5.19.2022

通过不同层次进行封装进行相关的内容，然后进行接口的调用。

操作系统，是管理的软件。进行硬件与软件的管理。理论的道理。

书写的所以的程序都是应用层的开发。

先用键盘进行开发应用。指令距离操作系统更加近

man命令查找说明

cp 拷贝文件（不可以拷贝目录）， -f强制拷贝。

mv 文件名 文件名 交换目录 可以进行重命名。 -i强制的移动。

不要轻易的改变文件。

Cat 打印文件内容，-n打印带上序号。

Echo 简单内容 >（输出重定向）书写到相关的内容 （输出到文件之中）

>>重定义，并且不会清空所有的东西在文件之中。

find \_\_\_ \_\_\_\_ 寻找文件

Grep “对象” 文件名 //找出相应内容所在位置信息。 -v（保下没有相关内容的具体消息）。行匹配工具。

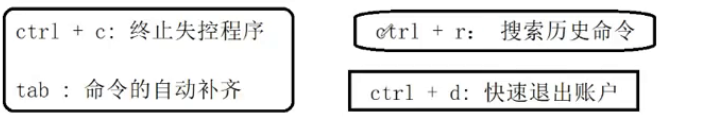
6.2.2022

通过压缩包进行东西部署,节约空间。

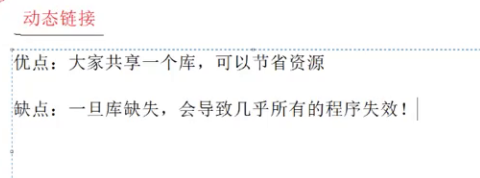
使用不同的压缩的包类型，使用的工具不同，进行东西的部署。

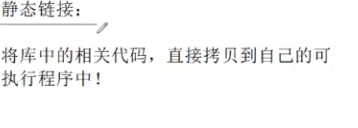
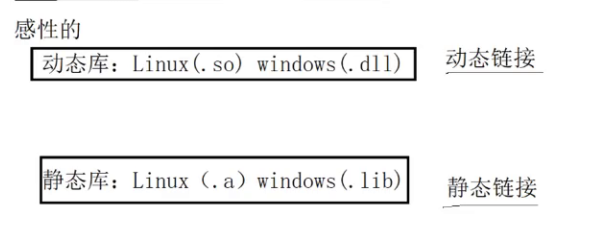
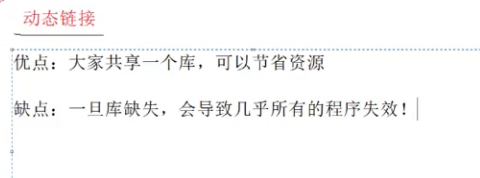
压缩过程zip（打包与压缩）unzip解压

tar zxf(vxf)压缩与解压（） -c指导路径



su登陆root用户，以及更换用户名。





利用gdb（工具的使用）进行调试代码：

list：

b 进行断点打入 r 跑程序 p + 相应打东西可以达到出现在屏幕上面 undisply 去掉循环的数字 n 进行循环的下一步 until 退出循环 info b查看断点

c 从一个点跑到另外一个断点 s 进入函数 finish 跑完这个函数就停下

项目构建：

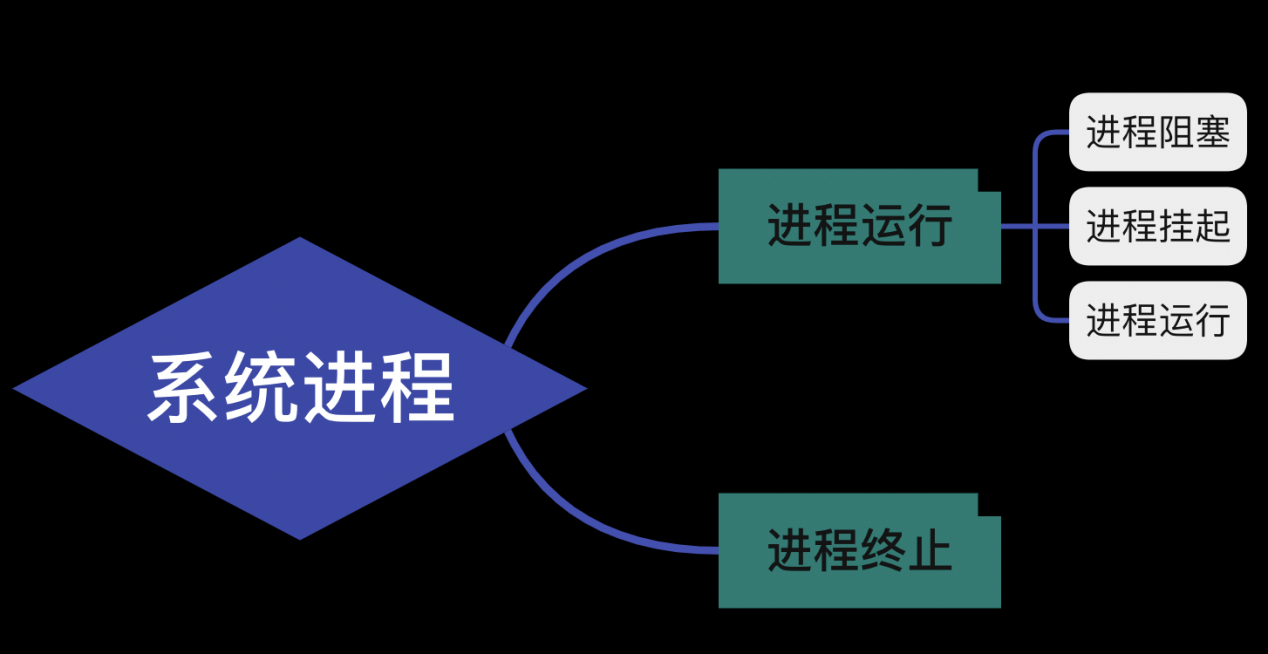
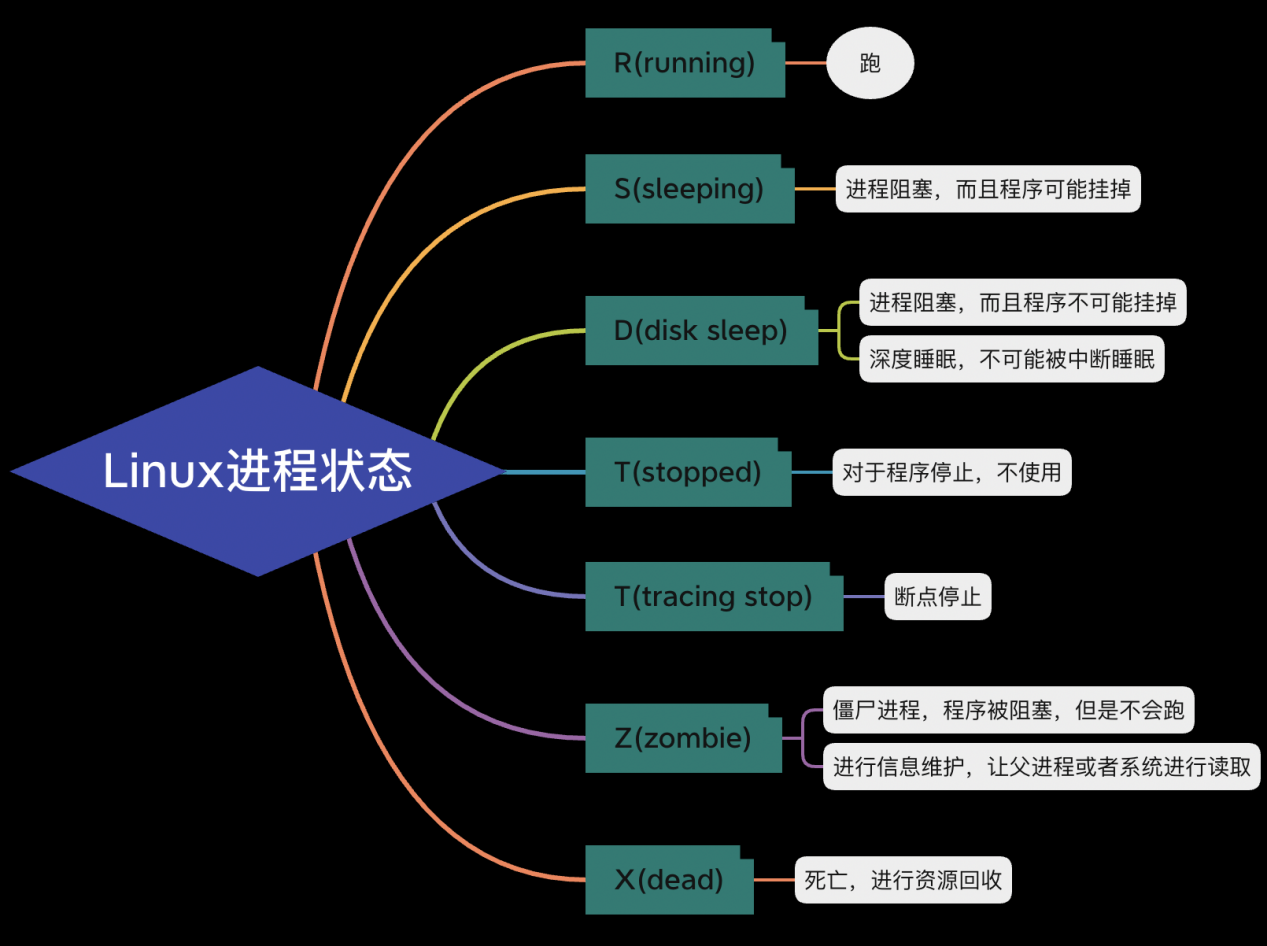
进行编辑相关的内容。使用makefile直接进行东西编译运行。并且需要在项目项目创建一个

