1.

Ctrl-a 回到第一个光标

Ctrl-e 到最后

Ctrl-u 删除所有输入

上 Ctrl-p 上一条指令

下 Ctrl-n 下一条指令

Ctrl-c 终止进程

连续按两次gg回到开头

按G回到末尾

2.

sudo apt-get update 更新软件资源到本地

sudo apt-get install “软件” 安装命令

sudo apt-get remove “软件” 卸载命令

Ubuntu安装包以.deb为结尾 （使用软件包安装 sudo dpkg -i 安装包名）

tar压缩：

tar zcvf 要生成的压缩包名 压缩材料

（tar zcvf test.tar.gz file1 dir2使用gzip方式压缩

tar jcvf test.tar.gz file1 dir2使用bzip2方式压缩

）

tar解压缩：

tar zxvf test.tar.gz使用gzip方式解压缩

tar jxvf test.tar.gz使用bzip2方式解压缩）

rar压缩：

rar a -r 要生成的压缩包名 压缩材料

(

rar a -r test.rar file1 dir2使用rar方式压缩

)

rar 解压：

unrar x rartest.rar

zip压缩：

zip -r testzip.zip file1 dir2

zip解压缩：

unzip testzip.zip

./当前目录

cd ~回到用户目录

reboot重启计算机

history 查看历史命令

pwd 查看当前目录

exit 退出用户

mkdir “文件夹名” 创建文件夹

rmdir 删除空文件夹

rm –rf 强制删除文件

cp 文件/目录 文件/目录 将文件拷贝到目录里

touch file\_name(s) 创建新的空文件

which 查看命令所在的目录

cat 查看文件内容

tac 倒着查看文件内容

tree 以树状显示当前目录

mv 文件/目录 文件/目录 移动文件

whoami 查看当前用户

sudo chmod u+x a.c修改文件权限 (文字设定法)

sudo chmod 471 a.c (数字设定法)

{

3个8进制数字分别代表User、Group、及Other的权限

r 对应4

w 对应2

x 对应1

}

sudo chown 新用户名 待修改文件 修改文件所属用户

sudo chgrp 新用户组名 待修改文件 修改文件所属用户组

sudo chown nobody:nogroup hello.cpp 一起改

find ./ -type ‘l’ 在当前目录下按文件类型找 （找文件）

find ./ -name ‘\*.jpg’ 在当前目录下按文件名字找

find ./ -maxdepth 1 -name ‘\*.jpg’ （限制层级深度为一层，应写为第一个参数）

（还有-size按大小）

grep –r ‘copy’ ./ -n（按文件内容搜索对象）

ps aux 查看当前执行的进程 (ps aux | grep ‘root’ ,两个命令结合使用)

ps ajx 参看pid/ppid/gid/sid

ps –Lf 进程id 可以查看线程号

3

bin：存放二进制可执行文件

boot：存放开机启动程序

dev：存放设备文件

home：存放普通用户

etc：用户信息和系统配置文件

lib：库文件

root：管理员宿主目录

usr：用户资源管理目录

4.

-l 详细信息

-a 显示隐藏文件

-d 显示目录

-a 全部

-r 递归

例子：ls –l hello.cpp

5.

Linux系统文件类型：8种（真正占用磁盘空间的是普通文件/目录文件/软连接，其他文件类型称为伪文件，只占用内存空间）

普通文件：-

目录文件：d

字符设备文件：c

块设备文件：b

软连接：l

管道文件：p

套接字：s

未知文件

6.

软连接：为保证软连接可以任意搬移，创建是务必对源文件使用绝对路径

ln –s “文件目录” “软连接名字” 创建软连接也就相当于Windows快捷方式

硬连接：操作系统给每一个文件赋予唯一的inode，当有相同inode的文件存在时，彼此同步。删除时只讲硬连接数减一，减为0时inode被释放

ln “文件目录” “软连接名字” 创建硬连接 文件之间同步

6.

makefile:

命名：makefile Makefile

1个规则，

目标：依赖条件

（一个他把缩进）命令

1. 目标的时间必须晚于依赖条件的时间，否则，更新目录
2. 依赖条件如果不存在，找寻新的规则去产生依赖

ALL:指定makefile的终极目标

2个函数： ($()相当于取值)

src=$(wildcard \*.c):匹配当前工作目录下的所有.c文件。将文件名组成列表，赋值给变量src。

obj=$(patsubst %.c,%.o,$(src))：将参数3中，包含参数1的部分，替换为参数2.

clean:（没有依赖）

-rm -rf $(obj) a.out “-“：作用是，删除不存在文件时，不报错

3个自动变量：

$@:在规则的命令中，表示规则的目标

$^:在规则的命令中，表示所有依赖条件

$<:在规则的命令中，表示第一个依赖条件,如果将该变量应用在模式规则中，它可将依赖条件列表中的依赖依次取出，套用模式规则。

模式规则：

%.o:%.c

gcc –c $< –o $@

静态模式规则：

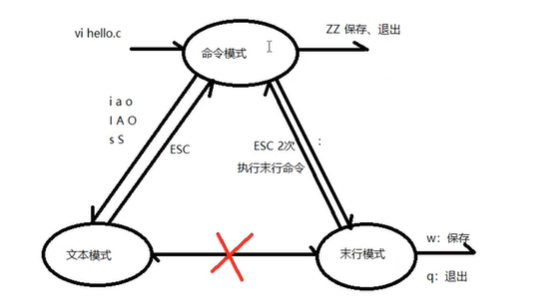
$(obj):%.o:%.c

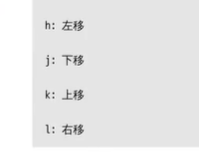
Gcc –c $< -o %@

伪目标：

.PHONY:clean ALL

7.





跳转到指定行：

1）88G(命令模式)

2）:88（末行模式）

3）连续按两次gg回到开头

4）按G回到末尾

5）自动格式化 gg=G（命令模式）

6）删除单个字符 x（命令模式）

7）删除光标至行尾D

8) 删除光标至行尾d0

9）复制一行 yy

10）粘贴 p：向后 P：向前

11）查找 1找设想内容，按“/“输入欲搜索关键字，回车，使用n检索下一个

2找看到的内容，将光标置于单词任意一个字符上，按“\*“

12）撤销 u（命令模式）

13）竖分屏 vsp（末行模式） Crtl+ww切换

8.gcc编译可以执行程序4步骤：预处理、编译。汇编。链接



-I 指定头文件目录位置

-c 只做预处理、编译、汇编 得到二进制文件

-g 编译是增加调试语句。主要支持gdb调试

-Wall 显示所有警告信息

-D 向程序动态注册宏定义

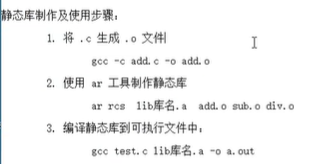
**C的头文件是在怎么和源文件中具体的函数方法联系起来；**

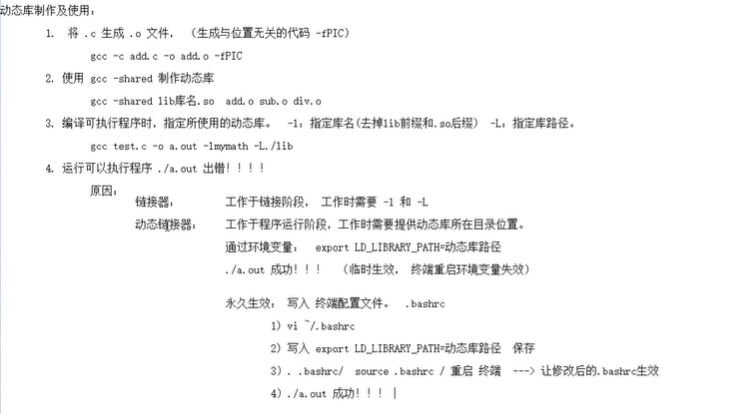
**头文件中有extern 函数原型  
这个在包含到你调用的函数的源代码后，编译器会产生一个占位符号。  
链接器会将这个占位符号对应到lib文件或者别的c/cpp中的符号，最终产生一个实际的调用。所以，要么.h文件中有函数的定义（函数体），要么除了.h还需要.lib，这样才能编译。（lib文件是Windows概念下的，对应到Linux下就是.o静态库文件/.so动态库文件）**

