目录

[1 Linux IO系统 2](#_Toc3121881)

[1.1 linux 系统基本介绍 2](#_Toc3121882)

[1.1.1 Linux 系统基本构成 2](#_Toc3121883)

[1.1.2 Linux 系统内存管理 3](#_Toc3121884)

[1.1.3 linux 系统调用介绍 4](#_Toc3121885)

[1.2 文件IO 4](#_Toc3121886)

[1.2.1 标准c的IO 4](#_Toc3121887)

[1.2.2 文件I/O系统调用 5](#_Toc3121888)

[1.2.3 读写文件缓冲区大小选择 9](#_Toc3121889)

[1.2.4 内核数据结构 9](#_Toc3121890)

[1.2.5 原子操作 9](#_Toc3121891)

[1.2.6 dup和dup2函数 11](#_Toc3121892)

[1.2.7 fcntl函数 13](#_Toc3121893)

[1.2.8 IOCTL函数 15](#_Toc3121894)

[1.2.9 I/O处理的五种模型 16](#_Toc3121895)

[1.2.10 文件锁 17](#_Toc3121896)

[1.2.11 存储映射 20](#_Toc3121897)

[1.3 文件和目录 23](#_Toc3121898)

[1.3.1 文件属性 23](#_Toc3121899)

[1.3.2 文件权限 26](#_Toc3121900)

[1.3.3 Linux 文件系统结构 29](#_Toc3121901)

[1.3.4 软连接和硬链接 30](#_Toc3121902)

[1.3.5 目录操作 33](#_Toc3121903)

[1.3.6 特殊设备文件 36](#_Toc3121904)

[2 进程 37](#_Toc3121905)

[2.1 进程的基本知识 37](#_Toc3121906)

[2.1.1 什么是进程 37](#_Toc3121907)

[2.1.2 c程序启动过程 37](#_Toc3121908)

[2.1.3 环境表 40](#_Toc3121909)

[2.1.4 非局部跳转 41](#_Toc3121910)

[2.2进程创建 46](#_Toc3121911)

[2.2.3进程创建 47](#_Toc3121912)

[2.2.3子进程继承 48](#_Toc3121913)

[2.2.4 进程链和进程扇 52](#_Toc3121914)

[2.2.5 守护进程、孤儿进程和僵尸进程 53](#_Toc3121915)

[2.3.1 wait系统调用 54](#_Toc3121916)

[2.3.2 exec函数 56](#_Toc3121917)

[5 网络编程 57](#_Toc3121918)

[5.1 网络编程基础知识 57](#_Toc3121919)

[5.1.1 TCP/IP 四层模型 57](#_Toc3121920)

[5.1.2 数据封装和IP地址 58](#_Toc3121921)

[5.1.3 网络端口号及TCP协议 58](#_Toc3121922)

[5.2 TCP编程 58](#_Toc3121923)

[5.2.1创建socket 58](#_Toc3121924)

[5.2.2 字节序 59](#_Toc3121925)

[5.2.3 地址结构 59](#_Toc3121926)

[5.2.4 ipv4 地址和字符地址之间的转换 59](#_Toc3121927)

[5.2.5 TCP客户端服务器编程模型 60](#_Toc3121928)

[5.2.6 基于tcp的服务器网络编程 61](#_Toc3121929)

[5.2.6 基于tcp的客户端网络编程 63](#_Toc3121930)

[5.2.6 tcp的连接和关闭过程 65](#_Toc3121931)

[5.2.7 服务器多并发概念 65](#_Toc3121932)

[5.2.8 多进程模型网络编程 66](#_Toc3121933)

[5.2.9 多线程模型网络编程 73](#_Toc3121934)

[5.3 UDP编程 77](#_Toc3121935)

[5.3.1 UDP 传输流程 77](#_Toc3121936)

[5.3.2 udp 常用系统调用 78](#_Toc3121937)

[5.3.3 udp编程 78](#_Toc3121938)

[5.3.4 域名解析 82](#_Toc3121939)

[5.3.5 通过域名访问网络编程 84](#_Toc3121940)

[5.3.6 广播 86](#_Toc3121941)

[6 Makefile 编写 86](#_Toc3121942)

[多进程和多线程思考 90](#_Toc3121943)

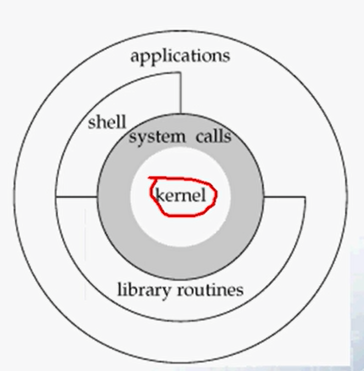
# 1 Linux IO系统

## linux 系统基本介绍

### 1.1.1 Linux 系统基本构成

Linux操作系统的基本构成：内核，系统调用，

c库,shell,文件系统，应用程序。



内核负责管理系统进程、内存、设备驱动程序、文件和网络系统，控制系统和硬件之间的通信。

内核源码：/usr/src目录下面。

shell 提供用户操作系统的接口。

/ 表示根文件系统

进程：操作系统分配资源的最小单位。

线程：一个进程内部由多个线程，线程共享内存。

initrd:最小文件系统。

Linux启动过程：bootloader🡪加载内核🡪加载最小文件系统🡪加载硬盘上的根文件系统🡪启动1号进程/sbin/init

ls -l /boot/ 目录里面 initrd.img即最小文件系统，vmlinuz-4.15.0加载的内核映像。

init进程任务：

执行/etc/init.d/目录下的所有脚本文件，启动某些系统服务。

执行/sbin/getty初始化0、1、2、（标准输入、标准输出、标准错误）

执行/bin/login启动用户登录程序

管理孤儿进程。

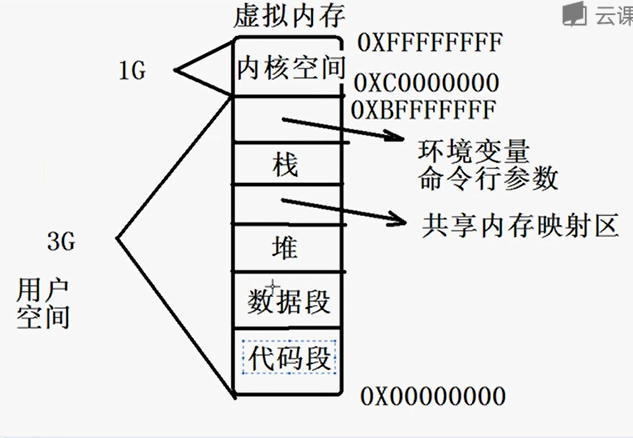
### Linux 系统内存管理

虚拟内存大小：和处理器位数有关。

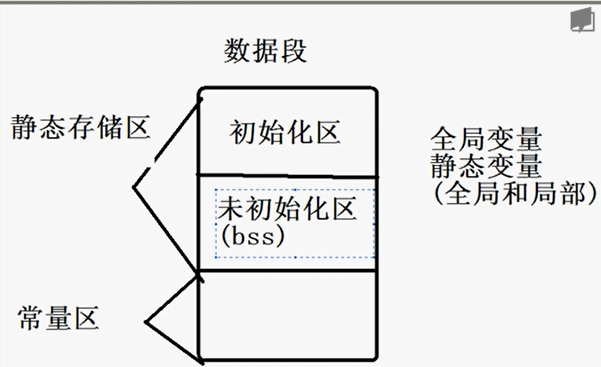
#### 进程内存分配：

一个进程的虚拟地址空间大小为2^n,n为处理器位数。典型的32位处理器共有4G虚拟地址空间。分为最高位1G内核空间，3G用户空间。

对一个用户进程虚拟地址空间分配：



数据段分为三段：



### linux 系统调用介绍

系统调用就是内核提供的一些特殊的函数，用户进程访问内核空间和使用内核服务必须使用系统调用。

进程状态：

内核态：进程运行再内核空间。

用户态：进程运行在用户空间。

## 文件IO

### 1.2.1 标准c的IO

typedef struct iobuf{

int cnt; //剩余的字节数

char \*ptr; //下一个字符位置

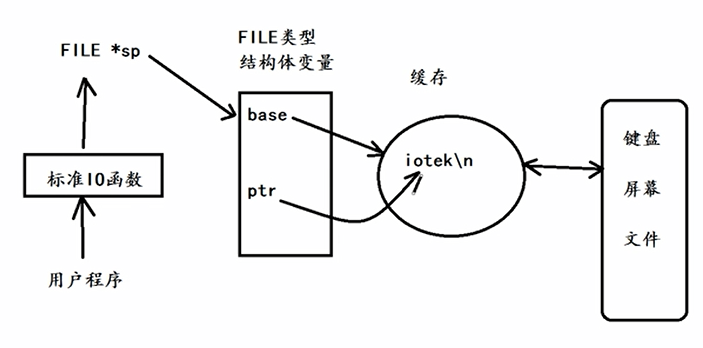
char \*base; //内存缓冲区的位置

int flag; //文件访问模式

int fd; //文件描述符，访问外存

}FILE;

标准c的IO都是带缓存的，所谓缓存意思就是对文件的读写是首先保存到内存缓冲区中的，延时一段时间再保存到外存去。



#### IO 缓存类型

全缓存：要求填满整个缓存区后才进行I/O系统调用操作。对于磁盘文件通常使用全缓存。

行缓存：涉及一个终端时（标准输入和输出），使用行缓存。行缓存满自动输出，碰到换行符自动输出。

无缓存：标准错误stderr不带缓存，使得错误信息能尽快显示出来。

行缓存案例：

#include<stdio.h>

int main(void)

{

printf("hello iotek");

while(1){

sleep(1);

}

return 0;

}

实验：

mkdir src bin obj include

gcc -o bin/line\_buf src/line\_buf.c

bin/line\_buf

发现并没有输出hello iotek。这是因为printf是行缓存的。缓存满，程序结束或者遇到换行符才会输出。

改程序：printf("hello iotek");为printf("hello iotek\n");再次运行程序发现输出了hello iotek。

### 1.2.2 文件I/O系统调用

系统调用都是不带缓存的，一般响应要求高的直接使用系统调用。

文件描述符：非负整数，标识一个文件。STDIN\_FILENO、STDOUT\_FILENO、STDERR\_FILENO对应标准输入0，标准输出1，标准错误2。

在ubuntu 中输入>man 系统调用名字 可以找到系统调用的帮助。

#### open函数

#inlcude<sys/types.h>//所有头文件都在内核源码include 目录下

#include<sys/stat.h>

#include<fcntl.h>

int opent(const char \*pathname,int flags);

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

参数：

pathname: 文件路径

flag: 用来说明对文件进行什么操作

O\_RDONLY： 只读

O\_WRONLY： 只写

O\_RDWR ： 读写

O\_NONBLOCK: 非阻塞模式执行文件操作函数

O\_CREAT： 如果文件不存在则创建该文件

O\_EXCL： 如果使用O\_CREAT选项且文件存在，则返回错误消息，可以测试文件是否存在

O\_DIRECTORY:如果pathname不是一个目录，则出错。测试是否一个目录。

O\_NOCTTY： 如果文件为终端，那么终端不可以调用open系统调用的那个进程的控制终端

O\_TRUNC： 只读或者只写打开模式，如果文件已经存在则删除文件中原有数据

O\_APPEND： 以追加的方式打开

mode：如果文件被新建，指定其权限未mode，mode是八进制权限码，0777表示文件所有者，组，其他人都拥有读写执行权限

#### close函数

#include<unistd.h>

int close(int fd);

功能：关闭一个打开的文件

返回：成功为0，出错-1

#### read函数

#include <unistd.h>

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

返回：读到的字节数，若已经到文件尾部返回0，出错-1。读操作从文件当前位移处开始，成功返回之前增加实际读得的字节数。

读到的实际数据数少于要求读取的字节数：

1>读到文件尾端

2>读终端设备，一次最多一行

3>进程由于信号中断

4>网络读，网络缓存机构造成小于所要求读的字节数。

#### write 函数

#include <unistd.h>

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

返回：成功返回写入的字节数，出错-1。

如果打开文件时指定了O\_APPEND,则从文件末尾开始写。

#### 复制文件案例：

//头文件io.h

#endif #ifndef \_\_IO\_H\_\_

#define \_\_IO\_H\_\_

**extern** **void** copy(**int** fdin,**int** fdout);

#endif

//c文件io.c，实现复制函数copy

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include "io.h"

#include <fcntl.h>

#define BUFFER\_LEN 1024

**void** copy(**int** fd1,**int** fd2)

{

**char** buffer[BUFFER\_LEN];

    ssize\_t size;

**while**(size=read(fd1,buffer,BUFFER\_LEN)>0){//读正确

**if**(write(fd2,buffer,size)!=size){

**fprintf**(stderr,"write error:%s\n",**strerror**(**errno**));

**exit**(1);

            }

    }

**if**(size<0){//读错误

        //errno是系统设置的错误编码，是上次系统调用产生错误时设置

**fprintf**(stderr,"read error:%s\n",**strerror**(**errno**));

**exit**(1);

        }

}

//cp.c文件，复制一个文件源代码

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include "io.h"

#include <fcntl.h>

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

{

**if**(argc!=3){

**fprintf**(stderr,"usage:%s srcfile desfile\n",argv[0]);

**exit**(1);

    }

**int** fdin,fdout;

    //打开一个只读文件

    fdin=open(argv[1],O\_RDONLY);

**if**(fdin<0){

**fprintf**(stderr,"open error:%s\n",**strerror**(**errno**));

**exit**(1);

    }**else** {

**printf**("open file:%d\n",fdin);

    }

    fdout=open(argv[2],O\_WRONLY|O\_CREAT|O\_TRUNC,0777);

**if**(fdout<0){

**fprintf**(stderr,"open error:%s\n",**strerror**(**errno**));

**exit**(1);

    }**else** {

**printf**("open file:%d\n",fdout);

    }

    copy(fdin,fdout);

    close(fdin);

    close(fdout);

**return** 0;

}

//编译为模块

gcc -c src/io.c -o obj/io.o -Iinclude

//编译cp 程序

gcc -o bin/cp -Iinclude obj/io.o src/cp.c

//测试是否成功复制

bin/cp src/copy.c ./copy.c

#### lseek 函数

#include<sys/types.h>

#include<uinistd.h>

off\_t lseek(int fd,off\_t offset,int whence);

返回：若成功则返回新的文件位移量，若出错为-1

功能：定位一个已经打开的文件。

offset :位移量

whence:定位的参考位置。

SEEK\_SET:文件开头

SEEK\_CUR:当前位置，此时offset可正可负

SEEK\_END:文件结尾，如果offset为正，会空出一些字节，这种文件称为空洞文件。

通过lseek获取文件大小，将whence设为SEEK\_END，offset=0,返回值就是文件大小。

lseek 还可以用来确定文件是否可以设置位移量。如果文件描述符引用的是一个管道或者FIFO,则lessk返回-1，并设置errno为EPIPE

printf("file length:%ld\n",lseek(fdin,0L,SEEK\_END));//获取文件大小

printf("file length:%ld\n",lseek(fdin,0L,SEEK\_CUR));//获取当前位置

### 1.2.3 读写文件缓冲区大小选择

df -k查看当前分区看到硬盘是哪个设备：

文件系统 1K-块 已用 可用 已用% 挂载点//根文件挂载点

/dev/sda1 30830500 18763656 10477700 65% /

sudo tune2fs -l /dev/sda1 查看硬盘具体信息发现有一项磁盘块大小

Block size: 4096

系统操作都是以Block size为单位的，所以在读写文件时，将缓冲区设为Block size大小，效率最高。

### 1.2.4 内核数据结构

一个打开的文件在内核中使用三种数据结构表示：

1. 文件描述符表

文件描述符标志fd

文件表项指针

1. 文件表项 : 操作文件的动态信息

文件状态标志：读、写、追加、同步、和非阻塞状态标志

当前文件偏移量

i节点表项指针

引用计数器

1. i节点 ：文件的静态信息

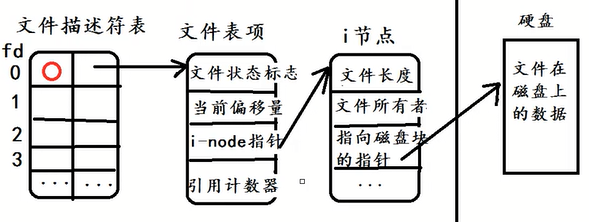
文件类型和对该文件的操作函数指针

当前文件长度

文件所有者

文件所在的设备、文件访问权限

指向文件数据在磁盘块上所在位置的指针等



### 1.2.5 原子操作

//文件读写并发出现竞争问题

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

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

{

**if**(argc<3){

**fprintf**(stderr,"usage:%s content destfile\n",argv[0]);

**exit**(1);

    }

**int** fd;

    fd=open(argv[2],O\_WRONLY);

**if**(fd<0){

**perror**("open error\n");

**exit**(1);

    }

    lseek(fd,0L,SEEK\_END);

    sleep(10);

**size\_t** size=**strlen**(argv[1])\***sizeof**(**char**);

**if**(write(fd,argv[1],size)!=size){

**perror**("write error\n");

**exit**(1);

    }

    close(fd);

**return** 0;}

#### 实验

touch append.txt //创建文件

//打开两个终端，同时执行下面两句命令,向append.txt写入数据

bin/file\_append aaaaaa append.txt

bin/file\_append AAAAAA append.txt

结果发现append.txt里面只有AAAAAA,原因是两次程序执行都是在先获得文件的尾部后休眠，休眠10s后再写入。那么他们获得的尾部偏移是一样的，后写入的覆盖了前面写入的字符。

