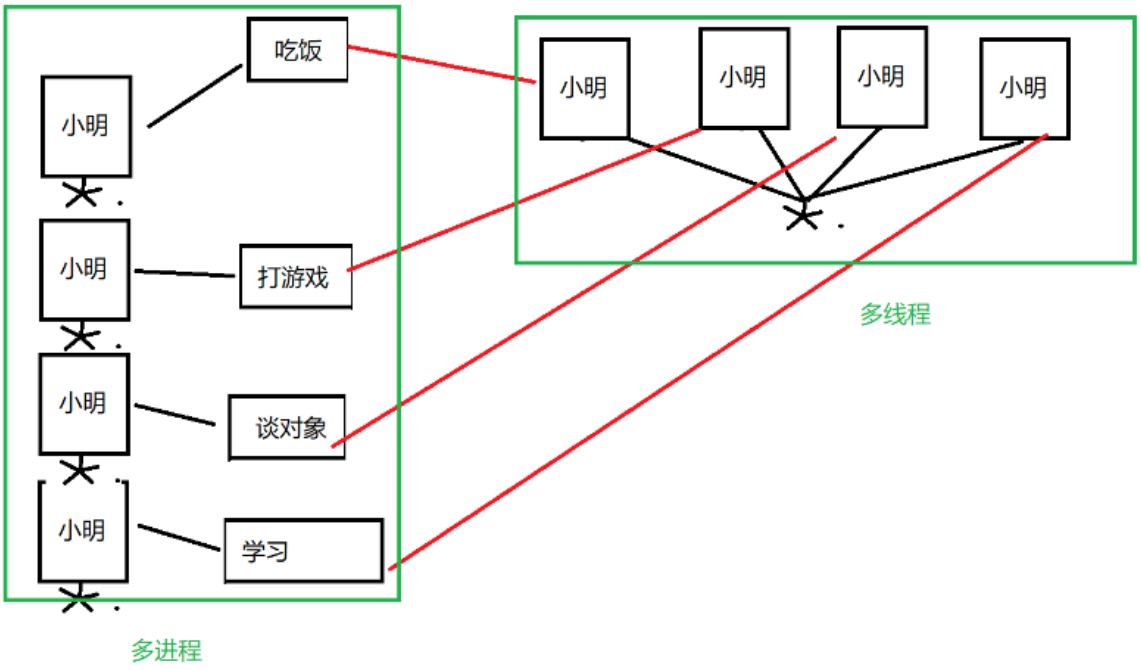
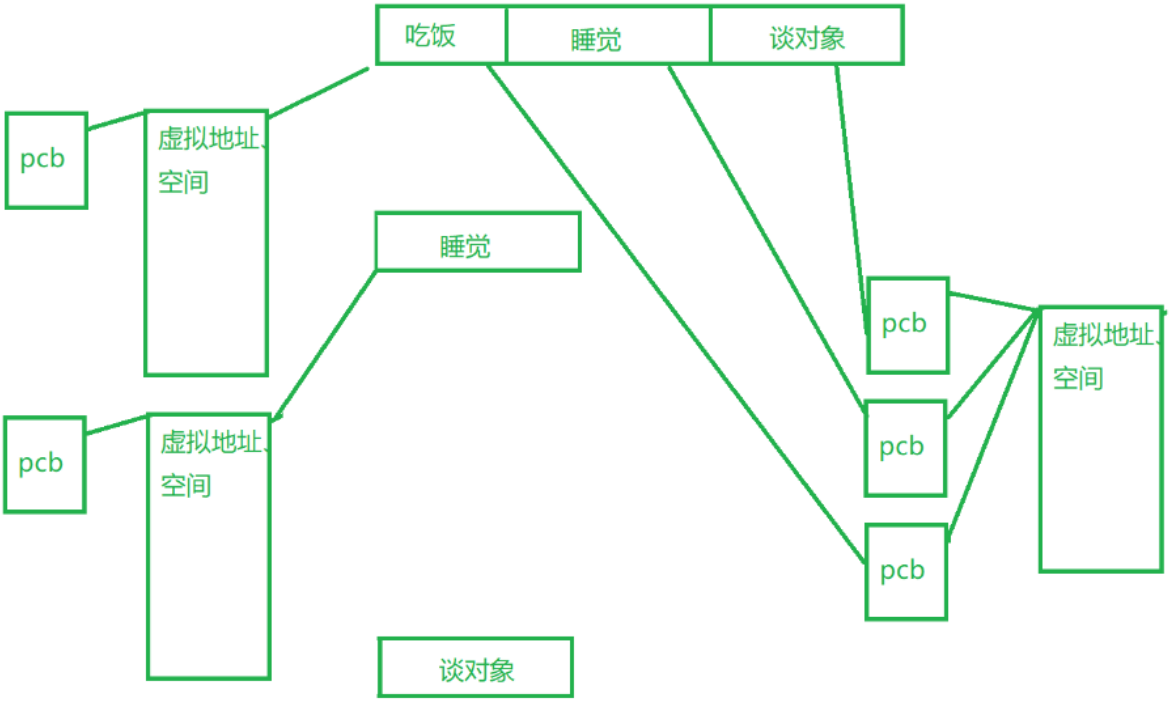
多线程：线程概念/线程控制/线程安全/知识的整合应用

