# LINUXC

## 编译

c语言用gcc

c++用g++

gcc 文件名 预处理，编译，汇编，链接生成a.out

通常所说的编译指前三步

-选项

-o 新名称 指定生成文件的名称

-c 只编译不链接(.o)

-S大写 生成汇编文件（.s）

-E 只进行预处理（.i）

-Wall 显示更多警告信息

-V 查看版本号

-g 生成文件后，gdb 文件名 命令进入gdb

将宏加上//,可用-D定义宏:

//#define SIZE 8

gcc -DSIZE=5 a.c

#pragma 可额外增加一些功能

1)#pragma GCC dependency “fname”

“fname”文件如果比当前文件新，则产生警告

2)#pragma GCC poison ab

ab不可用，如果使用则报错

3)#pragma pack(n)

指定对齐/补齐字节数(1,2)有意义

多源文件的编译

1. 生成.o文件
2. gcc \*.o 链接所有的.o文件

目录下提供相应的头文件，其中有对函数的声明，满足函数先声明，位置任意放

## 库

静态库（.a）：使用是代码复制到目标文件，目标文件独立于库文件，运行速度稍快，目标

文件变大，代码修改不方便。

1. 生成t.o文件
2. ar -r lib\*\*\*.a t.o 创建名为\*\*\*的静态库

调用库时：1）生成test.c的test.o

2)gcc test.o libxx.a (当前目录时)

（常用）gcc test.o -l\*\*\* -l 库所在路径

3）配置环境变量LIBRARY\_PATH,把库文件路径加入，然后可用

gcc test.o -l\*\*\*

共享库（.so）：使用是把函数地址放入目标文件，运行速度稍慢，目标文件小，修改方便

目标文件和共享库需要同时存在

1. 生成t.o文件 （可不写-fpic）
2. gcc -shared t.o -o lib\*\*\*.so

调用库时与静态库方式相同

ldd 名称 可查看包含的动态库，可检查哪些库文件未加入到运行环境变量中

## Function

### XSI IPC

XSI IPC包括共享内存，消息队列，信号量集，隶属于一个规范，有着共同的特征

每个XSI IPC结构都是在内核中存储和维护的，外部到内核要用key，内核中使用id标识

key\_t key有三种方式得到：

1. 使用宏 IPC\_PRIVATE做key，无法实现进程间通信（私有），极少使用
2. 把所有的key定义在一个头文件中，用宏定义
3. 使用ftok()函数生成key，参数：真实存在的路径和项目编号（0到255）

id的生成：ipc结构在内核中都用id做唯一标识，创建、获取id都有对应的函数

key\_t key=ftok("/home/jyl",222);//生成一个key

调用\*\*get()时，都有一个flags，创建时值为：

权限 | IPC\_CREAT或IPC\_EXCL

IPC\_CREAT - 不存在则创建，存在则获取

IPC\_EXCL - 如果存在则创建失败

IPC结构都有一个特殊的操作函数，提供查询，修改和删除功能

函数名：\*\*ctl() 如：shmct(),msgctl()

参数cmd提供功能：

IPC\_STAT 查属性、状态

IPC\_SET 修改权限

IPC\_RMID 删除ipc结构，按id删除

#### 共享内存

ipcs -m可以查看系统中共享内存的使用情况

每个进程内存独立，无法直接互访，共享内存就是内核管理一段内存（物理内存），这段内存允许每个进程进行映射，是速度最快的IPC，效率高。

如果多个进程写数据，会产生覆盖

1. 系统创建、获取共享内存（拿到物理内存）

shmget() 创建、获取共享内存，返回id

1. 挂接共享内存（映射）

shmat()

1. 使用共享内存
2. 脱接共享内存（解除映射）

shmdt()

1. 如果共享内存不再被使用，删除

shmctl()

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

const int BUFSIZE = 100;

const key\_t key = (key\_t)12345678;

======写入

int shmid=shmget(key,BUFSIZE,IPC\_CREAT);

char\* pDat=(char\*)shmat(shmid,0,0);

char buf[BUFSIZE] = {0};

while(1)

{

cin>>buf;

memcpy(pDat,buf,BUFSIZE);

}

// 分离

shmdt(pDat);

// 删除共享内存

shmctl(shmid, IPC\_RMID, 0);

======读取

int shmid=shmget(key,0,0);

char\* pDat=(char\*)shmat(shmid,0,0);

while(1)

{

sleep(1);

cout<<pDat<<endl;

}

shmdt(pDat);

======状态

struct shmid\_ds ds;

shmctl(shmid,IPC\_STAT,&ds); 取shm状态

挂接数nattch %d ds.shm\_nattch

ds.shm\_perm.mode=0640; 权限可以修改

shmctl(shmid,IPC\_SET,&ds);

#### 消息队列

队列中存放各种信息，每个进程可以把数据封存在消息中，再放入队列，每个进程都可以拿到消息队列，再从中取出、放入消息

1. 创建、获取消息队列 msgget
2. 发送、接收消息 msgsnd、msgrcv
3. 如果所有进程不再被使用，删除 msgctl()