#### 如何避免出现这种情况

采用append方式打开文件，而不是使用lseek。

注释掉  lseek(fd,0L,SEEK\_END);

更改open函数如下：

fd=open(argv[2],O\_WRONLY|O\_APPEND);//以追加方式写

再次实验发现append.txt 文件内容aaaaaaAAAAAAaaaaaaAAAAAA。写入正常

O\_APPEND模式写过程：

write(){

1. 从i节点中读取文件长度作为当前偏移量
2. 往文件中写入数据
3. 修改i节点中文件长度

}

文件追加：

对open使用O\_APPEND使得对文件偏移调整和追加称为原子操作

对open使用O\_CREAT和O\_EXEL使得文件是否存在和创建过程称为原子操作

### 1.2.6 dup和dup2函数

#include<unistd.h>

int dup(int oldfd);

int dup2(int oldfd,int newfd);

返回：成功返回新文件描述符，出错返回-1

功能：文件描述符的复制。实际意思是文件描述符表指针的复制。将oldfd的文件描述符表指针复制给newfd,以后对newfd的操作实际就是对原来oldfd关联的文件的操作。

作用：在进程间通信时可以用来改变进程的标准输入和标准输出设备。

oldfd:原先的文件描述符。newfd 新的描述符

由dup返回的新文件描述符一定是当前可用文件描述符中的最小数值。

dup2则可以用newfd参数指定新描述符的数值，如果newfd已经打开，则先将其关闭。如果oldfd等于newfd,则dup2返回newfd,而不关闭它。

#### 重定向：

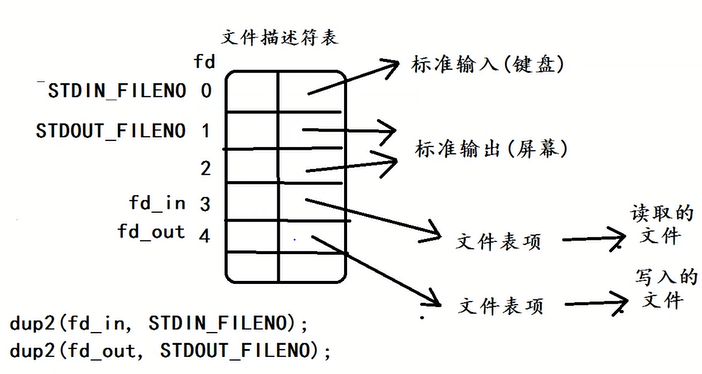
输入重定向 <

输出重定向 >

cat > cat.txt这样就把标准输出定向到cat.txt中。

如果文件不存在，重定向会新建文件；如果文件存在会覆盖原来文件内容。

追加重定向符号：>> << ；追加重定向不会覆盖原有内容。



STDIN\_FILENO的文件描述符表指针指向fd\_in关联的文件

STDOUT\_FILENO的文件描述符表指针指向fd\_out关联的文件

以后对STDIN\_FILENO的读取就是操fd\_in关联文件

STDOUT\_FILENO的写就是写入fd\_out关联的文件

//自己编程实现重定向功能，如果没有重定向就类似于echo回显功能，从键盘输入输出到屏幕

//有了重定向可以文件作为输入或者输出。

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include "io.h"

/\*

    bin/mcat + iotek.txt (+为输入重定向)

    bin/mcat - iotek.txt(-为输出重定向)

\*/

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

{

**int** fd\_in,fd\_out;

**int** flag=0;

**int** i;

**for**(i=1;i<argc;i++){

**if**(!**strcmp**("+",argv[i])){

            fd\_in=open(argv[++i],O\_RDONLY);

**if**(fd\_in<0){

**perror**("open error\n");

**exit**(1);

            }

**if**(dup2(fd\_in,STDIN\_FILENO)!=STDIN\_FILENO){

**perror**("dup2 error\n");

**exit**(1);

            }

            close(fd\_in);

        }**else** **if**(!**strcmp**("-",argv[i])){

            fd\_out=open(argv[++i],O\_WRONLY|O\_CREAT|O\_TRUNC,0777);

**if**(fd\_out<0){

**perror**("open error\n");

**exit**(1);

            }

            //将标准输出重定向到文件

**if**(dup2(fd\_out,STDOUT\_FILENO)!=STDOUT\_FILENO){

**perror**("dup2 error\n");

**exit**(1);

            }

            close(fd\_out);

        }**else**{//没有文件重定向

            flag=1;

            fd\_in=open(argv[i],O\_RDONLY);

**if**(fd\_in<0){

**perror**("open error\n");

**exit**(1);

            }

**if**(dup2(fd\_in,STDIN\_FILENO)!=STDIN\_FILENO){

**perror**("dup2 error\n");

**exit**(1);

            }

            copy(STDIN\_FILENO,STDOUT\_FILENO);

            close(fd\_in);

        }

    }

**if**(!flag)

        copy(STDIN\_FILENO,STDOUT\_FILENO);

**return** 0;

}

#### 实验

gcc -o bin/mcat obj/io.o src/mcat.c -Iinclude

bin/mcat + append.txt //输入重定向

aaaaaaAAAAAAaaaaaaAAAAAA //在屏幕显示append.txt内容

bin/mcat - append.txt //输出重定向，从键盘输入，写到append.txt

hello world //键盘输入hello world

^C

查看append.txt里面内容变为如下：

hello world

因为没有追加模式打开，因此覆盖原有内容。

chao@chao:~/linux/IO$ bin/mcat

hello china

hello china

### 1.2.7 fcntl函数

#include<unistd.h>

#include<fcntl.h>

int fcntl(int fd, int cmd);

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

int fcntl(int fdn ,int cmd, struct flock \*lock);

返回：若成功则依赖于cmd,若出错-1

功能：可以改变已经打开文件的性质

#### 使用：

1>复制一个现存的描述符，新文件描述符作为函数值返回（cmd=F\_DUPFD）

2>获得/设置文件描述符标志（cmd=F\_GETFD或F\_SETFD）

3>获得/设置文件状态标志（cmd=F\_GETFL或F\_SETFL）常用，通过第三个参数设置，可以更改的几个标志是:O\_APPEND,O\_NONBLOCK,SYNC,O\_ASYNC(O\_RDONLY、O\_WRONLY和O\_RDWR不适用)

4>获得/设置文件锁(cmd=F\_SETLK、cmd=F\_GETLK、F\_SETLKW) 第三个参数为struct flock结构体，文件锁。

//io.c中增加设置和清除文件状态标志的函数

//增加新的文件状态标志

**void** set\_fl(**int** fd,**int** flag)

{

    //获得原来的文件状态标志

**int** val =fcntl(fd,F\_GETFL);

    //增加新的文件状态标志

    val|=flag;

    //重新设在文件状态标志

**if**(fcntl(fd,F\_SETFL,val)<0){

**perror**("fcntl error");

    }

}

//清除指定的文件标志

**void** clr\_fl(**int** fd,**int** flag)

{

    //获得原来的文件状态标志

**int** val=fcntl(fd,F\_GETFL);

    //清除指定的文件标志

    val&=~flag;

    //重新设在文件状态标志

**if**(fcntl(fd,F\_SETFL,val)<0){

**perror**("fcntl error");

    }

}

//使用fcntl增加O\_APPEND功能

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include "io.h"

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

{

**if**(argc<3){

**fprintf**(stderr,"usage:%s content destfile\n",argv[0]);

**exit**(1);

    }

**int** fd;

    fd=open(argv[2],O\_WRONLY);

        set\_fl(fd,O\_APPEND);

**if**(fd<0){

**perror**("open error\n");

**exit**(1);

    }

    sleep(10);

**size\_t** size=**strlen**(argv[1])\***sizeof**(**char**);

**if**(write(fd,argv[1],size)!=size){

**perror**("write error\n");

**exit**(1);

    }

    close(fd);

**return** 0;

}

#### 实验：

重新编译copy.c

gcc -o obj/io.o -c src/io. c -Iinclude

gcc -o bin/fcntl obj/io.o src/fcntl.c -Iinclude

bin/fcntl aaaaaa append.txt

bin/fcntl AAAAAA append.txt

aaaaaaAAAAAA//append.txt内容，增加追加方式成功

### 1.2.8 IOCTL函数

#include<unistd.h>

#include<sys/ioctl.h>

int ioctl(int fd,int request,…)； //可变参数

返回：出错返回-1，成功其他数值

cat /proc/bus/input/devices 查看event\*对应哪个设备

//通过ioctl读keyboard

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include <linux/input.h>

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

{

**int** fd=-1;

**char** name[256]="unknown";

**struct** input\_event event;

**int** ret=0;

**if**((fd=open("/dev/input/event1",O\_RDONLY))<0){

**perror**("open error\n");

**exit**(1);

    }

    //获得设备的名字

**if**(ioctl(fd,EVIOCGNAME(**sizeof**(name)),name)<0){

**perror**("evdev ioctl error\n");

**exit**(1);

    }

**printf**("The device says its name is %s\n",name);

    //读写设备

**while**(1){

        ret=read(fd,&event,**sizeof**(event));

**if**(ret<0){

**printf**("read event error\n");

        }

**if**(EV\_KEY==event.type){

**printf**("key code is %d\n",event.code);

//如果是q退出，q的编码在内核里面是#define KEY\_Q           16

**if**(event.code==16){

**break**;

            }

        }

    }

}

#### 实验

chao@chao:~/linux/IO$ gcc src/ioctl.c -o bin/ioctl

chao@chao:~/linux/IO$ sudo bin/ioctl

[sudo] chao 的密码：

The device says its name is AT Translated Set 2 keyboard

key code is 28

key code is 50

mkey code is 50

key code is 38

lkey code is 38

key code is 16

qchao@chao:~/linux/IO$

### 1.2.9 I/O处理的五种模型

阻塞IO模型

非阻塞IO模型

IO多路转接模型

信号驱动IO模型

异步IO模型

### 1.2.10 文件锁

当多个用户共同使用、或者多个进程同时操作一个文件的时候，linux通常采用的方法是给文件上锁，来避免共享资源产生竞争状态。

文件锁按功能分类：

1. 共享读锁

文件描述符必须读打开

一个进程上了读锁，其他进程也可以上读锁进行读取

1. 独占写锁

文件描述符必须写打开

一个进程上了写锁，其它进程就不能上写锁和读锁进行读写操作

文件锁按类型分为建议性锁和强制性锁，建议性锁要求上锁文件的进程都要检测是否有锁存在，并尊重已有的锁。强制性锁由内核和系统执行的锁。

fcntl 不仅可以实施建议下锁而且可以实施强制性锁。

#include <unistd.h>

#include<fcntl.h>

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

cmd=F\_SETLK,F\_GETLK,F\_SETLKW //上锁，获取锁，,F\_SETLKW阻塞上锁，W等待的意思

struct flock{

short l\_type;

off\_t l\_start;

short l\_whence;

off\_t l\_len;

pid\_t l\_pid; //上锁进程的pid

}

l\_type:

锁类型：F\_RDLCK（共享读锁）,F\_WRLCK（独占性写锁）,F\_UNLCK（解锁一个区域）

l\_start、l\_whence

要枷锁或解锁的区域的起始地址，由l\_start和l\_whence两者决定。

l\_start是相对位移量，l\_whence则决定相对位移量的起点。

l\_len:

表示区域的长度

该区域可以在当前文件尾端处开始或越过其尾端处开始，但是不能在起始位置之前开始。

如若l\_len为0，则表示锁的区域从其起点开始直至最大可能位置为止，也就是不管添加了多少数据到该文件，它都处于锁的范围内。

为了锁整个文件，l\_start设为0，l\_whence说明为SEEK\_SET,l\_len设为0。

**//io.h中添加如下**

**extern** **int** lock\_reg(**int** fd, **int** cmd,**short** type,off\_t offset,**short** whence,off\_t length);

//阻塞读锁

#define READ\_LOCKW(fd,offset,whence,length) \

        lock\_reg(fd,F\_SETLKW,F\_RDLCK,offset,whence,length)

//非阻塞读锁

#define READ\_LOCK(fd,offset,whence,length) \

        lock\_reg(fd,F\_SETLK,F\_RDLCK,offset,whence,length)

//阻塞写锁

#define WRITE\_LOCKW(fd,offset,whence,length) \

        lock\_reg(fd,F\_SETLKW,F\_WRLCK,offset,whence,length)

//阻塞写锁

#define WRITE\_LOCK(fd,offset,whence,length) \

        lock\_reg(fd,F\_SETLK,F\_WRLCK,offset,whence,length)

//解锁

#define UNLOCK(fd,offset,whence,length)  \

        lock\_reg(fd,F\_SETLK,F\_UNLCK,offset,whence,length)

//io.c添加函数

**int** lock\_reg(**int** fd, **int** cmd,**short** type,off\_t offset,**short** whence,off\_t length)

{

**struct** flock flock;

    flock.l\_type=type;

    flock.l\_start=offset;

    flock.l\_whence=whence;

    flock.l\_len=length;

**if**(fcntl(fd,cmd,&flock)<0){

**perror**("fcntl error\n");

**return** 0;

    }

**return** 1;

}

//flock.c实现对文件加写锁，然后写入字符串的例子。

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include "io.h"

#include <fcntl.h>

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

{

/\*

参数：    1         2         3         4

        本程序 要写入的字符串 文件名  上锁还是解锁，字符串lock表示上锁

\*/

**if**(argc<4){

**printf**("usage:%s content file lock|unlock\n",argv[0]);

**exit**(1);

    }

    ssize\_t size=**strlen**(argv[1])\***sizeof**(**char**);

**int** fd=open(argv[2],O\_WRONLY|O\_CREAT,0777);

**if**(fd<0){

**perror**("open error\n");

**exit**(1);

    }

    sleep(5);

**if**(!**strcmp**("lock",argv[3])){

//第二个进程阻塞在此处，等待第一个进程解锁后才能加锁，程序才能向下继续执行

        WRITE\_LOCKW(fd,0,SEEK\_SET,0);

**printf**("lock success\n");

**printf**("lock pid:%d\n",getpid());

    }

**char** \*p=argv[1];

**int** i;

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

**if**(write(fd,(p+i),1)!=1){  //一次写入一个字符测试是否交叉？

**perror**("write error\n");

**exit**(1);

        }

**printf**("%d success write one character\n",getpid());

        sleep(1);

    }

**if**(!**strcmp**("lock",argv[3])){

        UNLOCK(fd,0,SEEK\_SET,0);

**printf**("unlock success\n");

**printf**("unlock pid:%d\n",getpid());

    }

    close (fd);

**return** 0;

}

//编写一个测试脚本start.h,内容如下

bin/flock aaaaaa demo.txt lock &

bin/flock AAAAAA demo.txt lock &

//命令行添加执行权限

chmod u+x start.h

#### 实验

chao@chao:~/linux/IO$ gcc -o bin/flock obj/io.o src/flock.c -Iinclude

chao@chao:~/linux/IO$ ./start.h

chao@chao:~/linux/IO$ lock success

lock pid:4506

4506 success write one character

4506 success write one character

4506 success write one character

4506 success write one character

4506 success write one character

4506 success write one character

unlock success

unlock pid:4506

lock success

lock pid:4507

4507 success write one character

4507 success write one character

4507 success write one character

4507 success write one character

4507 success write one character

4507 success write one character

unlock success

unlock pid:4507

chao@chao:~/linux/IO$

### 1.2.11 存储映射

存储映射是磁盘文件与存储空间的一个缓存相映射，对缓存数据的读写就相应的完成了文件的读写。该段缓存空间位于进程的共享内存区。

#include <sys/types.h>

#include<sys/mman.h>

void \*mmap(void \*addr,size\_t length,int prot,int flags,int fd,off\_t offset);

返回：成功则为映射区的起始地址，出错-1

功能：IO使一个磁盘文件与存储空间中的一个缓存相映射

int munmap(void \*addr,size\_t length);

返回：成功返回0，出错-1

功能：解除映射

mmap 函数从缓存中获取数据，就相当于读文件中的相应字节。与其类似，将数据存入缓存，则相应字节就自动地写入文件。这样，就可以在不使用read和write的情况下执行IO。

子进程继承父进程的存储映射区。

参数：

addr:存储映射区的起始地址，通常设为0，让系统分配。

length:需要映射的字节数

offset:映射字节在文件中的偏移量

prot:

PROT\_READ 映射区可读

PROT\_WRITE 映射区可写

PROT\_EXEC 映射区可执行

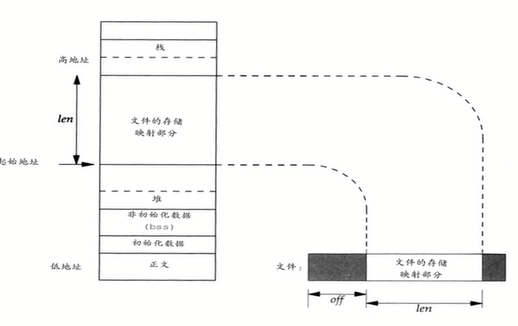
PROT\_NONE 映射区不可访问

flags :

MAP\_FIXED 返回地址必须等于addr,不鼓励使用

MAP\_SHARED 存储操作离开修改映射区文件内容

MAP\_PRIVATE 存储操作导致创建映射文件的副本，并对副本读写。



//mmap.c通过内存映射方式向文件写入20个字符

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

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

{

**if**(argc<2){

**printf**("usage:%s file\n",argv[0]);

**exit**(1);

    }

**int** fd=open(argv[1],O\_RDWR);

**if**(fd<0){

**perror**("open error\n");

**exit**(1);

}

/\*

对文件结尾后的空间读写，要预留空间。设置成一个空洞文件，这很重要,否则出现如下错误：

总线错误 (核心已转储)