内容可以直接进行书写相关的通过的过程。

8.5

makefile的文件开头可以是大写！还有相应的依赖文件的相关的内容。

stat可以查看相应的文件配置信息

8.12

8.12.2022

cpu

并行：多个程序运行在不同的核心之中

并发：多个程序只运行在一个核心之中（多重数组实现执行内容）

环境变量：不同的程序有不同的环境变量。

export 声明环境变量（等式）

命令行变量：1，普通变量 2，环境变量

PATH为环境变量的赋值内容$PATH进行赋值。（set查看本地变量）

echo $NAME看见常见的环#境变量

常见的环境变量：

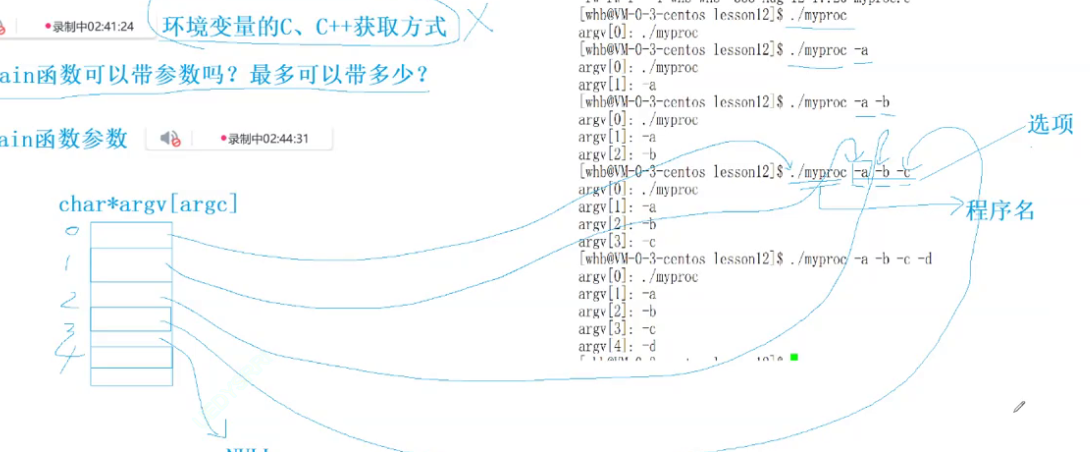
$HOSTNAME 用户名 $SHELL 命令行路径 $USER 用户名 $PWD 用户目位置

环境变量的导入:先进行声明（export），然后赋值（$ = )。

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

main函数参数：const char \* argv和int argc为命令行参数！

env为环境变量字符串



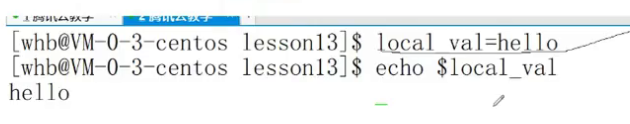
通过命令的填写可以进行，命令行参数的书写！aoti将字符串转换成为数字。Linux的指令的来源于此！

进程会被传入环境变量。

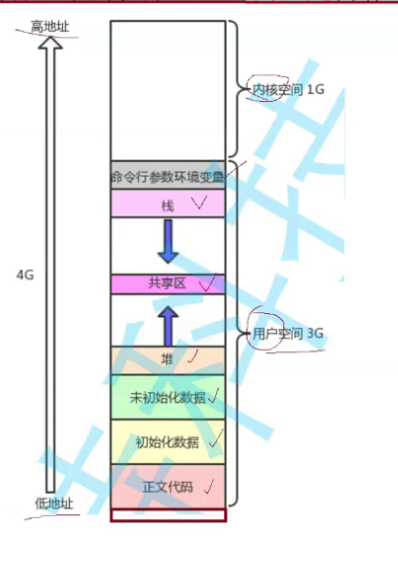
getenv（）进行环境变量的获取。环境变量可以进行用户等相关信息的使用，进行权限的判断对于相关的功能的使用！