<sys/msg.h>

必须用结构体

struct DDD

{

int x;

};

const int BUFSIZE = sizeof(DDD);

const key\_t key = (key\_t)6666;

======写入

程序结束时必须要删除消息队列，否则会造成读取错误

int msgid=msgget(key,IPC\_EXCL);

if(msgid > 0) msgctl(msgid,IPC\_RMID,0); //删除消息队列

msgid=msgget(key,IPC\_CREAT|0666);

while(1)

{

int xx;

cin>>xx;

DDD dd;

dd.x = xx;

msgsnd(msgid,&dd,BUFSIZE,IPC\_NOWAIT);

}

msgctl(msgid,IPC\_RMID,0); //删除消息队列

======读取

int msgid=msgget(key,IPC\_EXCL);

if(msgid < 0) return 0;

while(1)

{

sleep(1);

DDD dd;

msgrcv(msgid,&dd,BUFSIZE,0,0);

cout<<dd.x<<endl;

}

#### 信号量集

由信号量组成数组，信号量其实就是一个计数器，用于控制同时访问共享资源的进程、线程的总数（IPC中的信号只控制进程）

计数到0阻塞后续进程

信号量使用方法：

1. 先赋初值，允许并行的进程最大数量
2. 如果有进程访问，计数-1，到0就阻塞访问进程
3. 如果有进程访问结束，计数加1

信号量集就是一个计数器数组，只能控制进程数量而不能互发数据

<sys/sem.h>

int ArrayLen = 10;

//创建、获取信号量集的id 0666代表权限

int semid = semget((key\_t) 1234567, ArrayLen, 0666 | IPC\_CREAT);

//semctl(semid,0,IPC\_RMID); 删除信号量集合？

int res = semctl(semid, 0, SETVAL, 2);//下标为0的位置设置2个信号

res = semctl(semid, 1, SETVAL, 3);//下标为1的位置设置3个信号

for(int i=0;i<6;i++)

{

pid\_t pid=fork();

if(pid!=0) continue;

struct sembuf op;

op.sem\_num = 0;//对应下标为0的信号

op.sem\_op = -1;//对应信号量-1

op.sem\_flg = SEM\_UNDO;

if (semop(semid, &op, 1) == -1)

{//进行操作信号量的个数，即op的个数，需大于或等于1。常设置1，只完成对一个信号量的操作

cout<<"run failed"<<endl;

exit(0);

}

cout<<i<<endl;

string str;

cin>>str;

op.sem\_op=1;//对应信号量+1

semop(semid, &op, 1);

exit(0);

}

### 管道

管道分为有名管道和无名管道，有名管道自由建立管道文件（\*.pipe），用于进程间通信。无名管道由内核建立管道文件，用于fork创建的父子进程之间的通信。管道数据是先进先出，只是数据中转站，不存储数据（只写不读会阻塞），和文件的操作相同。一个程序写，另一个程序读(阻塞)。

### 有名管道

#include <fcntl.h>

#include <sys/stat.h>

const int BUFFER\_SIZE = 100;

const char \*pipName = "/home/jyl/NamedPip";

======写入

if (access(pipName, F\_OK) == -1)

{

int res = mkfifo(pipName, 0777);

if (res != 0) return 0;

}

int fd = open(pipName, O\_WRONLY);

char buf[BUFFER\_SIZE] = {0};

while(1)

{

cin>>buf;

int res = write(fd, buf, BUFFER\_SIZE);

if(res == -1) break;

}

close(fd);

======读取

const int BUFFER\_SIZE = 100;

const char \*pipName = "/home/jyl/NamedPip";

int fd = open(pipName, O\_RDONLY);

char buf[BUFFER\_SIZE] = {0};

while(1)

{

int res = read(fd, buf, BUFFER\_SIZE);//阻塞

if(res <= 0) break;

cout<<buf<<endl;

}

close(fd);

### 无名管道

const int bufLen = 100;

int fd[2]={};

pipe(fd);

pid\_t id = fork();

if(id == 0)

{

close(fd[1]);//只负责读，因此将写描述关闭(节约资源)

char val[bufLen] = {0};

while(1)

{

read(fd[0],val,bufLen);

cout<<val<<endl;

}

exit(0);

}

close(fd[0]);//只负责写，因此将读描述关闭(节约资源)

while(1)

{

char str[bufLen] = {0};

cin>>str;

write(fd[1],str,bufLen);

}

### 信号

#### 定义

信号是程序中段的一种方式，属于软件中断。

中断就是终止当前的代码，转而执行其他代码，中断有软件中断和硬件中断。

信号的本质是一个非负整数

unix常用信号1到48，linux常用信号1到64，每个信号都有宏名，以SIG开头。

信号处理采用异步处理。

1到31都是不可靠信号，这种信号不支持排队，有可能丢失，非实时信号。（同一信号时）

34到64都是可靠信号，支持排队，不会丢失，是实时信号。

#include <signal.h>

#### 信号处理

1. 默认处理 80%的默认处理都是中断进程
2. 忽略信号 可以忽略信号，不做处理
3. 自定义处理

信号9不可以被忽略，也不能被自定义

普通用户只能给自己的进程发信号，root可以给所有用户发信号

fork创建的子进程完全沿袭父进程对信号的处理。

vfork+exec创建的子进程可以沿袭SIG\_IGN和SIG\_DFL，不能沿袭自定义方式（改为默认）

signal(信号，信号处理函数指针)