\*/

    lseek(fd,19,SEEK\_END);

    write(fd,"0",1);//第20写0;

**char** \*addr;

    addr=mmap(0,20,PROT\_WRITE,MAP\_SHARED,fd,0);

**if**(addr<0){

**perror**("mmap error\n");

**exit**(1);

    }

    //修改存储映射区同步到文件

**int** i;

**for**(i=0;i<20;i++){

        \*(addr+i)='A'+i;

    }

**printf**("write success\n");

    munmap(addr,0);

    close(fd);

**return** 0;

}

#### 实验

gcc -o bin/mmap src/mmap.c

bin/mmap demo.txt

demo.txt里面是A-T

//利用mmap进行文件拷贝

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

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

{

**if**(argc<3){

**printf**("usage:%s srcfile desfile\n",argv[0]);

**exit**(1);

    }

**int** sfd=open(argv[1],O\_RDONLY);

**if**(sfd<0){

**perror**("open error\n");

**exit**(1);

    }

**int** dfd=open(argv[2],O\_RDWR|O\_CREAT|O\_TRUNC,0777);

**if**(dfd<0){

**perror**("open error\n");

**exit**(1);

    }

    off\_t len=lseek(sfd,0,SEEK\_END);//获取待读取的文件长度

**printf**("len:%ld\n",len);

    lseek(dfd,len-1,SEEK\_SET);//定位到待写文件的末尾

    write(dfd,"0",1);//文件结尾写入字符串结束符0

**char** \*a1 = mmap(0,len,PROT\_READ,MAP\_SHARED,sfd,0);

**if**(a1<0){

**perror**("mmap error\n");

**exit**(1);

    }

**char** \*a2 =mmap(0,len,PROT\_WRITE,MAP\_SHARED,dfd,0);

**if**(21<0){

**perror**("mmap error\n");

**exit**(1);

    }

**memcpy**(a2,a1,len);

    munmap(a1,0);

    munmap(a2,0);

    close(sfd);

    close(dfd);

**return** 0;

}

#### 实验

gcc -o bin/mmap\_copy src/mmap\_copy.c

bin/mmap demo.txt append.txt

demo.txt里面是A-T,执行过后，apend.txt里面内容和demo.txt一样。

## 文件和目录

### 1.3.1 文件属性

**struct** stat {

unsigned **long**  st\_dev; // 设备编号，文件系统

    unsigned **long**  st\_ino; //i-node 编号

    unsigned **int**   st\_mode; //文件类型（目录，普通文件，可执行文件）和权限0777

unsigned **int**   st\_nlink; //硬链接的连接数

 unsigned **int**   st\_uid; //用户id

    unsigned **int**   st\_gid; //用户组id

    unsigned **long**  st\_rdev;//真实的物理设备

**long**        st\_size; //文件大小字节单位

    unsigned **long**  st\_blksize; //磁盘块大小

    unsigned **long**  st\_blocks; //文件占用磁盘块数量

    unsigned **long**  st\_atime; //最后一次访问时间

    unsigned **long**  st\_atime\_nsec;

    unsigned **long**  st\_mtime; //最后一次修改时间

    unsigned **long**  st\_mtime\_nsec;

    unsigned **long**  st\_ctime; //文件状态改变时间

    unsigned **long**  st\_ctime\_nsec;

};

stat 命令查看某个文件的详细信息

stat demo.txt 获取demo.txt的信息

#include <sys/types.h>

#include <.sys/stat.h>

int stat(const char\*pathname,struct stat buf);

int fstat(int fd,struct stat \*buf);

int lstat(const char\*pathname,struct stat \*buf);

返回：若成功则返回0，若出错则为-1

功能：返回一个与pathname或fd指定的文件属性信息，存储在结构体buf中。

参数：

pathname:文件路径名字

buf:struct stat 结构体指针

lstat 函数类似于stat,但是当命名的文件是一个符号连接时，lstat返回该符号连接的有关信息，而不是由该符号连接引用的文件的信息。

#### linux 中七种文件和宏

普通文件： S\_ISREG()

目录文件： S\_ISDIR()

块特殊文件： S\_ISBLK()

字符特殊文件： S\_ISCHR()

FIFO命名管道： S\_ISSFIFO()

套接字socket: S\_ISSOCK()

符号连接： S\_ISLNK()

#### 实验

创建命名管道文件：mkfifo s.pipe

创建软连接文件：

chao@chao:~/linux/filedir$ date > date.txt

chao@chao:~/linux/filedir$ more date.txt

2018年 12月 17日 星期一 11:59:38 CST

chao@chao:~/linux/filedir$ ln -s date.txt l\_date

chao@chao:~/linux/filedir$ ls -l l\_date

lrwxrwxrwx 1 chao chao 8 12月 17 12:00 l\_date -> date.txt

//lstate.c分析文件类型

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

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

{

**if**(argc<2){

**printf**("usage:%s file \n",argv[0]);

**exit**(1);

    }

**int** i;

**struct** stat buff;

**for**(i=1;i<argc;i++){

**memset**(&buff,0,**sizeof**(buff));

**if**(lstat(argv[i],&buff)<0){

**perror**("lsate error\n");

**continue**;

        }

**printf**("%-20s",argv[i]);//输出文件名字

**if**(S\_ISREG(buff.st\_mode)){

**printf**("normal file");

        }**else** **if**(S\_ISDIR(buff.st\_mode)){

**printf**("directory");

        }**else** **if**(S\_ISBLK(buff.st\_mode)){

**printf**("block device");

        }**else** **if**(S\_ISCHR(buff.st\_mode)){

**printf**("chatacter file");

        }**else** **if**(S\_ISSOCK(buff.st\_mode)){

**printf**("socket file");

        }**else** **if**(S\_ISFIFO(buff.st\_mode)){

**printf**("fifo file");

        }**else** **if**(S\_ISLNK(buff.st\_mode)){

**printf**("link file");

        }**else**{

**printf**("unknown file");

        }

**printf**("\n");

    }

**return** 0;

}

#### 实验：

chao@chao:~/linux/filedir$ gcc -o bin/lstat src/lstat.c

hao@chao:~/linux/filedir$ bin/lstat s.pipe

s.pipe fifo file

chao@chao:~/linux/filedir$ bin/lstat l\_date

l\_date link file

chao@chao:~/linux/filedir$ bin/lstat bin

bin directory

chao@chao:~/linux/filedir$ bin/lstat date.txt

date.txt normal file

### 文件权限

9种文件访问权限：读、写、可执行

用户：S\_IRUSR S\_IWUSR S\_IXUSR

组： S\_IRGRP S\_IWGRP S\_IXGRP

其它：S\_IROTH S\_IWOTH S\_IXOTH

int fd=open(“data.txt”,O\_WRONLY,S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);

int fd=open(“data.txt”,O\_WRONLY,0666);

文件权限是8进制。7就是八进制的111，各位按权重依次是4，2，1。

//fileperm.c 分析文件的权限

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

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

{

**if**(argc<2){

**printf**("usage:%s file \n",argv[0]);

**exit**(1);

    }

**int** i;

**struct** stat buff;

**for**(i=1;i<argc;i++){

**memset**(&buff,0,**sizeof**(buff));

**if**(lstat(argv[i],&buff)<0){

**perror**("lsate error\n");

**continue**;

        }

**printf**("%-20s",argv[i]);//输出文件名字

        mode\_t mode=buff.st\_mode;

        //用户权限

**if**(S\_IRUSR&mode){

**printf**("r");

        }**else**{

**printf**("-");

        }

**if**(S\_IWUSR&mode){

**printf**("w");

        }**else**{

**printf**("-");

        }

**if**(S\_IXUSR&mode){

**printf**("x");

        }**else**{

**printf**("-");

        }

        //用户组权限

**if**(S\_IRGRP&mode){

**printf**("r");

        }**else**{

**printf**("-");

        }

**if**(S\_IWGRP&mode){

**printf**("w");

        }**else**{

**printf**("-");

        }

**if**(S\_IXGRP&mode){

**printf**("x");

        }**else**{

**printf**("-");

        }

//其它用户权限

**if**(S\_IROTH&mode){

**printf**("r");

        }**else**{

**printf**("-");

        }

**if**(S\_IWOTH&mode){

**printf**("w");

        }**else**{

**printf**("-");

        }

**if**(S\_IXOTH&mode){

**printf**("x");

        }**else**{

**printf**("-");

        }

**printf**("\n");

    }

**return** 0;

}

实验：

chao@chao:~/linux/filedir$ bin/fileperm bin/fileperm

bin/fileperm rwxr-xr-x

#### acess函数

#include <unistd.h>

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

返回：成功执行时返回0，若出错为-1

功能：检测是否可以对指定文件进行某种操作

mode:

R\_OK 判断文件是否有读权限

W\_OK 判断文件是否有写权限

X\_OK 判断文件是否有可执行权限

F\_OK 判断文件是否存在

//access.c 判断某个文件是否存在，本进程对文件是否有读写执行权限

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

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

{

**if**(argc<2){

**printf**("usage:%s file \n",argv[0]);

**exit**(1);

    }

**int** i;

**for**(i=1;i<argc;i++){

**if**(access(argv[i],F\_OK)<0){

**printf**("%s is not exist\n",argv[i]);

**continue**;

        }

**if**(access(argv[i],R\_OK)<0){

**printf**("%d can not read %s\n",getpid(),argv[i]);

        }**else**{

**printf**("%d can  read %s\n",getpid(),argv[i]);

        }

**if**(access(argv[i],W\_OK)<0){

**printf**("%d can not write %s\n",getpid(),argv[i]);

        }**else**{

**printf**("%d can  write %s\n",getpid(),argv[i]);

        }

**if**(access(argv[i],X\_OK)<0){

**printf**("%d can not exec %s\n",getpid(),argv[i]);

        }**else**{

**printf**("%d can exec %s\n",getpid(),argv[i]);

        }

    }

**return** 0;

}

#### 实验：

chao@chao:~/linux/filedir$ gcc -o bin/access src/access.c

chao@chao:~/linux/filedir$ bin/access date.txt

4221 can read date.txt

4221 can write date.txt

4221 can not exec date.txt

chao@chao:~/linux/filedir$ bin/access /etc/passwd

4228 can read /etc/passwd

4228 can not write /etc/passwd

4228 can not exec /etc/passwd

#### umask 函数

#include<sys/types.h>

#include<sys/stat.h>

mode\_t umask(mode\_t mode);

返回：以前的模式屏蔽字

功能：为进程设置模式屏蔽字，并返回以前的值。

mode:文件权限常量(S\_IRGRP、S\_IWGRP)

被umask设置过的权限不能再使用在创建文件的权限上

#include <sys/stat.h>

int chmod(const char \*pathname ， mode t mode);

intfchmod(intfd ， mode\_t mode);

mode:

用户：S\_IRUSR S\_IWUSR S\_IXUSR S\_IRWXU

组： S\_IRGRP S\_IWGRP S\_IXGRP S\_IRWXG

其它：S\_IROTH S\_IWOTH S\_IXOTH S\_IRWXO

想要改变一个文件的权限位需要条件：

进程的有效用户id必须等于文件的所有者ID

或者进程拥有超级用户权限

文件的拥有者和有效用户ID：

拥有者用户：创建文件的用户;有效用户：使用文件的用户

### Linux 文件系统结构

文件操作相关的最基本元素是：目录结构、索引节点和文件的数据本身。

目录结构：目录项

索引节点：i节点

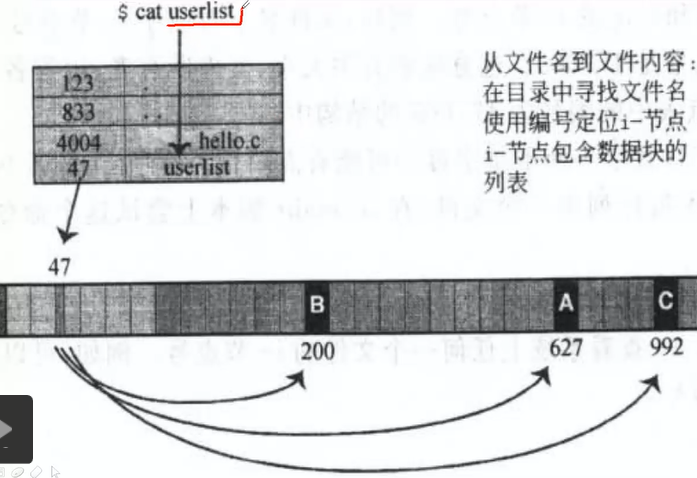
文件的数据

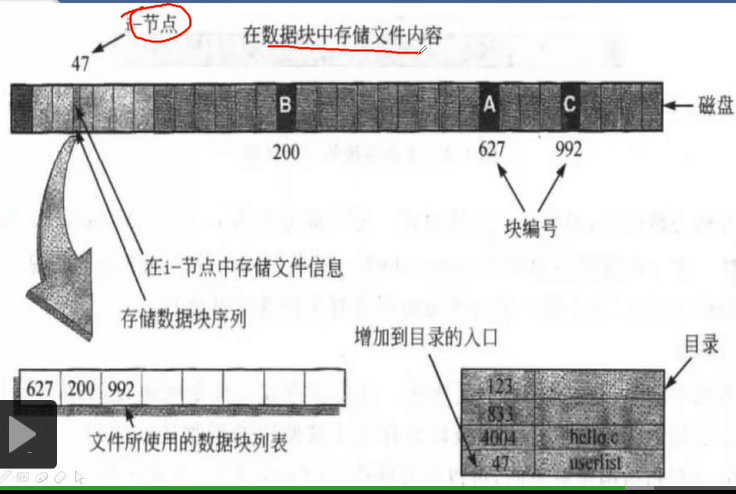
文件系统的三个区域：

超级块：存放文件系统本身的结构信息

i节点表：存放i节点信息列表（磁盘上面存储的i节点，内核进程操作文件也有一个对应的i节点，两者数据是同步的）

数据区： 存放文件内容





### 软连接和硬链接

#include<unistd.h>

int link(const char \*existingpath,const char \*newpath);

返回：成功0，出错-1

功能：创建一个执行现存文件连接(硬链接)

int unlink(const char \*pathname);

返回：成功返回0，出错返回-1

功能：删除pathname指定的硬链接，并将由pathname所引用的文件链接数减1

硬链接创建条件：

针对文件创建连接

必须在同一个分区

只有超级用户才能对目录建立连接

文件删除条件：

连接计数为0

无其他进程打开该文件

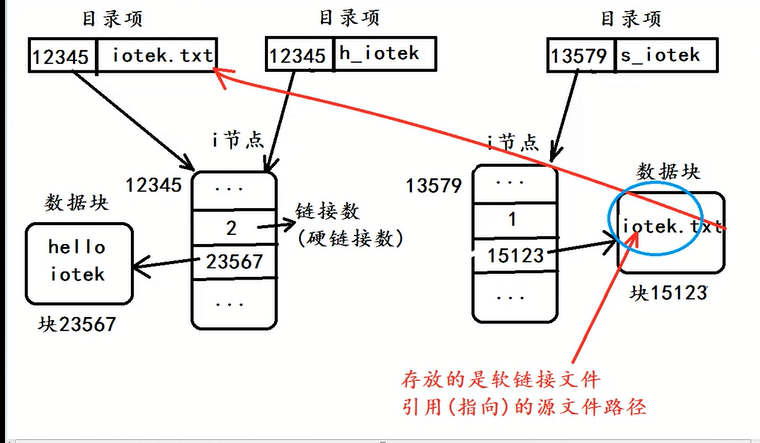
#### ln 命令

ln demo.txt h\_demo //创建硬链接文件

在demo.txt追加数据发现h\_demo中数据也更新了。

ln -s demo.txt s\_demo//创建软连接

硬链接没有创建i节点，也没有备份数据块，但是创建了一个新的目录项，该目录项和原来的目录项都指向该i节点，i节点保存了目录项的个数（就是硬链接的个数）。删除硬链接只是删除了目录项，然后i节点连接数减1，只有硬链接数为0才会真正释放数据。软连接创建了一个目录项和一个新的i节点 ，但是i节点本身指向的数据块存放的是是原来目录项的路径，而不是原来文件的数据。



**//hardlink.c 创建连接文件**

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

{

**if**(argc<3){

**printf**("usage:%s srcfile linkfile \n",argv[0]);

**exit**(1);

    }

**int** i;

**for**(i=2;i<argc;i++){

**if**(link(argv[1],argv[i])<0){

**perror**("link error\n");

**continue**;

        }

    }

**return** 0;

}

#include <unistd.h>

int remove(const char \*pathname);

返回：成功0，出错-1

功能：解除对一个文件或目录的连接，删除文件或者目录

对于文件：remove功能与unlink相同

对应目录：rmove的功能与rmdir相同。

int rename(const char \*oldname,const char \*newname);

返回：成功0，出错-1

功能：文件或者目录更名

#include<unistd.h>

int syslink(const char \*actualpath,const char \*sympath);

返回：成功返回0，出错返回-1

功能：创建一个符号软连接

int readlink(const char \*restrict pathname,char \*restrict buf,size\_t bufsize);

返回：成功返回读到的字节数，出错返回-1

功能：打开该连接本身，并读该连接中的名字。

备注：

创建符号连接并不要求actualpath路径中的文件真实存在

可以跨文件系统建立符号连接

//symlink.c 创建软链接并且通过软连接读取文件内容，并读取软连接内容

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include<unistd.h>

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

{

**if**(argc<3){

**printf**("usage:%s srcfile linkfiles \n",argv[0]);

**exit**(1);

    }

**if**(symlink(argv[1],argv[2])<0){

**perror**("symlink error\n");

**exit**(1);

    }

**char** buffer[1024];

    ssize\_t size;

**memset**(buffer,0,**sizeof**(buffer));

**int** fd=open(argv[2],O\_RDONLY);//打开链接文件

**if**(fd<0){

**perror**("open error\n");

**exit**(1);

    }

    //读取软连接文件引用的源文件内容,只读取了1024

**if**((size=read(fd,buffer,**sizeof**(buffer)))<0){

**perror**("read error\n");

    }**else**{

        write(STDOUT\_FILENO,buffer,size);

    }

**printf**("\n");

    //读取软连接文件本身的内容

**memset**(buffer,0,**sizeof**(buffer));

**if**((size=readlink(argv[2],buffer,**sizeof**(buffer)))<0){

**perror**("readlink error\n");

    }**else**{

        write(STDOUT\_FILENO,buffer,size);

    }

**printf**("\n");

    close(fd);

**return** 0;

}

#### 实验：

gcc -o bin/symlink src/symlink.c

bin/symlink date.txt ldate

12345678 //date.txt内容

date.txt //ldate内容

### 目录操作

#include<sys/types.h>

#include<sys/stat.h>

int mkdir(const char \*pathname,mode\_t mode);//mode权限位

返回：成功返回0，出错返回-1

功能：创建目录

int rmdir(const char \*pathname);

返回：成功返回0，出错返回-1

功能：删除目录

创建目录

创建一个新的空目录,.和..目录项时自动创建的

创建目录时，至少指定一个执行权限位

目录删除条件：

目录的链接数位2，只包含.和..