编译器不管头文件的，头文件只是用来被cpp文件包含的，被包含之后，它就成了那个cpp文件的一部分了，而编译器只编译.cpp文件，不会去单独编译一个头文件的，  
编译器这样做之后，针对每个编译过的cpp文件生成一个obj文件。  
然后链接器把所有这些obj文件连接成一个程序，

9.静态库：对空间要求较低。而对时间要求较高。动态库：对时间要求较低，对空间要求较高。

编译阶段出错有行号，链接阶段出错没有行号，且有collect2（链接器）





静态库动态库的区别：

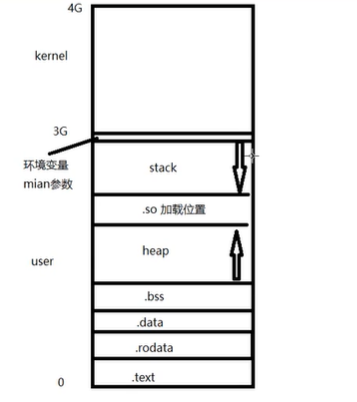
静态库在编译的时候会直接整合到目标程序中，所以利用静态函数库编译成的文件会比较大，这类函数库最大的优点就是编译成功的可执行文件可以独立运行，而不再需要向外部要求读取函数库的内容；但是从升级难易度来看明显没有优势，如果函数库更新，需要重新编译。

动态库与静态函数库被整个捕捉到程序中不同，动态函数库在编译的时候，在程序里只有一个“指向”的位置而已，也就是说当可执行文件需要使用到函数库的机制时，程序才会去读取函数库来使用；也就是说可执行文件无法单独运行。这样从产品功能升级角度方便升级，只要替换对应动态库即可，不必重新编译整个可执行文件。

注意上述是使用库编译成可执行文件的时候两种库的区别，

在生成动态库或者静态库后，库中其实都具有了生成库文件时用到的.cpp/.c文件

10.进程的内存分布



11.gdb调试工具：

基础指令：

-g：使用该参数编译可执行文件，得到调试表

gdb a.out

list: list/l 1 列出源码。根据源码指定行号设置断点

b：break/b 20 在20行处设置断点

d：delete/d 20 删除某个断点

run/r 运行程序

next/n 下一条指令（会越过函数）

step/s 下一条指令（会进入函数）

print/p p i 查看变量i的值

continue：继续执行断点后续程序

quit 退出gdb当前调试

其他指令：

run：进入gdb直接run就可以查看段错误

set args 设置main 函数命令行参数

run 参数1 参数2 设置main 函数命令行参数

start:直接从第一行开始进行调试

finsh:结束当前函数的调用

info b 查看设置的断点位置

b 20 if i = 5 ：设置条件断点

ptype arr:查看arr变量的变量类型

backtrace/bt:列出当前程序正存活的栈帧

frame/f 切换函数的栈帧

display:设置跟踪变量

undisplay：取消设置跟踪变量，使用跟踪变量的编号

使用gdb调试程序

g++ AddressBook.cpp -o AddressBook -g

gdb AddressBook

使用gdb调试进程

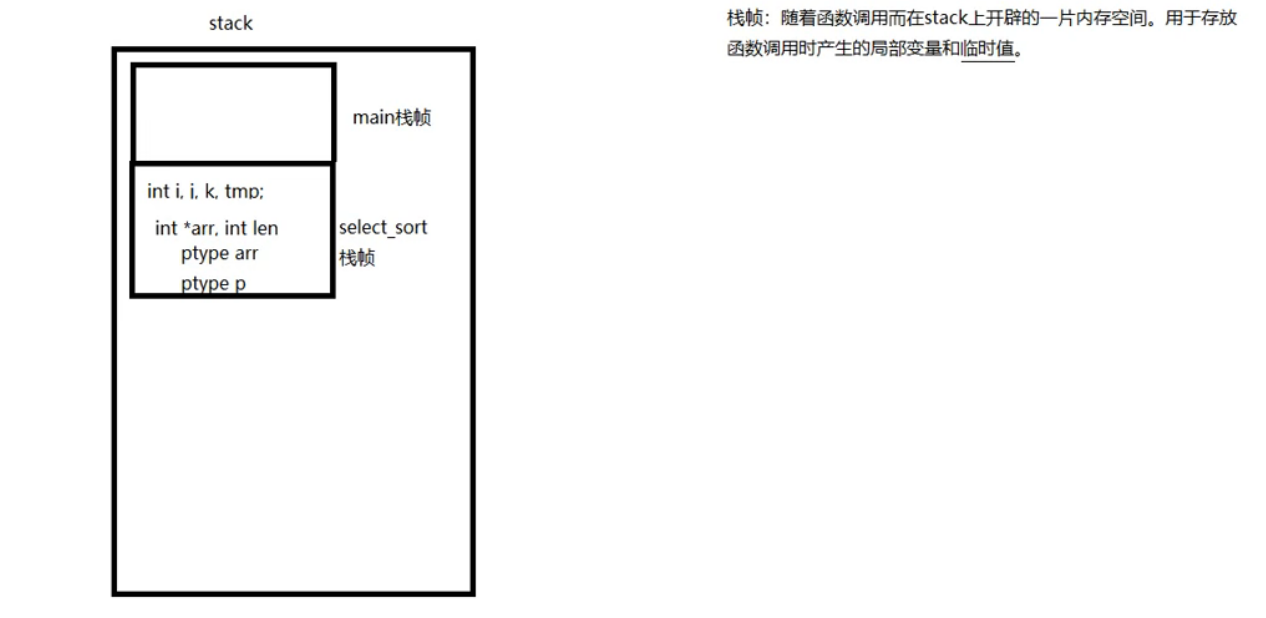
set follow-fork-mode child 命令设置gdb在fork之后跟踪子进程

set follow-fork-mode parent 设置跟踪父进程

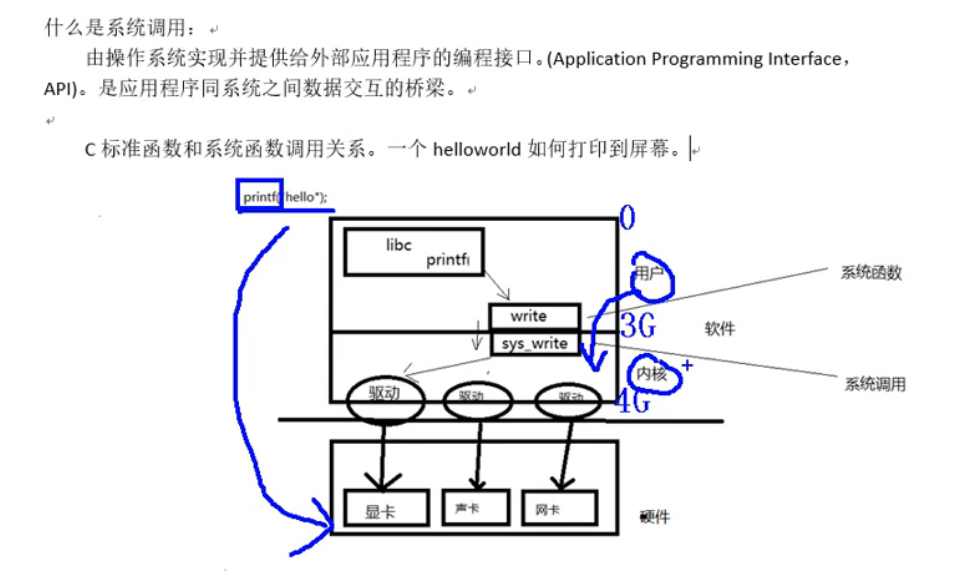
不进行设置默认跟踪父进程

注意：一定要在fork函数调用前设置才有效

12.