函数指针：1）SIG\_IGN 忽略

2）SIG\_DFL 默认

3）自定义处理函数

错误返回SIG\_ERR

#define MSG\_MY \_NSIG-1

======方式1

void sigProc(int sig)

{

cout<<"recv "<<sig<<endl;

}

signal(MSG\_MY,sigProc);

======方式2

void sigMsgProc(int sig, siginfo\_t \*info, void \*p)

{

cout<<"procees["<<info->si\_pid<<"]send ["<<sig<<"]:"<<endl;

int sigValInt = info->si\_int;

cout<<sigValInt<<endl;

//指针数据传递只适用当前程序

//char \*sigStr = (char\*)info->si\_ptr;

//cout<<sigStr<<endl;

}

struct sigaction action={};

action.sa\_flags=SA\_SIGINFO;

action.sa\_sigaction=sigMsgProc;

sigaction(MSG\_MY,&action,NULL);

#### 信号发送

|  |  |
| --- | --- |
| 类别 | 信号 |
| 键盘发送（少部分信号） | ctrl+C 2 SIGINT |
| ctrl+\ 3 SIGQUIT 结束进程 |
| 出错（少部分信号） | 段错误 11 SIGSEGV |
| 总线错误 7 SIGBUS |
| 整数除0 8 SIGFPE |
| 命令kill（全部信号） | kill -信号 进程id |
| 信号发送函数 | int raise(int sig)  只能给当前进程发信号，成功返回0，失败返回非0 |
| int kill（pid\_t pid， int sig）  pid>0 给pid对应的进程发信号  pid==0 发送给同组所有进程  pid==-1 发给所有有权限的进程  pid<-1 发给进程组（id等于pid绝对值）中的所有进程 |
| <unistd.h>  闹钟信号  unsigned int alarm(unsigned int seconds);  seconds 秒数后发送SIGALRM信号到当前进程  alarm（5） 5秒后结束进程  alarm（0） 取消闹钟  每次使用alarm时，之前使用的alarm失效  调用alarm时，以前设置的闹钟还没有执行，会返回之前的闹钟剩余秒数，如果以前设置闹钟执行了或者没有设置闹钟，则返回0 |

信号0用来测试是否有权限发信号，信号最大数值为\_NSIG-1

======方式1

//给当前进程发MSG\_MY信号

raise(MSG\_MY);

//给pid进程发信号

kill(pid,MSG\_MY);

======方式2

union sigval v;

v.sival\_int=10;

v.sival\_ptr = str;

sigqueue(pid,MSG\_MY,v);

### 信号集

用一个二进制位代表一个信号，用一个整数（sigset\_t 类型）代表信号集

sigset\_t set;

int sigemptyset(sigset\_t \*set); 将二进制位全部设置成0

int sigfillset(sigset\_t \*set); 将二进制全设成1

int sigaddset(sigset\_t \*set, int signum);

把信号signum加入信号集

int sigdelset(sigset\_t \*set, int signum);

把signum从信号集中删除

int sigismember(const sigset\_t \*set, int signum);

信号集中存储所有信号，每个信号都有相应二进制位置，且不重复

将信号加入，则将相应的位设成1，每个信号的存储位置在信号集中都有预设

信号屏蔽：延后信号的处理，等屏蔽结束后执行发过的信号，可靠信号支持排队，会执行发过的次数，不可靠信号无论发几次，只执行一次

sigprocmask(int how, sigset\_t \*new, sigset\_t \*old)

how是屏蔽方式：

SIG\_BLOCK 在原有基础上加新信号进行屏蔽

SIG\_UNBLOCK 减去原有信号屏蔽中的某些信号

SIG\_SETMASK 重新设置 （常用）

new是新的屏蔽信号集，old可以把之前的屏蔽信号集保存起来，如果old参数为空就是不保存

可以用old作为新函数中的new来取消屏蔽

sigpending(sigset\_t \*set)

取信号屏蔽期间来过的信号放入信号集

### 定时器

<sym/time.h>

linux为每个进程维护了三个计时器

真实计时器：程序运行时间（常用） SIGALRM

虚拟计时器：用户态消耗时间 SIGVTALRM

实用计时器：用户态和内核态消耗时间 SIGPROF

settimer() gettimer()

struct itimerval timer;

timer.it\_interval.tv\_sec=1; 时间间隔秒数

timer.it\_inerval.tv\_usec=1000; 微秒（必须）

timer.it\_value.tv\_sec=3; 三秒后开始

timer.it\_value.tv\_usec=0;

settimer(ITIMER\_REAL,&timer，0)

三个参数：真实计时器，新计时器，旧计时器

可用signal(SIGALRM,p)为真实计时器的信号定义一个行为

timer是为了设置计时器的属性，当使用settimer后，就可以使用真实计时器每隔n秒后发信号SIGALRM

usleep(1 000 000)

