**Linux程序设计复习资料**

考试题型：

填空题（10分）：**基本概念**理解和命令的使用

简答题（35分）：**核心概念**的理解（包含简单shell编程）

程序及理解（30分）：按照要求编写/补充/比较程序（如果需要函数，会将函数原型附上）

思考与论述（25分）：对Linux知识的应用和思考（可能需要伪代码）

1. 基本Linux命令
   1. 文件操作
      1. Touch

途

更新文件的访问和修改时间。

语法

touch [ -a ] [ -c ] [ -m ] [ -f ] [ -r RefFile ] [ Time | -t Time ] { File ... | Directory ... }

描述

touch 命令更新由 Directory 参数指定的每个目录下的由 File 参数指定的每个文件的访问和修改时间。如果没有指定 Time 变量值，touch 命令就使用当前时间。如果指定了一个不存在的文件，touch 命令就创建此文件，除非指定了 -c 标志。

touch 命令的返回码是时间没有被成功修改的文件数目（包括不存在的文件和没有创建的文件）。

注:

任何超出 2038 年（包含2038年）的日期都是无效的。

标志

-a 更改由 File 变量指定的文件的访问时间。不要更改修改时间，除非也指定了 -m 标志。

-c 如果文件不存在，则不要进行创建。没有写任何有关此条件的诊断消息。

-f 尝试强制 touch 运行，而不管文件的读和写许可权。

-m 更改 File 的修改时间。不要更改访问时间，除非也指定了 -m 标志。

-r RefFile 使用由 RefFile 变量指定的文件的相应时间，而不用当前时间。

Time 以 MMDDhhmm[YY] 的格式指定新时间戳记的日期和时间，其中：

MM

指定一年的哪一月（从 01 到 12）。

DD

指定一月的哪一天（从 01 到 31）。

hh

指定一天中的哪一小时（从 00 到 23）。

mm

指定一小时的哪一分钟（从 00 到 59）。

YY

指定年份的后两位数字。如果 YY 变量没有被指定，缺省值为当前年份。

-t Time 使用指定时间而不是当前时间。Time 变量以十进制形式 [[CC]YY]MMDDhhmm[.SS] 指定，其中：

CC

指定年份的前两位数字。

YY

指定年份的后两位数字。

MM

指定一年的哪一月（从 01 到 12）。

DD

指定一月的哪一天（从 01 到 31）。

hh

指定一天中的哪一小时（从 00 到 23）。

mm

指定一小时的哪一分钟（从 00 到 59）。

SS

指定一分钟的哪一秒（从 00 到 59）。

注:

touch 命令调用 utime () 子例程来更改所涉及文件的修改和访问时间。当没有真正拥有该文件，即使对文件有写许可权，使用标志时也可能使 touch 命令失败。

当使用 touch 命令时，如果接收到错误消息，不要指定完整路径名 /usr/bin/touch。

退出状态

命令返回以下出口值：

0 命令成功执行。所有请求的更改已完成。

>0 发生一个错误。

示例

要更新文件的访问和修改时间，请输入：

touch program。c

这会将 program.c 文件的上次访问和修改时间设置为当前日期和时间。如果 program.c 文件不存在，touch 命令就创建一个同名的空文件。

要避免创建新文件，请输入：

touch -c program.c

要仅更新修改时间，请输入：

touch -m \*.o

这会更新当前目录下的以 .o 扩展名结尾的文件的上次修改时间（不更新访问时间）。touch 命令经常以此方式使用，来改变 make 命令的结果。

要明确设置访问和修改时间，请输入：

touch -c -t 02171425 program.c

这会将访问和修改时间设置为当前年份的 2 月 17 日的 14:25（2:25 p.m.）。

要使用另一文件的时间戳记而不用当前时间，请输入：

touch -r file1 program.c

这会将 program.c 文件的时间戳记设置成与 file1 文件一样。

要使用指定时间而不是当前时间处理文件，请输入：

touch -t 198503030303.55 program.c

这会将 program.c 文件的时间戳记设置为 1985 年 3 月 3 日上午 3:03:55。

* + 1. Mkdir

linux mkdir 命令用来创建指定的名称的目录，要求创建目录的用户在当前目录中具有写权限，并且指定的目录名不能是当前目录中已有的目录。

1．命令格式：

mkdir [选项] 目录...

2．命令功能：

通过 mkdir 命令可以实现在指定位置创建以 DirName(指定的文件名)命名的文件夹或目录。要创建文件夹或目录的用户必须对所创建的文件夹的父文件夹具有写权限。并且，所创建的文件夹(目录)不能与其父目录(即父文件夹)中的文件名重名，即同一个目录下不能有同名的(区分大小写)。

3．命令参数：

-m, --mode=模式，设定权限<模式> (类似 chmod)，而不是 rwxrwxrwx 减 umask

-p, --parents 可以是一个路径名称。此时若路径中的某些目录尚不存在,加上此选项后,系统将自动建立好那些尚不存在的目录,即一次可以建立多个目录;

