Linux / UNIX编程

参考书籍：

《UNIX环境高级编程》

《Linux/UNIX系统编程手册》

# 1、编译与调试

## GCC程序编译

GCC编译器能将C、C++语言源程序、汇编程序编译、链接成可执行文件。在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。

**简介**

使用GCC编译程序时,编译过程可以被细分为四个阶段:

* 预处理(Pre-Processing)
* 编译(Compiling)
* 汇编(Assembling)
* 链接(Linking)

例：预处理做些什么？

答：主要是C语言编译器对各种预处理命令进行处理，包括头文件的包含、宏定义的扩展、条件编译的选择等。

编译时需要头文件；

链接时需要库文件；

**文件类型**

Gcc通过后缀来区别输入文件的类别：

* .c为后缀的文件: C语言源代码文件
* .a为后缀的文件: 是由目标文件构成的库文件
* .C，.cc或.cxx 为后缀的文件: 是C++源代码文件
* .h为后缀的文件: 头文件
* .i 为后缀的文件: 是已经预处理过的C源代码文件
* .ii为后缀的文件: 是已经预处理过的C++源代码文件
* .o为后缀的文件: 是编译后的目标文件
* .s为后缀的文件: 是汇编语言源代码文件
* .S为后缀的文件: 是经过预编译的汇编语言源代码文件。

**基本用法**

gcc最基本的用法是∶

gcc [options] [filenames]

options：编译器所需要的编译选项

filenames: 要编译的文件名。

**编译选项**

gcc编译器的编译选项大约有100多个，其中多数我们根本就用不到，这里只介绍其中最基本、最常用的参数。

**-o** **output\_filename**：确定可执行文件的名称为output\_filename。如果不给出这个选项，gcc就给出预设的可执行文件a.out。

**-c**：只编译，不连接成为可执行文件，编译器只是由输入的.c等源代码文件生成.o为后缀的目标文件。

-g：产生调试工具(GNU的gdb)所必要的符号信息，要想对编译出的程序进行调试，就必须加入这个选项。

-O：对程序进行优化编译、链接，采用这个选项，整个源代码会在编译、连接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、连接的速度就相应地要慢一些。

-O2：比-O更好的优化编译、连接，当然整个编译、连接过程会更慢

示例：

#include <stdio.h>

int main(void)

{

double counter;

double result;

double temp;

for (counter = 0; counter < 2000.0 \* 2000.0 \* 2000.0 / 20.0 + 2020;

counter += (5 - 1) / 4) {

temp = counter / 1979;

result = counter;

}

printf(Result is %lf\\n, result);

return 0;

}

1. gcc optimize.c -o optimize

time ./optimize

2. gcc –O optimize.c -o optimize

time ./optimize

对比两次执行的输出结果不难看出，程序的性能的确得到了很大幅度的改善。

**-Idirname**: 将dirname所指出的目录加入到程序头文件目录列表中。

C程序中的头文件包含两种情况∶

#include <A.h>

#include “B.h”

对于<>，预处理程序cpp在系统预设的头文件目录(如/usr/include)中搜寻相应的文件；而对于””，cpp在当前目录中搜寻头文件。这个选项的作用是告诉cpp，如果在当前目录中没有找到需要的文件，就到指定的dirname目录中去寻找。

例：gcc foo.c -I /home/include -o foo

注：-I后面目录，中间有没有空格都可以。

**-Ldirname**：将dirname所指出的目录加入到库文件的目录列表中。在默认状态下，连接程序ld在系统的预设路径中(如/usr/lib)寻找所需要的库文件，这个选项告诉连接程序，首先到-L指定的目录中去寻找，然后再到系统预设路径中寻找。

-**lname**：在连接时，装载名字为“libname.a”的函数库，该函数库位于系统预设的目录或者由-L选项确定的目录下。例如，-lm表示连接名为“libm.a”的数学函数库。（小写的L）

例：gcc foo.c -L /home/lib -lfoo -o foo

-static：静态链接库文件

例：gcc –static hello.c -o hello

库有动态与静态两种，**动态通常用.so为后缀，静态用.a为后缀**。例如：libhello.so libhello.a。当使用静态库时，连接器找出程序所需的函数，然后将它们拷贝到可执行文件，一旦连接成功，静态程序库也就不再需要了。然而，对动态库而言，就不是这样，动态库会在执行程序内留下一个标记‘指明当程序执行时，首先必须载入这个库。由于动态库节省空间，linux下进行连接的缺省操作是首先连接动态库。

演示：静态链接与动态链接可执行文件大小比较

-Wall：生成所有警告信息

-w：不生成任何警告信息

-DMACRO： 定义MACRO 宏，等效于在程序中使用#define MACRO

## GDB程序调试

**简介**

GDB是GNU发布的一款功能强大的程序调试工具。GDB主要完成下面三个方面的功能：

1、启动被调试程序。

2、让被调试的程序在指定的位置停住。

3、当程序被停住时，可以检查程序状态（如变量值）。

**GDB快速进阶**

1.编译生成可执行文件：

gcc -g tst.c -o tst

2.启动GDB

gdb tst

3. 在main函数处设置断点

break main

4. 运行程序

run

5. 单步运行

next

6. 继续运行

continue

**启动GDB**

两种方式，如下：

1. gdb 调试程序名

例：gdb helloworld

2. gdb

file 调试程序名

**GDB命令（演示）**

list(l) 查看程序（方便打断点）

打断点的四种方式，如下：

break(b) 函数名 在某函数入口处添加断点

break(b) 行号 在指定行添加断点

break(b) 文件名:行号 在指定文件的指定行添加断点

break(b) 行号 if 条件 当条件为真时，指定行号处断点生效，例b 5 if i=10，当i等于10时第5行断点生效

info break 查看所有设置的断点

delete 断点编号 删除断点

run(r) 开始运行程序

next(n) 单步运行程序(不进入子函数)

step(s) 单步运行程序(进入子函数)

continue(c) 继续运行程序

print(p) 变量名 查看指定变量值

finish 运行程序,直到当前函数结束

watch 变量名 对指定变量进行监控

quit(q) 退出gdb

## Makefile工程管理

**GNU make**

Linux程序员必须学会使用GNU make来构建和管理自己的软件工程。GNU 的make能够使整个软件工程的编译、链接只需要一个命令就可以完成。

**Makefile**

make在执行时, 需要一个命名为Makefile的文件。Makefile文件描述了整个工程的编译，连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译；需要创建那些库文件以及如何创建这些库文件、如何最后产生我们想要得可执行文件。

**Makefile(例)**

hello: main.o func1.o func2.o

gcc main.o func1.o func2.o -o hello

main.o : main.c

gcc –c main.c

func1.o : func1.c

gcc –c func1.c

func2.o : func2.c

gcc –c func2.c

.PHONY : clean

clean :

rm –f hello main.o func1.o func2.o

### Makefile(术语)

规则：用于说明如何生成一个或多个目标文件，规则格式如下：

targets : prerequisites

command

目标 依赖 命令

main.o : main.c

gcc –c main.c

目标？依赖？命令？

**\*\*命令需要以【TAB】键开始\*\***

### 目标

在Makefile 中，规则的顺序是很重要的，因为，**Makefile中只应该有一个最终目标**，其它的目标都是被这个目标所连带出来的，所以一定要让make知道你的最终目标是什么。一般来说，定义在Makefile中的目标可能会有很多，但是**第一条规则中的目标将被确立为最终的目标**。