## 线程概念：

多线程其实是实现多任务并发/并行处理的 / 多进程也是完成多任务的

通过多线程/多进程进行多任务处理的不同之处来理解什么是线程




多线程的处理思路：一个运行中的程序中，具有多个执行流，各自完成一个功能模块的实现

linux的操作系统中：认为一个pcd就是一个执行流(pcd是操作系统调用一段程序运行的实体--描述了程序的运行过程)

linux下的线程就是一个pcd

pcd：是一个进程

现在，多个pcd可以共用同一个虚拟地址空间，这些pcd共用了一个运行中程序的资源

linux下的线程就是一个pcd是一个轻量级进程，因为一个运行中程序的多个pcd共用同一份资源

一个运行中的程序就是一个进程，以前我们所说的进程，这时候再来解释，就是具有一个线程的进程，linux下的线程是一个pcb(轻量级进程)，是一个进程中的一条执行流；
这时候再来理解进程的话：进程就是一个线程组；

若以后面试官问到进程/线程
1、进程跟线程是要一起说的(否则单一说会有一种冲突感)
2、进程就是一个运行中的程序，操作系统会创建一个pcb(运行中程序的描述)，并且分配资源，通过pcb来调度运行整个程序
3、线程是一个进程中的执行流，但是linux下实现进程中的执行流的时候，使用了pcb实现
4、因此就说linux下的线程是一个pcb,称作轻量级进程,因为同一个进程中的线程共用进程分配的资源
5、而进程就是所有线程的统称，就是一个线程组，系统在运行程序，分配资源的时候是分配给线程组，分配给整个进程的
进程是资源分配的基本单位，线程是cpu调度的基本单位
进程就像是一个工厂，线程就是工厂里干活的工人

vfork--创建出来的子进程与父进程共用同一个虚拟地址空间，但是父进程会阻塞，因为同时运行会出现栈混乱
线程是一个pcb; 一个进程中若有多个线程，也就意味着有多个pcb,这些pcb共用同一个虚拟地址空间;
它是如何做到同时运行而不会出现栈混乱的情况的呢?
线程之间的独有与共享
**独有**:栈(每个线程一个，就可以避免出现栈混乱了)，寄存器(每一个pcb都是一个执行流)，信号屏蔽字(阻塞自己想阻塞的信号)，errno (每调用一次系统调用都会重置errno)
**共享**:代码段和数据段(虚拟地址空间)，信号的处理方式，I0信息， 工作路径，用户id,组id

多线程与多进程都能进行多任务处理，我们以后写代码的时候就要根据各自的优缺点以及应用场景来选择技术

多线程任务处理相较于多进程的**优点**:

1.线程间的通信更加灵活方便(出了进程间通信方式以外，还可以通过全局变量/函数传参实现通信)

2.线程的创建和销毁成本更低(线程间共享进程等的大部分资源)

3.同一个进程中的线程间调度成本更低(切换页表,数据)