8.13.2022

环境变量可以被子进程继承！本地变量不会被子进程继承下去



Linux大部分命令通过子命令运行，但是还有部分由于bash自己进行执行（叫做内建命令）。



stactic放在全局变量数据区域

没有父子修改数据的时候，父子进程公用一个数据！

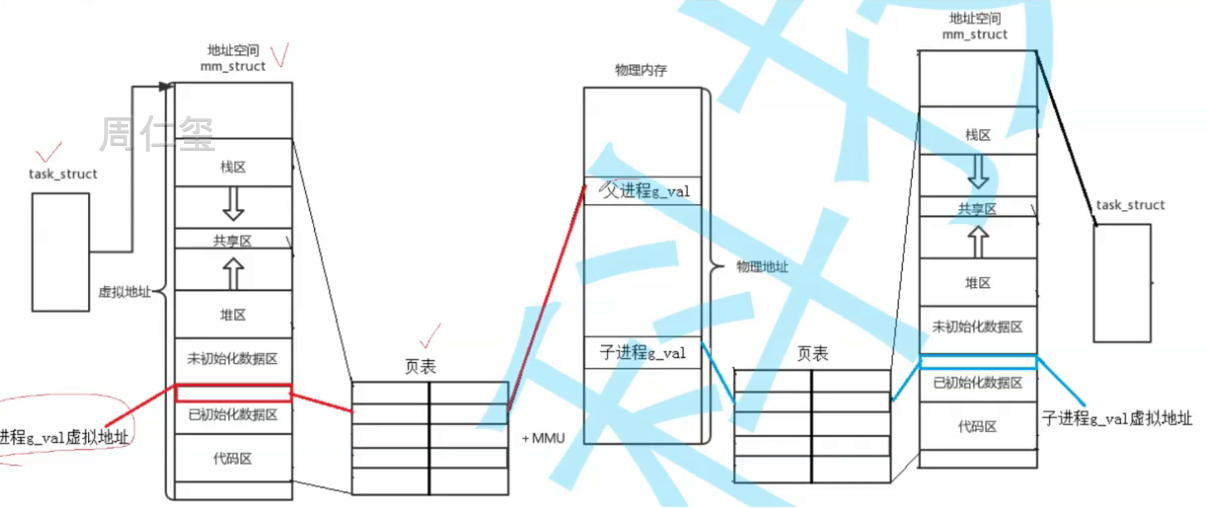
在c++/c之中使用的不是物理地址（虚拟地址，线性地址，逻辑地址[在Linux基本相同，，其他地方不同]）！

每一个进程在启动的时候，都会让系统创建一个地址空间。每一个进程都有自己的地址空间！

OS要进行地址空间的管理。

实际地址与虚拟地址进行映射。（自动生成映射表）

每一个进程有不同的task\_struct相应的关系，对于内存地址不同。



父子进程具有独立性，互相不影响

可以使用这一种东西进行内存保护，防止指针越界

8.15.2022

子进程与父进程共享代码，而且从子进程创建开始具体的过程！可以选择开始相应的位置开始。

映射表还有读写表之类的东西。

***进程终止：***

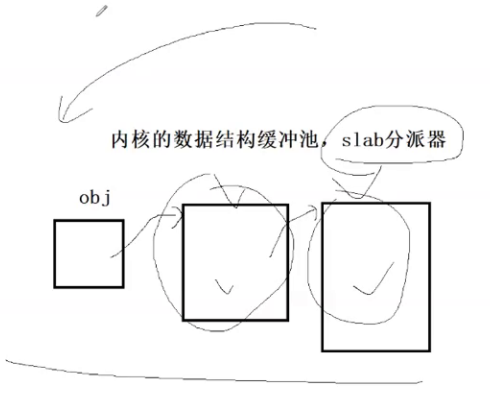
代码执行成功返回为0，否则为非零。（自己书写的！）这个东西是进程退出码，表现进程退出信息。

表现退出的信息，让父进程进行读取相关的内容。使用echo $?进行直接获取进程退出码！退出码，没有固定的关系。

exit（）进程退出，在任意位置，刷新缓冲区。\_exit()直接退出，不会刷新缓冲区。

main（）函数，return表示进程退出。

进程 = 内核结构 + 进程代码 和 数据。



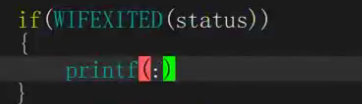
***进程等待：***

目的，获取子进程的退出信息。

#include<sys/wait.h> pid\_t waitpid(pid\_t pid,int \*status, int options);