### 文件名

make命令默认在当前目录下寻找名字为makefile或者Makefile的工程文件，当名字不为这两者之一时，可以使用如下方法指定：

make –f 文件名

或者

make –file 文件名

### 伪目标

Makefile中把那些**没有任何依赖只有执行动作的目标**称为“伪目标”（phony targets）。

.PHONY : clean

clean :

rm –f hello main.o func1.o func2.o

“.PHONY” 将“clean”目标声明为伪目标。

### 变量

hello: main.o func1.o func2.o

gcc main.o func1.o func2.o -o hello

思考1：如果要为hello目标添加一个依赖，如：func3.o，该如

何修改？

答案1：

hello: main.o func1.o func2.o func3.o

gcc main.o func1.o func2.o func3.o -o hello

答案2：

obj=main.o func1.o func2.o func3.o

hello: $(obj)

gcc $(obj) -o hello

在makefile中，存在系统**默认**的自动化变量

$^：代表所有的依赖文件

$@：代表目标

$<：代表第一个依赖文件

例：

hello: main.o func1.o func2.o

gcc main.o func1.o func2.o -o hello

=》

hello: main.o func1.o func2.o

gcc $^ -o $@

**杂项**

Makefile中“#”字符后的内容被视作**注释**。

hello: hello.c

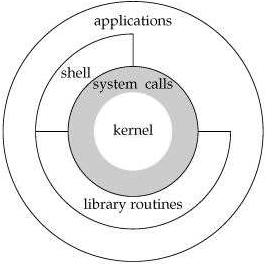
@gcc hello.c –o hello

@:取消回显 （演示）

# 2、UNIX基础知识

UNIX体系结构

操作系统、内核、系统调用、公用库函数、shell、应用程序。



Linux下一切皆文件。

在Linux系统中一切皆可以看成是文件，文件又可分为：普通文件、目录文件、链接文 件和设备文件。

**文件属性**：是指文件类型（普通文件or目录），文件大小、文件所有者、文件权限（其他用户能否访问该文件）以及文件最后的修改时间等。

stat和fstat函数包含所有文件属性的一个信息结构。

**文件名**：只有斜线（/）和空字符 这两个字符不能出现在文件名中。

斜线用来分割构成路径名的文件名，空字符用来终止一个路径名。

为了**可移植**性，POSIX.1推荐命名在一下字符内：字母(a-z,A-Z)，数字（0-9）、句点（.）、短横线（-）、和下划线（\_）。

创建新目录时系统会自动创建两个文件名：“**.**”和“**..**”，点指向当前目录，点点指向父目录。在最高层次的根目录中，点点与点相同。

以斜线（/）开头的路径名称为**绝对路径**名，否则称为**相对路径**名。

**文件描述符**（file descriptor）通常是一个 小的非负整数，内核用以标识一个特定进程正在访问的文件。 当内核打开一个现有文件或者创建一个新文件时，它都返回一个文件描述符。

按惯例，每当运行一个新程序时，所有的shell都为其打开**3个文件描述符**，即 标准输入（standard input）、标准输出（standard output）以及标准错误（standard error）。如果不做特殊处理，录入就像简单的命令ls，则这三个描述符都链接向终端。

blog：

<http://blog.csdn.net/cywosp/article/details/38965239>

**不带缓冲的I/O**

函数open、read、write、lseek以及close提供了不带缓冲的I/O。这些函数都使用文件描述符。

**标准I/O**

标准I/O函数为那些不带缓冲的I/O函数提供了一个带缓冲的接口。使用标准I/O函数无须担心如何选取最佳的缓冲区大小。

程序和进程

**程序**

程序（program）是一个存储在磁盘上某个目录中的可执行文件。内核使用exec函数（exec函数之一），将程序读入内存，并执行程序。

**进程和进程ID**

程序的执行实例被称为进程（process）。某些操作系统用任务（task）表示正在被执行的程序。

UNIX系统确保每一个进程都有一个唯一的数字标示符，称为进程ID（process ID）。进程ID总是一个非负整数。

**进程控制**

有3个用于进程控制的主要函数：fork、exec和waitpid。（exec函数有7种变体，但经常它们统称为exec函数。）

线程和进程

通常，一个进程只有一个控制线程（thread）——某一时刻执行的一组机器执行。对于某些问题，如果有多个控制线程分别作用于它的不同部分，那么解决起来就容易得多。另外，多个控制线程也可以充分利用多处理器系统的并行能力。

一个进程内的所有线程共享同一地址空间、文件描述符、栈以及与进程相关的属性。因为它们能访问同一存储区，所以各线程在访问共享数据时需要采取同步措施以避免不一致性。

与进程相同，线程也用ID标识。但是，线程ID只在它所属的进程内起作用。一个进程中的线程ID在另一个进程中没有意义。当在一进程中对某个特殊线程进行处理时，我们可以使用该线程的ID引用它。

**出错处理**

**出错恢复**

致命性和非致命性

**用户标识**

**用户ID**

getuid()

**组ID**

getgid()

**信号**

信号（signal）用于通知进程发生了某种情况。

进程处理信号有3中方式：

忽略信号、系统默认方式处理、捕捉该信号。

**时间值**

日历时间（时钟时间，UTC-格林尼治标准时间）、进程时间（CPU时间：系统CPU时间和用户CPU时间）

日历时间：格林尼治标准时间（UTC），可用于记录文件最近一次的修改时间。

进程时间：即CPU时间。包括时钟时间、用户CPU时间、系统CPU时间。

时钟时间：又称墙上时间，它是进程运行的时间总量，其值与系统中同时运行的进程数有关。

用户CPU时间：执行用户指令所用的时间量。

系统CPU时间：为该进程执行内核程序所经历的时间。例如，read或write调用在内核所花费的时间。

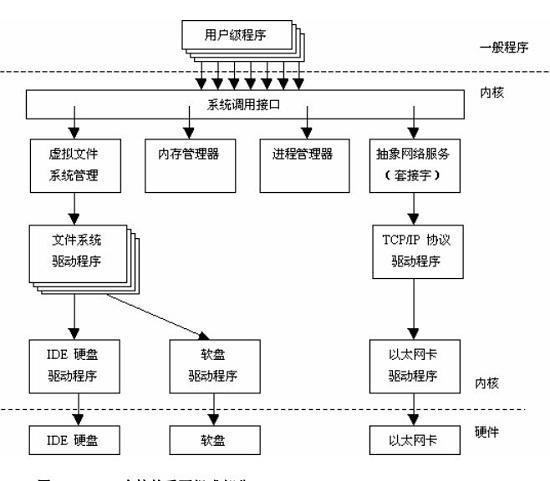
time命令

## linux内核

博客：<http://www.jianshu.com/p/c1ec431297f0>

内核是操作系统的核心，具有很多最基本功能，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

Linux 内核由如下几部分组成：内存管理、进程管理、设备驱动程序、文件系统和网络管理等。如图：



## UNIX标准化

### ISOC

### IEEE POSIX

POSIX（可移植操作系统接口Portable Operating System Interface的缩写）是一个最初由IEEE（电器及电子工程师协会），确切地说是其下属的可移植应用标准委员会（PASC）赞助下所开发的一系列标准。PASC标准的目的是提升应用程序在源码级别的可移植性。