4.线程的执行粒度更加细致

**缺点:**

线程间缺乏访问控制，有些系统调用(exit)或者异常是针对整个进程产生效果

多线程任务处理没有多进程任务处理稳定性高

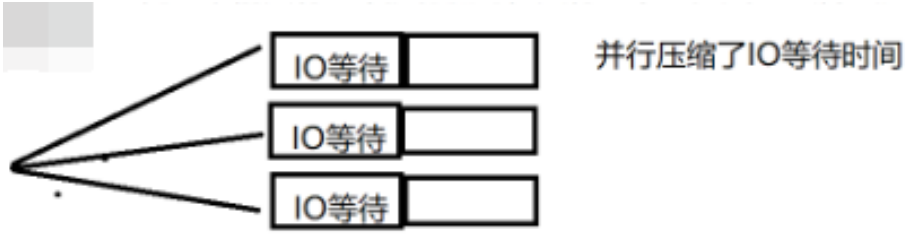
应用场景:多进程应用场景，对主程序的稳定性安全性要求更高的场景，比如shell/网络服务器;

多进程/多线程进行多任务的并发处理的共同优势:

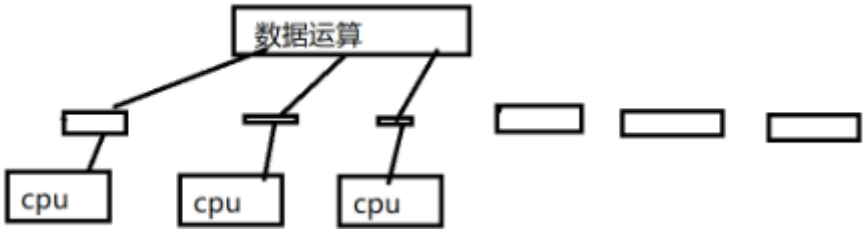
**IO密集型程序:**程序中大部分的工作都是进行IO　　占用cpu极少

可以在一个执行流中发起一个IO;避免了以前只有一个执行流时,只有一个IO完成了之后才能进

行下一个的情况，提高了IO效率




**CPU密集型程序**：程序中的大部分工作都是进行数据运算




在cpu密集型程序中，执行流的创建并不是越多越好，多了反而会提高cpu调度的成本

cpu密集型程序中，线程的创建最好是cpu核心数+1

# 线程控制:线程创建/线程终止/线程等待/线程分离

讲的是对线程所能进行操作接口的学习

实际上linux操作系统并没有给用户提供创建一个线程的系统调用接口，用户无法直接创建线程;

但是大佬们封装了一套线程库，通过库函数可以实现线程控制的各种操作

我们创建线程就是在用户态创建线程，可以实现在内核态创建一个轻量级进程实现调度

**线程创建:**

int pthread_create(pthread_ t *tid, pthread_attr _t  *attr, void* (*thread_ routine) (void *arg),void *arg)