pid表示是那一个子进程。Options用0表示阻塞等待。 wait从系统之中调用相关的内容。status可以拿到子进程的信息码。

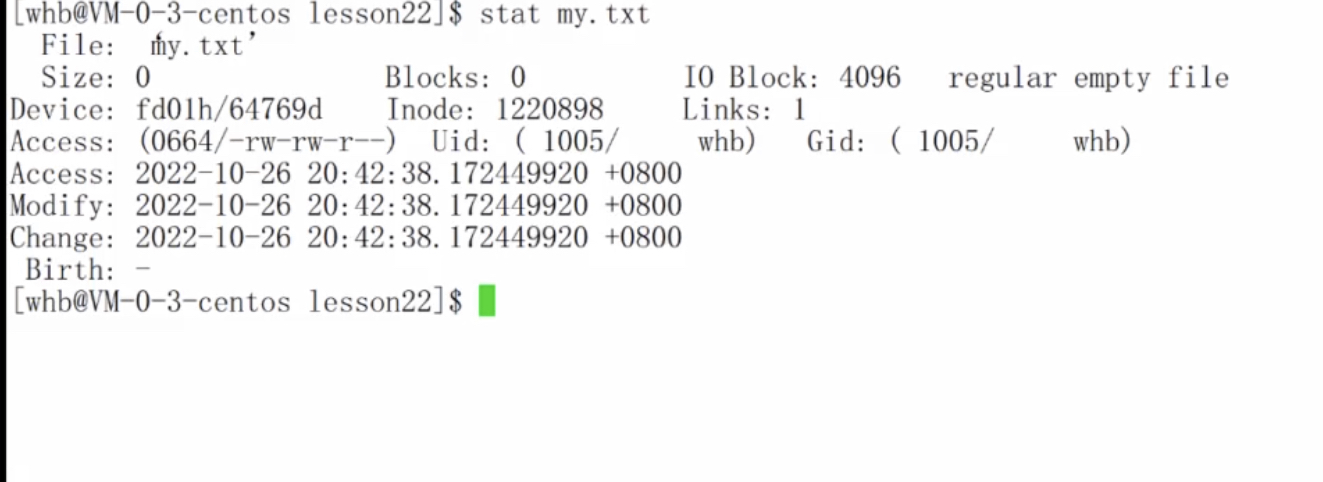
子进程退出信息写入task\_struct 。



系统启动首先进行BIOS进行启动相关的内容。

inode储存文件信息,确定数据文件信息。

删除文件，删除映射表的内容。ionde进行系统内容的寻找。



磁盘访问以扇区为单位（提高效率）。磁带是以线性表进行访问相关内容。

操作系统按照4kb进行读取，操作系统也是按照4kb进行储存。

对于磁盘进行控制，只需要管理相应位置的头指针（分片进行管理）。分治进行管理相关的内容。

Linux将内容与属性分开储存。属性储存在inode之中。

Linux底层没有文件名的概念。通过index进行标记文件。

使用ls -i 进行inode的查询！

In -s a b;软链接，让 b 作为 a 的快捷方式。

硬链接：不使用-s；

软链接：独立的文件（有自己的inode与indoe编号）。

硬链接：不是独立的文件（与原始文件共享同一个indoe）。

软链接相当与快捷方式。硬链接：新增指定文件名和inode的映射关系。

In建立映射关系。

使用unlink进行链接删除。

indoe其实加上指针，指向内容的地址（编号）。

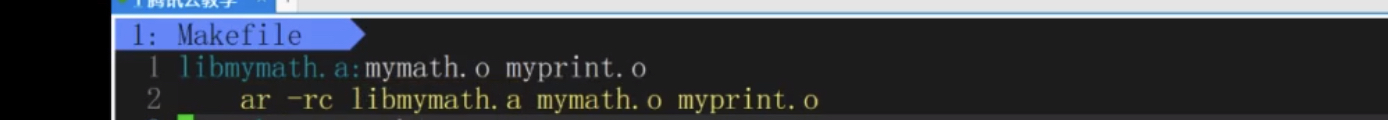
任何目录都有./这个文件名。./表示当前位置。

..指向上一级目录的indoe(.指向第一个目录)。

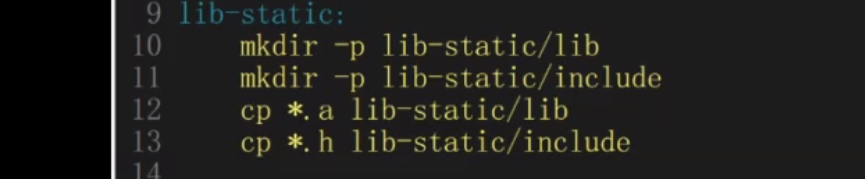
动静态库制作：

形成.o程序进行链接起来，进行相关内容的使用。

以lib开头，.a进行结尾。

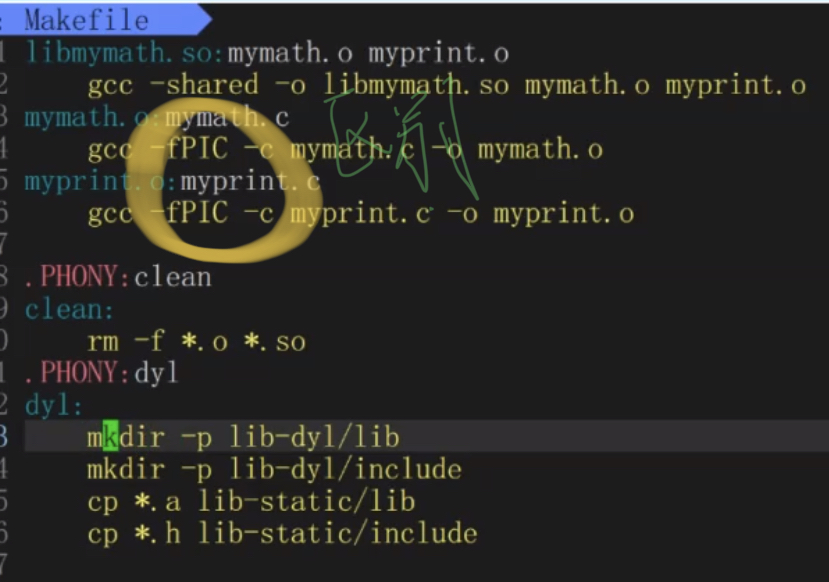


生产.a的库文件，其中ar使用的关于相关的组件。



创建文件之后统一进行删除相关的内容。

形成动静态库：



libmymatch.so：

使用动静态库的区别：相对位置使用的就是动态库，静态库固定的位置信息。

编译器在寻找头文件。动静库在/lib里面。把自己书写的静态库拷贝到系统库里面。

库首先要形成可执行程序（通过对于.a与.so,进行处理。gcc -l链接第三方库的名称）。

把库文件与头文件，拷贝到系统路径下面。不推荐这种方法，会污染系统库（名字相同等问题！）

gcc -I 指定路径（搜索路径）。进行更改寻找的寻找范围。

这个没有到系统文件里面的库进行链接的过程。

lld命令可以查看动态库的相应的内容。

gcc告诉库的位置与动态库，然后没有告诉程序相应的库的位置信息（动态库才会发生这种情况，静态库在形成程序的使用已经代码镶嵌到程序之中了，动态库没有将库放置在程序之中）。