符合POSIX.1标准的操作系统应向程序提供调用各项服务的API，POSIX.1文档对此作了规范。凡是共了上述API的操作系统都可被认为符合POSIX.1标准。

POSIX.1基于UNIX系统调用和C语言库函数，但无需与任何特殊实现相关。这意味着任何操作系统都可以实现该接口，而不一定要是UNIX操作系统。实际上，在不对底层操作系统大加改动的同时，一些厂商通过添加API已经使自己的专有操作系统符合了POSIX.1标准。

对原有POSIX.1标准的扩展也同样重要。POSIX.1b包含了一系列对基本POSIX标准的实时性扩展。POSIX.1c对POSIX线程作了定义，等等。

### Single UNIX Specification

## UNIX系统实现

前面讲述了3个由各自独立的组织所制定的标准：ISOC、IEEE POSIX以及Single UNIX Specification。

但是。标准只是接口的规范。这些标准是如何与现实世界相关联的呢？这些标准由厂商采用，然后转变成具体实现。

## 限制

两种类型的限制：

1. 编译时限制（eg: 短整型的最大值是多少?）
2. 运行时限制（eg: 文件名有多少个字符?）

编译时限制可在头文件中定义（静态的），程序在编译时可以包含这些头文件。但是，运行时（变动的）限制则要求进程调用一个函数获得限制值。

1. 编译时限制（头文件）。
2. 与文件或目录无关的运行时限制（sysconf函数）。
3. 与文件或目录有关的运行时限制（pathconf和fpathconf函数）

# 3、基本概念

## 内核

管理和分配计算机资源（即CPU、RAM和设备）的核心层软件。

### 内核的职责

内核执行的主要任务如下所示：

1. **进程调度**

计算机内均配备有一个或多个CPU（中央处理单元），以执行程序指令。与其他UNIX系统一样，Linux属于**抢占式多任务**操作系统。“多任务”意指多个进程（即运行中的程序）可同时驻留内存，且每个进程都能获得对CPU的使用权。“抢占”则是指一组规则。这组规则控制着哪些进程获得对CPU的使用，以及每个进程能使用多长时间，这两者都由内核进程调度程序（而非进程本身）决定的。

1. **内存管理**

物理内存（RAM）属于有限资源，内核必须以公平、高效地方式在进程间共享这一资源。与大多数现代操作系统一样，Linux也采用了虚拟内存管理机制，这项技术主要具有以下两方面的优势：

* 进程与进程之间、进程与内核之间彼此隔离，因此一个进程无法读取或修改内核或其他进程的内存内容。
* 只需将进程的一部分保持在内存中，这不但降低了每个进程对内存的需求量，而且还能在RAM中同时加载更多的进程。这也大幅提升了如下事件的发生概率，在任一时刻，CPU都有至少一个进程可以执行，从而使得对CPU资源的利用更加充分。

1. **提供了文件系统**

内核在磁盘之上提供有文件系统，允许对文件执行创建、获取、更新以及删除等操作。

1. **创建和终止进程**

内核可将新程序载入内存，为其提供运行所需的资源（比如，CPU、内存以及对文件的访问等）。这样一个运行中的程序我们称之为“进程”。一旦进程执行完毕，内核还要确保释放其占用资源，以供后续程序重新使用。

1. **对设备的访问**

计算机外接设备（鼠标、键盘、磁盘和磁带驱动器等）可实现计算机与外部世界的通信，这一通信机制包括输入、输出或是两者兼而有之。内核既为程序访问设备提供了简化版的标准接口，同时还要仲裁多个进程对每一个设备的访问。

1. **联网**

内核以用户进程的名义收发网络消息（数据包）。该任务包括将网络数据包路由至目标系统。

1. **提供系统调用应用编程接口（API）**

进程可利用内核入口点（也称为系统调用）请求内核去执行各种任务。

除了上述特性外，一般而言，诸如Linux之类的多用户操作系统会为每个用户营造一种抽象：虚拟私有计算机（virtual private computer）。这就是说，每个用户都可以登录进入系统，独立操作，而与其他用户大致无关。例如，每个用户都有输入自己的磁盘存储空间（主目录）。再者，用户能够运行程序，而每一程序都能从CPU资源中“分得一杯羹”，运转于自有的虚拟地址空间中。而且这些程序还能独立访问设备，并通过网络传递信息。内核负责解决（多进程）访问硬件资源时可能引发的冲突，用户和进程对此则往往一无所知。

### 内核态和用户态

现代处理器架构一般允许CPU至少在两种不同状态下运行，即：用户态和核心态（有时也称之为监管态supervisor mode）。执行硬件指令可使CPU在两种状态间来回切换。与之对应，可将虚拟内存区域划分（标记）为用户空间部分或内核空间部分。在用户态下运行时，CPU只能访问被标记为用户空间的内存，试图访问属于内核空间的内存会引发硬件异常。当运行于核心态时，CPU既能访问用户空间内存，也能访问内核空间内存。

仅当处理器在和心态运行时，才能执行某些特定的操作。这样的例子包括：执行宕机（halt）指令去关闭系统，访问内存管理硬件，以及设备I/O操作的初始化等。实现者们利用这一硬件设计，将操作系统置于内核空间。这确保了用户进程即不能访问内核指令和数据结构，也无法执行不利于系统的操作。

**参考博文：**

<https://www.cnblogs.com/zemliu/p/3695503.html>

<https://wenku.baidu.com/view/591e7faf7fd5360cbb1adb9b.html>

**内核态(Kernel Mode)与用户态(User Mode) 区别**

内核态：CPU可以访问内存所有数据, 包括外围设备, 例如硬盘, 网卡. CPU也可以将自己从一个程序切换到另一个程序。

用户态：只能受限的访问内存, 且不允许访问外围设备. 占用CPU的能力被剥夺, CPU资源可以被其他程序获取。

内核态和用户态是操作系统的两种运行级别，跟Intel CPU没有必然联系。

**为什么要有用户态和内核态？**

由于需要限制不同的程序之间的访问能力, 防止他们获取别的程序的内存数据, 或者获取外围设备的数据, 并发送到网络, CPU划分出两个权限等级 -- 用户态 和 内核态

**用户态与内核态的切换**

所有用户程序都是运行在用户态的, 但是有时候程序确实需要做一些内核态的事情, 例如从硬盘读取数据, 或者从键盘获取输入等. 而唯一可以做这些事情的就是操作系统, 所以此时程序就需要先操作系统请求以程序的名义来执行这些操作.

这时需要一个这样的机制: 用户态程序切换到内核态, 但是不能控制在内核态中执行的指令。

这种机制叫系统调用, 在CPU中的实现称之为陷阱指令(Trap Instruction)

他们的工作流程如下:

1、用户态程序将一些数据值放在寄存器中, 或者使用参数创建一个堆栈(stack frame), 以此表明需要操作系统提供的服务.