-v, --verbose 每次创建新目录都显示信息

--help 显示此帮助信息并退出

--version 输出版本信息并退出

4．命令实例：

实例1：创建一个空目录

命令：

mkdir test1

输出：

[root@localhost soft]# cd test

[root@localhost test]# mkdir test1

[root@localhost test]# ll

总计 4drwxr-xr-x 2 root root 4096 10-25 17:42 test1

[root@localhost test]#

实例2：递归创建多个目录

命令：

mkdir -p test2/test22

输出：

[root@localhost test]# mkdir -p test2/test22

[root@localhost test]# ll

总计 8drwxr-xr-x 2 root root 4096 10-25 17:42 test1

drwxr-xr-x 3 root root 4096 10-25 17:44 test2

[root@localhost test]# cd test2/

[root@localhost test2]# ll

总计 4drwxr-xr-x 2 root root 4096 10-25 17:44 test22

[root@localhost test2]#

实例3：创建权限为777的目录

命令：

mkdir -m 777 test3

输出：

[root@localhost test]# mkdir -m 777 test3

[root@localhost test]# ll

总计 12drwxr-xr-x 2 root root 4096 10-25 17:42 test1

drwxr-xr-x 3 root root 4096 10-25 17:44 test2

drwxrwxrwx 2 root root 4096 10-25 17:46 test3

[root@localhost test]#

说明：

test3 的权限为rwxrwxrwx

实例4：创建新目录都显示信息

命令：

mkdir -v test4

输出：

[root@localhost test]# mkdir -v test4

mkdir: 已创建目录 “test4”

[root@localhost test]# mkdir -vp test5/test5-1

mkdir: 已创建目录 “test5”

mkdir: 已创建目录 “test5/test5-1”

[root@localhost test]#

实例五：一个命令创建项目的目录结构

参考：http://www.ibm.com/developerworks/cn/aix/library/au-badunixhabits.html

命令：

mkdir -vp scf/{lib/,bin/,doc/{info,product},logs/{info,product},service/deploy/{info,product}}

输出：

[root@localhost test]# mkdir -vp scf/{lib/,bin/,doc/{info,product},logs/{info,product},service/deploy/{info,product}}

mkdir: 已创建目录 “scf”

mkdir: 已创建目录 “scf/lib”

mkdir: 已创建目录 “scf/bin”

mkdir: 已创建目录 “scf/doc”

mkdir: 已创建目录 “scf/doc/info”

mkdir: 已创建目录 “scf/doc/product”

mkdir: 已创建目录 “scf/logs”

mkdir: 已创建目录 “scf/logs/info”

mkdir: 已创建目录 “scf/logs/product”

mkdir: 已创建目录 “scf/service”

mkdir: 已创建目录 “scf/service/deploy”

mkdir: 已创建目录 “scf/service/deploy/info”

mkdir: 已创建目录 “scf/service/deploy/product”

[root@localhost test]# tree scf/

scf/

|-- bin

|-- doc

| |-- info

| `-- product

|-- lib

|-- logs

| |-- info

| `-- product

`-- service

`-- deploy

|-- info

`-- product

1. irectories, 0 files

[root@localhost test]#

* + 1. ls
    2. more
    3. mv
    4. chmod
    5. **tar**
    6. **find**

用于搜索文件。

* + - 1. 格式：find [path] [options] [tests] [actions]

find /usr/lib -name libgcc\_s.so.1 -print

* + - 1. Options:
         1. –depth 在查看目录本身之前，先搜索目录的内容
         2. –follow 跟随符号链接
         3. –maxdepths N 最多搜索N层目录
         4. –mount/-xdev 不搜索其他文件系统中的目录
      2. Tests：（可以有多个）
         1. –atime N 文件在N天之前被最后访问过
         2. –mtime N 文件在N天之前被最后修改过
         3. –name pattern 文件匹配提供的模式（用引号括起）
         4. –newer otherfile 文件要比otherfile新
         5. –type c 文件的类型为c，最常见的是d（目录），f（普通文件
         6. –user name 文件的拥有者为name

可以用圆括号来强制测试和操作符的优先级，需要用到转义字符\

find . \(-name "\_\*" -or -newer while2\) -type f -print

* + - 1. Action
         1. –exec command 执行一条指令。这个动作必须用\;字符对来结束
         2. –ok command 同上，但在执行命令，会要求用户进行确认。
         3. –print 打印文件名
         4. –ls 对当前文件使用命令ls-dils
    1. **Grep**

在文件中搜索字符串

* + - 1. 格式：grep [options] PATTERN [files]
      2. 选项：
         1. –c 输出匹配行的数目，而不是输出匹配的行
         2. –E 启用扩展表达式
         3. –h 取消每个输出行的普通前缀，即匹配查询模式的文件名
         4. –i 忽略大小写
         5. –l 只列出包含匹配行的文件名，而不输出真正的匹配行
         6. –v 对匹配模式取反
      3. 正则表达式
         1. ^指向一行的开头
         2. $指向一行的结尾
         3. .任意单个字符
         4. []包含
      4. 特殊的匹配模式
         1. [:alnum:]字母与数字
         2. [:alpha:]字母
         3. [:ascii:]ASCII字符
         4. [:blank:]空格或制表符
         5. [:digit:]数字
         6. [:lower:]小写字母
         7. [:space:]空白字符（划定单词的结尾）
      5. 指定用于扩展表达式-E选项，则可使用以下字符（**在使用前需要加上\字符**）
         1. ? 匹配是可选的，最多一次
         2. \*匹配0次或多次
         3. +匹配一次或多次
         4. {n}匹配n次
         5. {n,}匹配n次或n次以上
         6. {n,m}匹配n次到m次

grep Th.[[:space:]] words2.txt

grep –E [a-z]\{10\} words2.txt

* + 1. **ar**
    2. dd
    3. cat
  1. 进程管理
     1. ps
     2. pstree
     3. kill
  2. 编辑器
     1. vi
     2. emacs
  3. 编译、链接、调试
     1. gcc
        1. 四个阶段：预处理、编译、汇编、链接 (E、S、c、o)
        2. 使用方式：gcc [options] [filename]
        3. 选项：
           1. -E: 只对源程序进行预处理(调用cpp预处理器)
           2. -S: 只对源程序进行预处理、编译
           3. -c: 执行预处理、编译、汇编而不链接
           4. -o output\_file: 指定输出文件名（不指定为a.out）
           5. -g: 产生调试工具必需的符号信息
           6. -O/On: 在程序编译、链接过程中进行优化
           7. -Wall: 显示所有的警告信息
           8. -Idir: 指定额外的头文件搜索路径
           9. -Ldir: 指定额外的库文件搜索路径
           10. -lname: 链接时搜索指定的库文件
           11. -DMACRO[=DEFN]: 定义MACRO宏
  4. 库管理工具
     1. ar
  5. man和info查看帮助