暂停1秒=一百万us

### socket

使用socket文件做交互媒介，后缀是.sock,类型为s

本地 PF\_LOCAL 或PF\_UNIX

网络 PF\_INET IPV4

PF\_INET6 IPV6

UDP SOCK\_DGRAM 打包发送

TCP SOCK\_STREAM 数据流的方式发送

#### 服务器

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

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

{

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

if (sockSrv == -1)

{

perror("socket创建出错！");

return 0;

}

struct sockaddr\_in addrSrv;

addrSrv.sin\_family=AF\_INET;

addrSrv.sin\_port=htons(5555);//端口号

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

bzero(&(addrSrv.sin\_zero),8);

if (bind(sockSrv, (struct sockaddr \*)&addrSrv, sizeof(struct sockaddr)) == -1)

{

perror("bind出错！");

return 0;

}

if (listen(sockSrv, 5) == -1)//最多能接受5个客户端链接

{

perror("listen出错！");

return 0;

}

while(1)

{

struct sockaddr\_in addrClient;

socklen\_t sin\_size = sizeof(struct sockaddr\_in);

//接受链接

int sockConn = accept(sockSrv, (struct sockaddr \*)&addrClient, &sin\_size);

//发送

char sendBuf[100]={0};

sprintf(sendBuf,"%s",inet\_ntoa(addrClient.sin\_addr));

send(sockConn,sendBuf,strlen(sendBuf)+1,0);

//接收

const int bufSize = 100;

char recvBuf[bufSize ] = {0};

recv(sockConn,recvBuf,bufSize ,0);//返回值为接收到的实际大小,网络错误时，返回SOCKET\_ERROR

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

//关闭

close(sockConn);

}

return 0;

}

#### 客户端

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <unistd.h>

#include <netdb.h>

#include <netinet/in.h>

#include <sys/socket.h>

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

{

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

struct hostent \*host = gethostbyname("127.0.0.1");

struct sockaddr\_in addrSrv;

addrSrv.sin\_family=AF\_INET;

addrSrv.sin\_port=htons(5555);

addrSrv.sin\_addr = \*((struct in\_addr \*)host->h\_addr);

bzero(&(addrSrv.sin\_zero),8);

//连接

if (connect(sockClient, (struct sockaddr \*)&addrSrv, sizeof(struct sockaddr)) == -1)

{

perror("connect出错！");

return 0;

}

//接收

const int bufSize = 100;

char recvBuf[bufSize ]={0};

recv(sockClient,recvBuf,bufSize ,0);

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

//发送

send(sockClient,"clientSend",strlen("client")+1,0);

close(sockClient);

return 0;

}

### 进程

#### 定义

linux内核启动0进程，0进程启动1进程和2进程（有些linux只启动1进程），1，2再启动其他所有进程

进程常见的状态：

S 休眠状态

s 有子进程

o 可运行状态

R 运行状态

z 僵尸进程（已结束但资源未回收）

1. 父进程启动子进程，父子进程同时运行
2. 如果子进程先结束，子进程给父进程发信号，由父进程负责回收子进程的相关资源
3. 如果父进程先结束，自进程会把init（进程1）作为新的父进程
4. 如果子进程先结束，同时发出的信号父进程未收到或子进程未发信号，子进程就变成僵尸进程

linux下一个进程在内存里有三部分：数据段，堆栈段，代码段

堆栈段存放的是子进程的返回地址，子进程的参数以及子进程的局部变量，而数据段则存放程序的全局变量，常数以及动态数据分配的空间，系统如果同时运行多个相同的程序，他们之间就不可能使用同一个堆栈段和数据段。

#### PID

<unistd.h>

进程id（PID），非负整型，是唯一的，但也可以延迟重用

getpid() 取当前进程id

getppid() 取当前进程的父进程id

getuid()/geteuid() 取当前有效用户的id

#### fork

<unistd.h>

pid\_t fork() 失败返回-1

fork通过复制自身（父进程）创建子进程。（复制代码区之外的内存区域，但和父进程共享代码区）fork之前的代码，父进程执行一次，fork之后的代码父子进程分别执行一次

fork自身会有两次返回，在父进程返回子进程id，在子进程返回0.

fork之后，父子进程同时运行，谁先运行不确定，子进程和父进程（虚拟内存地址相同），映射的物理内存不同

文件操作时，复制文件描述符，不复制文件表，所以父子进程相当于共用文件指针，不会相互覆盖

fork函数执行后，系统就为新的进程准备数据段，堆栈段和代码段，首先让新的进程和旧的进程使用同一个代码段，因为他们的程序还是相同的，对于数据段和堆栈段，系统复制一份给新的进程，但实际上数据已经分开，相互之间不再有影响，不再共享任何数据。fork复制两个段实际上是逻辑上的，并非物理上的复制，实际执行fork时，物理空间上两个进程的数据段和代码段都还是共享的，当有一个进程写了某个数据时，这是两个进程之间的数据才有了区别，系统就将有区别的页从物理上分开。

父进程结束，子进程不结束。

pid\_t pid=fork();

if(pid==0)//子进程pid为0

{

.......子进程执行的代码

exit(0);

}

.........