无其他进程打开此目录。

#include<sys/type.h>

#include<dirent.h>

DIR \*opendir(const char \*pathname);

返回：成功返回目录指针，出错NULL

功能：打开目录

struct dirent \*readdir(DIR \*dp);

返回：成功返回指针，若在目录结尾或者出错返回NULL

功能：读取目录

void rewinddir(dir(DIR \*dp));

返回：成功返回0，出错返回-1

功能：重新定位从头开始读取

int closedir(DIR \*dp);

返回：成功返回0，出错返回-1

功能：关闭目录

struct dirent{

ino\_t d\_ino; //目录中文件I节点编号

char d\_name[NAME\_MAX+1];//目录里面文件的名字

}

//readdir.c读取目录里面的所有文件的名字

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include<unistd.h>

#include<dirent.h>

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

{

**if**(argc<2){

**printf**("usage:%s dir\n",argv[0]);

**exit**(1);

    }

**struct** stat buff;

**if**(lstat(argv[1],&buff)<0){

**perror**("lstat error\n");

**exit**(1);

    }

**if**(!S\_ISDIR(buff.st\_mode)){

**fprintf**(stderr,"%s is not diretory\n",argv[1]);

**exit**(1);

    }

    //打开目录

    DIR \*dir=opendir(argv[1]);

**struct** dirent \*ent;

**while**((ent=readdir(dir))!=NULL){

**printf**("%-20s %10ld\n",ent->d\_name,ent->d\_ino);

    }

    closedir(dir);

**return** 0;

}

int chdir(const char \*pathname);

int fchdir(int fd);

返回;成功返回0，出错-1

功能：指定当前工作目录

char \*getcwd(char \*buf,size\_t size);

返回：成功返回buf,出错返回NULL

功能：获得当前工作目录的绝对路径名。类似pwd命令

当前工作目录时一个进程的属性，所以它只影响调用chdir的进程本身，而不影响其他进程。

问题：改变了路径是否会影响shell的当前路径？

不会。

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include<unistd.h>

#include<dirent.h>

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

{

**if**(argc<2){

**printf**("usage:%s dir\n",argv[0]);

**exit**(1);

    }

**char** buff[1024];

**memset**(buff,0,**sizeof**(buff));

    //获得当前工作目录

**if**(getcwd(buff,**sizeof**(buff))!=NULL){

**printf**("current dir:%s\n",buff);

    }

    //切换目录

**if**(chdir(argv[1])<0){

**perror**("chdir error\n");

**exit**(1);

    }

**memset**(buff,0,**sizeof**(buff));

    //获得切换后的当前工作目录

**if**(getcwd(buff,**sizeof**(buff))!=NULL){

**printf**("current dir: %s\n",buff);

    }

**return** 0;

}

实验：

chao@chao:~/linux/filedir$ bin/chdir /home/chao/work/

current dir:/home/chao/linux/filedir

current dir: /home/chao/work

### 1.3.6 特殊设备文件

//分析文件的设备号，所有文件都有st\_dev 设备号，但只有块设备和字符设备有st\_rdev

#include<unistd.h>

#include<stdio.h>

#include<sys/mman.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include<unistd.h>

#include<dirent.h>

#include <sys/sysmacros.h>

#include <sys/types.h>

**void** out\_dev(dev\_t devno)

{

**int** mj =major(devno);

**int** mi =minor(devno);

**printf**("(%5d,%5d)",mj,mi);

}

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

{

**if**(argc<2){

**printf**("usage:%s dir\n",argv[0]);

**exit**(1);

    }

**struct** stat buff;

**int** i;

**for**(i=1;i<argc;i++){

**memset**(&buff,0,**sizeof**(buff));

**if**(lstat(argv[i],&buff)){

**perror**("lstat error\n");

**continue**;

        }

**printf**("%-20s",argv[i]);

    //判断是否是字符设备文件和块设备文件

**if**(S\_ISCHR(buff.st\_mode)||S\_ISBLK(buff.st\_mode)){

**printf**("device file:");

        out\_dev(buff.st\_rdev);//输出特殊设备文件设备号

        }

**printf**("  on:");

    out\_dev(buff.st\_dev);

**printf**("\n");

    }

**return** 0;

}

chao@chao:~/linux/filedir$ bin/devno /dev/sda1 /dev/rtc0 date.txt

/dev/sda1 device file:( 8, 1) on:( 0, 6)

/dev/rtc0 device file:( 249, 0) on:( 0, 6)

date.txt on:( 8, 1)

# 进程

## 进程的基本知识

### 2.1.1 什么是进程

程序：磁盘中的可执行文件

进程：程序的执行实例

进程ID: 每一个进程都有一个唯一的数字标识符，这个非负整数就是进程ID

ps -ef 显示所有进程

进程控制块：task\_struct 结构体

### 2.1.2 c程序启动过程

内核会启动特殊例程，该例程在main函数之前执行。

编译器在编译时会将启动例程编译进可执行文件中。

启动例程作用：

搜集命令行的参数传递给main函数的argc和argv

搜集环境信息构建环境表并传递给main函数。main函数有三个参数的原型

登记进程的终止函数。自己也可以定义一个终止函数在进程结束时调用。终止函数主要作用是释放资源。

正常终止：

从main函数返回

调用exit(0)

调用\_exit或者\_Exit

最后一个线程从其启动例程返回

最后一个线程调用pthread\_exit

异常终止

调用abort

接收到一个信号并终止

最后一个线程对取消请求做处理响应

exit(x) x非0

进程返回:

通常程序运行成功返回0，失败非0

shell中查看进程返回值(echo $?) 0代表成功返回，其他标识出错

#include<stdlib.h>

int atexit(void(\*function)(void));

返回：若成功则为0，出错-1

功能：向内核登记终止函数

作用：释放进程占用的资源

登记了多个终止函数，执行顺序是栈方式，先登记后执行。

//atexit.c

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include <sys/types.h>

**void** term\_fun1(**void**)

{

**printf**("first term function\n");

}

**void** term\_fun2(**void**)

{

**printf**("second term function\n");

}

**void** term\_fun3(**void**)

{

**printf**("third term function\n");

}

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

{

**if**(argc<3){

**printf**("usage:%s file [exit|\_exit|return]\n",argv[0]);

**exit**(1);

    }

**atexit**(term\_fun1);

**atexit**(term\_fun2);

**atexit**(term\_fun3);

**FILE** \*fp=**fopen**(argv[1],"w");

**fprintf**(fp,"hello iotek");//向文件写入数据，缓存方式写入

**if**(!**strcmp**(argv[2],"exit")){

**exit**(0);//标准c

    }**else** **if**(!**strcmp**(argv[2],"\_exit")){

        \_exit(0);//系统调用

    }**else** **if**(!**strcmp**(argv[2],"return")){

**return** 0;

    }**else**{

**fprintf**(stderr,"usage:%s file [exit|\_exit|return]\n",argv[0]);

    }

**exit**(0);

}

#### 实验：

chao@chao:~/linux/process$ gcc -o bin/atexit src/atexit.c

chao@chao:~/linux/process$ bin/atexit iotek.txt return

third term function

second term function

first term function

chao@chao:~/linux/process$ bin/atexit iotek.txt exit

third term function

second term function

first term function

chao@chao:~/linux/process$ more iotek.txt

hello iotek

chao@chao:~/linux/process$ rm iotek.txt

chao@chao:~/linux/process$ bin/atexit iotek.txt \_exit

chao@chao:~/linux/process$ ls

bin include iotek.txt obj src

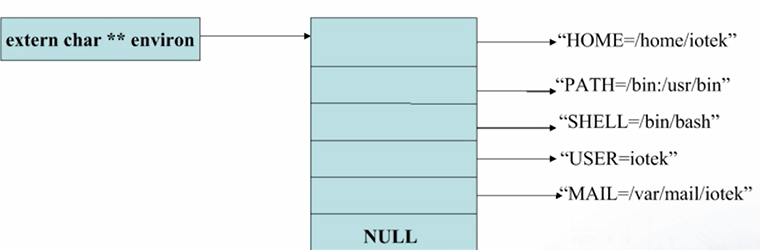
chao@chao:~/linux/process$ more iotek.txt

chao@chao:~/linux/process$

注意：发现使用\_exit结束程序不会执行atexit登记的函数，也不会刷新缓存，用标准c库函数写入文件的内容就会丢失。



### 2.1.3 环境表



有两种方法获取环境表：

1>三个参数的main函数，这样启动进程会把环境表作为参数传递给main函数。

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

2>通过外部的全局变量

etxern char \*\* environ;

#### 环境变量的操作函数：

#include<stdblib.h>

char \*getenv(const char \*name);

返回：指向与name关联的value的指针，若未找到则返回NULL

功能：获取环境变量值

int putenv(char \*str);

返回：成功返回0，出错返回非-1

功能：形式为name=value的字符串，将其放到环境表中。如果name已经存在，则先删除其原来的定义。

int setenv(const char \*name,const char \*value ,int rewrite);

返回：成功返回0，出错返回-1

功能：将name设置为value.如果在环境中name已经存在，那么若rewrite非0，则受先删除其现存的定义，若rewrite为0，则不删除其现存定义(name 不设置为新的value,而且也不出错)

int unsetenv(const char \*name);

返回：成功返回0，出错返回-1

功能：删除name的定义，即使不存在这种定义也不算出错。

注意：所有在进程中设置的环境变量只影响本进程，进程结束，环境变量就不存在。

系统配置文件设置的环境变量才是全局的。

//获取环境表例子

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include <sys/types.h>

/\*

方法1

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

{

    int i=0;

    char \*env;

    while((env=envp[i])!=NULL){

        printf("%s\n",env);

        i++;

    }

    return 0;

}\*/

 //方法2

**extern** **char** \*\*environ;

**void** show\_env()

{

**int** i=0;

**char** \*env;

**while**((env=environ[i])!=NULL){

**printf**("%s\n",env);

        i++;

    }

}

**int** main(**void**)

{

show\_env();

printf("------------------------------");

putenv("CITY=shanghai");

setenv("COMPANY","iotek",1);

//unsetenv(“CITY”);

show\_env();

**return** 0;

}

实验:bin/environment 显示当前环境变量

### 2.1.4 非局部跳转

#include<setjmp.h>

int setjmp(jmp\_buf env);

返回：直接调用返回0，若从longjmp调用返回非0（意思是该函数会被调用两次，在longjmp中还会再调用一次。）

功能：设置非局部跳转的跳转点

void longjmp(jmp\_buf env,int\_val);

功能：进行非局部跳转，val为setjmp的返回值。

goto语句限制再函数内部跳转，longjmp不限于函数内部。

//setjmp.c

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include <sys/types.h>

#include<setjmp.h>

#define TOK\_ADD 5

#define TOK\_SUB 6

**char** \*prompt="cal:";

**jmp\_buf** env;

**static** **int** is\_number(**char** \*item)

{

**int** len=**strlen**(item);

**int** i;

**for**(i=0;i<len;i++){

**if**(item[i]>'9'||item[i]<'0')

**return** 0;

    }

**return** 1;

}

**int** get\_token(**char** \*line)

{

**char** \*item=**strtok**(line," ");//以空格来分解命令

**if**(line!=NULL){

**if**(!**strcmp**("add",item))

**return** TOK\_ADD;

**if**(!**strcmp**("sub",item))

**return** TOK\_SUB;

   }**else**{//

**if**(is\_number(item)){

**int** i=**atoi**(item);

**return** i;

        }**else**{

**fprintf**(stderr,"arg not number\n");

            //进行非局部跳转

**longjmp**(env,1);

        }

    }

}

**void** cmd\_add(**void**)

{

**int** i=get\_token(NULL);

**int** j=get\_token(NULL);

**printf**("resule is %d\n",i+j);

}

**void** cmd\_sub(**void**)

{

**int** i=get\_token(NULL);

**int** j=get\_token(NULL);

**printf**("resule is %d\n",i-j);

}

**void** do\_line(**char** \*line)

{

**int** cmd=get\_token(line);//分解命令行参数

**switch**(cmd){

**case** TOK\_ADD:

            cmd\_add();

**break**;

**case** TOK\_SUB:

            cmd\_sub();

**break**;

**default**:

**fprintf**(stderr,"error command\n");

**break**;

    }

}

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

{

    ssize\_t size=**strlen**(prompt)\***sizeof**(**char**);

**char** buff[256];

    ssize\_t len;

**if**(**setjmp**(env)<0){

**perror**("setjmp error\n");

**exit**(1);

    }

    write(STDOUT\_FILENO,prompt,size);//显示提示符prompt

**while**(1){

        len=read(STDIN\_FILENO,buff,256);

**if**(len<0){

**break**;

        }

        buff[len-1]=0;

        do\_line(buff);//处理命令行输入

        write(STDOUT\_FILENO,prompt,size);//显示提示符prompt

    }

**return** 0;

}

实验：

在没有使用非局部跳转，就是红色代码部分时。会出现错误结果。

当输入的不是数字时应该立即停止本次运行，然而程序却继续执行输出了错误信息，因为错误信息并不致命，因此也不能使用exit来终止程序，这时候可以使用setjmp来重新运行程序。

chao@chao:~/linux/process$ bin/setjmp //没有加入setjmp

cal:add 2 f

arg not number

resule is 17 //输出了错误信息

chao@chao:~/linux/process$ bin/setjmp //加入setjmp

cal:add 2 3

resule is 5

cal:add 3 g

arg not number

#### 非局部跳转变量的值

//那些变量在局部跳转后值会不会能恢复到初始值

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include <sys/types.h>

#include<setjmp.h>

**int** g\_v;

**jmp\_buf** env;

**void** f2()

{

**longjmp**(env,1);

}

//参数依次是全局变量、静态变量、自动变量、寄存器变量、malloc分配变量、volatile变量

**void** f1(**int** g\_v,**int** s\_v,**int** a\_v,**int** r\_v,**int** m\_v,**int** v\_v)

{

**printf**("before longjmp\n");

**printf**("global %d,static %d,auto %d,reg %d,heap %d,vola %d\n",g\_v,s\_v,a\_v,r\_v,m\_v,v\_v);

  f2();

}

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

{

**static** **int** s\_v;

**int** a\_v;

**register** r\_v;

**int** \*m\_v=(**int**\*)**malloc**(**sizeof**(**int**));

**volatile** **int** v\_v;

    g\_v=1;

    s\_v=2;

    a\_v=3;

    r\_v=4;

    \*m\_v=5;

    v\_v=6;

**int** k=0;

**if**((k=**setjmp**(env))<0){

**perror**("setjmp error");

    }**else** **if**(k==1){//longjmp返回1，就是setjmp第二次调用的返回值

**printf**("after longjmp\n");

**printf**("global %d,static %d,auto %d,reg %d,heap %d,vola %d\n",g\_v,s\_v,a\_v,r\_v,\*m\_v,v\_v);

**exit**(0);

    }

    g\_v=10;

    s\_v=20;

    a\_v=30;

    r\_v=40;

    \*m\_v=50;

    v\_v=60;

    f1( g\_v,s\_v, a\_v, r\_v,\*m\_v,v\_v);