tid:输出型参数，用于向用户返回线程id,是后续线程操作的句柄

attr:用于设置线程属性，通常置NULL

thread_ routine: 线程的入口函数

arg:最终会通过线程入口函数的参数传递给线程的数据

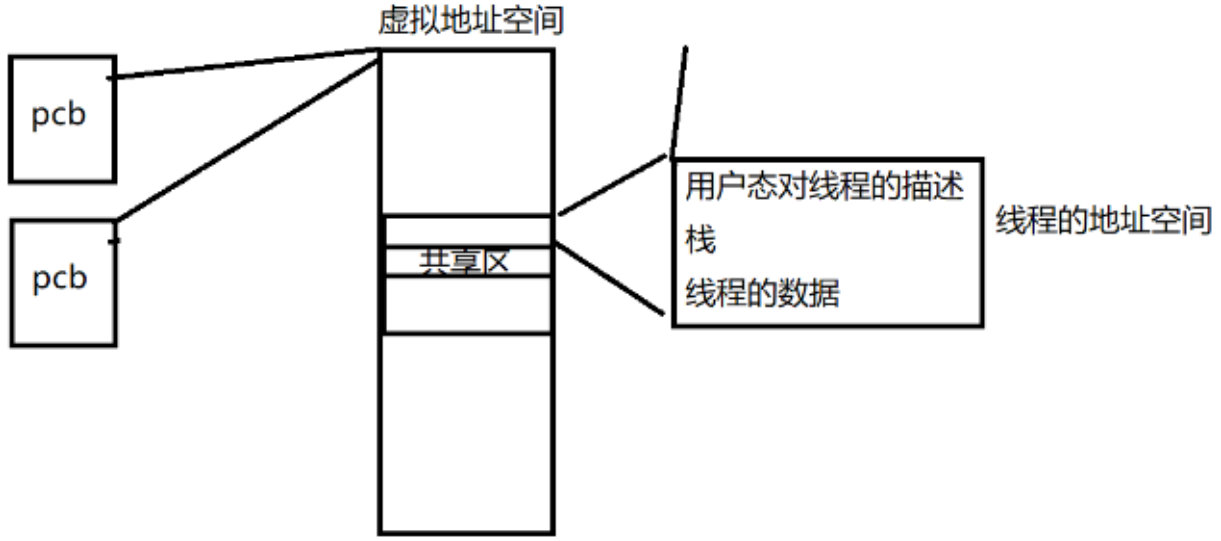
返回值:若成功则返回0， 失败返回一个非0值

使用ps  -L选项可以查看轻量级进程信息

LWP这一项是轻量级进程id,实际上就是pcb中的pid

在外边查看到的进程id实际是pcb中tgid-线程组id　　线程组id等于进程中主线程的轻量级进程id的值

tid是什么，如何成为线程的操作句柄?




创建线程，会在进程的虚拟地址空间中开辟一块相对独立的空间作为线程的地址空间，这个空

间内包含用户态对线程的描述，实现对线程的操作，线程栈...；创建线程成功之后，会将这

块空间的首地址返回给用户;

用户对线程进行操作，就可以通过首地址找到线程的数据，进行进行一系列的操作，

因此返回的线程地址空间的首地址就是线程的操作句柄，也就是pthread_create函数返回的tid

线程的地址空间也是在进程的地址空间之内的一部分

tid:线程空间在虚拟地址空间中的首地址
PID:实际上是线程组id  tgid = mainthread-> pcb-> pid
LWP:实际上是每个线程pcb中的pid  pcb-> pid

pthread_t pthread_ self() :返回调用线程的线程id
pthread_ create创建一个线程的接口应用
pthread_ self()获取线程id的接口应用
线程中的id (tid  pid  lwp)

**线程终止**:退出-一个线程
主动退出:线程入口函数中的return (main函数中的return是 用于退出进程的而不是主线程) ;
void pthread_exit(void *retval); 退出调用线程;  retval :返回一个退出返回值;
被动退出: int pthread_cancel(pthread_t thread);取消- 个线程; thread: 要取消的线程id

在主线程中调用pthread_ exit可以退出主线程，但是不会退出进程
只有所有的线程都退出了，进程才会退出

# 复习

多线程:
线程概念:
.线程是什么:
1.在最早学习进程的时候，进程是一个pcb, 是一个运行中程序的描述，linux下是一个task_struct结构体，描述了一个程序的运行信息,通过描述的这些信息，可以实现操作系统对运行中程序的调度;
2.在学习到线程这节课的时候，线程是进程中的一条执行流， linux 下线程是一个task_ struct结构体，是一个pcb, 这些pcb在一个运行中的程序中共用同一个虚拟地址空间，相较于传统的pcb更加轻量化,因此linux 下的线程是一个轻量级进程
进程与线程的关系就像是一个工厂与工人的关系;

多任务处理中:多进程就像多建几个工厂;而多线程就像在工厂中多招几个工人

多个线程共用同一个虚拟地址空间，如何做到不会栈混乱?
线程间的独有与共享:
线程独有的数据:栈，寄存器，信号屏蔽字，errno
线程共享的数据:虚拟地址空间，信号的处理方式，I0信息， 工作路径，用户id/组id