2、用户态程序执行陷阱指令

3、CPU切换到内核态, 并跳到位于内存指定位置的指令, 这些指令是操作系统的一部分, 他们具有内存保护, 不可被用户态程序访问

4、这些指令称之为陷阱(trap)或者系统调用处理器(system call handler). 他们会读取程序放入内存的数据参数, 并执行程序请求的服务

5、系统调用完成后, 操作系统会重置CPU为用户态并返回系统调用的结果

### 以进程及内核视角检视系统

见《Linux/UNIX系统编程手册（上）》P19，值得一看。

## shell

shell是一种具有特殊用途的程序，主要用于读取用户输入的命令，并执行相应的程序以响应命令。有时，人们也称之为命令解释器。

术语登录shell（login shell）是指用户刚登录系统时，由系统创建，用以运行shell的进程。

尽管某些操作系统将命令解释器集成于内核中，而对UNIX系统而言，shell只是一个用户进程。

设计shell的目的不仅仅是用于人机交互，对shell脚本（包含shell命令的文本文件）进行解释也是其用途之一。为实现这一目的，每款shell都内置许多通常与编程语言相关的功能，其中包括变量、循环和条件语句、I/O命令以及函数等。

## 用户和组

系统会对每个用户的身份做唯一标识，用户可隶属于多个组。

**参考博文：**

<https://yq.aliyun.com/articles/50327>

<http://blog.csdn.net/solaraceboy/article/details/78486048?locationNum=2&fps=1>

在/etc目录下，有三个文件：passwd、shadow、group，这三个配置文件用于系统帐号管理。/etc/passwd用于存放用户帐号信息，/etc/shadow用于存放每个用户加密的密码，/etc/group用于存放用户的组信息。

### /etc/passwd

系统密码文件/etc/passwd为每个用户都定义有一行记录。

[root@localhost etc]# cat passwd

root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

……

tcpdump:x:72:72::/:/sbin/nologin

czha:x:1000:1000:czha:/home/czha:/bin/bash

其中，每一行是由分号分隔的字串组成，它的格式如下：

username:password:uid:gid:gecos:homedir:shell

各域对应的中文说明如下:

用户名:密码:用户ID:组ID:用户全名:主目录:登录shell

（gecos是通用电子计算机操作系统的缩写，是Bell实验室中的一台大型主机。）

Unix系统最初是用明文保存密码的，后来由于安全的考虑，采用crypt()算法加密密码并存放在/etc/passwd文件。现在，由于计算机处理能力的提高，使密码破解变得越来越容易。/etc/passwd文件是所有合法用户都可访问的，大家都可互相看到密码的加密字符串，这给系统带来很大的安全威胁。现代的Unix系统使用影子密码系统，它把密码从/etc/pa sswd文件中分离出来，真正的密码保存在/etc/shadow文件中，shadow文件只能由**超级用户**访问。这样入侵者就不能获得加密密码串，用于破解。使用shadow密码文件后，/etc/passwd文件中所有帐户的password域的内容为"x"，如果password域的内容为"\*"，则 该帐号被停用。使用passwd这个程序可修改用户的密码。

### /etc/shadow

[root@localhost etc]# cat shadow

root:$6$qL2y9wls$uRdF7mhJQiV3uhVSaLZ2E3JsylTWCK8.C.2JBWtwHeQlpGIHXtDUu2/evW3/OrnGZtJ79CYItM1hjSdR0DQJG1:17410:0:99999:7:::

bin:\*:17110:0:99999:7:::

daemon:\*:17110:0:99999:7:::

……

tcpdump:!!:17211::::::

czha:$6$UJnxxGQuwdZNDcRE$1Mp9Iz3n5z.zRtx6daD2OLxuUFfWJ4EZIU8roWil7xDDqvf0wuBJu/rxU5vLGlNuYrWFdkK3fu/.kvg8eOllU/::0:99999:7:::

### /etc/group

/etc/group 的内容包括用户组（Group）、用户组口令、GID及该用户组所包含的用户（User），每个用户组一条记录；格式如下：

group\_name:passwd:GID:user\_list

在/etc/group 中的每条记录分四个字段：

第一字段：用户组名称；

第二字段：用户组密码；

第三字段：GID

第四字段：用户列表，每个用户之间用,号分割；本字段可以为空；如果字段为空表示用户组为GID的用户名； 我们举个例子：

### 超级用户

超级用户在系统中享有特权。超级用户账号的用户ID为0，通常登录名为root。在一般的UNIX系统上，超级用户凌驾于系统的权限检查之上。因此，无论对文件施以何种访问权限限制，超级用户都可以访问系统的任何文件，也能发送信号干预系统运行的所有用户进程。系统管理员可以使用超级用户账号来执行各种系统管理任务。

## 文件I/O模型

UNIX系统I/O模型最为显著的特性之一是其I/O通用性概念。也就是说，同一套系统调用（open()、read()、write()、close()等）所执行的I/O操作，可施之于所有文件类型，包括设备文件在内。（应用程序发起的I/O请求，内核会将其转化为相应的文件系统操作，或者设备驱动程序操作，以此来执行针对目标文件或设备的I/O操作。）因此，采用这些系统调用的程序能够处理任何类型的文件。

就本质而言，内核只提供一种文件类型：字节流序列，在处理磁盘文件、磁盘或磁带设备时，可通过lseek()系统调用来随机访问。

许多应用程序和函数库都将新行符（十进制ASCII码为10，有时亦称为换行）视为文本汇总一行的结束和另一行的开始。UNIX系统没有文件结束符的概念，读取文件时如无数据返回，便会认定抵达文件末尾。

### 文件描述符

I/O系统调用使用文件描述符——（往往是数值很小的）非负整数——来指代打开的文件。获取文件描述符的常用手法是调用open()，在参数中指定I/O操作目标文件的路径名。

通常，由shell启动的进程会继承3个已打开的文件描述符：

描述符0为标准输入，指代为进程提供输入的文件；

描述符1为标准输出，指代供进程写入输出的文件；

描述符2为标准错误，指代供进程写入错误消息或异常通告的文件；

在交互式shell或程序中，上述三者一般都指向终端。在stdio函数库中，这几种描述符分别与文件流stdin、stdout、stderr想对应。

### stdio函数库

C编程语言在执行文件I/O操作时，往往会调用C语言标准库的I/O函数。也将这样一组I/O函数称为stdio函数库，其中包括fopen()、fclose()、scanf()、printf()、fgets()、fputs()等。stdio函数位于I/O系统调用层（open()、close()、read()、write()等）之上。

## 程序

### 过滤器

从stdin读取输入，加以转换，再将转换后的数据输出到stdout，常常将拥有上述行为的程序称为过滤器，cat、grep、tr、sort、wc、sed、awk均在其列。

### 命令行参数

C语言程序可以访问命令行参数，即程序运行时在命令行中输入的内容。要访问命令行参数，程序的main()函数需做如下声明：

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

argc变量包含命令行参数的总个数，argv指针数组的成员指针则逐一指向每个命令行参数字符串。首个字符串argv[0]，标识程序名本身。

## 进程

简而言之，进程是正在执行的程序实例。执行程序时，内核会将程序代码载入虚拟内存，为程序变量分配空间，建立内核记账（bookkeeping）数据结构，以记录与进程有关的各种信息（比如，进程ID、用户ID、组ID以及终止状态等）。