/user/lib64普通用户按照库的位置。/lib64为系统库的相应的位置信息。

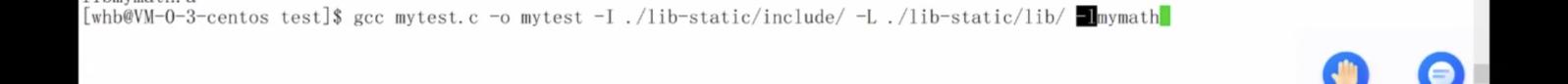
卸载库：指的是直接移除库。

库的使用：

静态库的使用

需要引用相应的名称。所有的头文件都放在/user/include/之中。

这个的安装到非系统之中的操作。



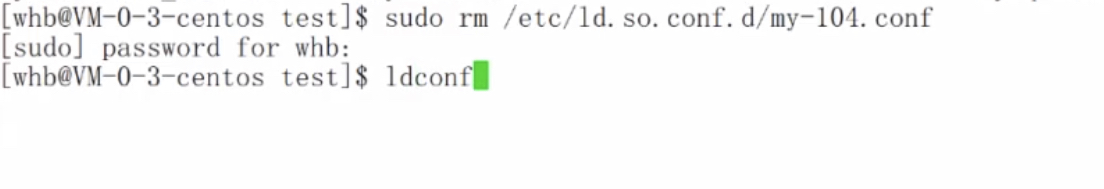
找头文件 库在哪里 链接库的名称

或者可以直接安装到系统之中，就不用上面的操作了。

动态库的使用

LD\_LIBRY\_PATH就是动态库的储存位置。要

ldd 程序名称 可以看出链接的第三方库的位置信息。



In -s 路径 可以直接链接申上去了。在系统之中建立软链接进行操作。

\*.so本身就是可指向程序。

动态库：可以被多个程序进行共享

11.2.2022

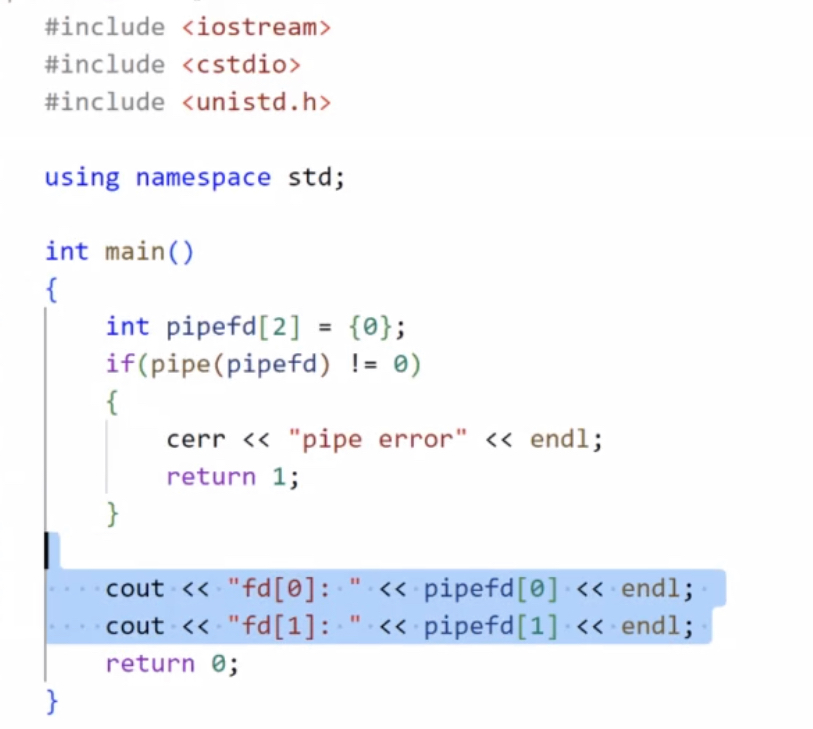
匿名管道：都是半双工（通信的条件）进行相关的操作，具体的东西不进行介绍了。一般都在父子进程之中进行相关的内容通信。读写的过程之中有相应的顺序，必须父大于子。如果管道满了，需要进行阻塞等待。

匿名管道创建过程：

int fd[2];

int pipe(int fd[2]);进行管道创建。管道基本上就可以看作是一个文件，然后被写入相关的东西和文件的写入基本相同。write，close，read等进行相关的操作！

* 当没有数据可读时
* O\_NONBLOCK disable: read调用阻塞，即进程暂停执行，一直等到有数据来到为止。
* O\_NONBLOCK enable：read调用返回-1，errno值为EAGAIN。
* 当管道满的时侯
* O\_NONBLOCK disable： write调用阻塞，直到有进程读走数据
* O\_NONBLOCK enable：调用返回-1，errno值为EAGAIN
* 如果所有管道写端对应的文件描述符被关闭，则read返回0
* 如果所有管道读端对应的文件描述符被关闭，则write操作会产生信号SIGPIPE,进而可能导致write进程  
  退出
* 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。
* 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。

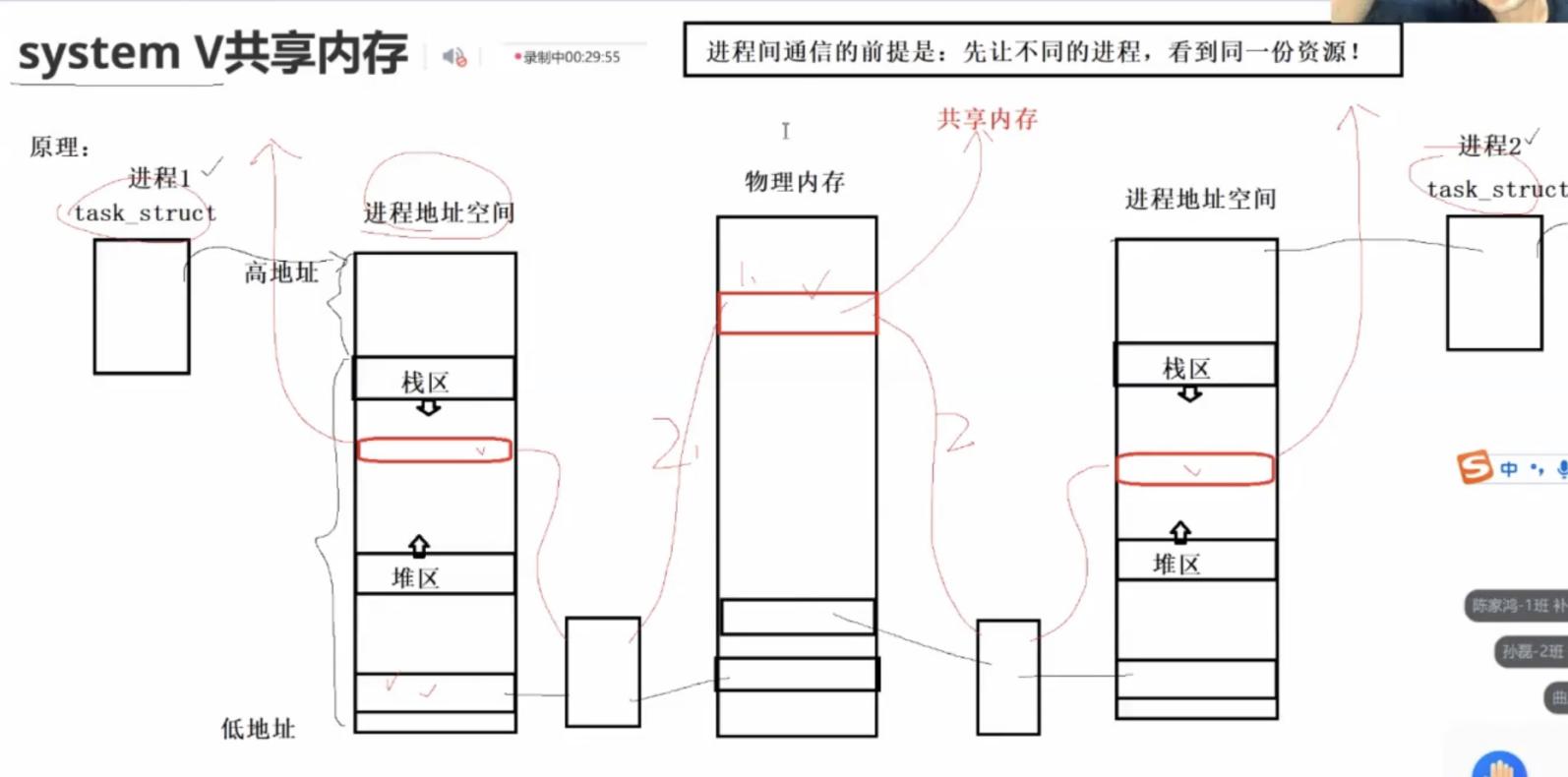


0下标为读，1下标为写。使用close进行管道的关闭等的内容。只可以单向通信半双工。使用write。可以让东西，进入管道，储存东西到管道里面！read可以从管道进行东西的读取。读取成功返回值为大于0，反之小于0.