**return** 0;

}

实验：

chao@chao:~/linux/process$ gcc -o bin/jmpval src/jmpval.c

chao@chao:~/linux/process$ bin/jmpval

global 10,static 20,auto 30,reg 40,heap 50,vola 60

after longjmp

global 10,static 20,auto 30,reg 4,heap 50,vola 60

gcc -O -o bin/jmpval src/jmpval.c

chao@chao:~/linux/process$ bin/jmpval

before longjmp

global 10,static 20,auto 30,reg 40,heap 50,vola 60

after longjmp

global 10,static 20,auto 3,reg 4,heap 50,vola 60

## 2.2进程创建

查看系统中的进程可以使用ps指令。

ps 指令查当前用户运行的进程

ps -ef 查看所有进程的详细信息

ps -aux 查看进程占用的内存和cpu时间比例

#### 2.2.1进程常见的状态：

运行状态：就绪态和正在运行，ps命令的STAT列为R

等待状态：等待事件发生，等待系统资源，ps命令的STAT列为S表示可中断的状态，D表示不可中断

停止状态：ps命令的STAT列为T,sleep命令

僵尸状态: 进程终止，但是在进程表项中仍有记录。ps命令的STAT列为Z

#### 2.2.2 进程标识：

pid\_t getpid(void);//进程id

uid\_t getuid(void);//实际用户id

uid\_t geteuid();//有效用户id

gid\_t getgid();//组id

pid\_t getppid(void);//父进程id

登陆用户实际用户，登陆用户改变权限到root用户运行程序，root的id就是有效id

**int** main(**void**)

{

**printf**("pid:%d\n",getpid());//进程id

**printf**("ppid:%d\n",getppid());//父进程id

**printf**("uid:%d\n",getuid());//实际用户id

**printf**("epid:%d\n",geteuid());//有效用户id

**printf**("gid:%d\n",getgid());//组id

**return** 0;

}

chao@chao:~/linux/process$ gcc -o bin/pid src/pid.c

chao@chao:~/linux/process$ bin/pid

pid:4292

ppid:2184

uid:1000

epid:1000

gid:1000

chao@chao:~/linux/process$ ps

PID TTY TIME CMD

2184 pts/0 00:00:00 bash

4293 pts/0 00:00:00 ps

chao@chao:~/linux/process$

shell执行的进程的父进程就是当前shell。

### 2.2.3进程创建

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void)

返回：fork返回两次子进程中为0，父进程中为子进程ID,出错-1

pid\_t vfork(void);

返回：vfork返回两次子进程中为0，父进程中为子进程ID,出错-1

fork来说哪个先运行不确定，并且复制子进程的内存空间

vfork子进程先运行但不复制父进程的内存空间。

**//fork创建子进程**

**int** main(**void**)

{

**printf**("pid:%d\n",getpid());//进程id

    pid\_t pid;

    pid=fork();

**if**(pid<0){

**perror**("fork error\n");

    }**else** **if**(pid>0){

        //父进程执行代码

**printf**("parent process pid:%d ppid:%d fork return pid:%d",getpid(),getppid(),pid);

    }**else** **if**(pid==0){

**printf**("child process pid:%d ppid:%d fork return pid:%d",getpid(),getppid(),pid);

    }

    //下面程序父子进程都要运行的

**printf**("pid:%d\n",getpid());

    sleep(1);

**exit**(0);

}

//fork2 父子进程是交替执行的

**int** main(**void**)

{

**printf**("current pid:%d\n",getpid());//进程id

    pid\_t pid;

    pid=fork();

**if**(pid<0){

**perror**("fork error\n");

    }**else** **if**(pid>0){

        //父进程执行代码

**int** i;

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

**printf**("this is parent process\n");

            sleep(1);

        }

    }**else** **if**(pid==0){

**int** i;

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

**printf**("this is child process\n");

            sleep(1);

        }

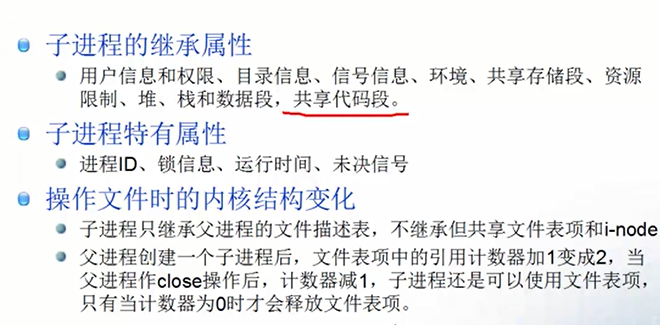
    }

**exit**(0);

}

### 2.2.3子进程继承

继承就是复制一份，复制后各自操作各自的互不干扰，而共享意味着只有一份，父子进程都操作同一份。



**//fork3 子进程继承的父进程数据**

**int** g\_v=30;

**int** main(**void**)

{

**int** a\_v=30;

**static** **int** s\_v=30;

**printf**("pid:%d\n",getpid());//进程id

    pid\_t pid;

    pid=fork();

**if**(pid<0){

**perror**("fork error\n");

    }**else** **if**(pid>0){

        //父进程执行代码

        g\_v=40;a\_v=40;s\_v=40;

**printf**("parent process pid:%d ppid:%d fork return pid:%d",getpid(),getppid(),pid);

**printf**("g\_v: %p,a\_v:%p,s\_v:%p\n",&g\_v,&a\_v,&s\_v);

    }**else** **if**(pid==0){

          g\_v=50;a\_v=50;s\_v=50;

**printf**("child process pid:%d ppid:%d fork return pid:%d",getpid(),getppid(),pid);

**printf**("g\_v: %p,a\_v:%p,s\_v:%p\n",&g\_v,&a\_v,&s\_v);

    }

    //下面程序父子进程都要运行的

**printf**("pid:%d\n",getpid());

**printf**("g\_v: %d,a\_v:%d,s\_v:%d\n",g\_v,a\_v,s\_v);

    sleep(1);

**exit**(0);

}

#### 实验

chao@chao:~/linux/process$ gcc -o bin/fork3 src/fork3.c

chao@chao:~/linux/process$ bin/fork3

pid:2790

parent process pid:2790 ppid:2687 fork return pid:2791g\_v: 0x5645140fa010,a\_v:0x7ffc98ad07d0,s\_v:0x5645140fa014

pid:2790

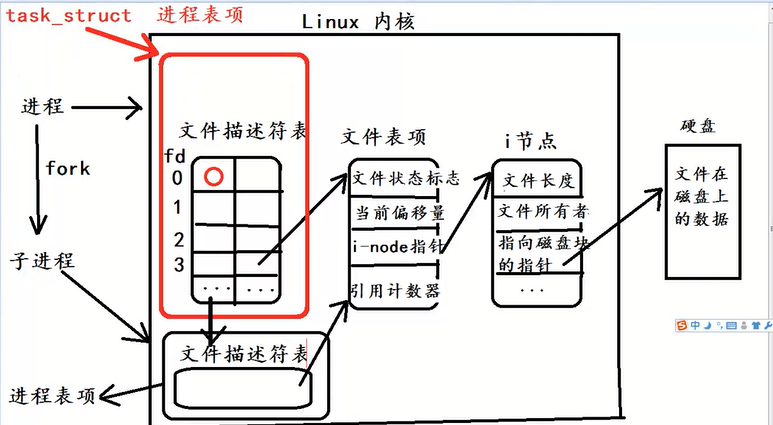
g\_v: 40,a\_v:40,s\_v:40

child process pid:2791 ppid:2790 fork return pid:0g\_v: 0x5645140fa010,a\_v:0x7ffc98ad07d0,s\_v:0x5645140fa014

pid:2791

g\_v: 50,a\_v:50,s\_v:50

发现子进程继承的变量和父进程中的变量的虚拟地址是一样的，但是变量的值不一样。这是因为父子进程运行的程序只有一份，变量的虚拟地址不变。但是数据段，堆栈段在物理内存中地址是各有一份，地址是不一样。



**//文件的缓冲区的继承fork4**

**int** g\_v=30;

**int** main(**void**)

{

**int** a\_v=30;

**static** **int** s\_v=30;

**printf**("pid:%d\n",getpid());//进程id

**FILE** \*fp=**fopen**("s.txt","w");

**int** fd=open("s\_fd.txt",O\_WRONLY|O\_CREAT|O\_TRUNC,S\_IRWXU|S\_IRWXG);

**char** \*s="hello iotek";

    ssize\_t size=**strlen**(s)\***sizeof**(**char**);

    //标准IO函数(带缓存)

**fprintf**(fp,"s:%s,pid:%d",s,getpid());

    write(fd,s,size);//不带缓存

    pid\_t pid;

    pid=fork();

**if**(pid<0){

**perror**("fork error\n");

    }**else** **if**(pid>0){

        //父进程执行代码

        g\_v=40;a\_v=40;s\_v=40;

**printf**("parent process pid:%d ppid:%d fork return pid:%d",getpid(),getppid(),pid);

**printf**("g\_v: %p,a\_v:%p,s\_v:%p\n",&g\_v,&a\_v,&s\_v);

    }**else** **if**(pid==0){

          g\_v=50;a\_v=50;s\_v=50;

**printf**("child process pid:%d ppid:%d fork return pid:%d",getpid(),getppid(),pid);

**printf**("g\_v: %p,a\_v:%p,s\_v:%p\n",&g\_v,&a\_v,&s\_v);

    }

    //下面程序父子进程都要运行的

**printf**("pid:%d\n",getpid());

    //printf("g\_v: %d,a\_v:%d,s\_v:%d\n",g\_v,a\_v,s\_v);

    //fprintf(fp,"pid:%d g\_v: %d,a\_v:%d,s\_v:%d\n",getpid(),g\_v,a\_v,s\_v);

    sleep(1);

**exit**(0);

}

实验:

chao@chao:~/linux/process$ more s\_fd.txt

hello iotek

chao@chao:~/linux/process$ more s.txt

s:hello iotek,pid:3378pid:3378 g\_v: 40,a\_v:40,s\_v:40

s:hello iotek,pid:3378pid:3379 g\_v: 50,a\_v:50,s\_v:50

发现在创建子进程之前，父进程调用系统调用write写入的数据，只有父进程写入的一份。而fprintf写入的数据，子进程也继承了一份。这是因为fpirntf写入是缓存性的，而缓存在堆中会被子进程继承。而数据真正的写入文件是在程序结束或者缓存满的时候，由于子进程继承了父进程的缓存数据，因此子进程写入了和父进程一样的数据。并且因为父子进程共享进程表项，因此对同一个文件的写入是交替的而不是覆盖的。

**/\*父进程设置文件的读写位置在结尾，在子进程中直接进行读写。发现数据追加到了文件末尾**

**证明了文件描述符被复制，但是文件描述符表是共享的。也就是在父进程lseek位置会影响到子进程\*/**

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

{

**if**(argc<2){

**fprintf**(stderr,"usage: %s file\n",argv[0]);

**exit**(1);

    }

**int** fd=open(argv[1],O\_WRONLY);

**if**(fd<2){

**perror**("open error\n");

**exit**(1);

    }

    pid\_t pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

    }**else** **if**(pid>0){

        //父进程将文件偏移量调整到文件尾部

**if**(lseek(fd,0L,SEEK\_END)<0){

**perror**("lseek error\n");

**exit**(1);

        }

    }**else**{

**char** \*str="hello child";

        ssize\_t size=**strlen**(str)\***sizeof**(**char**);

        sleep(3);

        //此处fd是从父进程中复制过来的，和父进程fd指向同一个文件

**if**(write(fd,str,size)!=size){

**perror**("write error\n");

**exit**(1);

        }

    }

**printf**("pid:%d finish\n",getpid());

    sleep(1);

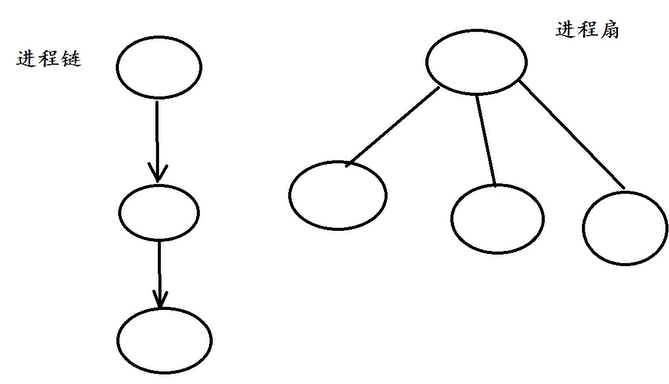
    //父子进程都要关闭自己的fd

    close(fd);

**exit**(0);

}

### 2.2.4 进程链和进程扇



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

{

**int** counter=0;

**if**(argc<2){

        counter=2;

    }

**else** {

        counter=**atoi**(argv[1]);

    }

**int** i=0;

    pid\_t pid;

**for**(;i<counter;i++){

        pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

/\*如果是父进程运行退出循环。因为父子进程交替运行，如果父进程不退出循环那么

它还会继续创建新的子进程。\*/

        }**else** **if**(pid>0) **break**;

    }

**printf**("pid: %d,ppid: %d\n",getpid(),getppid());

sleep(1);

**exit**(0);

}

/\*

进程扇：一个进程创建多个子进程，

counter创建子进程个数，由命令行传入

\*/

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

{

**int** counter=0;

**if**(argc<2){

        counter=2;

    }

**else** {

        counter=**atoi**(argv[1]);

    }

**int** i=0;

    pid\_t pid;

**for**(;i<counter;i++){

        pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

/\*如果是子进程运行退出循环。因为父子进程交替运行，如果子进程不退出循环那么

它还会继续创建新的子进程。\*/

        }**else** **if**(pid==0) **break**;

    }

**printf**("pid: %d,ppid: %d\n",getpid(),getppid());

    sleep(1);

**exit**(0);

}

### 2.2.5 守护进程、孤儿进程和僵尸进程

守护进程daemon是生存期长的一种进程，他们通常在系统启动时启动，在系统关闭时终止。

所有守护进程都以超级用户的优先权运行。

守护进程没有控制终端

守护进程的父进程都是init进程。

父进程结束，子进程称为孤儿进程，会由1号进程领养（init）。发现ubutu孤儿进程父进程并不是1。

子进程结束但是没有完全释放内存（内核中的task\_struct没有释放），该进程就成为僵尸进程。当僵尸进程的父进程结束后就会被Init进程领养，最终被收回。

如和避免僵尸进程：

父进程回收，父进程每隔一段实际来查询子进程是否结束并回收，调用wait()或者waitpid(),通知内核释放僵尸进程。

采用信号SIGCHLD通知处理，并在信号处理程序中调用wait函数。

让僵尸进程称为孤儿进程，由init进程回收。

**//孤儿进程，发现ubuntu孤儿进程的父进程不是init进程1**

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

{

    pid\_t pid;

        pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

        }**else** **if**(pid>0){

**printf**("%d deaded\n",getpid());

**exit**(0);

        }**else** **if**(pid==0){

            sleep(10);//等待父进程先结束

**printf**("pid:%d,ppid:%d\n",getpid(),getppid());

        }

**exit**(0);

}

**// 生成僵尸进程**

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

{

        pid\_t pid;

        pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

        }**else** **if**(pid==0){

**printf**("pid:%d,ppid:%d\n",getpid(),getppid());

**exit**(0)//子进程结束称为僵尸进程

        }

**while**(1){//父进程做循环

        sleep(1);

    }

**exit**(0);

}

打开新终端使用命令：ps -aux | grep process\_zombie查看进程状态。

kill -9 2484 结束pid为2484的进程。

### 2.3.1 wait系统调用

#include <sys/types.h>

#include <syt/wait.h>

pid\_t wait(int \*status);

返回：成功返回子进程ID,出错返回-1

功能：等待子进程退出并收回，防止孤儿进程产生。

pid\_t waipid(pid\_t pid, int \*status,int options);

返回：成功返回子进程ID,出错返回-1

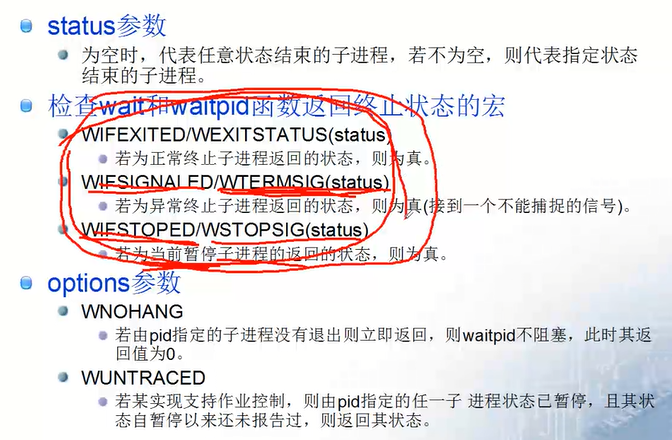
功能：wait函数的非阻塞版本

wait和waitpid函数区别：

在一个子进程终止前，wait使其调用者阻塞

waitpid有一选择项，可使调用者不阻塞

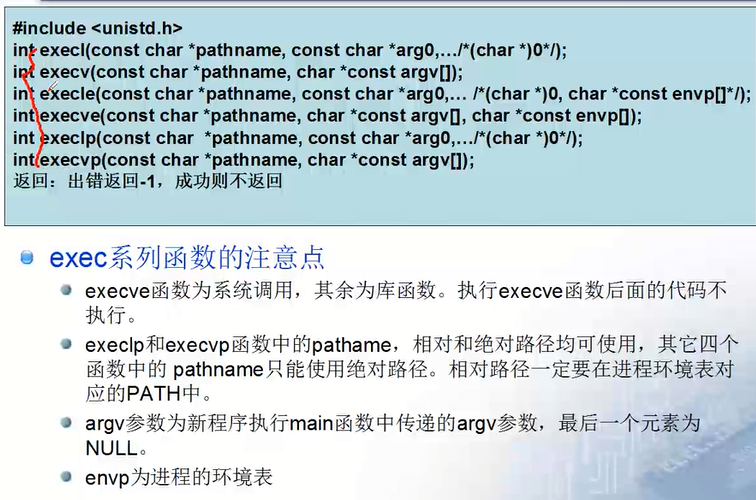
waitpid等待一个指定的子进程，而wait等待所有的子进程，返回任一终止子进程的状态。



wait返回的状态码并不是exit时的状态码，所以需要宏来获得真正的状态。exit退出码不为0并不表示是异常终止

kill -l查看所有信号

### 2.3.2 exec函数



参数列表就是传给新程序的argv[]