在内核看来，进程是一个个实体，内核必须在它们之间共享各种计算机资源。对于像内存这样的受限资源来说，内核一开始会为进程分配一定数量的资源，并在进程的生命周期内，统筹该进程和整个系统对资源的需求，对这一分配进行调整。程序终止时，内核会释放所有此类资源，供其他进程重新使用。其他资源（如CPU、网络带宽等）都属于可再生资源，但必须在所有进程间平等共享。

### 进程的内存布局

逻辑上将一个进程划分为以下几个部分（也称为阶段）：

文本：程序的指令。

数据：程序使用的静态变量。

堆：程序可从该区域动态分配额外内存。

栈：随函数调用、返回而增减的一片内存，用于为局部变量和函数调用链接信息分配存储空间。

### 创建进程和执行程序

进程可使用系统调用fork()来创建一个新的进程。调用fork()的进程被称为父进程，新创建的进程被称为子进程。内核通过对父进程的**复制**来创建子进程。子进程从父进程处继承数据段、栈段、以及堆段的副本后，可以修改这些内容，不会影响父进程的“原版内容”。（在内存中被标记为只读的程序文本段则由父、子进程共享。）

然后，子进程要么去执行与父进程共享代码段中的另一组不同函数，或者，更为常见的情况是使用系统调用execve()去加载并执行一个全新程序。execve()会销毁现有的文本段、数据段、栈段及堆段，并根据新程序的代码，创建新段来替代它们。

以execve()为基础，C语言库还提供了几个相关函数，接口虽然略有不同，但功能全都相同。以上所有库函数的名称均已字符串“exec”打头，在函数间差异无关宏旨的场合。

### 进程ID和父进程ID

每一个进程都有一个唯一的整数型进程标识符（PID）。此外，每一个进程还具有一个父进程标示符（PPID）属性，用以标识请求内核创建自己的进程。

### 进程终止和终止状态

可使用以下两种方式之一来终止一个进程：

（1）使用\_exit()系统调用（或相关的exit()库函数），请求退出；

（2）向进程传递信号，将其“杀死”。

无论以何种方式退出，进程都会生成“终止状态”，一个非负小整数，可供父进程的wait()系统调用检测。在调用\_exit()的情况下，进程会指明自己的终止状态。若由信号来“杀死”进程，则会根据导致进程“死亡”的信号类型来设置进程的终止状态。（有时会将传递进\_exit()的参数称为进程的“退出状态”，以示与终止状态有所不同，后者要么指传递给\_exit()的参数值，要么表示“杀死”进程的信号。）

根据管理，终止状态为0表示进程“功成身退”，非0则表示有错误发生。大多数shell会将前一执行程序的终止状态保存于shell变量$?中。

### 进程的用户和组标示符（凭证）

### 特权进程

### 能力

### init进程

系统引导时，内核会创建一个名为init的特殊进程，即“所有进程之父”，该进程的相应程序文件为/sbin/init。系统的所有进程不是由init（使用fork()）“亲自”创建，就是由其后代进程创建。Init进程的进程号为1，且总是以超级用户权限运行。谁（哪怕是超级用户）都不能“杀死”init进程，只有关闭系统才能终止该进程。Init的主要任务是创建并监控系统运行所需的一系列进程。

### 守护进程

守护进程指的是具有特殊用途的进程，系统创建和处理此类进程的方式与其他进程相同，但以下特征是其所独有的：

* “长生不老”。守护进程通常在系统引导时启动，直至系统关闭前，会一直“健在”。
* 守护进程在后台运行，且无控制终端供其读取或写入数据。

守护进程中的例子有syslogd（在系统日志中记录消息）和httpd（利用HTTP分发Web页面）。

环境列表

每个进程都有一份环境列表，即在进程用户空间内存中维护的一组环境变量。这份列表的每一元素都由一个名称及其相关值组成。由fork()创建的新进程，会继承父进程的环境副本。这也为父进程间通信提供了一种机制。当进程调用exec()函数族替换当前正在运行的程序时，新程序要么继承老程序的环境，要么在exec()调用的参数汇总指定新环境并加以接收。

在绝大多数shell中，可使用export命令来创建环境变量（C shell使用setenv命令），如：export MYVAR=’Hello world’

C语言程序可使用外部变量（char \*\*environ）来访问环境，而库函数也允许进程去获取或修改自己环境中的值。

### 资源限制

每个进程都会消耗诸如打开文件、内存以及CPU时间之类的资源。使用系统调用setrlimit()，进程可为自己消耗的各类资源设定一个上限。此类资源限制的每一项均有两个相关值：软限制（soft limit）限制了进程可以消耗的资源总量，硬限制（hard limit）软限制的调整上限。非特权进程在针对特定资源调整软限制值时，可将其设置为0到相应硬限制值间的任意值，但硬限制值只能低，不能高。

由fork()创建的新进程，会继承其父进程对资源限制的设置。

使用ulimit命令（在C shell中为limit）可调整shell的资源限制。Shell为执行命令所创建的子进程会继承上述资源设置。

## 内存映射

调用系统函数mmap()的进程，会在其虚拟地址空间中创建一个新的内存映射。

映射分为两类：

文件映射：将文件的部分区域映射入调用进程的虚拟内存。映射一旦完成，对文件映射内容的访问则转化为对相应内存区域的字节操作。映射页面会按需自动从文件中加载。

相映成趣的是并无文件与之相对应的匿名映射，其映射页面的内容会被初始化为0。

由某一进程所映射的内存可以与其他进程的映射共享。达成共享的方式有二：其一是两个进程都针对某一文件的相同部分加以映射，其二是由fork()创建的子进程从父进程处继承映射。当两个或多个进程共享的页面相同时，进程之一对页面内容的改动是否为其他进程所见呢？这取决于创建映射时所传入的标志参数。若传入标志位私有，则某进程对映射内容的修改对于其他进程是不可见的，而且这些改动也不会真的落实到文件上；若传入标志位共享，对映射内容的修改就会为其他进程所见，并且这些修改也会造成对文件的改动。内存映射用途很多，其中包括：以可执行文件的相应段来初始化进程的文本段、内从（内容填充为0）分配、文件I/O（即映射内存I/O）以及进程间通信（通过共享映射）。

## 静态库和共享库

所谓目标库是这样的一种文件：将（通常是逻辑相关的）一组函数代码加以编译，并置于一个文件中，供其他应用程序调用。这一做法有利于程序的开发和维护。现代UNIX提供两种类型的对象库：静态库和共享库。

### 静态库

静态库是早期UNIX系统中唯一的一种目标库。本质上说，静态库是对已编译目标模块的一种结构化整合。要使用静态库中的函数，需要在创建程序的链接命令中指定相应的库。主程序会对静态库中隶属于各自目标模块的副本，将其复制到最终的可执行文件中，这就是所谓静态链接。对于所需库内抽取所需库内的各自目标模块，采用静态链接方式生成的程序都存有一份副本。这会引起诸多不便。其一，在不同的可执行文件中，可能都存有相同目标代码的副本，这是对磁盘空间的浪费。同理，调用同一库函数的程序，若均以静态链接方式生成，且又于同时加以执行，这会造成内存浪费，因为每个程序所调用的函数都各有一份副本驻留在内存中，此其二。此外，如果对库函数进行了修改，需要重新加以编译、生成新的静态库，而所有需要调用该函数“更新版”的应用，都必须与新生成的静态库重新链接。