在多任务处理中到底是多线程好，还是多进程好:分析各自的优缺点，针对不同的应用场景而定
多线程的优点:
1.线程间的通信更加简单灵活(进程间通信能实现的方式，线程间都可以，除此之外全局数据，函数传参)
2.线程的创建与销毁成本更低(线程只是进程中一条执行流， 共享了进程的大部分资源)
3.同-个进程中的线程间调度成本更加低(不需要重新加载数据/页表信息..)
多进程的优点: .
1.因为进程间的独立性，因此多进程程序更加健壮稳定/线程就不够了，因为有些系统调用和异常是针对整个进程的多进程适用场景(shell/网络服务器)
多进程/多线程进行并发/并行任务处理的优势:
I0密集型程序:在程序中进行大量的I0操作，但是对cpu资源要求并不多
在操作系统层面实现平衡化(循环调度;循环对一个线程中的IO发出请求,不需要等待上一个完成后才能进行下一个的请求)， 并行压缩了I0的等待时间
CPU密集型程序:在程序中进行大量的数据运算,对cpu资源要求更多
多执行流操作，可以更加充分的利用cpu资源;在多cpu情况下， 多个执行流可以实现并行处理，提高效率;
在这种情况下，到底线程创建多少个比较合适: cpu核心数+1 (线程创建的太多，反而会增加调度成本)

1.线程是什么，与进程的关系是什么样的
2.线程间数据的独有与共享
3.多线程与多进程的优缺点分析以及多执行流任务操作的优势

线程控制:线程创建/线程终止/线程等待/线程分离
操作系统并没有提供系统调用接口实现线程的控制，而当前即将要讲的线程控制的接口，都是大佬封装的库函数

通过库函数创建线程:在内核中创建pcb实现调度,并且为了便于在用户态用户的操作，因此在用户态进行了封装描述
用户态进行的封装描述:用户态线程--在内核中通过一个轻量级进程实现调度
线程创建:
int pthread_ create(pthread _t *tid，pthread_ attr_ t *attr, void*(thread_ routine)(void *arg)，void *arg)
tid:每个线程在进程虚拟地址空间中都有一个相对独立的空间，返回的tid就是这个空间在虚拟地址空间中的首地址
返回值:成功返回0;失败返回非0值的-一个错误编号
线程中的id: pcb->pid pcb->tgid tid
pthread_ t pthread. self(void) ://获取当前调用线程的tid

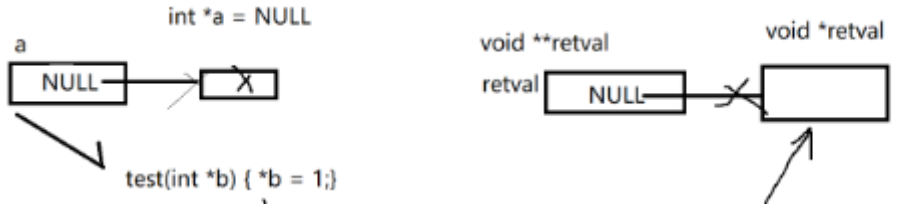
线程的终止:
线程主动退出:
1.在线程入口函数中，调用return; (main函数中的return, 退出的并不是主线程，而是进程)
2. void pthread_ exit(void *retval); (主线程也可以退出， 但是主线程退出，进程并不一定退出，所有线程退出，进程才会退出)
线程被动取消:
1. int pthread_ cancel(pthread_ t tid);取消一个正在运行的线程

1.进行线程操作使用的是库函数，理解用户态线程与轻量级进程
2.线程创建的接口使用,以及几个id理解
3.线程终止的几种接口的应用

线程等待:等待一个指定线程的退出;获取这个退出线程的返回值;并且允许系统回收这个线程占用的资源:
但是:并不是所有的线程都需要被等待，因为线程有一个属性，默认叫joinable,处于这个属性的线程，退出后不会自动回收资源;需要其它线程进行等待处理;
如何等待:
int pthread_ join(pthread_ t tid, void **retval);
tid:用于指定要等待的线程
retval:输出性参数， 返回线程的退出返回值

retval为什么是一一个二级指针:
int a= 10 test(int *b){*b= 1;} test(&a) a=? ->

线程的返回值是一个一级指针，因此应该使用二级指针(也就是对一级指针取地址)
int *a{这个变量拥有一块空间一可以存放另与块空间的地址}




对指针解引用，就是访问指针保存的地址所指向的空间

在函数中定义char *buf = "leihoua~"与char buf[] = "leihoua~ “有什么区别?
"leihoua~"这个字符串是一个常量， 存放在正文段
char*buf=常量的地址　　将常量的地址赋值给了buf，buf这个指针就指向了常量的空间
char buf[] =常量　　给buf开辟空间，将常量的数据赋值给了这块空间