考核方法：

用程序实现某个命令的功能

考察命令的应用场景

1. Shell编程
   1. Shell是什么？
      1. 一种命令解释器和编程环境
      2. 用户和操作系统之间的接口
      3. 作为核外程序而存在
   2. Shell编程的基本要素
      1. 如何执行Shell脚本，有何差异？
         1. 方法一：

$ sh script\_file

* + - 1. 方法二：

$ chmod +x script\_file（chown，chgrp命令可选，将文件设置为可执行）

$ ./script\_file

* + - 1. 方法三：

$ source script\_file或者

$ .script\_file

* + 1. Shell变量（几种变量，如何**使用**）
       1. 用户变量
          1. 用户在shell脚本里定义的变量。在shell中，使用变量之前并不需要事先为它们做出声明。通过使用它们来创建它们。在默认情况下，所有变量都被看做字符串并以字符串来存储。
          2. 赋值和使用

var=value（**在为变量赋值时，只需要使用变量名。如果字符串里包含空格，则必须用双引号括起来**。）

echo $var（通过在变量前加一个$来访问它的内容）

* + - * 1. read命令，将用户的输入赋值给一个变量
        2. 引号的用法

双引号：脚本文件中的参数以空白字符分隔（例如一个空格，一个制表符等），必须给参数加上引号。在双引号中，$，\，`这些字符的特殊含义还是存在的。

单引号：**忽略所有的特殊字符**

* + - 1. 环境变量
         1. Shell环境提供的变量。通常使用大写字母做名字
         2. $HOME 当前用户的登录目录
         3. $PATH 以冒号分隔的用来搜索命令的目录清单
         4. $PS1 命令行提示符，通常是“$”
         5. $PS2 辅助提示符，用来提示后续输入，通常是“>”字符
         6. $IFS输入区分隔符。当shell读取输入数据时会把一组字符看成是单词之间的分隔符，通常是空格、制表符、换行符等
      2. 参数变量和内部变量
         1. 参数变量和内部变量为调用脚本程序时如果带有参数，对应的参数和**额外**产生的一些变量。
         2. $# 传递到脚本程序的参数个数
         3. $0 脚本程序的名字
         4. $1，$2 脚本程序的参数
         5. $\* 一个全体参数组成的清单，它是一个独立的变量，各个参数之间用环境变量IFS中的第一个字符分隔开
         6. $@ “$\*”的一种变体，它不使用IFS环境变量

2、3使用的时候结合1

* + 1. Shell的几种程序结构
       1. 条件测试（关注字符串的比较、文件比较）
          1. test命令，格式为test expression或[ expression ]（[和检查条件之间必须有空格）
          2. **字符串比较**

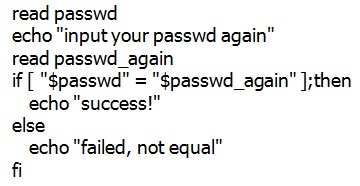
str1=str2，两个字符串相同则结果为真

str1!=str2，两个字符串不相同则结果为真

-z str，字符串为空则结果为真

-n str，字符串不为空则结果为真

例子：



* + - * 1. 算数比较

expr1 -eq expr2，两个表达式相等则结果为真

expr1 -ne expr2，两个表达式不等则结果为真

expr1 -gt expr2，expr1大于expr2则结果为真

expr1 –ge expr2，expr1大于或等于expr2则结果为真

expr1 –lt expr2，expr1 小于 expr2 则结果为真

expr1 –le expr2，expr1 小于或等于 expr2 则结果为真

* + - * 1. **文件比较**

–d file，文件是个目录则结果为真

–e file，文件存在则结果为真

–f file，文件是一个普通文件则结果为真

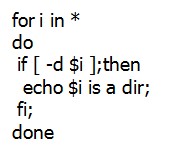
–s file，文件长度不为0则结果为真

–r file，文件可读则结果为真

–w file，文件可写则结果为真

–x file，文件可执行则结果为真

例子：



* + - * 1. 逻辑操作

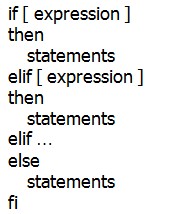
!expr，逻辑表达式求反

expr1 –a expr2，and

expr1 -o expr2，or

* + - 1. 判断结构（if，case）
         1. If语句

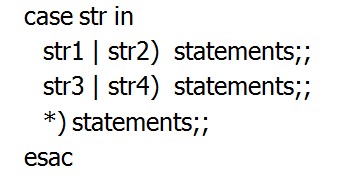
形式：



同一行上有多个命令用“;”分隔

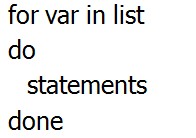
* + - * 1. Case语句

形式：（**注意两个分号**）



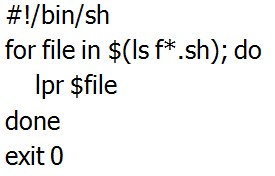
* + - 1. 循环语句
         1. For语句

形式:



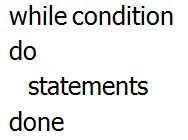
适用于对一系列字符串进行循环处理（list可以是一系列字符串的排列）

例子：

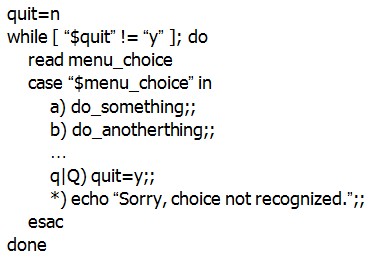


* + - * 1. While语句

形式：

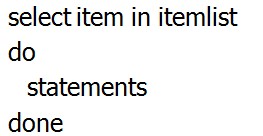


例子：



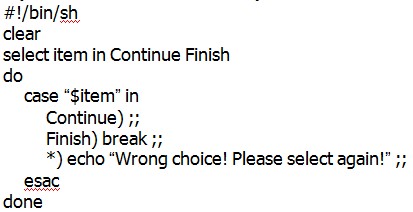
* + - * 1. Until语句（不推荐使用，与while相反，循环执行直到条件为真）
        2. Select语句

形式：

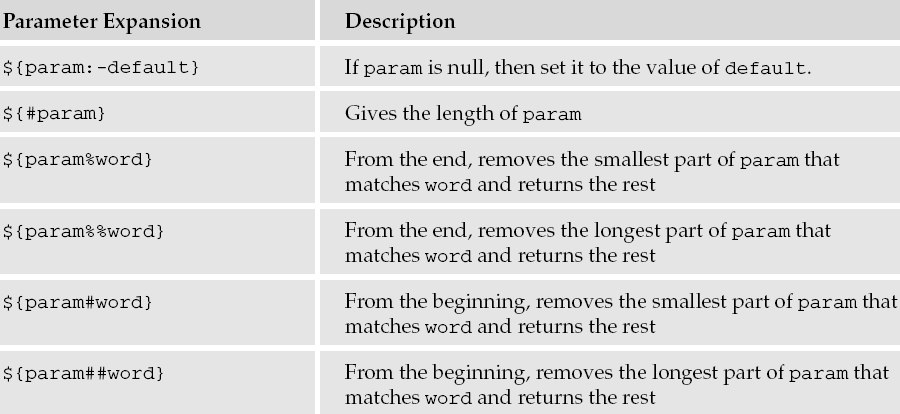


作用：生成**菜单列表**

例子：



* + 1. 参数扩展
       1. $i\_tmp转换为${i}\_tmp
       2. 算数扩展：x = $(($x+1))，区别于$(command)捕捉命令输出
       3. 更复杂的形式



* + - 1. 练习：

bar=/usr/local/etc/local/network;

echo ${bar%local\*}

bar=/usr/local/etc/local/network;

echo ${bar%%local\*}

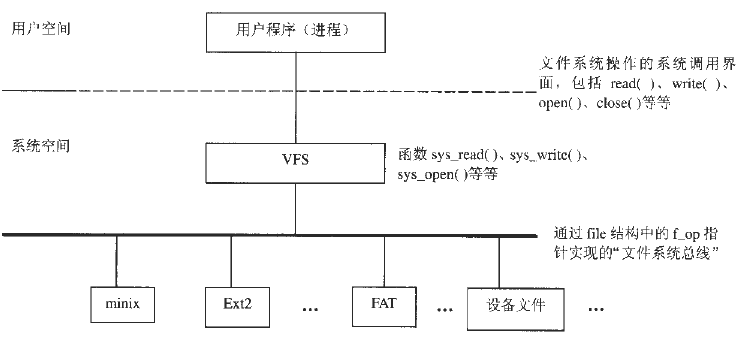
foo=/usr/bin/x11/startX;echo ${foo#\*/}

foo=/usr/bin/x11/startX;echo ${foo##\*/}

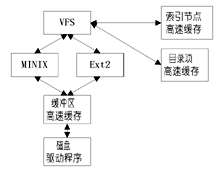
考核方式：

简单Shell编程

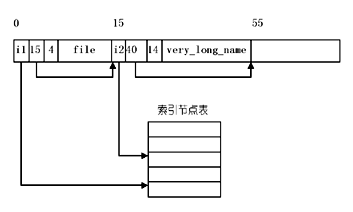
1. 文件系统
   1. 文件系统基础
      1. VFS模型（Visual File system Switch）
         1. VFS的作用就是采用标准的Unix系统调用读写位于不同物理介质上的不同**文件系统**。VFS是一个可以让open()、read()、write()等系统调用不用关心底层的**存储介质和文件系统类型**就可以工作的**粘合层**。屏蔽底层系统的具体形式。
         2. 图片：