### 共享库

设计共享库的目的是为了解决静态库所存在的问题。

如果将程序链接到共享库，那么链接器就不会把库中的目标模块复制到可执行文件中，而在可执行文件中写入一条记录，以表明可执行文件在运行时需要使用该共享库。一旦在运行时将可执行文件载入内存，一款名为“动态链接器”的程序会确保将可执行文件所需的动态库找到，并载入内存，随后实施运行时链接，解析可执行文件中的函数调用，将其与共享库中相应的函数定义关联起来。在运行时，共享库代码在内存中只需保留一份，且可供所有运行中的程序使用。

经过编译处理的函数仅在共享库内保存一份，从而节约了磁盘空间。另外，这一设计还能确保各类程序及时使用到函数的最新版本，只需将带有函数新定义的共享库重新加以编译即可，程序会在下次执行时自动使用新函数。

## Linux下进程间通信及同步

Linux系统上运行有多个进程，其中许多都是独立运行。然而，有些进程必须相互合作以达成预期目的，因此彼此间需要通信和同步机制。

读写磁盘文件中的信息是进程间通信的方法之一。可是，对许多程序来说，这种方法既慢又缺乏灵活性。因此，像所有现代UNIX实现那样，Linux也提供了丰富的进程间通信(IPC)机制，如下所示：

* 信号(signal)：用来表示事件的发生。
* 管道和FIFO：用于在进程间传递数据。
* 套接字：供同一台主机或是联网的不同主机上所运行的进程之间传递数据。
* 文件锁定：为防止其他进程读取或更新文件内容，允许某进程对文件的部分区域加以锁定。
* 消息队列：用于在进程间交换信息（数据包）
* 信号量(semaphore):用来同步进程动作。
* 共享内存：允许两个及两个以上进程共享一块内存。当某进程改变了共享内存的内容时，其他所有进程会立即了解到这一变化。

UNIX系统的IPC机制种类如此繁多，有些功能还互有重叠，部分原因是由于各种IPC机制是在不同的UNIX实现上演变而来的，需要遵循的标准也各不相同。例如，就本质而言，FIFO和UNIX套接字功能相同，允许同一系统上并无关联的进程彼此交换数据。二者之所以并存于现代UNIX系统之中，是由于FIFO来自System V,而套接字则源于BSD。

## 信号

尽管上一节将信号视为 IPC 的方法之一，但其在其他方面的广泛应用则更为普遍，因此值得深入讨论。

人们往往将信号称为"软件中断"。进程收到信号，就意味着某一事件或异常情况的发生。信号的类型很多，每一种分别标识不同的事件或情况。采用不同的整数来标识各种信号类型，并以SIGxxxx形式的符号名加以定义。

内核、其他进程（只要具有相应的权限）或进程自身均可向进程发送信号。例如，发生下列情况之一时，内核可向进程发送信号。

* 用户键入中断字符（通常为Control-C）。
* 进程的子进程之一已经终止。
* 由进程设定的定时器（告警时钟）已经到期。
* 进程尝试访问无效的内存地址。

在shell中，可使用kill命令向进程发送信号。在程序内部，系统调用kill()可提供相同的功能。

收到信号时，进程会根据信号采取如下动作之一。

* 忽略信号。
* 被信号"杀死"。
* 先挂起，之后再被专用信号唤醒。

就大多数信号类型而言，程序可选择不采取默认的信号动作，而是忽略信号（当信号的默认处理行为并非忽略此信号时，会派上用场）或者建立自己的信号处理器。信号处理器是由程序员定义的函数，会在进程收到信号时自动调用，根据信号的产生条件执行相应动作。

信号从产生直至送达进程期间，一直处于挂起状态。通常，系统会在接收进程下次获得调度时，将处于挂起状态的信号同时送达。如果接收进程正在运行，则会立即将信号送达。然而，程序可以将信号纳入所谓"信号屏蔽" 以求阻塞该信号。如果产生的信号处于"信号屏蔽"之列，那么此信号将一直保持挂起状态，直至解除对该信号的阻塞。（亦即从信号屏蔽中移除。）

## 线程

在现代UNIX实现中，每个进程都可执行多个线程。可将线程想象为共享同一虚拟内存及一干其他属性的进程。每个线程都会执行相同的程序代码，共享同一数据区域和堆。可是，每个线程都拥有属于自己的栈，用来装载本地变量和函数调用链接信息。

线程之间可通过共享的全局变量进行通信。借助于线程 API 所提供的条件变量和互斥机制，进程所属的线程之间得以相互通信并同步行为-尤其是在对共享变量的使用方面。此外，利用2.10节所述的IPC和同步机制，线程间也能彼此通信。

线程的主要优点在于协同线程之间的数据共享（通过全局变量）更为容易，而且就某些算法而论，以多线程来实现比之以多进程实现要更加自然。再者，显而易见，多线程应用能从多处理器硬件的并行处理中获益匪浅。

## 进程组和shell任务控制

shell执行的每个程序**都会在一个新进程内**发起。比如，shell创建了3个进程来执行以下管道命令（在当前的工作目录下，根据文件大小对文件进行排序并显示）：

$ ls -l | sort -k5n | less

除Bourne shell以外，几乎所有的主流shell都提供了一种交互式特性，名为任务控制。该特性允许用户同时执行并操纵多条命令或管道。在支持任务控制的shell中，会将管道内的所有进程置于一个新进程组或任务中。（如果情况很简单，shell命令行只包含一条命令，那么就会创建一个只包含单个进程的新进程组。）进程组中的每个进程都具有相同的**进程组标识符**（以整数形式），其实就是进程组中某个进程（也称为进程组组长process group leader）的进程ID。

内核可对进程组中的所有成员执行各种动作，尤其是信号的传递。如下节所述，支持任务控制的shell会利用这一特性，以挂起或恢复执行管道中的所有进程。

## 会话、控制终端和控制进程

会话指的是一组进程组（任务）。会话中的所有进程都具有相同的会话标识符。会话首进程（session leader）是指创建会话的进程，其进程ID会成为会话ID。

使用会话最多的是支持任务控制的shell，由shell创建的所有进程组与shell自身隶属于同一会话，shell是此会话的会话首进程。

通常，会话都会与某个控制终端相关。控制终端建立于会话首进程初次打开终端设备之时。对于由交互式shell所创建的会话，这恰恰是用户的登录终端。一个终端至多只能成为一个会话的控制终端。

打开控制终端会致使会话首进程成为终端的控制进程。一旦断开了与终端的连接（比如，关闭了终端窗口），控制进程将会收到SIGHUP信号。

在任一时点，会话中总有一个前台进程组（前台任务），可以从终端中读取输入，向终端发送输出。如果用户在控制终端中输入了"中断"（通常是Control-C）或"挂起"字符（通常是Control-Z），那么终端驱动程序会发送信号以终止或挂起（亦即停止）前台进程组。一个会话可以拥有任意数量的后台进程组（后台任务），由以"&"字符结尾的行命令来创建。