13.



14.文件描述符就是一个整数

open函数：

int open(char \*pathname, int flags) #include<unistd.h>

参数：

pathname:欲打开的文件路径名

flags：文件打开方式，O\_RDONLY|O\_WRONLY|O\_RDWD #include<fcntl.h>

返回值：

成功：返回打开文件所得到对应的文件描述符（整数）

失败：-1，设置errno

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

参数：

pathname:欲打开的文件路径名

flags：文件打开方式，O\_RDONLY|O\_WRONLY|O\_RDWD

mode:参数三使用的前提是，参二指定了O\_CREAT。取值8进制数，用来描述文件的访问权限。rwx 0664

创建文件最终得到的权限= mode & ~umask

返回值：

成功：返回打开文件所得到对应的文件描述符（整数）

失败：-1，设置errno

close函数

int close(int fd)

参数：

是前面open函数返回的文件描述符

错误处理函数： 与errno相关

printf（“xxx error： %d\n”,errno）

char \*strerror(int errnum);

printf(“%s\n”,strerror(errnum))

void perror(const char \*s);

perror(“open error”);等于实现了printf(“%s\n”,strerror(errnum))

read函数：

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

参数：

fd：文件描述符

buf：存数据的缓冲区

count：缓冲区大小

放回值：

0表示读到文件末尾

成功：读到的字节数

失败：-1，设置errno

-1：并且errno = EAGIN 或EWOULDBLOCK,说明不是read失败，而是read在以非阻塞方式去读一个设备文件（网络文件）并且文件无数据

write函数：

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

参数：

fd：文件描述符

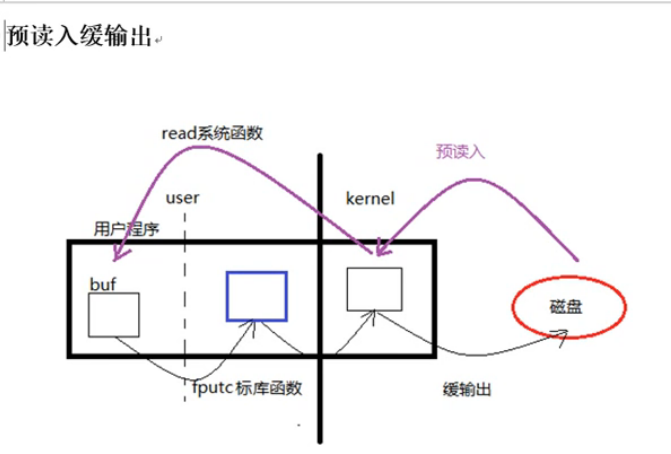
buf：待写出数据的缓冲区

count：数据大小

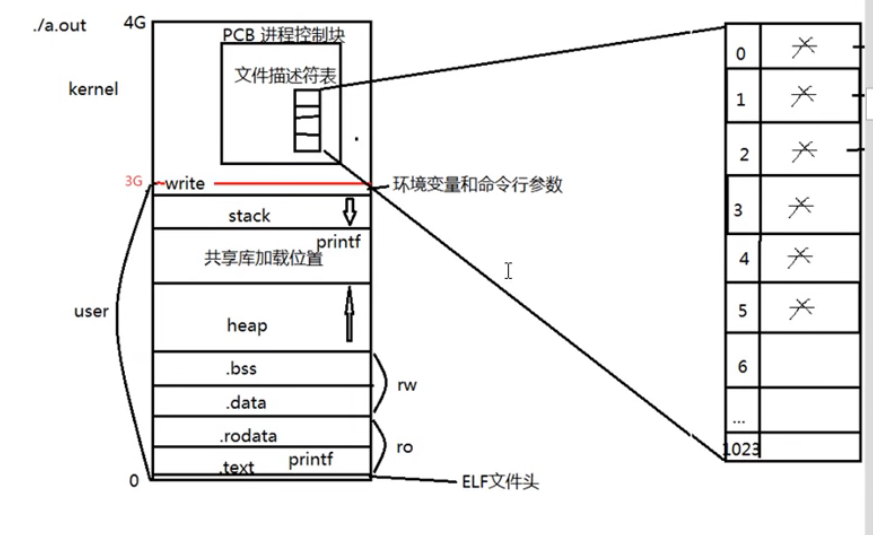
放回值：

成功：写入的字节数

失败：-1，设置errno



文件描述符：



PCB进程控制块，本质结构体

成员，文件描述符表

文件描述符：0/1/2/3/…./1024 表中可用的最小的

0 – STDIN\_FILENO

1 – STDOUT\_FILENO

2 – STDERR\_FILENO

15.

阻塞和非阻塞： 是设备文件，网络文件的属性

产生阻塞的场景。读设备文件/和读网络文件。（读常规文件无阻塞概念）

/dev/tty –-终端文件。（默认终端文件都是阻塞文件）

open(“/dev/tty”, O\_RDWR|O\_NONBLOCK) --设置/dev/tty非阻塞状态。（默认为阻塞状态）

fcntl函数:在不打开文件的状况下改变文件状态

int fcntl(int fd, int cmd, ... /\* arg \*/ );变参函数

int flags = fcntl(fd, F\_GETFL)

flags |= O\_NOBLOCK

fcntl(fd, F\_SETGL, flags)

获取文件状态：F\_GETFL

设置文件状态：F\_SETFL

lseek函数：修改文件偏移量（读写位置）

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

参数：

fd：文件描述符

offset：偏移量

whence：起始偏移位置： SEEK\_SET/SEEK\_CUR/SEEK\_END

返回值：

成功：较起始位置偏移量

失败：-1，设置errno

应用场景：

1.文件的读和写使用的是同一偏移位置

2.使用lseek获取。扩展文件的大小/也可使用truncate函数直接拓展

3.使用lseek函数拓展文件大小，要想是文件真正拓展，要进行IO操作（比如说使用write函数）

od -tcx filename 查看文件的16进制表示形式

od -tcd filename 查看文件的10进制表示形式

int ftruncate(int fildes, off\_t length);也可以用于扩展文件大小，参数1文件描述符，参数2需要扩展的大小，需要写权限，才能扩展文件大小

16.