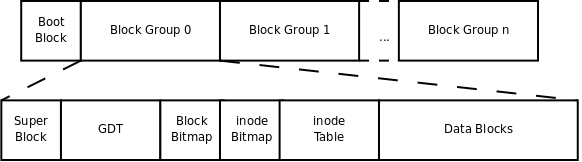
* + - 1. 组成：



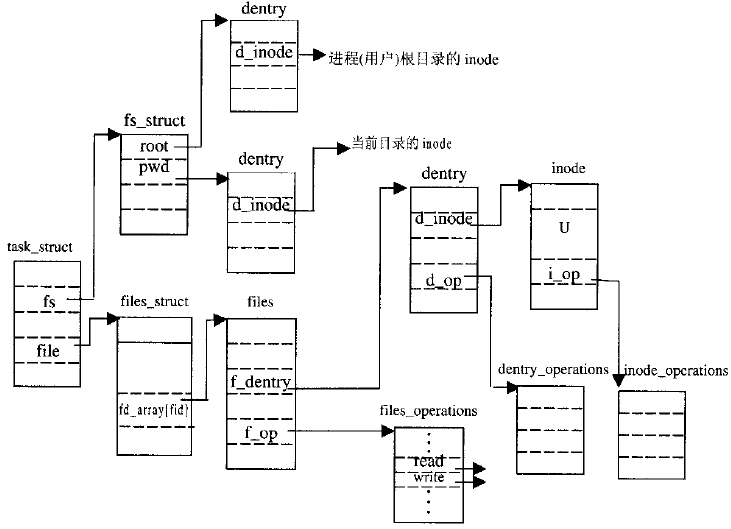
* + - * 1. super block：一个已安装的文件系统的控制信息
        2. i-node object：**索引节点**，存储了文件的相关信息，代表一个物理节点
        3. file object：已打开文件在**内存**中的表示，主要用于建立进程和磁盘上的文件对应关系
        4. dentry object：**目录项**对象，一个路径的各个组成部分，不管是目录还是文件，都是一个dentry对象（对应到目录包含的inode上）目录项内容包括索引节点编号，目录项名称长度以及名称



* + 1. EXT2文件系统中数据的存储
       1. 结构（最小存储单位为块，大小在格式化时确定）下图为一个磁盘分区格式化成ext2文件系统后的存储布局



* + - 1. 超级块：描述整个分区的文件信息，如块大小，系统版本号等。（所有块组中都相同）
      2. GDT（Group Descriptor Table块组描述符表）：整个分区分成多少个块组，就有多少个块组描述符。每个块组描述符存储一个块组的描述信息，如例如inode表起始位置，数据区起始位置（所有块组中都相同）
      3. Block bitmap（块位图）：描述block group中每个block是否被使用。占据一个块的大小。1代表占用，0代表未占用。
      4. Inode bitmap：同块位图。
      5. Inode table：块组中所有inode构成的inode表，每个inode对应一个物理文件。（与inode bitmap混合使用，大小在格式化的时候就确定了）
      6. Data blocks：
         1. 对于常规文件，文件的数据存储在数据块中
         2. 目录文件：该目录下所有的**文件名和目录名**存储在数据块中（名称+inode节点号）
         3. 符号链接：如果目标路径名较短则直接保存在inode中以便更快地查找，如果目标路径名较长则分配一个数据块来保存
         4. 设备文件，fifo，socket文件等没有数据块，设备文件的major ，minor number保存在inode中
    1. 进程和文件的关系
       1. 每个进程通过“打开文件”（open()）与具体的文件建立起连接，或者说建立起一个读写的上下文。这种连接以一个**file数据结构**为代表，结构中有个file\_operations结构指针f\_op
       2. 将file结构中的指针f\_op设置成指向某个具体的file\_operations结构变量，就指定了这个文件所属的文件系统，并与具体文件系统所提供的一组函数挂上了钩。
       3. 结构图：



* + 1. 7种类型文件的差异
       1. 普通文件-：包含了某种形式的数据。
       2. 目录f：包含了其他文件的名字和inode号
       3. 字符设备文件c：用于系统中的无缓冲写的设备文件
       4. 块设备文件b：用于系统中的缓冲写设备文件，如磁盘设备。系统中的所有设备要么是字符文件，要么是块设备文件。
       5. 套接字文件s：用于网络间通信
       6. 管道p：用于进程间通信
       7. 符号链接文件l：这种文件指向另一个文件

在上篇文章（linux的文件属性模式）中，我说道：Linux的文件，有个16位的字来表示文件的类型和属性信息，其中4位表示文件的类型信息，剩下的12位表示文件的模式。其中12位在那篇文章中已经解释清楚了，本文就来解释剩下4位是如何编排的。

linux下面文件类型一共有7种，分别是：

1. 普通文件，regular file，这是最常见的文件类型，这种文件包含了某种形式的数据。至于这种数据是文本还是二进制数据对于内核而言并无区别。

2. 目录文件，deirctory file，这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。

3. 字符特殊文件，character special file，这种文件用于系统中的无缓冲写的设备文件。

4. 块特殊文件，block special file，这种文件用于系统中的缓冲写设备文件，最常见典型有磁盘设备。系统中的所有设备要么是字符特殊文件，要么是块特殊文件。

5. FIFO，这种文件用于进程间通信，有时也将其称为命名管道。

6. 套接口，socket，这种文件用于进程间网络通信，亦可用于一台宿主主机上的进程之间的非网络通信。

7. 符号连接，symbolic link，这种文件指向另一个文件。

既然用4位来表示，那么一共就可以表示16种文件类型，去除掉0000，那么还可以表示15种文件，linux一共用来7种，那么这7种的位图分布是什么样子的呢？先来揭晓答案吧，然后再给出得到答案的方法，位图分布如下（注意后面12位是权限位图）：

fifo : 0001 000000000000

character : 0010 000000000000

directory : 0100 000000000000

block : 0110 000000000000

regular : 1000 000000000000

symbolic link : 1010 000000000000

socket : 1100 000000000000

* + 1. 硬链接和符号链接
       1. 硬链接
          1. 不同的文件名对应同一个inode
          2. 不能跨越文件系统
          3. 对应系统调用link
       2. 符号链接
          1. 存储被链接文件的**文件名(而不是inode)**实现链接
          2. 可跨越文件系统
          3. 对应系统调用symlink
  1. 文件操作与管理
     1. 常用系统调用
        1. 基本操作：open、close、read、write、lseek
           1. Open