命名管道：mkifo(const char \*filename,mode\_t mode);子进程之间进行通信。 使用open进行打开命名管道。

-———

进程通信的本质让不同的资源看到同一份资源。



具体的使用规则类比 上面的相识的东西！看到同一份资源才可以进行相关的内容通信（不同程序之中的关系）。

使用下面这个参数进行通信！

共享内存以4KB进程开辟，但是OS赋予的空间时候是4KB的倍速。

**#include** **<sys/shm.h>**

int **shmget**(key\_t key, size\_t size, int shmflg);

返回小于0，创建不成功！对于0创建成功！key为对于访问的密钥，size为共享内存的大小。

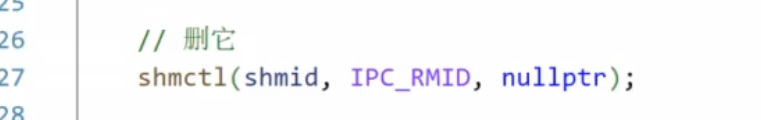
运行这个程序之后，推出之后！第二次，重新不可以运行，只有删除生成程序文件之后，才可以。或者重新启动系统。

shmflg的相关东西

|  |
| --- |
| IPC\_CREAT：当shmflg&IPC\_CREAT为真时，如果内核中不存在键值与key相等的共享内存，则新建一个共享内存；如果存在这样的共享内存，返回此共享内存的标识符 |
| IPC\_CREAT|IPC\_EXCL：如果内核中不存在键值 与key相等的共享内存，则新建一个共享内存；如果存在这样的共享内存则报错 |

共享内存

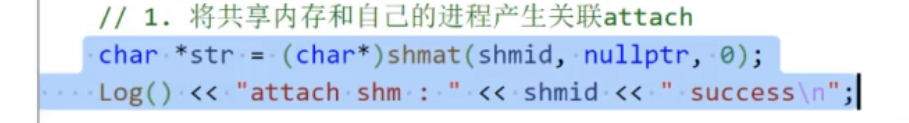
ipcs -m 查看创建的共享内存。ipcrm -m shmid(的相应名称)，可以删除内存共享的东西！

、

使用这个空间与malloc的空间的使用方法基本相同（除了不可以free以为的都可以进行操作）！

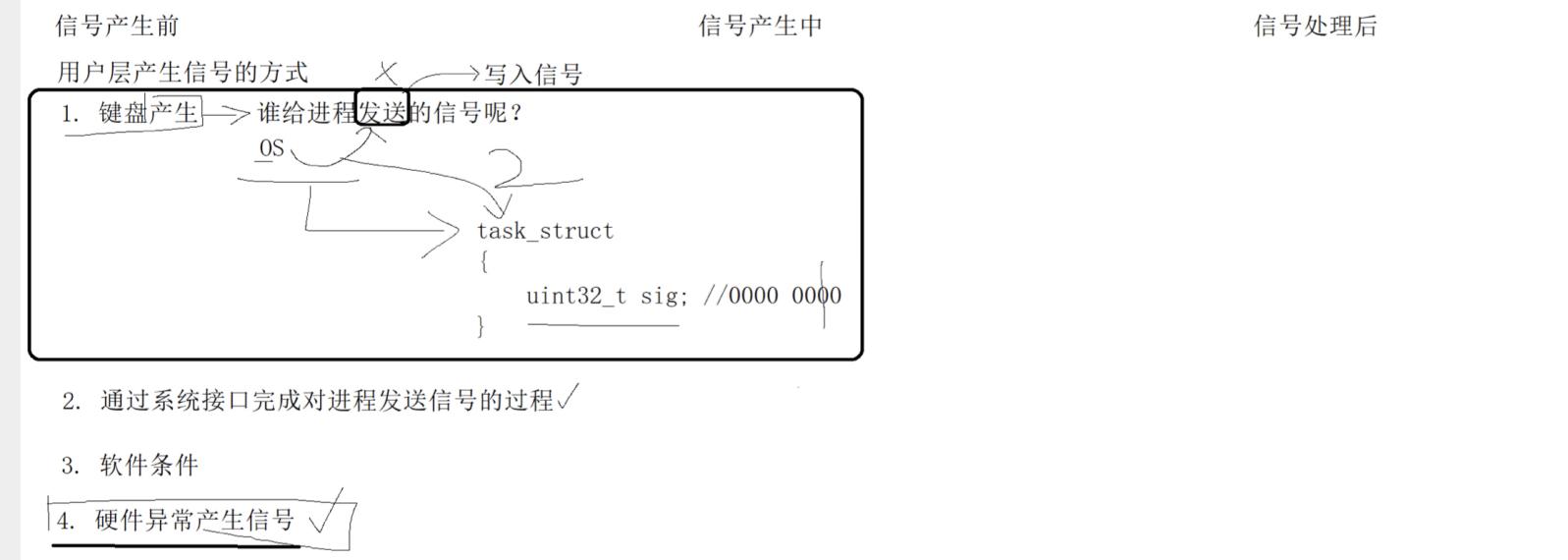
这个使用的PCB里面套struct ipc\_perm结构，进行内存的东西进行维护。

获取共享地址。通过shmat进行操作相关的内容。

klik

信号：是异步（不同的优先基本）！ 进程不需要立即处理信号。产生了信号，但是这个信号不一定要被处理。信号有优先级别。 信号有：1，默认动作 2，忽略 3，自定义动作 信号的处理。

在进程PCB（进程结构体之中）之中，进行信号的记住通过位图去实现相关的内容（uint32\_t sig）,通过比特位进行判断。而且在里面都是在内核数据结构！信号都是要系统OS进程接受的相关的内容。



中断： 硬件向cpu发送针脚的具体内容信息（中断信息），然后软件接收到具体信息（中断数组），执行中断表。

abrot可以进行信号终止。

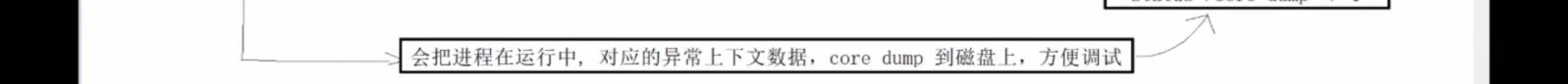
int kill(pid\_t ,int sig);向任意进程传输任意的信号。返回值为0与1的相关的大小。通过arg的命令可以杀相关的内容。