支持任务控制的shell提供如下命令：列出所有任务，向任务发送信号，以及在前后台任务之间来回切换。

## 伪终端

伪终端广泛应用于各种应用领域，最知名的要数telnet和ssh之类提供网络登录服务的应用，以及X Window系统所提供的终端窗口实现。

## 日期和时间

进程涉及两种类型的时间。

* 真实时间：指的是在进程的生命期内（所经历的时间或时钟时间），以某个标准时间点（日历时间）或固定时间点（通常是进程的启动时间）为起点测量得出的时间。在UNIX系统上，日历时间是以国际协调时间（简称UTC）1970年1月1日凌晨为起始点，按秒测量得出的时间，再进行时区调整（定义时区的基准点为穿过英格兰格林威治的经线） 。这一日期与UNIX系统的生日很接近，也被称为纪元（Epoch）。
* 进程时间：亦称为CPU时间，指的是进程自启动起来，所占用的CPU时间总量。可进一步将CPU时间划分为系统CPU时间和用户CPU时间。前者是指在内核模式中，执行代码所花费的时间（比如，执行系统调用，或代表进程执行其他的内核服务）。后者是指在用户模式中，执行代码所花费的时间（比如，执行常规的程序代码）。

time命令会显示出真实时间、系统CPU时间，以及为执行管道中的多个进程而花费的用户CPU时间。

## 客户端/服务器架构

本书有多处论及客户端/服务器应用程序的设计和实现。

客户端/服务器应用由两个组件进程组成。

* 客户端：向服务器发送请求消息，请求服务器执行某些服务。
* 服务器：分析客户端的请求，执行相应的动作，然后，向客户端回发响应消息。

有时，服务器与客户端之间可能需要就一次服务而进行多次交互。

客户端应用通常与用户打交道，而服务器应用则提供对某些共享资源的访问。一般说来，都是众多客户端进程与为数不多的一个或几个服务器端进程进行通信。

客户端和服务器既可以驻留于同一台计算机上，也可以位于联网的不同计算机上。客户端和服务器使用2.10节所讨论的IPC机制来实现彼此通信。

服务器可以提供各种服务，如下所示。

* 提供对数据库或其他共享信息资源的访问。
* 提供对远程文件的跨网访问。
* 对某些商业逻辑进行封装。
* 提供对共享硬件资源的访问（比如，打印机）。
* 提供WWW服务。

将某项服务封装于单独的服务器应用中，这一做法原因很多，举例如下。

* 效率：较之于在本地的每台计算上提供相同资源，在服务器应用管理之下提供资源的一份实例，则要节约许多。
* 控制、协调和安全：由于资源（尤其是信息资源）的统一存放，服务器既可以协调对资源的访问（例如，两个客户端不能同时更新同一信息），还可以保护资源安全，令其只对特定客户端开放。
* 在异构环境中运行：在网络中，客户端和服务器应用所运行的硬件平台和操作系统可以不同。

## 实时性

实时性应用程序是指那些需要对输入做出及时响应的程序。此类输入往往来自于外接的传感器或某些专门的输入设备，而输出则会去控制外接硬件。具有实时性需求的应用程序示例包括自动化装配流水线、银行ATM机，以及飞机导航系统等。

虽然许多实时性应用程序都要求对输入做出快速响应，但决定性因素却在于要在事件触发后的一定时限内，保证响应的交付。

要提供实时响应，特别是在短时间内加以响应，就需要底层操作系统的支持。由于实时响应的需求与多用户分时操作系统的需求存在冲突，大多数操作系统"天生"并不提供这样的支持。虽然已经设计出不少实时性的UNIX变体，但传统的UNIX实现都不是实时操作系统。Linux的实时性变体也早已诞生，而近期的Linux内核正转向对实时性应用原生而全面的支持。

为支持实时性应用，POSIX.1b定义了多个POSIX.1扩展，其中包括异步I/O、共享内存、内存映射文件、内存锁定、实时性时钟和定时器、备选调度策略、实时性信号、消息队列，以及信号量等。虽然这些扩展还不具备严格意义上的"实时性"，但当今的大多数UNIX实现都支持上面提到的全部或部分扩展（本书将讲解Linux所支持的POSIX.1b特性）。

本书会以术语"真实时间（real time）"来指代日历时间或经历时间的概念，而术语"实时性（realtime）"则是指操作系统或应用程序具备本节所述的响应能力。

## /proc文件系统

类似于其他的几种UNIX实现，Linux也提供了/proc文件系统，由一组目录和文件组成，装配（mount）于/proc目录下。

**/proc文件系统**是一种虚拟文件系统，以文件系统目录和文件形式，提供一个指向内核数据结构的接口。这为查看和改变各种系统属性开启了方便之门。此外，还能通过一组以/ proc/PID形式命名的目录（PID即进程ID）查看系统中运行各进程的相关信息。

通常，/proc目录下的文件内容都采取人类可读的文本形式，shell脚本也能对其进行解析。程序可以打开、读取和写入/proc目录下的既定文件。大多数情况下，只有特权级进程才能修改/proc目录下的文件内容。

本书在讲解各种Linux编程接口的同时，也会对相关的/proc文件进行介绍。12.1节将就该文件系统的总体信息做进一步介绍。尚无任何标准对/proc文件系统进行过规范，书中与该文件系统相关的细节均为Linux专有。

# 4、系统编程概念

## 系统调用

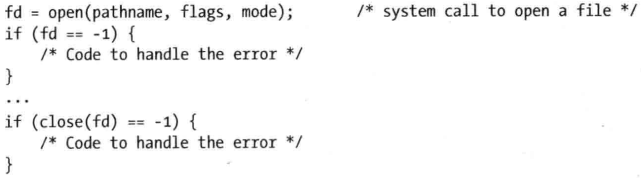
## 库函数

GUN C（glibc）

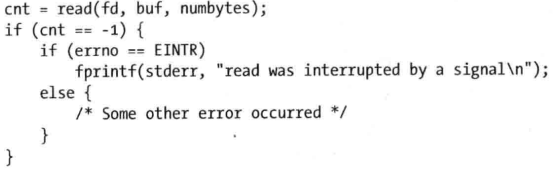
嵌入式设备领域、受限内存条件下的C语言函数库：uClibc、dietlibc。

### 处理系统调用错误

每个系统调用的手册页记录有调用可能的返回值，并指出了哪些值表示错误。通常，返回值为-1表示出错。因此，可使用下列代码对系统调用进行检查：



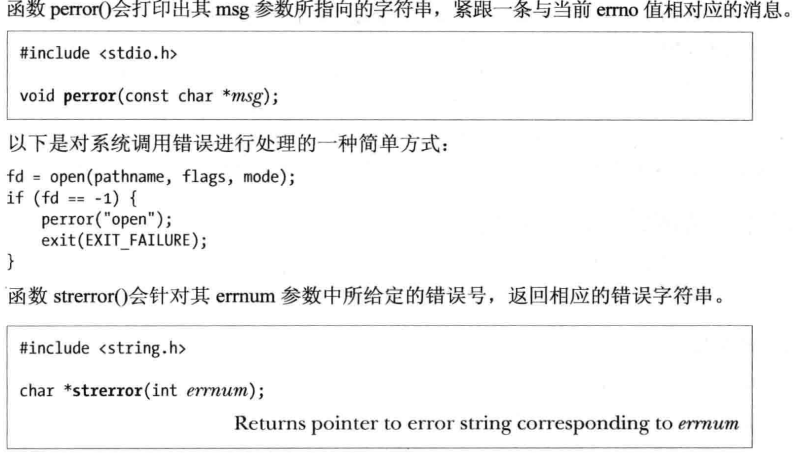
。。。。。。

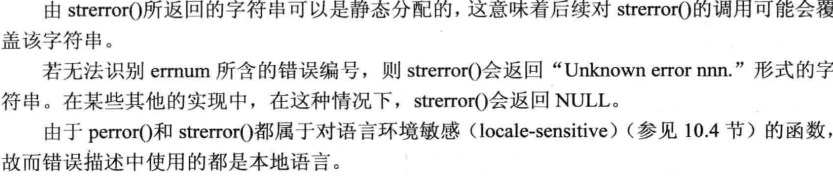


如果调用系统调用和库函数成功，errno绝不会被重置为0，故此，该变量值不为0，可能是之前系统调用失败造成的。此外，SUSv3允许在函数调用成功时，将errno设置为非零值（当然，几乎没有函数会这么做）。因此，在进行错误检查时，必须坚持首先检查函数的返回值是否表明调用出错，然后再检查errno确定错误原因。