创建一个新的文件描述符（失败返回-1）

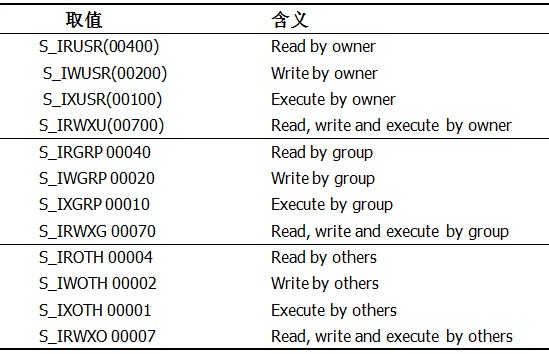
int open(const char \*pathname, int flags)

int open(const char \*pathname, int flags, mode\_t mode)

flags：

O\_RDONLY，O\_WRONLY，O\_RDWR，O\_APPEND，O\_TRUNC，O\_CREAT（按mode中给出的访问形式创建文件），O\_EXCL（与O\_CREAT合用）

**Mode：**



* + - * 1. Close

int close(int fd)

Return: 0 if success; -1 if failure

* + - * 1. Read

ssize\_t read(int fd, void \*buf, size\_t count)

返回值: 读到的字节数，若已到文件尾为0，若出错为-1

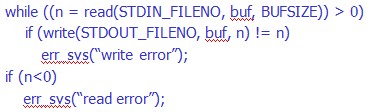
从fd中读出count个字符到buf中，返回实际读取的字符数

* + - * 1. Write

ssize\_t write(int fd, const void \*buf, size\_t count)

返回值: 若成功为已写的字节数，若出错为-1

例子：



* + - * 1. Lseek

Lseek系统调用对文件描述符fildes的读写指针进行设置。也就是说，可以用来设置文件读写位置。

off\_t lseek(int fildes, off\_t offset, int whence)

lseek返回从文件头到文件指针被设置处的字节偏移值，失败时返回-1。lseek仅将当前的文件位移量记录在内核中，**并不引起任何I/O操作**

offset用来指定位置，而whence定义offset的用法：

SEEK\_SET：offset是个绝对位置

SEEK\_CUR：offset是一个相对于当前位置的相对位置

SEEK\_END：offset是相对于文件尾的的相对位置

* + - 1. dup、dup2（13章有详细描述）
         1. int dup(int oldfd)

复制文件描述符，使我们能够通过两个或者多个不同的文件描述符来访问同一个文件。

int dup2(int oldfd, int newfd)

Return: the new file descriptor if success; -1 if failure

* + - 1. fcntl（改变文件的属性，如阻塞/非阻塞等；文件中的区域锁）
         1. int fcntl(int fd, int cmd)

int fcntl(int fd, int cmd, long arg)

int fcntl(int fd, int cmd, struct flock \*lock)

* + - * 1. 对打开的文件描述符进行各种操作，包括对他们进行复制，获取和设置文件描述符标志、获取和设置文件状态标志，以及管理建议性文件锁等。
        2. 对不同操作的选择是通过选取参数命令cmd不同的值来实现的。根据所选的命令不同，可能还需要第三个参数。

F\_DUPFD，复制描述符

（dup(fd) <-> fcntl(fd, F\_DUPFD, 0)，

dup2(fd, fd2) <-> close(fd2); fcntl(fd, F\_DUPFD, fd2)）

F\_GETFD/F\_SETFD，访问/设置文件描述符标志close-on-exec flag，决定是否在成功调用了某个exec之后关闭该文件描述符

F\_GETFL/F\_SETFL，访问/设置文件状态标志和文件访问标志

* + 1. 与文件属性相关的操作
       1. stat/lstat/fstat函数和struct stat结构
          1. int stat(const char \*file\_name, struct stat \*buf)

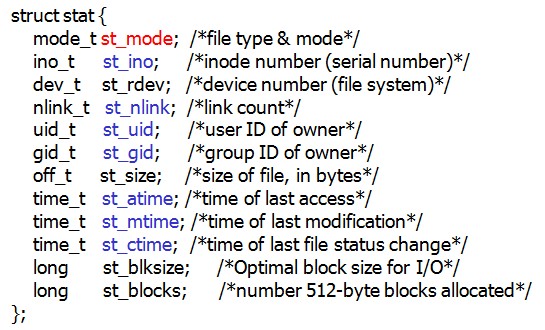
int fstat(int filedes, struct stat \*buf)

int lstat(const char \*file\_name, struct stat \*buf)

* + - * 1. fstat系统调用返回与打开的文件描述符相关的文件的状态信息，该信息将会写入buf中

stat和lstat返回的是通过文件名查到的状态信息。它们产生相同的结果，担当文件是一个符号链接时，lstat返回的是该符号链接本身的信息，而stat返回的是该链接指向的文件的信息。

* + - * 1. struct stat结构



* + - * 1. SUID(s): run with effective user ID. euid是进程访问资源的依据，SUID的功能就是改变euid

**SGID(g)**: Files created in directory inherit the same group ID as the directory.

Sticky bit(t): Only the owner of the file and directory may delete them. **仅对目录有效**

* + - 1. umask，chmod，chown等
         1. mode\_t umask(mode\_t mask)

为进程设置文件存取权限屏蔽字，并返回以前的

* + - * 1. int chmod(const char \*path, mode\_t mode)

改变文件或目录的访问权限，mode参考上文

* + - * 1. int chown(const char \*path, uid\_t owner, gid\_t group)
      1. link/unlink/symlink/readlink
         1. int link(const char \*oldpath, const char \*newpath)

创建一个指向已有文件的新链接

* + - * 1. int unlink(const char \*pathname)

删除一个文件的目录项并减少它的链接数（可能会删除文件本身）

* + - * 1. int symlink(const char \*oldpath, const char \*newpath)

创建一个符号链接，不会增加文件的链接数，所以不会像硬链接一样放置文件被删除

* + - * 1. int readlink(const char \*path, char \*buf, size\_t bufsiz)