传入参数：

1. 指针作为函数参数
2. 通常用const关键字修饰
3. 指针指向有效区域，在函数内部做读操作

传出参数：

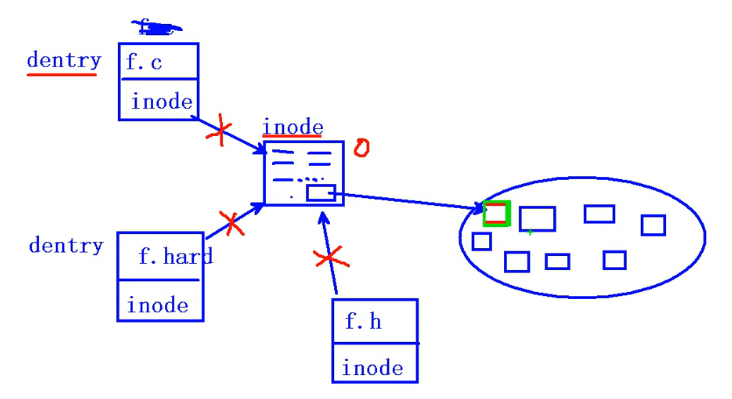
1. 指针作为函数参数
2. 在函数调用之间，指针指向的空间可以无意义，但必须有效
3. 在函数内部，做写操作
4. 函数调用结束后，充当函数返回值

传入传出参数：

1. 指针作为函数参数
2. 在函数调用之前，指针指向的空间有实际意义
3. 在函数内部，先做读操作，后写操作
4. 函数调用结束后，充当函数返回值

17.

inode其本质也是一个结构体，存储文件的属性信息



dentry:目录项

18.

stat函数：获取文件属性

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

参数：

pathname：文件路径

stabuf：（传出参数）存放文件属性

返回值：

成功：0

失败：-1 设置errno

获取文件大小：buf.st\_size

获取文件类型：buf.st\_mode

获取文件权限：buf.st\_mode

lstat函数：获取文件属性

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

参数：

pathname：文件路径

stabuf：（传出参数）存放文件属性

返回值：

成功：0

失败：-1 设置errno

lstat 不会穿透符号连接，stat会穿透符号链接

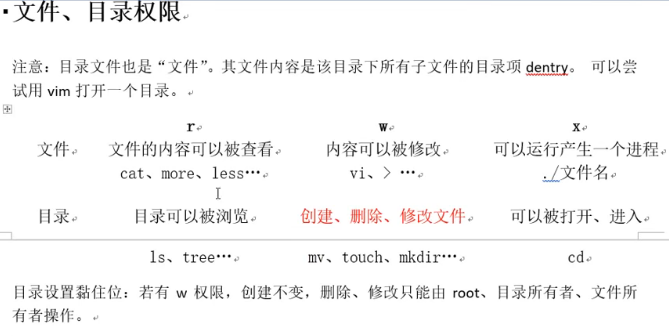
link函数：创建硬连接

unlink函数：删除硬连接（只有当打开文件的进程都关闭了，系统才会择机释放）

19.

隐式回收：当进程结束运行时，所有该进程打开的文件会被关闭，申请的内存空间会被释放。

20.



21.目录操作函数：

DIR \*opendir(char \*name)

int closedir(DIR \*dp)

struct dirent \*readdir(DIR \*dp)

struct dirent{

inode

char dname[256]

…

}

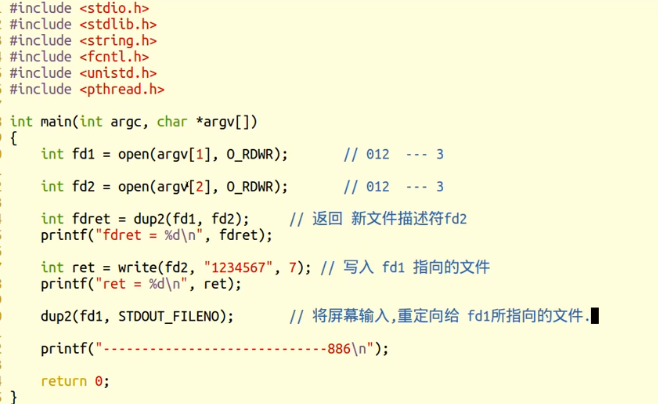
22.重定向

dup和dup2

int dup2(int oldfd, int newfd);

放回值：新文件描述符newfd

例子：dup2（2， 4）就是把后面的指向前面的



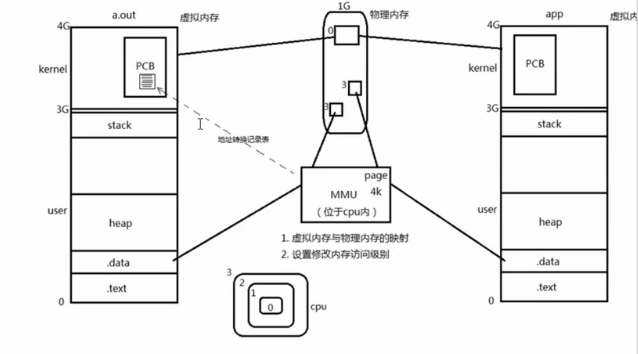
23.

进程：

程序：死的。只占用磁盘空间。 --剧本

进程：活的。运行起来的程序，占用内存、cpu等系统资源。 --戏

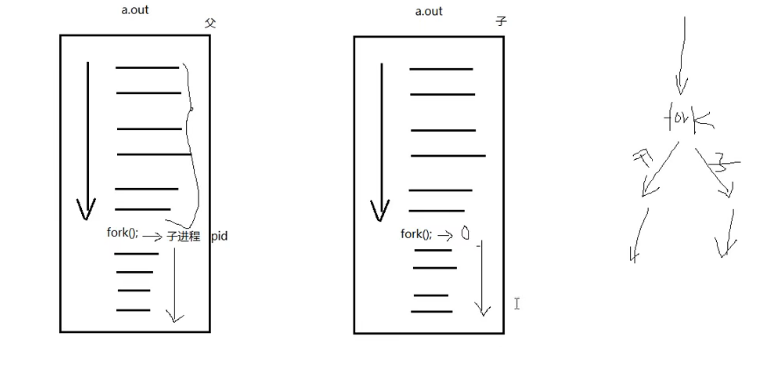
MMU：虚拟内存映射单元



进程控制块：PCB，进程控制块是一个结构体

结构体中包含很多，其中重点掌握：进程id/进程状态（初始态/就绪态/运行态/终止态/挂起态）/进程切换时需要保存和恢复的一些CPU寄存器/描述虚拟地址空间的信息/描述控制终端的信息/当前工作目录位置/umask掩码/文件描述符表包含很多指向file结构体的指针/和信号相关的信息/用户id和组id/会话和进程组/进程可以使用的资源上限

24.



25.

fork函数：

pid\_t fork(void)

创建子进程，父子进程各自返回，父进程返回紫禁城pid。子进程返回0

getpid();获取当前进程的pid

getppid()；获取子进程的pid

特别的，fork之后父进程先执行还是子进程先执行不确定，取决于内核所使用的调度算法

父子进程相同：

刚fork后。data段/test段/堆/栈/环境变量/全局变量/宿主目录位置/进程工作目录位置/信号处理方式

父子进程不同：

进程id/返回值/各自的父进程/进程创建时间/闹钟/未决信号集

父子进程共享：

读时共享，写时复制。（父子进程不共享全局变量）

1. 共享文件描述符2.mmamp建立的映射区

26.

exec函数族(使进程执行某一程序，调用成功不会返回，只有失败才会返回-1)

先只学习两个函数

int execl(const char \*path, const char \*arg, ...

/\* (char \*) NULL \*/);执行自己的进程

int execlp(const char \*file, const char \*arg, ...

/\* (char \*) NULL \*/);执行系统的进程

参1：程序名

参2：argv0

参3：argv1

…：argvN

哨兵:NULL

因为是变参函数，所以通过在传入的最后一个参数后再加NULL，可以告诉函数表示传入参数结束，这个操作就叫做加哨兵

例子：

execlp(“ls”, “ls”, “-l”, NULL);

execl(“/bin/ls”, “ls”, “-l”, NULL);

27.