#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<fcntl.h>

#include<memory.h>

#include<sys/stat.h>

#include <sys/types.h>

#include<sys/wait.h>

**char** \*cmd1="cat";

**char** \*cmd2="/bin/cat";

**char** \*argv1="/etc/passwd";

**char** \*argv2="/etc/group";

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

{

    pid\_t pid;

    pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

    }**else** **if**(pid==0){

        //必须绝对路径执行，第一个参数argv是新程序的名字，可以是相对路径也可以是绝对路径

**if**(execl(cmd2,cmd1,argv1,argv2,NULL)<0){

**perror**("execl error\n");

**exit**(1);

        }**else**{

**printf**("execl %s success\n",cmd1);//成功调用不返回，此句不会执行

        }

    }

    wait(0);

**printf**("------------------\n");

    pid=fork();

**if**(pid<0){

**perror**("fork error\n");

**exit**(1);

}**else** **if**(pid==0){

 //相对路径执行，也可以绝对路径执行第一个参数argv是新程序的名字，可以是相对路径也

//可以是绝对路径

**char** \*argv[4]={cmd1,argv1,argv2,NULL};

**if**(execvp(cmd1,argv)<0){

**perror**("execl error\n");

**exit**(1);

        }**else**{

**printf**("execl %s success\n",cmd1);//成功调用不返回，此句不会执行

        }

    }

    wait(0);

}

2.3.3 system函数

#include<stdlib.h>

int system(const char \*command);

返回：成功返回执行命令时的状态，出错-1

功能：简化exec函数的使用

system函数内部构建一个子进程，由子进程调用exec函数

**char** \*cmd1="date";

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

{

**system**("clear");

**system**(cmd1);

}

# 5 网络编程

## 5.1 网络编程基础知识

### 5.1.1 TCP/IP 四层模型

**应用层：**Telnet(远程登陆);FTP(文件传输);SMTP(简单邮件传输);DNS(域名);

SNMP(简单网络管理)http(超文本传输)

**传输层：TCP,UDP**

**网络层：IP，ARP(**地址解析协议)

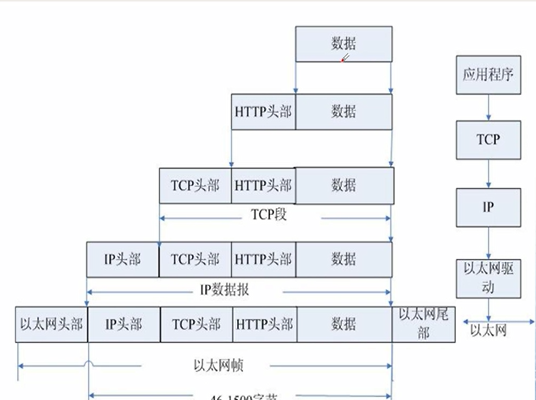
**物理层：**

应用层指定数据格式，传输层指定如何传输数据，网络层负责寻找路径，

物理层负责收发数据。

### 5.1.2 数据封装和IP地址

数据封装：

****

IP 地址：

长度32位，四个字节。使用点分十进制记录。

IP地址组成：网络类型 +网络ID+ 主机ID

192.168.0.112

C类网络类型：前3位110标识是网络类型，最后8位主机地址ID

中间为网络地址ID。前面24组成网段。

### 5.1.3 网络端口号及TCP协议

ip地址对应于一个网卡，那么网络端口号标识与哪个进程通信。

more /etc/services 用于查看各种服务的端口号。2000以下端口号用于特定服务，自己应用的特殊服务应该使用2000以上的端口。

80作为网页服务器的访问端口；21Ftp服务。

## 5.2 TCP编程

### 5.2.1创建socket

#include<sys/socket.h>

int socket(int domain,int type,int protocol)

返回值：成功返回描述符，出错返回-1

参数：

domain： 网络层协议

AF\_INET ipv4

AF\_INET6 ipv6

protocol:

通常为0，表示按给定的域和套接字类型选择默认协议。

type: 传输层协议

SOCK\_STREAM tcp协议

SOCK\_DGRAM udp协议

### 5.2.2 字节序

大端字节序和小端字节序，网络传输采用大端字节序。

uint32\_t htonl(uint32\_t hostlong);//主机字节序转换为网络字节序host to net long

uint16\_t htons(uint16\_t hostshort)// 主机字节序转换为网络字节序16位数据

uint32\_t ntohl(uint32\_t netlong);//网络字节序转换为主机字节序

uint16\_t ntohs(uint16\_t netshort)//

### 5.2.3 地址结构

/\*linux地址格式，不常用\*/

#include <sys/socket.h>

struct sockaddr{

unsigned short sa\_family; // internet 地址簇 ，AF\_XXX

char sa\_data[14]; // 14bbytes 的协议地址

}

sa\_data 包含了一些远程电脑的地址、端口和套接字的数目

sa\_family 一般来说，IPV4使用AF\_INET.

在传递给需要地址结构的函数时，把指向该结构的指针转换成（struct sockaddr \*）传递进去。

//Ipv4地址格式，常用

struct in\_addr{

in\_addr\_t s\_addr; //in\_addr\_t 定义成uint32\_t

}

struct sockaddr\_in{

short int sin\_family; //地址类别AF\_INET.，主机字节序

unsigned short int sin\_port;//端口号 in\_port\_t 定义成uint16\_t，网络字节序

struct in\_addr sin\_addr; //远程ipv4地址

unsigned char sin\_zero[8] //0填充位

}

函数默认是Linux地址格式，需要将网络格式强制转换为linux地址格式。 因为网络格式方便，所以一般使用网络格式进行数据设置，设置完再转换为Linux地址格式。

### 5.2.4 ipv4 地址和字符地址之间的转换

#include <arpa/inet.h>

const char \*inet\_ntop(int domain,const void \*restrict addr,char \*restrict str,socklen\_t size)

返回：成功返回地址字符串指针，出错返回NULL

功能：网络字节序转换成点分十进制。ntop net to point

int inet\_pton(int domain,const char \*restrict str,void \*restrict addr);

返回：成功返回1，无效格式返回0，出错返回-1

功能:点分十进制转换为网络字节序

domain: internet 地址簇，如AF\_INET

addr: internet 地址，32位IPV4地址（网络字节序）

str: 地址字符串（点分十进制）

size: 地址字符串大小

例子：

struct sockaddr\_in sin; //定义一个sockaddr\_in 结构体

char buf[16]; //存放转换后的点分十进制格式ip地址，字符串格式。12个数字，三个点，//1个字符串结束符

memset(&sin,0,sizeof(sin));

sin.sin\_family=AF\_INET; //地址协议类型ipv4

sin.sin\_port=htons((short)3001); //填写端口号(网络字节序)

if(inet\_pton(AF\_INET,”192.168.2.1”,&sin.sin\_addr.s\_addr)<=0)

{//错误处理

}

printf(“%s\n”,inet\_ntop(AF\_INET,&sin.sin\_addr.s\_addr,buf,sizeof(buf)));

### 5.2.5 TCP客户端服务器编程模型

客户端调用序列：

调用socket 函数创建套接字

调用connect 连接服务器端

调用I/O函数(read/write)与服务器通讯

服务器端调用序列

调用socket 函数创建套接字

调用bind绑定本地地址和端口

调用Listen启动监听

调用accept从已连接队列中提取客户连接

调用I/O函数(read/write)与客户端通讯

调用close关闭套接字

绑定地址函数：

#include<sys/socket.h>

int bind(sockfd,const struct sockaddr \*addr,socklen\_t len);

返回：成功返回0，出错返回-1

查找绑定到的套接字的地址：

#include<sys/socket.h>

int getsockname(int sockfd,struct sockaddr \*restrict addr,socklen\_t \*restrict alenp);

返回：成功返回0，出错返回-1

获取对方地址：

#include<sys/socket.h>

int getpeername(int sockfd,struct sockaddr \*restrict addr,socklen\_t \*restrict alenp);

返回：成功返回0，出错返回-1

监听连接：

#inlcude<sys/socket.h>

int listen(int sockfd,int backlog);

返回：成功返回0，出错返回-1。backlog 指定进行客户端连接排队的队列长度。

接收连接：

int accept(int sockfd,struct sockaddr\*restrict addr,socklen\_t \*restrict len);

地址参数用来存储来自客户端的地址。

返回值：fd不是服务器之前创建的socketfd。这是和客户端通信的socketfd。

请求连接：

int commect(int sockfd,const struct sockaddr \*addr,socklen\_t len);

返回：成功返回0，出错返回-1

close((int sockfd) //关闭网络连接

### 5.2.6 基于tcp的服务器网络编程

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<signal.h>

#include<time.h>

int sockfd;

//

void sig\_handler(int signo)

{

**if**(signo==SIGINT){

    printf("server close\n");

//步骤6

    close(sockfd);

    exit(1);

}

}

void out\_addr(struct sockaddr\_in \*clientaddr)

{

    int port=ntohs(clientaddr->sin\_port);

    char ip[16];

    memset(ip,0,sizeof(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,sizeof(ip));

    printf("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

void do\_serverice(int fd)

{

    long t =time(0);

     char \*s =ctime(&t);

    size\_t size=strlen(s)\*sizeof(char);

**if**(write(fd,s,size)!=size){

        perror("write error");

    }

}

int main(int argc, char \*argv[])

{

**if**(argc<2){

        printf("usage:%s #port\n",argv[0]);

        exit(1);

    }

**if**(signal(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

    perror("signal sigint error");

    exit(1);

}

/\* 步骤1：创建socket套接字

 socket 创建在内核中\*/

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

struct sockaddr\_in serveraddr;

memset(&serveraddr,0,sizeof(serveraddr));

serveraddr.sin\_family=AF\_INET;

//端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

serveraddr.sin\_port=htons(atoi(argv[1]));

/\*步骤2

一台主机有多个网络接口和ip地址，如果只关心某个地址的请求连接，可以指定一个具体的ip.要响应所有接口ip就需要一个特殊的地址INADDR\_ANY.

#define INADDR\_ANY (uint32\_t )0x00000000

\*/

serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

**if**(bind(sockfd,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

    perror("bind error");

    exit(1);

}

/\*步骤3

最多同时10个连接。相当listen于创建了一个单独的线程用来处理连接请求

\*/

**if**(listen(sockfd,10)<0){

    perror("listen error");

    exit(1);

}

/\*步骤4：调用accept函数从队列中获取一个客户端的请求连接，并返回新的socket描述符，如果没有客户端连接，调用此函数后会阻塞\*/

struct sockaddr\_in clientaddr;

socklen\_t clientaddr\_len=sizeof(clientaddr);

**while**(1){

    int fd=accept(sockfd,(struct sockaddr\*)&clientaddr,& clientaddr\_len);

**if**(fd<0){

        perror("accept error");

**continue**;

}

/\*步骤5：调用IO函数(read/write)和连接的客户端进行双向通信\*/

out\_addr(&clientaddr);

do\_serverice(fd);

/\*步骤6：通信完成，关闭和客户端连接的socket\*/

close(fd);

}

**return** 0;

}

实验测试：

./time\_tcp\_service 8888 //服务器启动

client:127.0.0.1(34494) connected //测试结果

//开启新终端，使用telnet测试

chao@chao:~$ telnet 127.0.0.1 8888

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character is '^]'.

Mon Dec 10 14:33:45 2018 //测试结果得到服务器发送时间

Connection closed by foreign host.

//使用http测试，在浏览器输入

<http://127.0.0.1:8888/>

Mon Dec 10 15:13:59 2018 //测试结果得到服务器发送时间

注意：服务器需要指定端口号，而客户端不需要指定，由系统分配。

### 5.2.6 基于tcp的客户端网络编程

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

int main(int argc, char \*argv[])

{

**if**(argc<3){

        printf("usage:%s ip port\n",argv[0]);

        exit(1);

    }

    /\*步骤1\*/

     int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

**if**(sockfd<0){

        perror("socket error");

        exit(1);

    }

    struct sockaddr\_in serveraddr;

    memset(&serveraddr,0,sizeof(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    //端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

    serveraddr.sin\_port=htons(atoi(argv[2]));

    //将ip地址转化为网络字节序

    inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    /\*步骤2：客户端调用connect函数连接到服务器端\*/

**if**(connect(sockfd,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0)

    {

        perror("connect error");

        exit(1);

    }

    /\*步骤3：读写\*/

    char buffer[1024];

    memset(buffer,0,sizeof(buffer));

    size\_t size;

**if**((size=read(sockfd,buffer,sizeof(buffer)))<0){

        perror("read error");

        }

**if**(write(STDOUT\_FILENO,buffer,size)!=size){

        perror("write error");

    }

    /\*步骤4：关闭socket\*/

    close(sockfd);

**return** 0;

}

实验：

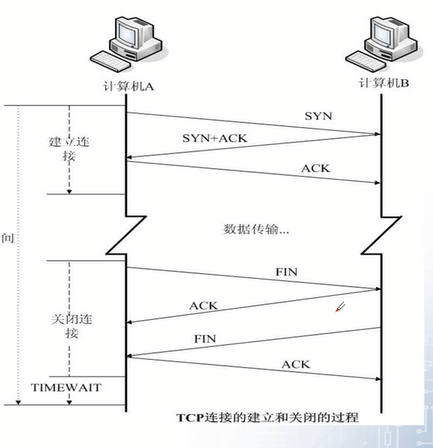
chao@chao:~/linux/network$ ./time\_tcp\_service 8888

client:127.0.0.1(34690) connected

chao@chao:~/linux/network$ ./time\_tcp\_client 127.0.0.1 8888

Mon Dec 10 16:21:34 2018

### 5.2.6 tcp的连接和关闭过程



计算机A是客户端，计算机B是服务器

建立连接需要三次握手：

1. 由客户端发送SYN报文
2. 由服务器返回SYN+ACK报文
3. 最后由客户端发送ACK建立最终连接。

关闭需要两次握手

由提出关闭方发送FIN报文，当对方返回ACK时连接关闭。

gcc -o msg.o -c msg.c //将msg.c编译成模块

### 5.2.7 服务器多并发概念

//连接成功后，向客户端发送系统时间

void do\_serverice(int fd)

{

size\_t len;

char buffer[20];

if((len=read(fd,buff,20))<0){

perror(“read error”);

}

    long t =time(0);

     char \*s =ctime(&t);

    size\_t size=strlen(s)\*sizeof(char);

**if**(write(fd,s,size)!=size){

        perror("write error");

    }

}

实验：

重新编译服务器程序，运行：

./time\_tcp\_service 8888

连接客户端1：

连接客户端2：

./time\_tcp\_client 127.0.0.1 8888

发现只有客户端1连接成功，客户端2没有连接成功，而且服务器和客户端都阻塞。

服务器和客户端都阻塞原因：

服务器等待读客户端发送的数据，而客户端等待服务器发送的数据。因此形成了死锁。

客户端1连接成功，客户端2没有连接成功的原因：

服务器只有一个进程，服务器阻塞因而无法继续执行，也就没有办法接受第二个客户端的连接请求。

解决办法：

服务器针对每一个客户端的请求创建一个子进程 ，该子进程单独服务于该客户端。

还可以使用多线程及IO多路转换

/

### 5.2.8 多进程模型网络编程

#### 自定义应用协议网络通信读写函数：

//头文件msg.h

#ifndef \_\_MSG\_H\_\_

#define \_\_MSG\_H\_\_

#include <sys/types.h>

typedef struct{

    char head[10];//协议头部

    char checknum;//校验码

    char buff[512];

}Msg;

//发送自定义协议的message,发送的数据存放在buff中。

extern int write\_msg(int sockfd,char \*buff,size\_t len);

extern int read\_msg(int sockfd,char \*buff,size\_t len);

#endif

//读写函数msg.c

#include "msg.h"

#include <unistd.h>

#include <string.h>

#include <memory.h>

//计算校验码

static unsigned char msg\_check(Msg \*message)

{

    unsigned char s=0;

    int i;

    for(i=0;i<sizeof(message->head);i++){

        s+=message->head[i];

    }

    for(i=0;i<sizeof(message->buff);i++){

        s+=message->buff[i];

    }

    return s;

}

//发送自定义协议的message,发送的数据存放在buff中。

 int write\_msg(int sockfd,char \*buff,size\_t len)

{

    Msg message;

    memset(&message,0,sizeof(message));

    strcpy(message.head,"iotek2012");

    memcpy(message.buff,buff,len);

    message.checknum=msg\_check(&message);

    if(write(sockfd,&message,sizeof(message))!=sizeof(message)){

        return -1;

    }

}

 int read\_msg(int sockfd,char \*buff,size\_t len)

{

    Msg message;

    memset(&message,0,sizeof(message));

    size\_t size;

    if((size=read(sockfd,&message,sizeof(message)))<0){

        return -1;

    }else if(size==0){

    return 0;

    }

    unsigned char s=msg\_check(&message);

    //验证数据

    if((s==(unsigned char)message.checknum)&&(!strcmp("iotek2012",message.head)))

    {

        memcpy(buff,message.buff,len);

        return sizeof(message);

    }

    return -1;

}

#### 自定义应用协议网络通信服务器程序：

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<signal.h>

#include<time.h>

#include "msg.h"

#include<errno.h>

#include <sys/wait.h>

**int** sockfd;

//

**void** sig\_handler(**int** signo)

{

**if**(signo==SIGINT){

**printf**("server close\n");

        //步骤6

        close(sockfd);

**exit**(1);

    }

**if**(signo==SIGCHLD){

**printf**("child process deaded...\n");

        wait(0);

    }

}

**void** out\_addr(**struct** sockaddr\_in \*clientaddr)

{

**int** port=ntohs(clientaddr->sin\_port);

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

void do\_serverice(int fd)

{

    /\*和客户端双向通信

\*/

    char buff[512];

    while(1){//循环读直到数据读完

        memset(buff,0,sizeof(buff));

        printf("start read and write...\n");

        size\_t size;

        if((size=read\_msg(fd,buff,sizeof(buff)))<0){

            perror("protocal error");

            break;

        }else if(size==0){//注意：如果读过程客户端断开，那么读完数据，读取长度0

            break;

        }else{

            printf("%s\n",buff);

            if(write\_msg(fd,buff,sizeof(buff))<0){

//注意：如果读过程客户端断开，那么写错误EPIPE，类似管道

                if(errno==EPIPE){

                    break;

                }

                perror("protocal error");

                }

        }

    }

}

int main(int argc, char \*argv[])

{

    if(argc<2){

        printf("usage:%s #port\n",argv[0]);

        exit(1);

    }

if(signal(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

    perror("signal sigint error");

    exit(1);

}

if(signal(SIGCHLD,sig\_handler)==SIG\_ERR){//注册信号处理函数

    perror("signal sigchld error");

    exit(1);

}

/\* 步骤1：创建socket套接字

 socket 创建在内核中\*/

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

struct sockaddr\_in serveraddr;

memset(&serveraddr,0,sizeof(serveraddr));

serveraddr.sin\_family=AF\_INET;

//端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

serveraddr.sin\_port=htons(atoi(argv[1]));

/\*步骤2

一台主机有多个网络接口和ip地址，如果只关心某个地址的请求连接，可以指定一个具体的ip.要响应所有接口ip就需要一个特殊的地址INADDR\_ANY.