读取一个符号链接的值

* + 1. 与目录相关的操作
       1. mkdir/rmdir/opendir/closedir/readdir/telldir/seekdir
          1. int mkdir(const char \*pathname, mode\_t mode)

创建一个空目录

* + - * 1. int rmdir(const char \*pathname)

删除一个空目录

* + - * 1. DIR \*opendir(const char \*name)

打开一个目录并创建一个目录流

* + - * 1. int closedir(DIR \*dir)

关闭目录流并释放相关资源

* + - * 1. struct dirent \*readdir(DIR \*dir)

返回一个指针，指向结构中下一个目录项的有关资料

* + - * 1. off\_t telldir(DIR \*dir)

返回值记录着一个目录流里的当前位置

* + - * 1. void seekdir(DIR \*dir, off\_t offset)

设置目录流dir的目录流指针

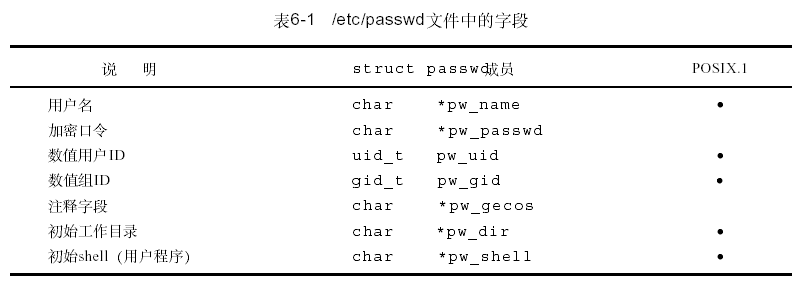
* + 1. 其他
       1. Linux错误处理风格
          1. 许多系统调用和函数都会因为各种各样的原因失败，它们会在失败时设置外部变量errno的值来指明失败的原因。有两个很有用的函数可以用来报告错误。
          2. Char \*strerror(int errnum)

把错误码映射成字符串

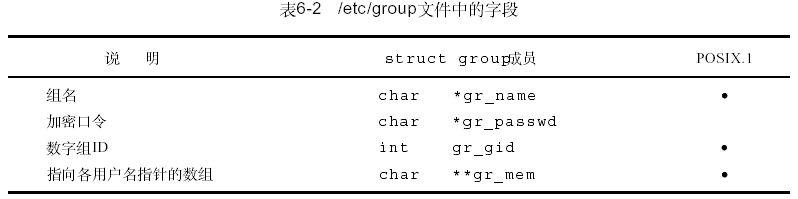
* + - * 1. Void perror(char \*s)

同样把当前错误映射到一个字符串，并输出到标准输出，在字符串之前加上s

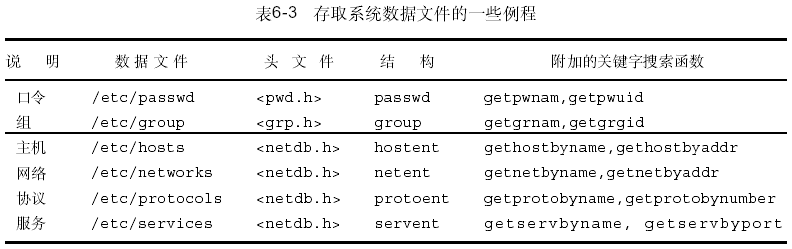
* + - 1. 文件系统数据（/etc/passwd，/etc/group，/etc/service等，详细看Linux系统基础）
         1. Passwd



* + - * 1. Group



* + - * 1. Service



* + - 1. passwd命令

考核要点：

根究上述API，充分了解Linux文件的各种属性以及读写处理

上述属性包括**读写/权限/大小/时间/连接数**等

读写：

权限：

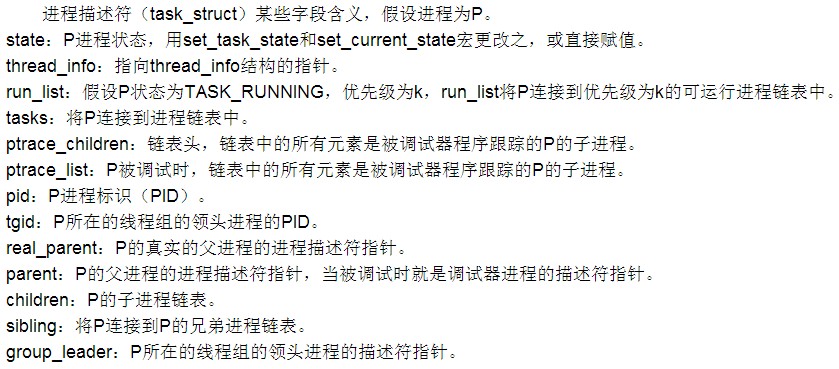
大小：st\_size包括普通文件、目录、符号链接，而hole文件不占据磁盘空间

时间：st\_atime最后访问时间，st\_mtime最后修改时间，st\_ctime表示inode状态最后修改时间

连接数：

目录操作

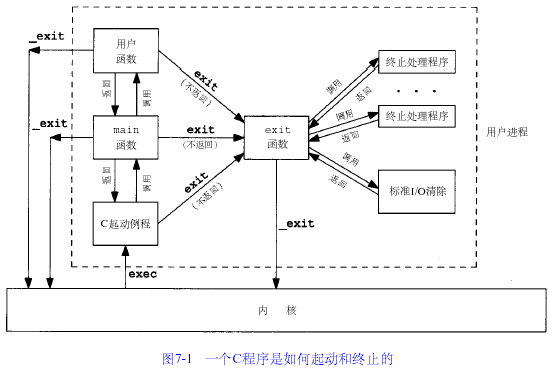
1. 进程管理
   1. 进程知识
      1. 进程的概念；进程描述符的内容
         1. 进程：An address space with one or more threads executing within that address space, and the required system resources for those threads。一个其中运行着一个或多个线程的**地址空间**和这些线程所需要**系统资源**。
         2. 进程描述符的内容：struct task\_struct