除了9号信号不会被捕捉，其他的会被捕捉。

操作系统就是一个死循环。还是需要进硬件的处理相关的问题。

是谁在推动操作系统做一系列的动作呢？

硬件 --时钟硬件 --给os发送时钟中断 alarm



core dump，一般是被关闭的。

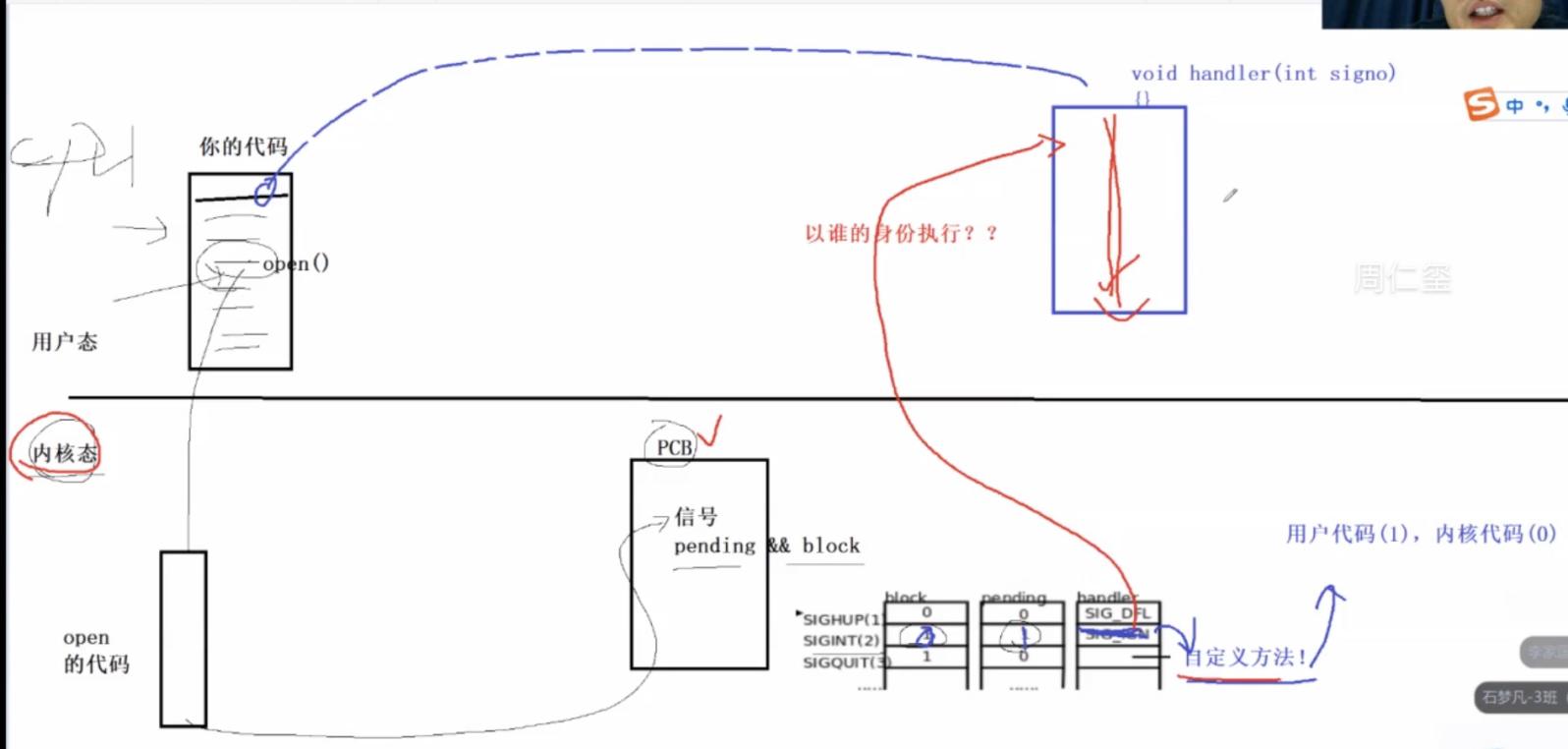
sigpending（）；获取进程pid位图信息。

信号集合：

11.8进程信号

内核态可以访问内存，但是不一定会使用相关的内容。

你无形之中都在切换相应的不同的相同级别的代码。内核的东西更换，时间片到了之后，切换内核级页表，保护上下文，然后继续更换相关的信息。



用户书写的代码，相应使用用户态进行指向（防止发生相应的冲突的等的相关的内容）。

信号在内核态转化为用户态是获取相应的信息，然后进行内容处理。

#include <signal. h>

int sigaction (int signum, const struct sigaction \*act, struct sigaction \*oldact)；

可重入函数：

volatile关键字：不准备让编译器做任何的优化。

父进程需要等待子进程的相关内容。

11.10.2022

创建一个程序：PCB，页表，物理空间。

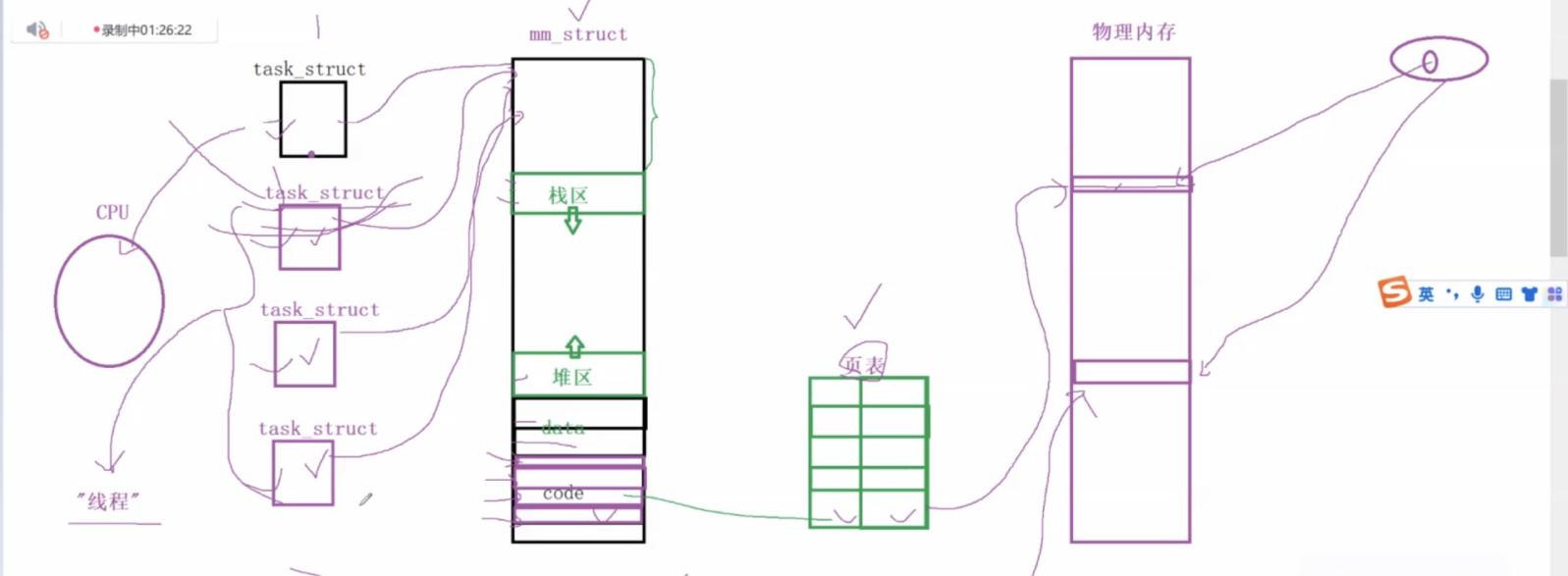
线程

1在进程内部运行的执行流

2. 线程比进程比调度成本更低

3. 线程是CPU调度的基本单位

线程跑的是一部分的代码。



线程：进程：n：1.