#define INADDR\_ANY (uint32\_t )0x00000000

\*/

serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

if(bind(sockfd,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

    perror("bind error");

    exit(1);

}

/\*步骤3

最多同时10个连接。相当listen于创建了一个单独的线程用来处理连接请求

\*/

if(listen(sockfd,10)<0){

    perror("listen error");

    exit(1);

}

/\*步骤4：调用accept函数从队列中获取一个客户端的请求连接，并返回新的socket描述符，如果没有客户端连接，调用此函数后会阻塞\*/

struct sockaddr\_in clientaddr;

socklen\_t clientaddr\_len=sizeof(clientaddr);

while(1){

    int fd=accept(sockfd,(struct sockaddr\*)&clientaddr,& clientaddr\_len);

if(fd<0){

        perror("accept error");

        continue;

}

/\*步骤5：调用IO函数(read/write)和连接的客户端进行双向通信

    启动新进程

\*/

    pid\_t pid=fork();

    if(pid<0){

        continue;

    }else if(pid==0){ //子进程继承父进程fd

        out\_addr(&clientaddr);

        do\_serverice(fd);

    /\*步骤6：通信完成，关闭和客户端连接的socket\*/

        close(fd);

        break;

    }else {          //父进程

        close(fd); //不再需要使用fd

    }

}

return 0;

}

#### 自定义应用协议网络通信客户端程序：

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

#include"msg.h"

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

{

**if**(argc<3){

**printf**("usage:%s ip port\n",argv[0]);

**exit**(1);

    }

    /\*步骤1\*/

**int** sockfd=socket(AF\_INET,SOCK\_STREAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    //端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

    serveraddr.sin\_port=htons(**atoi**(argv[2]));

    //将ip地址转化为网络字节序

    inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    /\*步骤2：客户端调用connect函数连接到服务器端\*/

**if**(connect(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0)

    {

**perror**("connect error");

**exit**(1);

    }

    /\*步骤3：读写\*/

**char** buff[512];

**size\_t** size;

**char** \*prompt=">";

**while**(1){

**memset**(buff,0,**sizeof**(buff));

        write(STDOUT\_FILENO,prompt,1);

        size=read(STDIN\_FILENO,buff,**sizeof**(buff));

**if**(size<0) **continue**;

        buff[size-0] ='\0';

**if**(write\_msg(sockfd,buff,**sizeof**(buff))<0){

**perror**("write msg error");

**continue**;

        }**else**{ //写成功

**if**(read\_msg(sockfd,buff,**sizeof**(buff))<0){

**perror**("read msg error");

**continue**;

            }**else**{//读成功

**printf**("%s\n",buff);

            }

        }

   }

    /\*步骤4：关闭socket\*/

    close(sockfd);

**return** 0;

}

#### 多客户端连接实验：

服务器启动：

root@chao:/home/chao/linux/network/msg# ./echo\_tcp\_service 8888

client:127.0.0.1(39572) connected

start read and write...

hello world //接收

start read and write...

client:127.0.0.1(39580) connected

start read and write...

hello china //接收

start read and write...

hello china1 //接收

start read and write...

hello world1 //接收

start read and write...

客户端1连接：

chao@chao:~$ cd linux/network/msg/

chao@chao:~/linux/network/msg$ ./echo\_tcp\_client 127.0.0.1 8888

>hello world //发送

hello world

>hello world1

hello world1

客户端2连接

chao@chao:~$ cd linux/network/msg/

chao@chao:~/linux/network/msg$ ./echo\_tcp\_client 127.0.0.1 8888

>hello china

hello china

>hello china1

hello china1

### 5.2.9 多线程模型网络编程

以分离线程模式启动线程，当子线程结束会自释放占用的资源。不能用pthread\_join回收子线程。因为pthread\_join会导致主控线程阻塞，而无法接受新的连接。

#### 基于线程的服务器程序

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<signal.h>

#include<time.h>

#include "msg.h"

#include<errno.h>

#include <sys/wait.h>

#include <pthread.h>

**int** sockfd;

//

**void** sig\_handler(**int** signo)

{

**if**(signo==SIGINT){

**printf**("server close\n");

        //步骤6

        close(sockfd);

**exit**(1);

    }

}

**void** out\_addr(**struct** sockaddr\_in \*clientaddr)

{

**int** port=ntohs(clientaddr->sin\_port);

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

**void** do\_serverice(**int** fd)

{

    /\*和客户端双向通信\*/

**char** buff[512];

**while**(1){//循环读直到数据读完

**memset**(buff,0,**sizeof**(buff));

**size\_t** size;

**if**((size=read\_msg(fd,buff,**sizeof**(buff)))<0){

**perror**("protocal error");

**break**;

        }**else** **if**(size==0){//注意：如果读过程客户端断开，那么读完数据，读取长度0

**break**;

        }**else**{

**printf**("%s\n",buff);

**if**(write\_msg(fd,buff,**sizeof**(buff))<0){

    //注意：如果读过程客户端断开，那么写错误EPIPE，类似管道

**if**(**errno**==EPIPE){

**break**;

                }

**perror**("protocal error");

                }

        }

    }

}

**void** out\_fd(**int** fd)

{

**struct** sockaddr\_in addr;

    socklen\_t len =**sizeof**(addr);

//从fd中获得连接客户端的相关信息

**if**(getpeername(fd,(**struct** sockaddr\*)&addr,&len)<0){

**perror**("getpeername error");

**return** ;

    }

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

**int** port=ntohs(addr.sin\_port);//转为主机字节序

    inet\_ntop(AF\_INET,&addr.sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("%16s(%5d)closed!\n",ip,port);

}

**void** \*th\_fn(**void**\*arg)

{

**int** fd=(**int**)arg;

    do\_serverice(fd);

    out\_fd(fd);

    close(fd);

**return** (**void**\*)0;

}

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

{

**if**(argc<2){

**printf**("usage:%s #port\n",argv[0]);

**exit**(1);

    }

**if**(**signal**(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

**perror**("signal sigint error");

**exit**(1);

    }

/\* 步骤1：创建socket套接字

 socket 创建在内核中\*/

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

serveraddr.sin\_family=AF\_INET;

//端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

serveraddr.sin\_port=htons(**atoi**(argv[1]));

/\*步骤2

一台主机有多个网络接口和ip地址，如果只关心某个地址的请求连接，可以指定一个具体的ip.要响应所有接口ip就需要一个特殊的地址INADDR\_ANY.

#define INADDR\_ANY (uint32\_t )0x00000000

\*/

serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

**if**(bind(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("bind error");

**exit**(1);

}

/\*步骤3

最多同时10个连接。相当listen于创建了一个单独的线程用来处理连接请求

\*/

**if**(listen(sockfd,10)<0){

**perror**("listen error");

**exit**(1);

}

/\*步骤4：调用accept函数从队列中获取一个客户端的请求连接，并返回新的socket描述符，如果没有客户端连接，调用此函数后会阻塞\*/

pthread\_attr\_t attr;

    pthread\_attr\_init(&attr);

    pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED);

**while**(1){

**int** fd=accept(sockfd,NULL,NULL);

**if**(fd<0){

**perror**("accept error");

**continue**;

        }

/\*步骤5：调用IO函数(read/write)和连接的客户端进行双向通信

    启动线程

\*/

        pthread\_t th;

**int** err;

**if**((err=pthread\_create(&th,&attr,th\_fn,(**void**\*)fd))!=0){

**perror**("pthread\_creat error");

        }

       pthread\_attr\_destroy(&attr);

    }

**return** 0;

}

#### 基于线程的客户程序

因为客户端不需要多线程，因此程序不用改，可以使用多进程的客户端程序。

实验：

先启动服务器，然后由客户端发送数据给服务器

chao@chao:~/linux/network/echo\_tcp\_thread$ ./echo\_tcp\_service 8888

hello china 0

hello world 0

hello world 1

hello china 1

chao@chao:~/linux/network/echo\_tcp\_thread$ ./echo\_tcp\_client 127.0.0.1 8888

>hello world 0

hello world 0

>hello world 1

hello world 1

chao@chao:~/linux/network/echo\_tcp\_thread$ ./echo\_tcp\_client 127.0.0.1 8888

>hello china 0

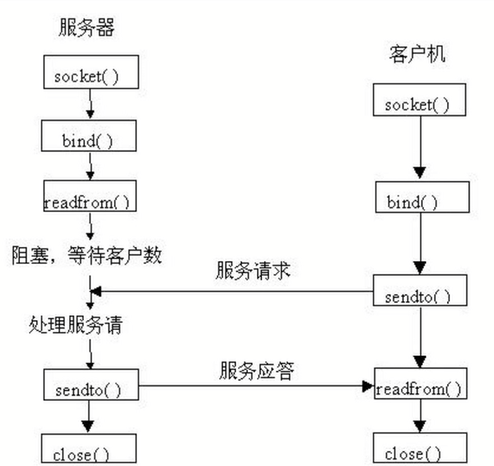
hello china 0

>hello china 1

hello china 1

# 5.3 UDP编程

## 5.3.1 UDP 传输流程



注意：客户端bind()并不是必要的。

## 5.3.2 udp 常用系统调用

#include<sys/socket.h>

ssize\_t send(int sockfd,const void \*buf,size\_t nbytes,int flag);//sockfd本机创建的sockfd

返回：成功返回发送字节数，出错返回-1

ssize\_t sento(int sockfd,const void \*buf,size\_t nbytes,int flag,const struct sockaddr \*destaddr,

socklen\_t destlen);

返回：成功返回发送的字节数，出错返回-1

#inlcude<sys/socket.h>

ssize\_t recv(int sockfd,void \*buf,size\_t nbytes,int flag);

ssize\_t recvfrom(int sockfd,void \*restrict buf,size\_t len,int flag,struct sockaddr \*restrict addr,

socklen\_t \*restrict addrlen);

返回：成功返回接收的字节数，出错返回-1

### 5.3.3 udp编程

#### 服务器程序：

#include<sys/socket.h>

#include<netdb.h>

#include <arpa/inet.h>

#include<unistd.h>

 #include<string.h>

 #include<stdlib.h>

#include<stdio.h>

#include<signal.h>

#include<time.h>

**int** sockfd;

**void** sig\_handler(**int** signo)

{

**if**(signo==SIGINT){

**printf**("server close\n");

        close(sockfd);

**exit**(1);

    }

}

**void** out\_addr(**struct** sockaddr\_in \*clientaddr)

{

**int** port=ntohs(clientaddr->sin\_port);

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

**void** do\_service(**int** fd)

{

**struct** sockaddr\_in clientaddr;

    socklen\_t len=**sizeof**(clientaddr);

**char** buffer[1024];

**memset**(buffer,0,**sizeof**(buffer));

**if**(recvfrom(sockfd,buffer,**sizeof**(buffer),0,(**struct** sockaddr\*)&clientaddr,&len)<0)

        {

**perror**("recvfrom error\n");

        }**else**{

            out\_addr(&clientaddr);

**printf**("client send info: %s\n",buffer);

**long** **int** t =**time**(0);

**char** \*ptr=**ctime**(&t);

**size\_t** size=**strlen**(ptr)\***sizeof**(**char**);

**if**(sendto(sockfd,ptr,size,0,(**struct** sockaddr\*)&clientaddr,len)<0){

**perror**("send error\n");

            }

        }

}

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

{

**if**(argc<2){

**printf**("usage:%s #port\n",argv[0]);

**exit**(1);

    }

**if**(**signal**(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

**perror**("signal sigint error");

**exit**(1);

    }

    //步骤1创建socket

    sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**int** ret;

**int** opt=1;

    //设置套接字选项，使刚刚停掉的端口可以立即使用，还可以添加超时

**if**(setsockopt(sockfd,SOL\_SOCKET,SO\_REUSEADDR,&opt,**sizeof**(opt))<0){

**perror**("setsockopt error");

**exit**(1);

    }

**struct** timeval timeout={10,0};

//即timeout={4,0};或者timeout.tv\_sec=4; timeout.tv\_usec=0;

//设置接收超时

setsockopt(sockfd,SOL\_SOCKET,SO\_RCVTIMEO,(**char**\*)&timeout,**sizeof**(**struct** timeval));

    //步骤2调用bind函数对socket和地址进行绑定

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    serveraddr.sin\_port=htons(**atoi**(argv[1]));

    serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

**if**(bind(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("bind error");

**exit**(1);

    }

**while**(1){

        do\_service(sockfd);

    }

}

#### 客户端程序：

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

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

{

**if**(argc<3){

**printf**("usage:%s ip port\n",argv[0]);

**exit**(1);

    }

**int** sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    serveraddr.sin\_port=htons(**atoi**(argv[2]));

    inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    /\*

        connect 并不会发起真正的连接，只是将服务器地址保存在内核中，这样以后发送

和接收函数都可以不指定服务器地址。也不会接受除该服务器以外的其他计算机发来的信息。

这样可以避免不必要的错误。

    \*/

**if**(connect(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("connect error");

**exit**(1);

    }

**char** buffer[1024]="hello iotek";

    /\*if(sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

        perror("sendto error\n");

        exit(1);\*/

**if**(send(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("send error\n");

**exit**(1);

    }**else** {

**memset**(buffer,0,**sizeof**(buffer));

        //已经有服务器信息了，不需要再指定

**if**(recv(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("recv error");

**exit**(1);

        }**else**{

**printf**("%s",buffer);

        }

    }

    close(sockfd);

**return** 0;

}

实验：

chao@chao:~/linux/network/time\_udp$ ./time\_udp\_server 8888

recvfrom error //由于添加了超时未接收数据错误

: Resource temporarily unavailable //由于添加了超时未接收数据错误

client:127.0.0.1(38696) connected //客户端连接

client send info: hello iotek //客户端连接

recvfrom error

: Resource temporarily unavailable

chao@chao:~/linux/network/time\_udp$ ./time\_udp\_client 127.0.0.1 8888

Thu Dec 13 10:06:06 2018

### 5.3.4 域名解析

#### 本地虚拟机ip和域名对应关系：

chao@chao:~$ more /etc/hosts

127.0.0.1 localhost

127.0.1.1 chao-virtual-machine

#### 域名解析函数：

struct hostent{

char \*h\_name; //正式主机名

char \*\*h\_aliases; //别名，字符串数组

int h\_addrtype; //协议类型

int h\_length; //网络地址大小

char \*\*h\_addr\_list; //指向网络地址的指针

}

#include<netdb.h>

struct hostent \*gethostent(void);//循环获取/etc/hosts里面的每一行

//只获取匹配的行，但是该函数是线程不稳定的，容易出现问题，不建议用

struct hostent \*gethostbyname(const char \*hostname);

void sethostent(int stayopen);

void endhostent(void);

#### 给虚拟机设置域名

第一个chao 是主机名；第二个www.chao-ubuntu18.com.cn称为别名，还可以设置第三个别名，也就是有多个别名。

root@chao:/home/chao# gedit /etc/hosts

添加一行： 192.168.1.118 chao www.chao-ubuntu18.com.cn

域名解析编程实例：

#include<netdb.h>

#include<stdio.h>

#include <stdlib.h>

#include<memory.h>

#include <arpa/inet.h>

**void** out\_addr(**struct** hostent \*h)

{

**printf**("hsotname:%s \n",h->h\_name);

**printf**("addrtype:%s\n",h->h\_addrtype==AF\_INET?"IPV4":"IPV6");

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

    inet\_ntop(h->h\_addrtype,h->h\_addr\_list[0],ip,**sizeof**(ip));

**printf**("ip address:%s\n",ip);

}

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

{

**if**(argc<2){

**fprintf**(stderr,"usage:%s host\n",argv[0]);

**exit**(1);

    }

**struct** hostent \*h;

**while**((h=gethostent())!=NULL){

**if**(!**strcmp**(argv[1],h->h\_name)){  //判断主机名是否相等

            out\_addr(h);

**exit**(0);

        }**else**{

**int** i=0;

**while**(h->h\_aliases[i]!=NULL){

**if**(!**strcmp**(argv[1],h->h\_aliases[i])){ //判断别名是否相等

                    out\_addr(h);

**exit**(0);

                }

                i++;