* + 1. 基于内核对进程的控制理解进程的创建过程；init进程对于其他应用进程的意义
       1. 进程的创建过程：一般而言，进程都是由另一个我们称之为父进程的进程启动的，被父进程启动的进程叫做子进程。
       2. Init进程

Linux系统启动时，它将运行一个名为init的进程，该进程是系统运行的第一个进程，它的进程号为1。可以把init进程看做操作系统的进程管理器，它是所有其他进程的祖先进程。其他系统进程要么是由init进程启动的，要么是由被init进程启动的其他进程启动的。

* + 1. 进程的终止
       1. 正常终止
          1. 从main返回
          2. 调用exit函数（库函数，执行一些清除处理，对于所有打开的流调用fclose函数，再进入内核）
          3. 调用\_exit函数（系统调用，立即进入内核）
       2. 异常终止
          1. 调用abort函数
          2. 由一个信号终止



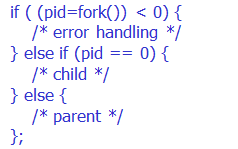
* 1. 进程基本系统调用
     1. fork，exec，wait，exit
        1. fork
           1. pid\_t fork(void)

返回值，在父进程中：子进程的pid

在子进程中：0

失败：-1

* + - * 1. 子进程是父进程的复制品。例如，子进程获得父进程数据空间、堆和栈的复制品。注意，这是子进程所拥有的拷贝。父、子进程并不共享这些存储空间部分。如果正文段是只读的，则父、子进程共享正文段。
        2. 这个系统调用复制当前进程，在进程表中创建一个新的表项，新表项的许多属性与当前进程是一样的，但有自己的数据空间、环境和文件描述符（拷贝）。
        3. 例子：



* + - * 1. 使用场景：

父进程复制自己，去执行不同的任务（网络服务器），不同代码段

进程需要执行不同的程序（之后跟随exec去执行其他的函数），不同程序

* + - 1. exec
      2. wait
         1. pid\_t wait(int \*status)
         2. 当一个进程或异常终止时，
      3. exit
    1. zombie进程

希望进程通知父进程它是如何停止的。

* + - 1. 父进程已经终止的所有进程（孤儿进程），他们的父进程都改为init进程
      2. 子进程在父进程之前终止：
         1. SIGHLD信号
         2. 子进程结束后，父进程通过wait/waitpid得到相关信息，进行处理
         3. 一个已经终止、但是其父进程尚未对其进行善后处理（获取终止子进程的相关信息、释放它仍占用的资源）的进程被称为僵尸进程。
      3. 解决方案：
         1. Wait
         2. SIGCHLD
         3. 忽略SIGCHLD
  1. 父子进程间的共享数据
     1. 有哪些类型的共享数据
        1. 实际用户ID，实际组ID，有效用户ID，有效组ID
        2. 添加组ID
        3. 进程组ID
        4. 对话期ID
        5. 控制终端
        6. 设置-用户-ID标识和设置-组-ID标识
        7. 当前工作目录
        8. 根目录
        9. 文件创建方式屏蔽字
        10. 信号屏蔽和排列
        11. 对任一打开的文件描述符的close-on-exec标识
        12. 环境
        13. 连接的共享存储段
        14. 内存映射
        15. 资源限制
     2. race condition的处理方法
  2. 信号的概念（软件中断，处理异步事件）
     1. 一些常用的信号（Ctrl-C：SIGINT；Ctrl-/：SIGQUIT；Ctrl-Z：SIGTSTP；SIGCHLD）
     2. 对信号的处理方式
        1. 忽略信号（不能忽略的信号：SIGKILL，SIGSTOP；一些硬件异常信号）
        2. 捕捉信号
        3. 执行系统默认动作
     3. 理解信号的异步执行
  3. 关于信号的系统调用
     1. signal，sigaction
     2. kill，raise；alarm，pause（kill函数和kill命令）
     3. 理解：进入信号阻塞、信号集（sigset\_t）的概念，信号掩码
     4. sigpromask，sigpending，sigsuspend，sigaction（用信号机制解决race condition）

1. 进程间通信
   1. IPC
      1. 信号
      2. pipe
      3. FIFO
      4. System V IPC（信号集、消息队列、共享内存）
      5. socket
   2. 管道
      1. 系统调用pipe（先pipe后fork）；管道的读写（同步）
      2. **Question**：用pipe实现popen，pclose；用pipe实现shell中的重定向和管道
   3. 命名管道（FIFO）
      1. mkfifo（mknod）命令和mkfifo（mknod）函数
      2. FIFO的打开和读写（同步）
      3. 用FIFO实现C/S结构的应用程序
   4. System V IPC
   5. **核心概念理解**：信号量集（semaphore set）、共享内存、消息队列
2. Socket通信
   1. 网络基础
      1. TCP/IP复习
   2. socket基础
      1. connection五元组<协议、本地地址、本地端口、远程地址、远程端口>
      2. socket三元组<协议、本地地址、本地端口>
      3. socket/bind/connect
   3. socket程序设计模型
      1. 基于连接的通信、无连接通信
      2. 了解：两种模型的比较。无连接通信的应用范围？要考虑哪些因素？
   4. 多客户的问题
      1. 多进程、多线程
      2. 五种I/O模式：阻塞模式、非阻塞模式、多路复用I/O、信号驱动I/O、异步I/O模式
   5. 并发服务器模型