孤儿进程：父进程先于子进程结束，则子进程成为孤儿进程，子进程的父进程成为init进程，称为init进程领养孤儿进程。

僵尸进程：进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸进程（kill命令对其无效）

28.

wait函数： 回收子进程退出资源，阻塞回收任意一个

pid\_t wait(int \*wstatus);

参数：（传出）回收进程的状态

返回值：成功：回收进程的pid

失败：-1，errno

函数作用：

1. 阻塞等待子进程退出
2. 清理子进程残留在内核中的pcb资源
3. 通过传出参数，得到子进程的结束状态

获取子进程正常终止值：

WIFEXITED(wstatus)—》为真—》调用WEXITSTATUS(wstatus)—》得到子进程的退出值

WIFSIGNALED(wstatus)—》为真—》调用WTERMSIG(wstatus)—》得到导致子进程一场终止的信号编号

waitpid函数：指定对某一个进程进行回收，可以设置非阻塞或阻塞。

pid\_t waitpid(pid\_t pid, int \*wstatus, int options);

返回值：

>0 ：表示成功回收的子进程pid

0 ：函数调用时，参3指定WNOHANG，并且，没有子进程结束。

-1：失败。errno

参数：

pid：指定回收的进程pid

>0 :指定的一个进程

-1 回收任意子进程

0回收和当前调用waitpid一个组的所有子进程

< -1回收指定进程组内的任意子进程

wstatus：（传出）回收进程的状态

options：WNOHANG，指定回收方式为非阻塞

总结：

一次wait/waitpid函数调用，只能回收一个子进程

想要回收多个子进程，使用循环

waitpid(-1, &status, 0) == wait(&status)

29.

进程间通信的常用方式，特征

管道：简单

信号：开销小

mmp映射：用在非血缘关系进程间

socket（本地套接字）：稳定

管道：默认管道大小4096

实现原理：

内核借助环形队列机制，使用内核缓冲区实现

特征：

1. 伪文件
2. 管道中的数据只能一次读取
3. 数据在管道中只能单向流动（确保管道只有一个读端和一个写端，虽然其实允许多读端，多写端）

局限性：

1.自己写，不能自己读

2.数据不可以反复读

3.采用的通信方式是半双工通信

4.管道必须作用在有血缘关系的进程之间

30.

pipe函数：创建并打开管道

int pipe(int fd[2])

参数：

fd[0]：读端

fd[1]：写端

返回值：

成功：0

失败：-1 .errno

管道的读写行为：

读管道：

1. 管道有数据，read返回实际读到的字节数
2. 管道无数据：1）无写端，read返回0（类似读到文件结尾）

2）有写端，read阻塞等待

写管道：

1. 无读端，异常终止，由SIGPIPE导致
2. 有读端，1）管道已满，阻塞等待

2）管道未满，返回写入的字节数

管道的优劣：

优点：

简单，相比信号，套接字实现进程间通信，简单的多

缺点：

1. 只能单向通信，双向通信需要建立两个管道
2. 只能用于父子，兄弟进程（有共同祖先）间通信，该问题后来使用fifo有名管道解决

pipe管道：用于有血缘关系的进程通信

父子进程间通信

兄弟进程间通信

fifo管道：可以用于无血缘关系的进程通信

命名管道：mkfifo

无血缘关系进程间通信

读端：open（fifo， O\_RDONLY）

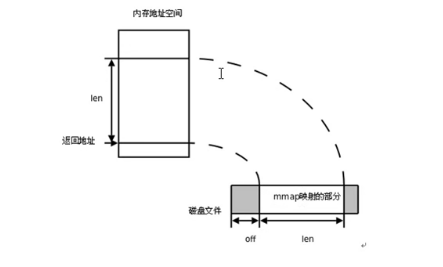
写端：open（fifo， O\_WRONLY）

文件实现进程间通信：

依赖打开的文件是内核中的一块缓冲区，多个无血缘关系的进程，可以同时访问该文件

31.

共享内存映射：将磁盘上的东西映射到内存



void \*mmap(void \*addr, size\_t length, int prot, int flags,

int fd, off\_t offset);

作用创建共享内存映射。

参数：

addr，指定映射区的首地址，通常传NULL，表示让系统自动分配

length：共享内存映射区的大小（<= 文件的实际大小）

prot：共享内存映射区的读写属性，PROT\_READ/PROT\_WRITE

flags:标注共享内存的共享属性。MAP\_SHARED/MAP\_PRIVATE（共享表示对内存的修改会反应到磁盘上）

fd：用于创建共享内存映射区的那个文件

offset：默认0，表示映射文件全部。偏移位置。必须是4k的整数倍

返回值：

成功：返回映射区的首地址

失败：MAP\_FAILED（void \*(-1)），errno

int munmap(void \*addr, size\_t length); 释放映射区。

参数：

addr：mmp的返回值

返回值：

成功：0

失败：-1 设置errno

(^@:文件空洞)

使用注意事项：

1. 用于创建映射区的文件大小为0，实际指定非0大小创建映射区，出“总线错误”
2. 用于创建映射区的文件大小为0， 实际指定0大小创建映射区，出“无效参数”
3. 用于创建映射区的文件读写属性为，只读。映射区属性为读、写。出“无效参数”
4. 创建映射区需要read权限。当访问权限指定为“共享”MAP\_SHARED,mmap的读写权限，应该小于等于文件的open权限。只写不行
5. 文件描述符fd，在mmp创建映射区完成即可关闭，后续访问文件，用地址。
6. offset必须是4096的整数倍。（MMU映射的最小单位就是4K）
7. 对申请的映射区内存不能越界访问
8. munmap用于释放的地址，必须是mmap申请返回的地址
9. 映射区访问权限为“私有”MAP\_PRIVATE,对内存所做的所有修改，只在内存有效，不会反应到物理磁盘上。
10. 映射区访问权限为“私有”MAP\_PRIVATE,只需要open文件有读权限，用于创建映射区即可

mmap函数的保险调用方法：

1. open（O\_RDWR）
2. mmap（NULL， 有效文件大小， PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0）
3. 一定要检查返回值

32.

父子进程使用mmap进程间通信：

父进程先创建映射区。open（O\_RDWR） mmap(MAP\_SHARED)

指定MAP\_SHARED权限

fork（）创建子进程

一个进程读，另外一个进程写

无血缘guanxi1进程间mmap通信：

两个进程打开同一个文件，创建映射区（同一个文件创建的映射区是相同的）

指定flags为MAP\_SHARED

一个进程写入，另外一个进程读出。

注意：无血缘关系进程间通信，mmap：数据可以重复读取。

fifo：数据只能读取一次（队列形式）

匿名映射只能用于血缘关系进程间通信

33.