            }

        }

    }

    endhostent();//释放调用gethhostent 占用的资源

**printf**("no %s exist\n",argv[1]);

**return** 0;

}

#### 实验：

root@chao:/home/chao/linux/network/dns# gcc gethost2.c -o gethost2

root@chao:/home/chao/linux/network/dns# ./gethost2 chao

hsotname:chao

addrtype:IPV4

ip address:192.168.1.118

root@chao:/home/chao/linux/network/dns# ./gethost2 www.chao-ubuntu18.com.cn

hsotname:chao

addrtype:IPV4

ip address:192.168.1.118

### 5.3.5 通过域名访问网络编程

客户端程序需要更改，服务器程序不变。下面程序是由time\_udp\_client更改

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

**int** is\_host(**struct** hostent \*host,**char** \*name)

{

**if**(!**strcmp**(host->h\_name,name))

**return** 1;

**int** i=0;

**while**(host->h\_aliases[i]!=NULL){

**if**(!**strcmp**(host->h\_aliases[i],name))

**return** 1;

            i++;

        }

**return** 0;

}

unsigned **int** get\_ip\_by\_name(**char** \*name)

{

    unsigned **int** ip=0;

**struct** hostent \*host;

**while**((host=gethostent())!=NULL){

**if**(is\_host(host,name)){

**memcpy**(&ip,host->h\_addr\_list[0],4);

**break**;

        }

    }

    endhostent();

**return** ip;

}

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

{

**if**(argc<3){

**printf**("usage:%s ip port\n",argv[0]);

**exit**(1);

    }

**int** sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    serveraddr.sin\_port=htons(**atoi**(argv[2]));

    //inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    unsigned **int** ip=get\_ip\_by\_name(argv[1]);//得到网络字节序地址

**if**(ip!=0){

        serveraddr.sin\_addr.s\_addr=ip;

    }**else**{

        inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    }

 /\*

        connect 并不会发起真正的连接，只是将服务器地址保存在内核中，这样以后发送

和接收函数都可以不指定服务器地址。也不会接受除该服务器以外的其他计算机发来的信息。

这样可以避免不必要的错误。

    \*/

**if**(connect(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("connect error");

**exit**(1);

    }

**char** buffer[1024]="hello iotek";

    /\*if(sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

        perror("sendto error\n");

        exit(1);\*/

**if**(send(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("send error\n");

**exit**(1);

    }**else** {

**memset**(buffer,0,**sizeof**(buffer));

        //已经有服务器信息了，不需要再指定

**if**(recv(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("recv error");

**exit**(1);

        }**else**{

**printf**("%s",buffer);

        }

    }

    close(sockfd);

**return** 0;

}

实验：

./time\_udp\_severce 8888

./time\_udp\_client chao 8888

### 5.3.6 广播

192.168.1.255是192.168.1子网的广播地址

# 6 cmake

## 6.1 cmake简介

CMake 是一个跨平台的自动化建构系统,它使用一个名为 CMakeLists.txt 的文件来描述构建过程,可以产生标准的构建文件,如 Unix 的 Makefile 或Windows Visual C++ 的 projects/workspaces 。文件 CMakeLists.txt 需要手工编写,也可以通过编写脚本进行半自动的生成。CMake 提供了比 autoconfig 更简洁的语法。在 linux 平台下使用 CMake 生成 Makefile 并编译的流程如下:

1、编写 CmakeLists.txt。

2、执行命令“cmake PATH”或者“ccmake PATH”生成 Makefile ( PATH 是 CMakeLists.txt 所在的目录 )。

3、使用 make 命令进行编译。

## 6.2cmake 安装

准备工作：官网下载[cmake-3.6.3.tar.gz](https://cmake.org/files/v3.6/cmake-3.6.3.tar.gz)（https://cmake.org/download/）

1.解压文件tar -xvf cmake-3.6.3.tar.gz，并修改文件权限chmod -R 777 cmake-3.6.3

2.检测gcc和g++是否安装，如果没有则需安装gcc-g++：sudo apt-get install build-essential（或者直接执行这两条命令sudo apt-get install gcc,sudo apt-get install g++）

3.进入cmake-3.6.3 进入命令 cd cmake-3.6.3

4.执行sudo ./bootstrap   
5.执行sudo make   
6.执行 sudo make install   
7.执行 cmake –version，返回cmake版本信息，则说明安装成功

## 6.3 cmake使用

### 6.3.1 单文件工程

main.c文件

#include<stdio.h>

int main(void)

{

printf("hello iotek\n");

while(1)

{

sleep(1);

}

return 0;

}

为了构建该项目,我们需要编写文件 CMakeLists.txt， 并将其与 main.cpp 放在 同一个目录。CMakeLists.txt：

PROJECT(main)

CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6)

AUX\_SOURCE\_DIRECTORY(. DIR\_SRCS)

ADD\_EXECUTABLE(main ${DIR\_SRCS})

CMakeLists.txt 的语法比较简单,由命令、注释和空格组成,其中命令是不区分大小写的,符号"#"后面的内容被认为是注释。命令由命令名称、小括号和参数组成,参数之间使用空格进行间隔。例如对于清单2的 CMakeLists.txt 文件:第一行是一条命令,名称是 PROJECT ,参数是 main ,该命令表示项目的名称是 main 。第二行的命令限定了 CMake 的版本。第三行使用命令 AUX\_SOURCE\_DIRECTORY 将当前目录中的源文件名称赋值给变量 DIR\_SRCS 。 CMake 手册中对命令 AUX\_SOURCE\_DIRECTORY 的描述如下:

|  |  |
| --- | --- |
|  | aux\_source\_directory(<dir> <variable>) |

该命令会把参数 <dir> 中所有的源文件名称赋值给参数 <variable> 。 第四行使用命令 ADD\_EXECUTABLE 指示变量 DIR\_SRCS 中的源文件需要编译 成一个名称为 main 的可执行文件。

完成了文件 CMakeLists.txt 的编写后需要使用 cmake 或 ccmake 命令生成Makefile 。 ccmake 与命令 cmake 的不同之处在于 ccmake 提供了一个图形化的操作界面。cmake 命令的执行方式如下:

|  |  |
| --- | --- |
|  | cmake [options] <path-to-source> |

这里我们进入了 main.cpp 所在的目录后执行 “cmake .” 后就可以得到 Makefile 并使用 make 进行编译。

### 6.3.2处理多源文件目录的方法

CMake 处理源代码分布在不同目录中的情况也十分简单。现假设我们的源代码分布情况如下:

main.c

src/test.h

src/test.c

main.c:

#include<stdio.h>

#include<unistd.h>

#include"src/test.h"

int main(void)

{

printf("hello main\n");

test();

while(1)

{

sleep(1);

}

return 0;

}

test.c

#include<stdio.h>

#include"test.h"

void test()

{

printf("hello test\n");

}

test.h

#ifndef \_\_TEST\_H

#define \_\_TEST\_H

void test();

#endif

其中 src 目录下的文件要编译成一个链接库。

第一步，项目主目录中的 CMakeLists.txt

在目录 step2 中创建文件 CMakeLists.txt 。文件内容如下:

清单 3 目录 step2 中的 CMakeLists.txt

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 1 PROJECT(main)  2 CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6)  3 ADD\_SUBDIRECTORY( src )  4 AUX\_SOURCE\_DIRECTORY(. DIR\_SRCS)  5 ADD\_EXECUTABLE(main ${DIR\_SRCS}  )  6 TARGET\_LINK\_LIBRARIES( main Test ) |

相对于清单 2，该文件添加了下面的内容: 第三行，使用命令 ADD\_SUBDIRECTORY 指明本项目包含一个子目录 src 。第六行，使用命令 TARGET\_LINK\_LIBRARIES 指明可执行文件 main 需要连接一个名为Test的链接库 。

第二步，子目录中的 CmakeLists.txt

在子目录 src 中创建 CmakeLists.txt。文件内容如下:

清单 4. 目录 src 中的 CmakeLists.txt

|  |  |
| --- | --- |
| 1  2 | 1 AUX\_SOURCE\_DIRECTORY(. DIR\_TEST1\_SRCS)  2 ADD\_LIBRARY ( Test ${DIR\_TEST1\_SRCS}) |

在该文件中使用命令 ADD\_LIBRARY 将 src 目录中的源文件编译为共享库。

第三步，执行 cmake

至此我们完成了项目中所有 CMakeLists.txt 文件的编写,进入目录 step2 中依次执行命令 “cmake .” 和 “make”

在执行 cmake 的过程中,首先解析根目录 中的 CMakeLists.txt ,当程序执行命令 ADD\_SUBDIRECTORY( src ) 时进入目录 src 对其中的 CMakeLists.txt 进行解析。

# 7 Makefile 编写

**编译一个程序需要四个步骤：**

1. 预处理
2. 编译
3. 汇编
4. 链接

**写一个好的makefile必须包含以下功能：**

1. 先编译再链接

2、修改了哪个文件就单独编译哪个文件再链接

3、修改头文件，就单独编译用到了此头文件的文件再链接

要实现上面的编译功能，就需要编写makefile

**什么时候才会重新编译一个文件？**

1、”依赖”文件比”目标”文件新

2、没有”目标”这个文件

**makefile 的例子**

例子1：

test:a.c b.c a.h

gcc -o test a.c b.c

其中冒号前面的test被称为目标后面的a.c,b.c,a.h被称为依赖,其中任何一个修改都会导致test这个目标重新编译。这就是重新编译的条件1

例子2：

test :a.o b.o

gcc -o test a.o b.o

a.o :a.c

gcc -c -o a.o a.c //-c表示编译不链接

b.o:b.c

gcc -c -o b.o b.c

如果a.o 在之前的编译时不存在，那么重新编译时就会编译它，这就是重新编译的条件2

上面makeflie的缺点是如果有很多个文件就必须写很多行的编译文件。可以修改如下：

例子3：

test :a.o b.o

gcc -o test a.o b.o

%.o:%.c

gcc -c -o $@ $< //$@表示目标；$< 表示第一个依赖；$^表示所有的依赖

上面这个makefile的缺陷是如果修改了.h文件那么使用.h文件的目标不会被重新编译修改为：

例子4：

test :a.o b.o

gcc -o test a.o b.o

a.o:a.c a.h

%.o:%.c

gcc -c -o $@ $<

上面这个例子也有缺陷，就是需要手工确定依赖的.h文件，需要修改

例子5：

test :a.o b.o

gcc -o test a.o b.o

a.o:a.c a.h

%.o:%.c

gcc -Wp,-MD, $@.d -c -o $@ $<//-Wp,-MD,会生自动成目标依赖的头文件列表文件$@.d文件名为a.o.d

clean:

rm \*.o test

上面程序依然有缺陷

例子6：

objs := a.o b.o

test:$(objs)

gcc -o test $^

# .a.o.d .b.o.d

#对objs里面的每一个文件f，如果是.开头的隐藏文件都加一个后缀.d，执行完得到文

#件.a.o.d .b.o.d

dep\_files := $(foreach f,$(objs),.$(f).d)

#取出已经有的.d文件，被展开为已经存在的、使用空格分开的、匹配此模式的所有文件列表

dep\_files := $(wildcard $(dep\_files))

#如果dep\_files不为空包含

ifneq ($(dep\_files),)

include $(dep\_files)

endif

%.o : %.c

gcc -Wp,-MD,.$@.d -c -o $@ $<

clean:

rm \*.o test

**实现一个可以编译一个工程的Makefile**

子目录里面的makefile：

# 罗列里面所有的文件，包括子目录

obj-y += disp\_manager.o

obj-y += fb.o

obj-y += test/ #该目录的子目录test下的所有.o文件

顶层目录的makefile

CROSS\_COMPILE = arm-linux-

AS = $(CROSS\_COMPILE)as

LD = $(CROSS\_COMPILE)ld

CC = $(CROSS\_COMPILE)gcc

CPP = $(CC) -E

AR = $(CROSS\_COMPILE)ar

NM = $(CROSS\_COMPILE)nm

STRIP = $(CROSS\_COMPILE)strip

OBJCOPY = $(CROSS\_COMPILE)objcopy

OBJDUMP = $(CROSS\_COMPILE)objdump

# export导出这些变量让其他文件引用

export AS LD CC CPP AR NM

export STRIP OBJCOPY OBJDUMP

# -Wall列出警告，-O2优化级别2，-g 显示调试信息,:=这种等号可以追加选项，并且值固定不变，而=在编#译使用时才确定值，

CFLAGS := -Wall -O2 -g

#头文件目录，在当前目录（就是shell执行pwd获得的目录）里面的include

CFLAGS += -I $(shell pwd)/include

#链接数学库和freetype库

LDFLAGS := -lm -lfreetype

export CFLAGS LDFLAGS

TOPDIR := $(shell pwd)

export TOPDIR

TARGET := show\_file

obj-y += main.o

obj-y += display/

obj-y += draw/

obj-y += encoding/

obj-y += fonts/

#执行./目录执行Makefile.build文件，把文件拷贝靠该makefile内部执行

all :

make -C ./ -f $(TOPDIR)/Makefile.build

$(CC) $(LDFLAGS) -o $(TARGET) built-in.o

clean:

rm -f $(shell find -name "\*.o")

rm -f $(TARGET)

distclean:

rm -f $(shell find -name "\*.o")

rm -f $(shell find -name "\*.d") #删除.d文件

rm -f $(TARGET)

Makefile.BUILD文件

PHONY := \_\_build

\_\_build:

obj-y :=

subdir-y :=

include Makefile

# $(filter %/, $(obj-y))只保留所有/的项，也就是子目录，例子：

# obj-y := a.o b.o c/ d/

# $(filter %/, $(obj-y)) : c/ d/

# \_\_subdir-y : c d

# subdir-y : c d

\_\_subdir-y := $(patsubst %/,%,$(filter %/, $(obj-y)))

subdir-y += $(\_\_subdir-y)

# c/built-in.o d/built-in.o

# 编译\

subdir\_objs := $(foreach f,$(subdir-y),$(f)/built-in.o)

# a.o b.o 把符合模式的去掉，去掉带/的，就是去掉子目录

cur\_objs := $(filter-out %/, $(obj-y))

dep\_files := $(foreach f,$(cur\_objs),.$(f).d)

dep\_files := $(wildcard $(dep\_files))

ifneq ($(dep\_files),)

include $(dep\_files)

endif

PHONY += $(subdir-y)

\_\_build : $(subdir-y) built-in.o

$(subdir-y):

make -C $@ -f $(TOPDIR)/Makefile.build

built-in.o : $(cur\_objs) $(subdir\_objs)

$(LD) -r -o $@ $^

dep\_file = .$@.d

%.o : %.c

$(CC) $(CFLAGS) -Wp,-MD,$(dep\_file) -c -o $@ $<

.PHONY : $(PHONY)

**makefile种使用的函数**

$(filter PATTERN…,TEXT)

函数名称：过滤函数—filter。

函数功能：过滤掉字串“TEXT”中所有不符合模式“PATTERN”的单词，保留所

有符合此模式的单词。可以使用多个模式。模式中一般需要包含模式字

符“%”。存在多个模式时，模式表达式之间使用空格分割。

返回值：空格分割的“TEXT”字串中所有符合模式“PATTERN”的字串。

函数说明：“filter”函数可以用来去除一个变量中的某些字符串，我们下边的例子中

就是用到了此函数。

示例：

sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo

使用“$(filter %.c %.s,$(sources))”的返回值给 cc 来编译生成目标“foo”，函数返回

值为“foo.c bar.c baz.s”

格式：$(patsubst <pattern>,<replacement>,<text> )   
名称：模式字符串替换函数——patsubst。  
功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）  
返回：函数返回被替换过后的字符串。

示例：

$(patsubst %.c,%.o,x.c.c bar.c)

把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.o”

# 多进程和多线程思考

如果总结多进程和多线程的区别，你肯定能说，前者开销大，后者开销较小。确实，这就是最基本的区别。线程函数的可重入性：  
说到函数的可重入，和线程安全，我偷懒了，引用网上的一些总结。  
  
线程安全：概念比较直观。一般说来，一个函数被称为线程安全的，当且仅当被多个并发线程反复调用时，它会一直产生正确的结果。  
  
可重入：概念基本没有比较正式的完整解释，但是它比线程安全要求更严格。根据经验，所谓“重入”，常见的情况是，程序执行到某个函数foo()时，收到信 号，于是暂停目前正在执行的函数，转到信号处理函数，而这个信号处理函数的执行过程中，又恰恰也会进入到刚刚执行的函数foo()，这样便发生了所谓的重 入。此时如果foo()能够正确的运行，而且处理完成后，之前暂停的foo()也能够正确运行，则说明它是可重入的。  
线程安全的条件：  
要确保函数线程安全，主要需要考虑的是线程之间的共享变量。属于同一进程的不同线程会共享进程内存空间中的全局区和堆，而私有的线程空间则主要包括栈和寄 存器。因此，对于同一进程的不同线程来说，每个线程的局部变量都是私有的，而全局变量、局部静态变量、分配于堆的变量都是共享的。在对这些共享变量进行访 问时，如果要保证线程安全，则必须通过加锁的方式。  
可重入的判断条件：  
  
要确保函数可重入，需满足一下几个条件：  
1、不在函数内部使用静态或全局数据   
2、不返回静态或全局数据，所有数据都由函数的调用者提供。   
3、使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据。  
4、不调用不可重入函数。  
  
可重入与线程安全并不等同，一般说来，可重入的函数一定是线程安全的，但反过来不一定成立。它们的关系可用下图来表示：  
  
比如：strtok函数是既不可重入的，也不是线程安全的；加锁的strtok不是可重入的，但线程安全；而strtok\_r既是可重入的，也是线程安全的。  
  
如果我们的线程函数不是线程安全的，那在多线程调用的情况下，可能导致的后果是显而易见的——共享变量的值由于不同线程的访问，可能发生不可预料的变化，进而导致程序的错误，甚至崩溃。