........父进程执行的代码

#### vfork

<unistd.h>

pid\_t vfork() 失败返回-1

vfork不复制父进程任何内存空间，创建的子进程占用父进程的内存空间运行，父进程在此时被阻塞。vfork和exec系列函数结合使用才有意义。vfork负责创建子进程，exec系列函数负责提供新的程序被执行。未使用exec系列函数则行为不确定。

一个进程一旦调用exec系列函数，它本身就终止了，系统把代码段替换成新的程序代码，废弃原来的数据段和代码段，并为新程序分配新的数据段和代码段，唯一留下的就是进程号。对系统而言，还是同一个进程，不过已经是另一个程序了。

exec系列函数（6种）不是新建一个进程，而是修改进程。用新的程序替换旧的，不改变pid（占用父进程的空间运行）

execl(“exe全路径”，“命令”，“选项”，“参数”,NULL）;

execlp 包含系统路径

execle 包含环境表

execl("/bin/ls","ls","-l","/home",NULL);//详细显示home目录下所有文件

execlp("ls","ls","-l","/home",NULL);

用于打开其他程序，父进程结束，子进程也结束。

pid\_t pid = vfork();

if(pid == 0)//子进程pid为0

{

子进程执行的代码打开a这个程序

execl("/home/jyl/a",0);

}

cout<<"baseRun"<<endl;

while(1){}

........父进程执行的代码

#### 终止进程

进程终止的正常情况：

1. main函数中执行return
2. 调用exit函数
3. 调用.exit或者\_Exit
4. 进程的最后一个线程执行了返回语句
5. 进程的最后一个线程调用pthread\_exit函数

异常终止：

1. 调用abort，产生SIGABRT信号
2. 进程接收到某些信号
3. 最后一个线程对“取消”请求做出响应

<unistd.h>

void \_exit(int status);

<stdlib.h>

void \_Exit(int status);

status:-1 非正常 0正常

<stdlib.h>

void exit(int status);

status: -1非正常， 0正常

<stdlib.h>

void exit(int status);

在退出之前会执行被on\_exit()和atexit()注册的函数

atexit(函数指针) 注册放在exit之前

#### 进程等待

<sys/wait.h>

pid\_t wait(int \*status) status作为返回值类型

成功返回子进程id，失败返回-1

用于父进程等待子进程的结束，等待任意一个子进程结束都会返回

返回结束子进程的状态和退出码

exit（100），则返回的退出码为100

wait（）可以回收僵尸子进程

宏函数：

WIFEXITED(status)可以判断是否正常结束

wait if exit end

WEXITSTATUS(status)可以得到退出码

（0到255有效）

exp：

pid\_t pid=fork();

if(pid==0){.........exit(100);}

int status;

int pid2=wait(&status);

等待子进程结束后执行下面的代码

if(WIFEXITED(status))

printf(“%d\n”,WEXITSTATUS(status));

pid\_t waitpid(pid\_t pid, int \*status, int options)

pid: -1 等待任意一个子进程结束

>0 等待指定子进程结束

<-1 取绝对值，绝对值代表一个进程组，等待这个进程组中任

意一个子进程结束

0 等待任意和父进程同一个进程组的子进程结束

options： WNOHANG 没有子进程结束，立即返回0

wait no hang

0 一直等待子进程结束

### 线程

#### 定义

每个进程的内部，都支持多线程并行。线程不需要有自己独立的内存空间，只是拥有一个独立的栈，一个进程的所有线程共享进程的资源。

进程中必须有一个主线程，主线程结束，进程随之结束，进程结束导致所有线程结束。

多线程之间代码是乱序执行，每个线程内部代码是顺序执行

POSIX规范中对线程做了完整定义，其中所有函数几乎都以 pthread\_开头

#include<pthread.h>

#### 创建

int pthread\_creat(线程id，属性默认为0，函数指针，函数参数的指针)

函数： void\* (\*pfun)(void \*)

错误处理是通过返回错误码，不是使用errono，失败返回非0，成功返回0

pthread\_t id;

int res = pthread\_creat(&id,0,pfun,0) 开启并运行线程

if(res) print(“%s\n”,strerror(res));

线程函数内部调用，参数同线程函数返回值

pthread\_exit(void \*returnval)

#### 终止

线程的取消就是给目标线程发CANCEL信号

目标线程可做出三种选择：忽略、立刻停止、延迟停止

pthread\_cancel(id)给目标线程发取消信号

可以在线程内设置CANCEL信号的处理

//设置不支持CANCEL信号

pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE,0);

//设置CANCEL信号的处理方式 pthread\_setcanceltype(PTHREAD\_CANCEL\_ASYNCHRONOUS,

PTHREAD\_CANCEL\_DEFERRED );