信号共性：

简单、不能携带大量信息，满足条件才能发送

信号的特质：

信号是软件层面上的“中断”，一旦信号产生，无论程序执行到什么位置，必须立即停止运行，处理信号，处理结束，在继续执行后续指令。

所有信号的产生及处理全部都是由【内核】完成的

产生信号：

1. 按键产生
2. 系统调用产生
3. 软件条件产生
4. 硬件异常产生
5. 命令产生

递达：产生并且送达的进程 （信号递达后直接会被内核处理掉）

未决：产生和递达之间的状态。主要由于阻塞导致该状态

信号的处理方式：

1. 执行默认动作
2. 忽略（丢弃）
3. 捕捉（调用户处理函数）

默认动作细分：

Term：终止进程

lgn：忽略信号（默认即对该信号执行忽略操作）

Core：终止进程，生成Core文件。（查验进程死亡原因，用于gdb调试）

Stop：停止（暂停）进程

Cont：继续运行进程。

SIGKILL和SIGSTOP信号，不允许忽略和捕捉，只能执行默认动作。甚至不能将其设置为阻塞。

阻塞信号集（信号屏蔽字）：

本质：位图。用来记录信号的屏蔽状态，一旦被屏蔽的信号，在解除屏蔽前，一直处于未决状态。

未决信号集：

本质：位图。用来记录信号的处理状态，该信号集中的信号，表示已经产生，但尚未被处理

SIGPIPE：当某一进程试图向管道，FIFO或套接字写入信息时，如果这些设备并无相应的阅读进程，系统将产生该信号。

SIG\_IGN：信号被忽略

信号4要素：（信号使用之前必须先确认4要素再使用）

1. 编号
2. 名称
3. 事件
4. 默认处理动作

kill命令和kill函数：

int kill（pid\_t pid, int signum）

pid: >0：发送信号给指定进程

=0：发送信号给跟调用跟调用kill函数的那个进程处于同一进程组的进程

< -1:取绝对值，发送信号给该绝对值对应进程组的所有成员

= -1:发送信号给，有权限发送的所有进程

signum：待发送的信号

返回值：

成功：0

失败：-1 errno

alarm函数：定时发送SIGALRM信号给当前进程(使用自然时间计时法)

unsigned int alarm(unsigned int seconds);

参数：

seconds：定时秒数

返回值：上次定时剩余时间。

无错误现象

alarm（0）：取消闹钟

每个进程有且只有唯一一个定时器

time命令：查看程序执行时间。

实际执行时间 = 系统时间 + 用户时间 + 等待时间。

程序运行的瓶颈在于IO，优化程序，首选优化IO

setitimer函数：（可代替alarm函数，精度微秒us，可以实现周期定时）

int setitimer(int which, const struct itimerval \*new\_value,

struct itimerval \*old\_value);

参数：

which：

ITIMER\_REAL采用自然计时法->SIGALRM

ITIMER\_VIRTUAL采用用户空间计时法->SIGVTALRM

ITIMER\_PROF 采用内核加用户空间计时法->SIGPROF

new\_value:定时秒数 (类型是一个结构体)

old\_value: 传出参数，上次定时剩余秒数(类型是一个结构体)

返回值：

成功： 0

失败： -1 errno

34.

信号集操作函数：

sigset\_t set: 自定义信号集

int sigemptyset（sigset\_t \*set）清空信号集

int sigfillset(sigset\_t \*set)全部置1

int sigaddset(sigset\_t \*set, int signum)将一个信号添加到信号集中

int sigdelset(sigset\_t \*set, int signum)将一个信号从集合中移除

以上函数的返回值都是成功返回0，失败返回-1 设置errno

int sigismember(const sigset\_t \*set, int signum)判断一个信号是否在集合中。 在放回1， 不在返回 -1

设置信号屏蔽字和解除屏蔽字：

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

参数：

how：

SIG\_BLOCK: 设置阻塞

SIG\_UNBLOCK: 取消阻塞

SIG\_SETMASK:用自定义set替换mask

set：自定义set

oldest：旧有的mask

返回值：

成功： 0

失败： -1 errno

查看未决信号集

int sigpending(sigset\_t \*set);

参数：

set：传出的未决信号集

返回值：

成功： 0

失败： -1 errno

信号屏蔽字就叫mask，操作系统自带的

未决信号集叫pending

35.

signal函数：注册一个信号捕捉函数

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

参数：

signum：待捕捉的信号

handler：自定义函数名（只能是返回值为空，参数为int的函数，其参数也就是被捕捉到的信号）

返回值：

成功：返回上一次处理这个信号处理函数的指针。

失败：返回SIG\_ERR，设置errno

sigaction函数：

int sigaction(int signum, const struct sigaction \*act,

struct sigaction \*oldact);

参数：

signum:待不住的信号

act：结构体。要设置对信号新的处理方式

oldact：结构体，原来对信号的处理方式

返回值：

成功： 0

失败： -1 errno

1. 捕捉函数执行期间，信号屏蔽字 由mask –》 sa\_mask,捕捉函数执行结束，恢复回mask
2. 捕捉函数执行期间，本信号自动被屏蔽（sa\_flags = 0）
3. 捕捉函数执行期间，被屏蔽信号多次发送，解除屏蔽后只处理一次。

36.

SIGCHLD产生的条件

子进程终止时：

子进程接收到SIGSTOP信号停止时

子进程处在停止太，接收到SIGONT后唤醒时

借助SIGCHLD信号回收子进程

37.

守护进程：

daemon进程。通常运行于操作系统后台，脱离控制终端，一般不与用户直接交互，周期性的等待某个事件发生或周期性的执行某一动作。不受用户登入注销影响，通常采用以d结尾的命名方式。

创建守护进程：

最关键的一步是调用setsid函数创建一个新的session，并成为session leader

守护进程创建步骤：

1. fork子进程，父进程退出
2. 子进程调用setsid（）创建新会话
3. 通常根据需要，改变当前目录位置chdir（）,防止目录被卸载
4. 通常根据需要，重设文件权限掩码unmask，影响新文件的创建权限。

022 — 755 345 – 432

1. 通常根据需要，关闭/重定向文件描述符
2. 守护进程业务逻辑。while（）

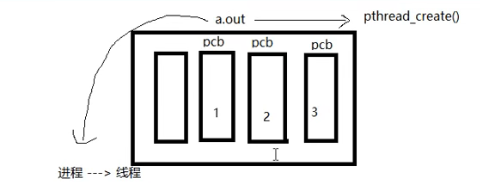
38.

LWP:线程,轻量级的进程

区别

进程：有独立的 进程地址空间。有独立的pcb，进程是最小的分配资源单位

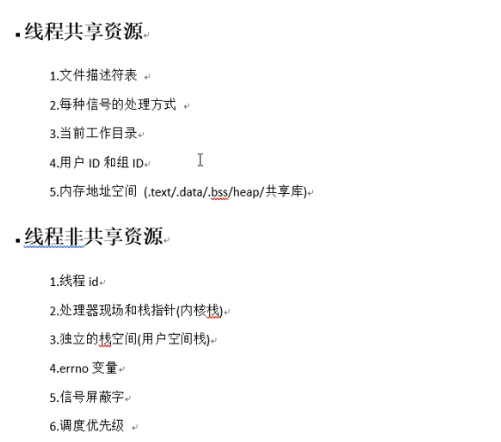
线程：没有独立的进程地址空间，有独立的pcb，线程是最小的执行单位



线程共享：

独享栈空间（内核栈、用户栈）

共享./text./data./rodataa./bsss heap 🡪共享全局变量（不共享errno）



39.

