1;

while (\*addr==-1);

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

}while (\*addr);

shmctl(shmid,IPC\_RMID,0); /\*撤消共享存储区，归还资源\*/

exit(0);

}

main( )

{

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

if (!i) server( );

system(“ipcs -m”);

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

if (!i) client( );

wait(0);

wait(0);

}

四、程序说明

1、为了便于操作和观察结果，用一个程序作为“引子“，先后fork()两个子进程，server和client，进行通信。

2、server端建立一个key为75的共享区，并将第一个字节置为-1，作为数据空的标志。等待其他进程发来的消息。当该字节的值发生变化时，表示收到了信息，进行处理。然后再次把它的值设为-1，如果遇到的值为0，则视为为结束信号，取消该队列，并退出server。server每接收到一次数据后显示“(server)received”。

3、client端建立一个key为75的共享区，当共享取得第一个字节为-1时，server端空闲，可发送请求。client随即填入9到0。期间等待 server 端的再次空闲。进行完这些操作后，client退出。client每发送一次数据后显示“(client)sent”。

4、父进程在server和client均退出后结束。

五、运行结果

和预想的完全一样。但在运行过程中，发现每当client发送一次数据后，server要等待大约0.1秒才有响应。同样，之后client又需要等待大约0.1秒才发送下一个数据。

六、程序分析

出现上述应答延迟的现象是程序设计的问题。当client端发送了数据后，并没有任何措施通知server端数据已经发出，需要由client的查询才能感知。此时，client端并没有放弃系统的控制权，仍然占用CPU的时间片。只有当系统进行调度时，切换到了server进程，再进行应答。这个问题，也同样存在于server端到client的应答过程中。

七、思考题

1、比较两种消息通信机制中数据传输的时间

由于两种机制实现的机理和用处都不一样，难以直接进行时间上的比较。如果比较其性能，应更加全面的分析。

（1）消息队列的建立比共享区的设立消耗的资源少。前者只是一个软件上设定的问题，后者需要对硬件的操作，实现内存的映像，当然控制起来比前者复杂。如果每次都重新进行队列或共享的建立，共享区的设立没有什么优势。

（2）当消息队列和共享区建立好后，共享区的数据传输，受到了系统硬件的支持，不耗费多余的资源；而消息传递，由软件进行控制和实现，需要消耗一定的cpu的资源。从这个意义上讲，共享区更适合频繁和大量的数据传输。

（3）消息的传递，自身就带有同步的控制。当等到消息的时候，进程进入睡眠状态，不再消耗cpu资源。而共享队列如果不借助其他机制进行同步，接收数据的一方必须进行不断的查询，白白浪费了大量的cpu资源。可见，消息方式的使用更加灵活。

实验二 银行家算法

　一、实验指导

　　要解释银行家算法，必须先解释操作系统安全状态和不安全状态。

　　安全状态：如果存在一个由系统中所有进程构成的安全序列P1，…，Pn，则系统处于安全状态。安全状态一定是没有死锁发生。

　　不安全状态:不存在一个安全序列。不安全状态不一定导致死锁。

　　那么什么是安全序列呢？

　　安全序列：一个进程序列{P1，…，Pn}是安全的，如果对于每一个进程Pi(1≤i≤n），它以后尚需要的资源量不超过系统当前剩余资源量与所有进程Pj (j < i )当前占有资源量之和。

　　银行家算法：

　　按银行家制定的规则为进程分配资源，可以使系统保持在安全状态，具体方法如下：

·进程首次申请资源的分配

如果系统现存资源可以满足该进程的最大需求量，则按当前的申请量分配资源，否则推迟分配。

·进程在执行中继续申请资源的分配

若该进程已占用的资源与本次申请的资源之和不超过对资源的最大需求量，且现存资源能满足该进程尚需的最大资源量，则按当前申请量分配资源，否则推迟分配。

·至少一个进程能完成

在任何时刻保证至少有一个进程能得到所需的全部资源而执行到结束。

银行家算法通过动态地检测系统中资源分配情况和进程对资源的需求情况来决定如何分配资源，并能在确保系统处于安全状态时才把资源分配给申请者，从而避免系统发生死锁。

二、实验目的

掌握银行家算法思想，并能编程实现。

三、实验内容和要求

1、在Linux环境下编译运行程序；  
2、按照教材3.7.2节的算法编写；  
3、输入数据从文本文件中读出，不得从键盘录入；  
4、主要数据结构的变量名和教材中的一致，包括Available、Max、Allocation、Need、Request、Work、Finish。并作以下改动：在变量名后加下划杠和姓名缩写。例如，张三丰同学的变量名为： Available\_zsf， Max\_zsf， Allocation \_zsf……  
5、程序可支持不同个数的进程和不同个数的资源；  
6、验证教材中的“银行家算法示例”中的例子（包括可成功分配、不可分配）。