TCB？ Linux是使用PCB进行模拟TCB（线程），轻量化线程。 进程 = 承担分配资源的基本实体（进程的基座实体）。向系统申请资源的基本的单位。

线程是调度的基本单位。

#include <pthread. h)

int pthread create (pthread t \*thread, const pthread attr\_t \*attr,void \* (\*start routine) (void \* , void \*arg) ;

虚拟地址到实际地址，通过MMU进行管理。

线程的优点：对于数据进行加密与解密等的问题。

线程的缺点：线程的性能损失。

pthread库，几乎所有Linux都有这个原生库。

**pthread\_self**(void);获取运行的进程id

 int **pthread\_join**(pthread\_t thread, void \*\*value\_ptr);进程退出，一般要join如果不进行就会造成内存泄露。 后面的参数可以获取退出码。

线程异常等于进程异常，线程是进程的最小单位。线程的健壮性比较差，会影响其他人。

void **pthread\_exit**(void \*value\_ptr);进行线程终止

int **pthread\_cancel**(pthread\_t thread);取消线程的指向，退出不成功返回-1.

线程的控制：

syscall

线程分离：

线程在默认在被创建的时候，线程必须要是被joinable。

不关系线程的退出码，可以使用pthreas\_detach(pthread\_t thread);进行主线程分离，处理相关的情况，主线程不关注线程的具体的运行结果。

1. 立即分离，延后分离 --线程活着

延长后退出，不关注线程的死活。

2，新线程分离，但是主线程先退出（进程退出），所有的线程都会退出。

一般分离线程，主线程都不会退出main thread都是常驻的。

可以使用exit也可以退出进程，任何线程调用都表示进程退出，而且还可以提取推出码。

临界区：访问临界资源的代码叫做临界区。

临界资源：多个指向流可以访问的资源

互斥：访问某一个资源的时候，只可以有一种执行流进行访问。

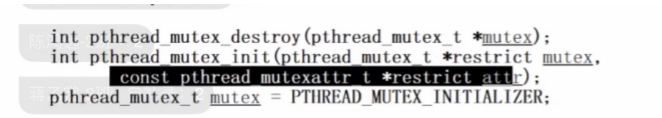
判断不是原子的进行执行，线程的是加减不是原子的。

**#include** **<unistd.h>**

int **usleep**(useconds\_t microseconds); 进行程序休眠，执行的时间以微秒为单位。

互斥锁：

静态或者全局的可以使用宏进行初始化。



pthread\_mutex\_lock();阻塞示申请，这可能一个人使用锁，等待锁被使用之后，然后直接申请。pthread\_mutex\_try（）；如果说被使用的话就直接退出，如果没有被使用就直接申请。

临界区，只要对临界区加锁，而且加锁的粒度约细越好

加锁的本质是让线程执行临界区代码串行化

加锁是一套规范，通过临界区对临界资源进行访问的时候，要加就都要加

任何资源需要访问加锁的区域都需要进行申请锁，其他都是要看到这一把锁。

这个锁也是临界资源。

申请锁与竞争锁的过程都是原子的。

11.13

解锁，就是转换的的使用人。

锁的申请实现：

多个线程访问寄存器，但是互相不影响。

线程上锁：总线进行上锁，

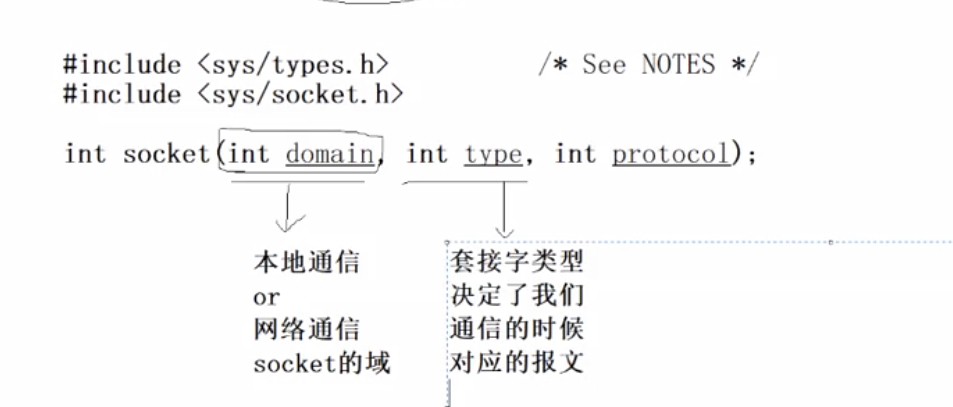
2022.11.23

网络需要进行稳定性，好管理性。层状的网络协议都可以直接进行通信。

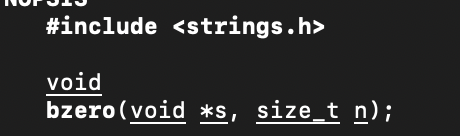
2022.11.27

网络的服务

UDP的服务相联：



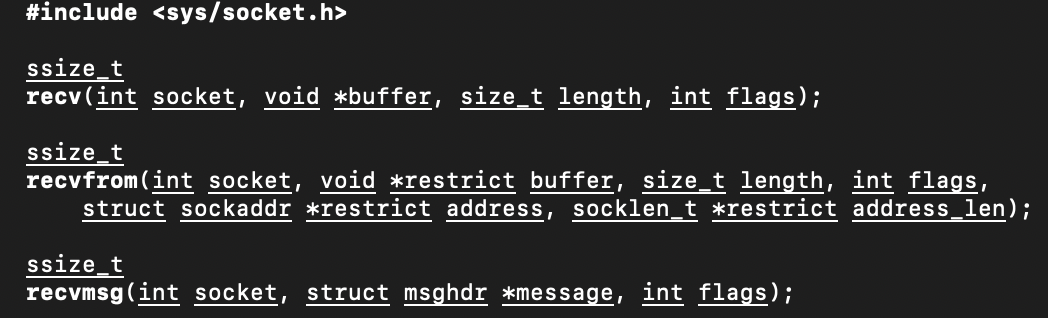
第一个参数为通信的类型，第二的参数的形式为数据流或者ip数据报，等的相关的内容。 后面一个为协议类型，一般设置为数字零。 返回为文件描述符号。

控制s的相应的大小进行控制。



进行联络ip通信的相关的内容。

服务器的一切都是死循环。



参数读取相关的东西。

客户信息通信：创建套接字，初始化服务器地址，发送数据（接受数据），关闭套接字。