线程ID于线程号不是一个概念：

线程ID是在进程当中标识线程身份的

线程号是标识身份给cpu用的，让cpu分配时间

线程控制原语：

pthread\_t pthread\_self(void);获取线程id。线程id是在进程地址空间内部，用来表示线程身份的id号。

返回值：本线程id

int pthread\_create(pthread\_t \*tid, const pthread\_attr\_t \*attr, void \*(\*start\_rountn)(void \*)), void \*arg)

参数：

参1：传出参数，表示新创建的子线程id

参2：表示线程属性，传NULL表示使用默认属性

参3：子线程回调函数。创建成功，pthread\_create函数返回时，该函数会被自动调用

参4：参3的参数，没有的话，传NULL

返回值：

成功：0

失败：errno

40.

exit（）表示退出进程

return表示返回到调用者那里去

pthread\_exit()表示将当前线程退出

线程中检查出错返回

fprintf(stderr, “xxx error: %s\n”, strerror(ret));

41.

void pthread\_exit(void \*retval)

退出当前线程

参数：

retval：退出值，无退出值时，NULL

int pthread\_join(pthread\_t thread, void \*\*retval);

阻塞回收线程

参数：

thread：待回收的线程id

retval：传出参数，回收的那个线程的退出值

线程异常结束，值为-1

返回值：

成功：0

失败：errno

int pthread\_detach(pthread\_t thread);

设置线程分离后，线程终止，会自动清理pcb，无需回收

参数：

thread：待分离的线程id

int pthread\_cancel(pthread\_t thread);

杀死特定的一个线程，需要到达一个取消点

参数：

thread：待杀死的线程id

返回值：

成功：0

失败：errno

如果子线程没有到达取消点，那么pthread\_cancel无效

可以在程序中，手动添加一个取消点，使用pthread\_testcancel();

成功被pthread\_cancel()杀死的线程，返回-1，使用pthread\_join回收

42.

线程进程控制原语对比：

线程控制原语 进程控制原语

pthread\_create() fork()

pthread\_self() getpid

pthread\_exit() exit()

pthread\_join() wait()/waitpid()

pthread\_cancel() kill()

pthread\_detach()

进程回收必须是父进程回收子进程，线程没有

43.

线程属性：（其中有很多其他属性，这边只学习了分离属性）

int pthread\_attr\_init(pthread\_attr\_t \*attr);

线程属性结构体初始化

参数：

attr：待初始化的线程属性结构体

返回值：

成功：0

失败：errno

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

设置线程属性结构体为分离属性

参数：

attr：前面初始化好的线程属性结构体

detachstate：

PTHREAD\_CREATE\_DETACHED 设置线程属性为分离态

PTHREAD\_CREATE\_JOINABLE 设置线程属性为非分离态

返回值：

成功：0

失败：errno

int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

线程属性结构体初始化

参数：

attr：待销毁的的线程属性结构体

返回值：

成功：0

失败：errno

设置线程分离属性

pthread\_attr\_t attr 创建一个线程属性结构体变量

pthread\_attr\_init(&attr)初始化线程属性

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);设置线程属性为分离态

pthread\_create(&tid, &attr, tfn, NULL);借助修改后的线程属性直接创建为分离态的新线程。

pthread\_attr\_destroy(&attr);销毁线程属性

44.

线程同步：

协同步调：对公共区域数据按序访问。防止数据混乱，产生与时间有关的错误。

锁的使用：

建议锁：对公共数据进行保护。所有线程【应该】在访问公共数据前先拿锁再访问。但锁本身不具有强制性。（互斥锁、读写锁）

互斥量（互斥锁）mutex

pthread\_mutex\_t类型，其本质是一个结构体，为简化理解，可以简单当成整数看待。

pthread\_mutex\_init函数

pthread\_mutex\_destroy函数

pthread\_mutex\_lock函数

pthread\_mutex\_trylock函数

pthread\_mutex\_unlock函数

以上5个函数的返回值都是成功0，失败返回错误号。

使用互斥锁的的一般步骤

1. pthread\_mutex\_t lock;创建锁
2. pthread\_mutex\_init;初始化
3. pthread\_mutex\_lock;加锁
4. 访问共享数据
5. pthread\_mutex\_unlock;解锁
6. pthread\_mutex\_destroy;销毁锁

注意事项：

尽量保持锁的粒度，越小越好，在访问完共享区域要立即解锁

互斥锁，初值为1。（pthread\_mutex\_init调用成功）

加锁，--操作 阻塞线程

解锁，++操作 唤醒阻塞在锁上的线程

try锁：尝试加锁，成功--。失败，返回，同时设置错误号为EBUSY

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

初始化锁，可以想象为锁的值为1

参数：

mutex：互斥量/互斥锁（应定义为全局变量，这样任意线程都可以访问）

attr：设置锁属性的参数

返回值：

成功：0

失败：errno

初始化互斥量：（两种方式）

1. pthread\_mutex\_init（&mutex，NULL）;动态初始化
2. pthread\_mutex\_t mutex= PTHREAD\_MUTEX\_INITIALIZER;静态初始化

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

销毁锁

参数：

mutex：互斥量

返回值：

成功：0

失败：errno

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

加锁.可以想象锁 --

参数：

mutex：互斥量

返回值：

成功：0

失败：errno

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

pthread\_mutex\_trylock() 是 pthread\_mutex\_lock() 的非阻塞版本。

参数：

mutex：互斥量

返回值：

成功：0

失败：errno

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

解锁， 可以想象锁++

参数：

mutex：互斥量

返回值：

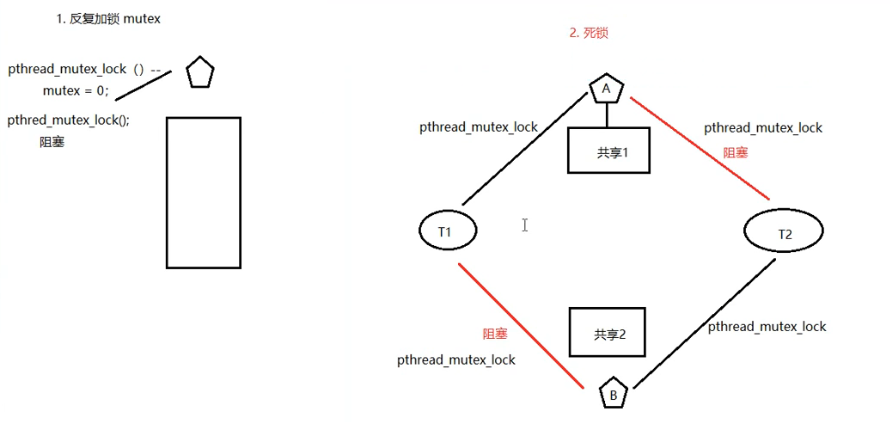
成功：0

失败：errno

45.

死锁：是使用锁不恰当导致的现象。

1. 对一个锁进行反复lock
2. 两个线程各自持有一把锁，请求另一把锁



45.

读写锁：

锁只有一把，以读方式给数据加--读锁。以写方式给数据加--写锁

读写锁特性：

锁只有一把。以读方式给数据加锁--读锁。以写方式给数据加锁--写锁

写独占、读共享。

写锁优先级高

相较于互斥量而言，当读线程多的时候，可以提高访问效率。

pthread\_rwlock\_t类型，用于定义一个读写锁

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock,

const pthread\_rwlockattr\_t \*restrict attr);

读写锁初始化

参数：

rwlock：读写锁（应定义为全局变量，这样任意线程都可以访问）

attr：设置锁属性的参数

返回值：

成功：0

失败：errno

初始化读写锁：（两种方式）

1. pthread\_rwlock\_init（&rwlock，NULL）;动态初始化
2. pthread\_rwlock\_t rwlock = PTHREAD\_RWLOCK\_INITIALIZER;静态初始化

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);

读模式加锁

参数：

rwlock：读写锁

返回值：

成功：0

失败：errno

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock);

尝试读模式加锁

参数：

rwlock：读写锁

返回值：

成功：0

失败：errno

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

写模式加锁

参数：

rwlock：读写锁

返回值：

成功：0

失败：errno

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock);

尝试写模式加锁

参数：

rwlock：读写锁

返回值：

成功：0

失败：errno

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

解锁

参数：

rwlock：读写锁

返回值：

成功：0

失败：errno

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

销毁读写锁

参数：

rwlock：读写锁

返回值：

成功：0

失败：errno

46.