#### 等待

pthread\_t ptId=pthread\_self(); 查看当前线程的id

int pthread\_equal(id1,id2); 相等返回非0，否则返回0

pthread\_join(ptId,0); 等待ptId线程的结束

char \*res;

pthread\_join(id,(void\*\*)&res);//线程的返回值为指针，所以用pthread\_join第二个参数接收

printf(“%s”,res);

同步方式：被其他线程等待时，只有pthread\_join()函数返回时，才释放自己占用的资源

异步方式：运行结束立即释放资源，未被其他线程等待

int pthread\_detach(id)

执行后线程不能被pthrea\_join同步

#### 锁

pthread\_mutex\_t lock;//声明互斥变量

pthread\_mutex\_init(&lock,0);//初始化互斥量

pthread\_mutex\_lock(&lock);//加锁

.......访问

pthread\_mutex\_unlock(&lock);//解锁

pthread\_mutex\_destroy(&lock);//销毁锁

#### 信号量

信号量是一个计数器，用来控制访问临界资源的线程最大并行数量，当信号量的初始值为1时，效果等同于互斥量

信号量不是最早的POSIX线程相关规范，因此信号量相关定义是在semaphore.h中，其中定义的信号量用于线程（也可以用于进程，linux不支持）。

sem\_t sem;

//0代表线程计数（其他值代表进程计数，linux不支持）

sem\_init(&sem,0,计数初始值);

//获取信号量 计数减1 减到0时就代表上锁，此时阻塞

sem\_wait(&sem);

......访问资源

释放信号量（计数加1）

sem\_post(&sem);

删除信号量

sem\_destroy(&sem)

### 文件

<unistd.h>

void\* sbrk(n)

n>0 分配n个字节的空间

n<0 释放n个字节的空间

n==0 返回当前的位置

sbrk底层自动维护一个位置指针，通过位置的移动分配、释放内存

分配时，指针向后移动n个字节，分配结束后返回值为之前的指针的位置

释放时，指针向回移动n个字节，返回值为之前的指针位置（无意义）

sbrk单向连续，先进后出（先分配的后释放），分配方便，释放不方便

int brk（void \*）

失败返回-1

释放方便，分配不方便（brk的指针不移动）

exp：

void \*p=sbrk(0); 取当前位置

brk（p+4）； 分配4个字节的空间

brk（p+8）； 分配4个字节的空间（上一步已分配过4个）

brk（p+4）； 释放4个字节空间

brk（p）； 释放所有已分配的内存

<sys/mman.h> memory manage

void \* mmap(0,4,

首地址，0代表内核指定 分配4字节（映射1页）

PROT\_READ|PROT\_WRITE,

权限

MAP\_PRIVATE|MAP\_ANONYMOUS，

共有或私有 匿名（针对物理内存）

0,

目标处（如fd），可从文件fd处分配内存，fd必须有足够的大小，0代表系统从硬盘分配

0)

偏移量

mmap用来映射内存，可以是硬盘或文件

munmap(void\* ,4) 释放4字节

内核中提供系统函数，如sbrk（），可通过sbrk访问内核

time命令可以测试程序在用户层和内核层的运行时间

linux中，几乎一切都被看成了文件

<unistd.h> 读写

<fcntl.h>

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

新建文件，需要第三个参数，如果只打开则不需要

exp：

int fd=open(“a.txt”,O\_CREAT|O\_RDWR,0666);

三位八进制权限，权限屏蔽写（2）

int fd1=open(“a.txt”,O\_RDONLY);

if(fd==-1) perror(“open”),exit(-1);

fd为文件描述符

文件表会记录在硬盘上的文件信息（i节点）。打开一个文件，用文件表记录文件，文件描述符在文件表中，指代文件，描述符为数字，从3开始，0，1，2被系统占用

O\_RDONLY 0

O\_WRONLY 1

O\_RDWR 2

O\_APPEND 1024 追加方式打开

O\_CREAT 不存在会新建，存在则打开

close（int fd） 关闭文件

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

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

成功则返回读写的字节数，失败返回-1

读一般用：fd=open(“a.txt”,O\_RDONLY);

写一般用：fd=open(“a.txt”,O\_CREAT|O\_TRUNC,0666);

标c都有输入、输出缓冲区，而uc函数在用户层无缓冲区，当频繁输入，输出时，uc函数最好定于一个缓冲区。

char buf[] 先输入到buf，然后一次性输入到位置

int dup(int oldfd) 复制一个文件描述符，新符号由内核指定

int dup(int oldfd, int newfd) 自定义新描述符，如果已存在，则关闭原来的，然后再使用

失败返回-1

off\_t lseek(int fd,偏移量，偏移起始位置)

SEEK\_SET SEEK\_CUR SEEK\_END

fcntl(int fd, int cmd, ....)