四、实验程序

五、验证数据和运行结果

六、思考和分析

实验三 内存管理实验

# （一）常用页面置换算法模拟实验

一、实验指导

一、虚拟存储系统

UNIX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，核心配置了四种数据结构：页表、页框号、访问位、修改位、有效位、保护位等。

二、页面置换算法

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。该程序通过查找页表，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/O将所缺之页调入内存，然后修改页表。如果内存已满，则须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有

1、最佳置换算法（Optimal）

2、先进先出法（Fisrt In First Out）

3、最近最久未使用（Least Recently Used）

4、最不经常使用法（Least Frequently Used）

5、最近未使用法（No Used Recently）

二、实验目的

通过模拟实现请求页式存储管理的几种基本页面置换算法，了解虚拟存储技术的特点，掌握虚拟存储请求页式存储管理中几种基本页面置换算法的基本思想和实现过程，并比较它们的效率。

三、实验内容

1、访问页面号以数字为模拟，输入数据从文本文件中读出，数字间以空格为间隔，假设有50个页面数据。

2、编写程序分别使用下述四个算法计算访问命中率：最佳淘汰算法（OPT），先进先出的算法（FIFO），最近最久未使用算法（LRU），最不经常使用算法（LFU）。

3、输出访问顺序和命中率结果。

四、实验程序

五、验证数据和运行结果

六、思考和分析

1、比较几种算法的命中率。

# （二）Linux下的内存分配与回收的管理

一、实验目的

在Linux环境下利用下列系统调用malloc()， free()编写一段程序实现内存分配与回收的管理。

二、实验内容和要求

1. 返回已分配给变量的内存地址；
2. 返回释放后的内存地址；
3. 释放已分配的内存空间后，返回释放内存后未使用内存的大小。

三、实验参考程序

/\* MALLOC.C: This program allocates memory with

\* malloc, then frees the memory with free.

\*/

#include <stdlib.h> /\* For \_MAX\_PATH definition \*/

#include <stdio.h>

#include <malloc.h>

#include <iostream.h>

void main()

{

int \*string;

/\* Allocate space for a path name \*/

string =(int\*) malloc(10 );

if( string == NULL )

printf( "Insufficient memory available\n" );

else

{

printf( "Memory space allocated for path name\n" );

/\*printf ("string=%d\n",string);\*/

cout<<"string="<<string<<endl;

free( string );

printf( "Memory freed\n" );

}

int \*stringy;

/\* Allocate space for a path name \*/

stringy =(int\*) malloc(12 );

if( stringy == NULL )

printf( "Insufficient memory available\n" );

else

{

printf( "Memory space allocated for path name\n" );

/\*printf ("string=%d\n",string);\*/

cout<<"stringy="<<stringy<<endl;

free( stringy );

printf( "Memory freed\n" );

}

}

四、运行结果

五、思考和分析

实验四 文件管理实验

一、实验目的

学会使用库函数和系统调用实现文件的基本操作。

二、实验内容

1、利用函数fopen(), fread(), fwrite(), fclose() 实现简单的文件备份， 即将一个文件的内容拷贝到另一个文件中去。

2、利用系统调用函数open(), read(), write(), close() 实现上述功能。

三、实验程序

四、运行结果

五、思考和分析

实验五 磁盘管理实验

一、实验目的

掌握几种基本的磁盘调度算法。

二、实验内容和要求

1、采用模拟先来先服务法（First-Come, First-Served，FCFS），最短寻道时间优先法（Shortest Seek Time First， SSTF），电梯法三种磁盘调度算法，对一组请求访问磁道序列输出为每种调度算法的磁头移动轨迹和移动的总磁道数。

2、请求访问磁道序列的数据放在文件里，数据间以空格间隔。

三、实验程序

四、运行结果

五、思考和分析