少数系统调用（比如，getpriority()）在调用成功后，也会返回-1。要判断此类系统调用是否发生错误，应在调用前将errno置为0，并在调用后对其进行检查（上述手法同样适用于某些库函数）。

系统调用失败后，常见的做法之一是根据errno值打印错误消息。提供库函数perror()和strerror()，就是出于这一目的。



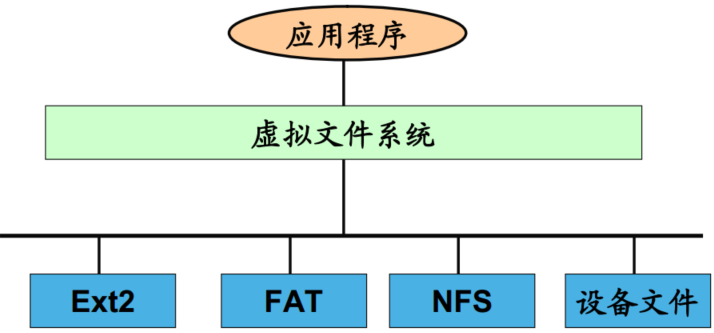


### 处理来自库函数的错误

# 文件编程

## 文件系统

Linux支持多种文件系统，如ext2、ext3、minix、iso9660、fat、vfat、nfs等。在这些具体文件系统的上层，Linux提供了虚拟文件系统（VFS）来隐藏各种文件系统的具体细节，为访问它们提供统一的接口。



## 系统调用和库函数

所有的操作系统都提供多种服务的入口点，由此程序向内核请求服务。各种版本的UNIX实现都提供良好定义、数量有限、直接进入内核的入口点，这些入口点被称为**系统调用**（system call）。UNIX所使用的技术是为每个系统调用在标准C库中设置一个具有同样名字的函数。用户进程用标准C调用序列来调用这些函数，然后，函数又用系统所要求的技术调用相应的内核服务。应用程序既可以调用系统调用，也可以调用库函数。很多库函数则会调用系统调用。

系统调用依赖于linux系统，C语言库函数与操作系统是独立的，在任何操作系统下，使用C语言库函数操作文件的方法都是相同的。

文件I/O函数（非缓冲的，即系统调用）：create、open、read、write、lseek、close

### 文件描述符

对于内核而言，所有打开的文件都通过文件描述符引用。

文件描述符为一个非负整数（>= 0）。

打开一个现有文件或创建一个新文件时，内核向进程返回一个文件描述符。

（1）标准文件描述符

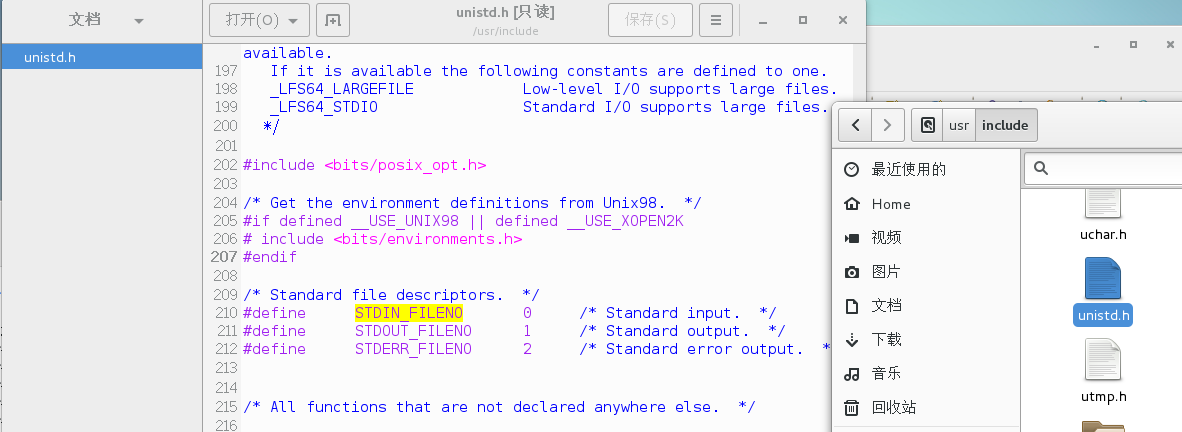
Unix系统的shell将以下三个文件描述符与相应功能关联起来了：

文件描述符0 与标准输入相关联 STDIN\_FILENO

文件描述符1 与标准输出相关联 STDOUT\_FILENO

文件描述符2 与标准错误相关联 STDERR\_FILENO

注：这几个常量在头文件<unistd.h>中定义



（2）linux的文件描述符数量限制

<http://blog.csdn.net/superchanon/article/details/13303705>

# UNIX标准及实现

# 文件I/O

开始讨论UNIX系统。

可用的文件I/O函数——打开文件、读文件、写文件等。UNIX系统中的大多数文件I/O只需要用到5个函数：open、read、write、lseek以及close。然后说明不同缓冲区长度对read和write函数的影响。

说明：文件I/O函数经常被称为不带缓冲的I/O（unbuffered I/O）。

术语 **不带缓冲** 指的是每个read和write都调用内核中的一个系统调用。这些不带缓冲的I/O函数 不是ISO C的组成部分，但是，它们是POSIX.1和Signal UNIX Specification的组成部分。

### 知识点：

#### 1 不带缓冲IO和带缓冲IO

概念、调用机制、优缺点

<http://blog.csdn.net/carolzhang8406/article/details/7227761>

<http://blog.csdn.net/WANGYONGZIXUE/article/details/46421931>

# 知识点补充：

## 幻数 是什么意思？

就是具体的数，反映不出数字所代表的意义所以应该尽量用符号常量（宏定义）来代替，以提高可读性。 比如说array[325]这里的235就是你说的Magic Number,没有变量名的数字就是幻数，建议大家不要用这样的数，解决的办法是const u32 SIZE=235；或者是#define SIZE 235

一个数，由写程序的人规定，所取的数值没有必然的条件限制，由写程序的人按自己意愿确定的数，也就是MAGIC NUMBER。

## 显示所有指令

先來瞭解一下Linux有多少指令呢？在文字模式下，你可以輸入 g 之後直接按下兩個[Tab]按鍵，看看總共有多少以 g 開頭的指令可以讓你用？

<http://linux.vbird.org/linux_basic/0160startlinux.php#manual>