1)可以复制文件描述符号（cmd取F\_DUPFD）

复制文件描述符时，和dup类似，不同点为当重名时，不会关闭已经使用的描述符，返回值是大于等于第三个参数

2)可以获取、设置文件的状态（cmd取F\_SETFL,F\_GETFL）

权限标识判断：

int flags=fcntl（fd，F\_GETFL）;

1.if（（flags&3）==O\_RDONLY）

3代表二进制11，O\_RDONLY为0

判断只读时使用，因为只读为0，任何值的二进制位与0都为0，flags&3取值只能是0，1，2中的一个

2.其他情况

if（flags&O\_APPEND）

3)实现文件锁（cmd取F\_SETLK,F\_GETLK,F\_SETKW）

fcntl（fd，cmd，&lock）；

读锁：共享锁，锁定其它写操作、允许读操作

写锁：锁定其它进程的操作

文件锁对应一个结构体：

struct flock{

short l\_type; 锁的类型

short l\_whence; 锁的开始位置

int l\_start； 偏移量 和锁的开始位置同时使用

int l\_len; 锁定长度，字节为单位

pid\_t l\_pid; 加锁的进程id（一般为-1）

}

l\_type: F\_RDLCK,F\_WRLCK,F\_UNLCK

l\_whence: SEEK\_SET,SEEK\_CUR,SEEK\_END

l\_pid: 只有在F\_GETLK时有效，其他置-1即可

exp：

int fd=open（.......）

描述锁: struct flock lock={F\_RDLCK,SEEK\_SET,0,20,-1}

加锁： int res=fcntl(fd,F\_SETLK,&lock);

if(res==-1)perror(“lock”),exit(-1);

解锁：

lock.l\_type=F\_UNLCK;

res=fcntl(fd,F\_SETLK,&lock);

if(res==-1)perror(“unlock”),exit(-1);

文件锁不能锁定硬盘上的文件，不能锁定read，write函数，只能阻止其他进程的上锁行为。

正确用法：在调用读函数前上读锁，调用写函数前上写锁。用if语句判断是否上锁成功，如果成功则执行读写

可使用F\_SETLKW实现加不上锁，则等待上锁。参数为F\_GETLK时，不是获取锁，而是测试某个锁是否能加上。如果已经有一个进程1中的锁la在运行，当前进程2中锁lb如果不可以加，则lb的参数都变成la的参数（其中lb的l\_pid变成进程1的id）如果lb可以加上，lb中的（\_type变为F\_UNLCK）其他不变

<sys/stat.h>

struct stat st;

int res=stat(“a.txt”,&st);

if(res==-1)perror(“stat”),exit(-1);

printf(“size=%d\n”,st.st\_size);

通过stat可得到文件的很多信息，stat结构体包括：st\_vid,st\_gid,

st\_nlink,st\_mode,st\_size,st\_atime,st\_mtime,st\_ctime.......

<unistd>

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

mode: R\_OK,W\_OK,X\_OK,F\_OK(存在)

存在权限则返回0，不存在则返回-1

mode可写多个组合：R\_OK|W\_OK

<sys/stat.h>

umask(0022) 0代表是八进制，后面三位对应用户

设置新屏蔽，返回旧屏蔽

对u不屏蔽，对g，o屏蔽写权限

exp：

mode\_t m=umask（0022）；

int fd=open(“a.txt”,O\_CREAT|O\_RDWR,0666)

创建文件时被屏蔽权限（0022）

umask(m); 恢复之前的权限屏蔽

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

修改文件的权限

<sys/types.h>

truncate()/ftruncate() 可指定文件大小

exp： truncate(“a.txt”,100);

ftruncate(fd,100);

<sys/types.h>

目录操作 （目录中都包含.和..目录）

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

rmdir(const char \*pathname)

chdir(const char \*pathname)

getcwd(char \*buf, long size);

读入当前位置放入到buf中，当前目录大小值放入size

<dirent.h>

DIR \* dir=opendir(“../ab”);

打开上级目录下的ab目录

if(dir==NULL)perror(“opendir”),exit(-1);

while(1){

struct dirent \* ent=readdir(dir);

返回一个目录，并指向下一条目录

if(ent==NULL) break; 注意判断.目录

printf(“%d,%s\n”,ent->d\_type,ent->d\_name);

}

type：4是目录

8是普通文件

rewinddir()

telldir()

seekdir() 类似文件指针的用法

动态调用共享库

<dlfcn.h>

代码在运行时才确定调用哪个函数

相关函数：dlopen() dlclose()

dlsym() dlerror()

dlsym()从一个打开的库文件中获取一个函数的指针

dlopen()第一个参数是共享库文件名，第二个是加载方式

runtime loading RTLD\_NOW open的同时加载

RTLD\_LAZY 延迟加载

exp: void \* handle=dlopen(“libxx.so”,RTLD\_NOW)；

char \*error=dlerror();

int (\*p)(int,int)；

p=dlsym(handle,”ab”);

ab为libxx.so中的函数名，ab形式与p指向形式相同

int d=p(1,1);

dlclose(handle);

gcc dl.c -ldl

## gdb

L 查看源码

b 行数

或 b main 设置运行断点

r 运行

n 运行下一行

s 进入函数内运行

q 退出

## cmake

编写CMakeLists.txt文件

cmake 目录

make

cmake命令生成makefile

make命令进行编译

基本结构：

cmake\_minimum\_required (VERSION 2.6)

project (工程名)

add\_executable(工程名 源文件.cpp)