条件变量：

本身不是锁，但是通常结合锁来使用

pthread\_ \_t类型用于定义一个条件变量。pthread\_cond\_t cond;

初始化条件变量：

1. pthread\_cond\_init（&cond，NULL）;动态初始化
2. pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;静态初始化

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,

const pthread\_condattr\_t \*restrict attr);

条件变量初始化

参数：

cond：条件变量（应定义为全局变量，这样任意线程都可以访问）

attr：设置条件变量属性的参数

返回值：

成功：0

失败：errno

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex);

阻塞等待一个条件变量

函数作用：

1. 阻塞等待条件变量cond（参1）满足
2. 释放已掌握的互斥锁（解锁互斥量）相当于pthread\_mutex\_unlock(&mutex);

1,2两步为一个原子操作（原子操作即两部操作不可分）

1. 当被唤醒，pthread\_cond\_wait函数返回时，解除阻塞并重新申请获取互斥锁相当于pthread\_mutex\_lock(&mutex);

参数：

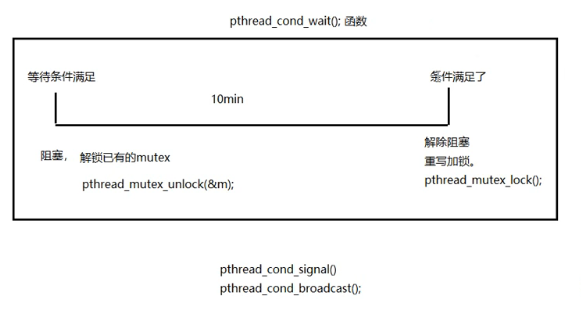
cond：条件变量

mutex：互斥锁

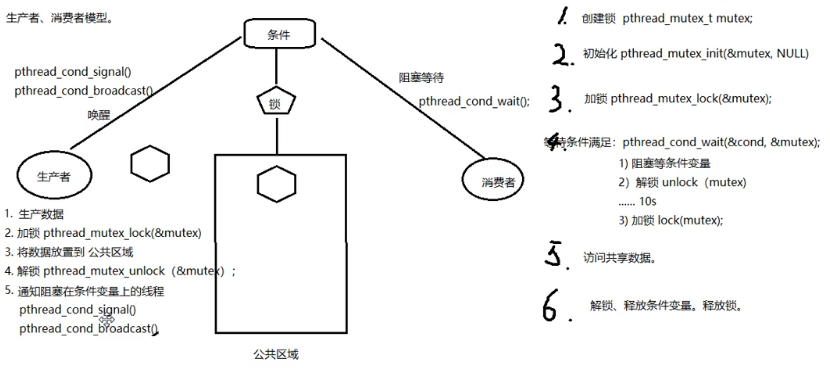
返回值：

成功：0

失败：errno



条件变量实现生产者消费者

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex,const struct timespec \*restrict abstime);

参数：

cond：条件变量

mutex：互斥锁

abstime：超时等待时间

返回值：

成功：0

失败：errno

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

唤醒阻塞在条件变量上的（至少）一个线程

参数：

cond：条件变量

返回值：

成功：0

失败：errno

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

唤醒阻塞在条件变量上的所有线程

参数：

cond：条件变量

返回值：

成功：0

失败：errno

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

销毁条件变量

参数：

cond：条件变量

返回值：

成功：0

失败：errno

信号量：（和信号无关）应用于线程、进程间同步

相当于初始化值为N的互斥量，N值，表示可以同时访问共享数据区的线程数

sem\_t sem 定义类型

sem\_init();初始化信号量

sem\_destroy();销毁信号量

sem\_wait(); 一次调用，做一次--操作，当信号量的值为0是，再次--就会阻塞（对比pthread\_mutex\_lock）

sem\_post(); 一次调用，做一次++操作，当信号量的值为N是，再次++就会阻塞（对比pthread\_mutex\_unlock）

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

参数：

sem：信号量

pshared： 0：用于线程间同步

1：用于进程间同步

value：N值，指定同时访问的线程数

返回值：

成功：0

失败：1

int sem\_destroy(sem\_t \*sem);

参数：

sem：信号量

返回值：

成功：0

失败：1

int sem\_wait(sem\_t \*sem);

参数：

sem：信号量

返回值：

成功：0

失败：1

int sem\_trywait(sem\_t \*sem);

wait非阻塞版本

参数：

sem：信号量

返回值：

成功：0

失败：1

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);

参数：

sem：信号量

abs\_timeout：绝对时间，与相对时间不同，使用时需要注意， 结构体，设置超时时间，超时返回

返回值：

成功：0

失败：1

int sem\_post(sem\_t \*sem);

参数：

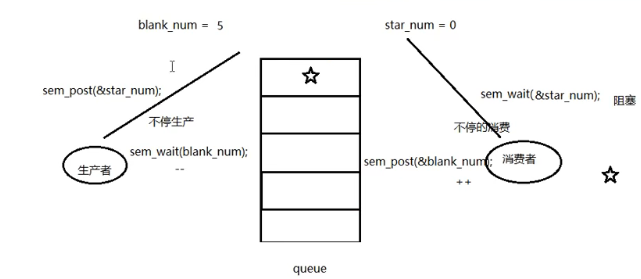
sem：信号量

返回值：

成功：0

失败：1

信号量实现生产者消费者



## 多线程函数总结：

int pthread\_attr\_setinheritsched (pthread\_attr\_t \*attr, int inherit);

设置线程是否继承父进程调度策略

参数：

attr：线程属性结构体地址

inheritsched：是否继承父线程的调度策略

PTHREAD\_EXPLICIT\_SCHED：不继承，只有不继承父线程的调度策略才可以设置线程的调度策略

PTHREAD\_INHERIT\_SCHED：继承父进程的调度策略

返回值：

成功的情况下，返回值为0，失败返回非0值，errno不会被设置

int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy);

设置线程的调度策略属性

参数：

attr：线程属性结构体地址

policy：调度策略

SCHED\_FIFO：抢占式调度，同一优先级中，一旦运行到设置了这个参数的线程CPU将会一直被该线程所占领，不会分配资源给其他实时线程，会分配一点资源给非实时线程

SCHED\_RR：轮询式调度，同一优先级总，遇到这个设置的线程，将会给其运行一段时间后，又继续给下一个人运行（相当于大家平均运行），会分配一点资源给非实时线程上面的两种是针对静态优先级1-99的实时线程才能设置的。

SCHED\_OTHER：其他普通的调度策略，仅能设置与0静态优先级，也就是非实时线程，让这条线程成为一个由系统去自动根据动态优先级分配资源的任务。

返回值：

成功的情况下，返回值为0，失败返回非0值，errno不会被设置

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr,

size\_t stacksize);

设置线程的堆栈大小

参数：

attr 是线程属性变量；

stacksize 则是设置的堆栈大小。

返回值：

成功的情况下，返回值为0，失败返回非0值，errno不会被设置

int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct sched\_param \*param);

设置线程的优先级

参数：

attr ：是线程属性变量；

param ：是一个结构体，需要设定线程的优先级

返回值：

成功的情况下，返回值为0，失败返回非0值，errno不会被设置

pthread\_condattr\_setclock函数：

功能：此函数用于设置pthread\_cond\_timewait函数使用的时钟ID