### 代码使用外部定义

#### 外部头文件

CMakeLists.txt文件中：

set(EXT\_INFO 123)

configure\_file (

"${PROJECT\_SOURCE\_DIR}/cfg.h.in"

"${PROJECT\_BINARY\_DIR}/cfg.h"

)

include\_directories("${PROJECT\_BINARY\_DIR}")

编写 cfg.h.in

#define EXT\_INFO @EXT\_INFO@

cmake会根据cfg.h.in在PROJECT\_BINARY\_DIR 目录下自动生成cfg.h（CMakeLists.txt中将该目录加入头文件包含目录）

源代码中可以应用该头文件，并使用 EXT\_INFO

#### 外部开关

＝＝＝＝＝＝定义开关

编写 cfg.h.in时定义一个开关变量

#cmakedefine ABCD

＝＝＝＝＝＝开关控制

CMakeLists.txt文件中控制开关：

１．选项控制

option (ABCD

"………….." ON)

通过该option命令ON 可以让cmake生成的cfg.h中包含#define ABCD

通过该option命令OFF则取消#define ABCD

２．函数检查

include (${CMAKE\_ROOT}/Modules/CheckFunctionExists.cmake)

///////include (CheckFunctionExists)

check\_function\_exists (log ABCD)

如果函数存在则可以让cmake生成的cfg.h中包含#define ABCD

＝＝＝＝＝＝开关使用

１．CMakeLists.txt文件中可以使用对应定义

if (ABCD)

。。。。

endif (ABCD)

２．代码中可以使用该ABCD)宏

### 子目录链接

代码目录下包含AA目录

AA目录中包含t1.cpp，其中实现fun函数

AA目录中包含一个 CMakeLists.txt：

add\_library(AA t1.cpp)

(可以对应生成libAA.a)

代码目录中可以直接声明AA中的函数fun，并调用

代码目录中CMakeLists.txt:

cmake\_minimum\_required (VERSION 2.6)

project (tt)

include\_directories ("${PROJECT\_SOURCE\_DIR}/AA")

add\_subdirectory (AA)

set (EXTRA\_LIBS ${EXTRA\_LIBS} AA)

add\_executable(tt main.cpp)

target\_link\_libraries (tt ${EXTRA\_LIBS})

### 安装

代码目录中CMakeLists.txt:

install (TARGETS tt DESTINATION bin)

（放到文件末尾）

可以将生成的执行文件tt拷贝到　/usr/local/bin目录下

install (FILES t1.h DESTINATION include)

可以将文件t1.h拷贝到　/usr/local/include目录下

### 测试

代码目录中CMakeLists.txt末尾：

include(CTest)

add\_test (ttRun tt argsss)

set\_tests\_properties (ttRun PROPERTIES PASS\_REGULAR\_EXPRESSION "100")

测试用例名为 ttRun，argss为启动参数

程序输出中是否包含字符串１００

使用ctest命令进行测试

可以使用宏进行用例编写

macro (do\_test arg result)

add\_test (testName${arg} Tutorial ${arg})

set\_tests\_properties (testName${arg}

PROPERTIES PASS\_REGULAR\_EXPRESSION ${result})

endmacro (do\_test)

do\_test (25 "25 is 5")

do\_test (-25 "-25 is 0")

### 编译时运行程序

代码目录下包含AA目录

AA目录中包含t2.cpp:

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

{

if(argc != 2) cout<<"erro"<<endl;

char \*FilePathName = argv[1];

FILE \*pf = fopen(FilePathName,"w");

fprintf(pf,"xxx");

fclose(pf);

return 0;

}

接收文件名参数，进行写文件操作。

AA目录中对应的CMakeLists.txt文件：

add\_executable(t2 t2.cpp)

add\_custom\_command (

OUTPUT ${CMAKE\_CURRENT\_BINARY\_DIR}/t2Ret.txt

COMMAND t2 ${CMAKE\_CURRENT\_BINARY\_DIR}/t2Ret.txt

DEPENDS t2

)

include\_directories( ${CMAKE\_CURRENT\_BINARY\_DIR} )

add\_library(AA t1.cpp ${CMAKE\_CURRENT\_BINARY\_DIR}/t2Ret.txt)

add\_executable编译该程序，再运行该程序，传入t2Ret.txt文件路径作为参数，让生成的t2程序生成该文件，并通过add\_library加入项目输出

### 项目打包

代码目录中CMakeLists.txt末尾：

include (InstallRequiredSystemLibraries)

set (CPACK\_RESOURCE\_FILE\_LICENSE

"${CMAKE\_CURRENT\_SOURCE\_DIR}/License.txt")

set (CPACK\_PACKAGE\_VERSION\_MAJOR "${×××\_VERSION\_MAJOR}")

set (CPACK\_PACKAGE\_VERSION\_MINOR "${×××\_VERSION\_MINOR}")

include (CPack)

License.txt源代码打包时使用

版本信息可以使用CMakeLists中的宏定义

打包程序命令：

cpack --config CPackConfig.cmake

打包源代码命令：

cpack --config CPackSourceConfig.cmake