1.为什么要进行线程等待----一个线程默认属性为joinable,这种线程退出后不会自动释放资源，因此需要被等待进行处理
2.如何等待int pthread, join(pthread, _t tid, void **retval);//等待一个线程退出，并且获取返回值
3.线程等待过程中涉及到的指针的操作

等待线程退出，原因是joinable属性的线程资源无法自动被回收;过程等待线程退出，获取返回值，释放资源在如果不想获取线程的返回值的情况下，能不能直接释放线程资源，不需要再去等待----线程分离的实现

线程分离:设置线程属性，从joinable设置 为detach,处于detach属性的线程退出后，会自动释放资源(这种线程不需要被等待)pthread_ join是线程等待--阻塞函数(如果线程没退出，就一直等待)

如何分离一个线程:
int pthread detach(pthread_ t tid);分离一个指定的线程 - -设置这个指定线程的属性为detach
分离一个线程，可以在任意位置，任意线程中完成，

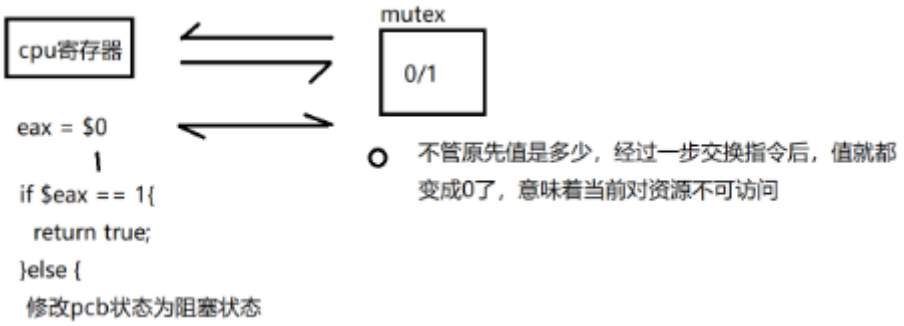
1.线程分离是在干什么---设置属性
2.被分离的线程有什么效果---退出后自动释放资源，不需要被等待
3.如何分离--- pthread_ detach

线程创建;线程终止;线程等待;线程分离
1.等待的作用，为什么需要等待
2.分离的作用,
3.接口的使用

# 线程安全:

多个执行流对临界资源进行争抢访问，而不会造成数据二义或者逻辑混乱;称这段争抢访问的过程是线程安全的;线程安全的实现:如何保证多个执行流对临界资源进行争抢访问而不会造成数据二义
同步:通过条件判断，实现对临界资源访问的时序合理性
互斥:通过唯一访问，实现对临界资源访问的安全性
互斥的实现技术:互斥锁/信号量
实现互斥的原理:只要保证同一时间只有一个执行流能够访问资源就是互斥
　　对临界资源进行状态标记:没人访问的时候标记为1, 表示可访问;有人正在访问的时候， 就标记为0,表示不可访问;在对临界资源进行访问之前先进行状态的判断，决定是否能够访问，不能访问则使其休眠
能够实现互斥的技术中:互斥锁
互斥锁:其实就是一个计数器，只有0/1的计数器，用于标记资源当前的访问状态 1-可访问 0-不可访问
互斥锁想要实现互斥，每个线程在访问临界资源之前都要先访问同一个互斥锁(加锁)；
意味着互斥锁本身就是一个临界资源，(涉及 到计数器的修改修改过程必须保证安全，因为如果连自己都保护不好，就不能保护别人)
互斥锁的计数器操作如何实现原子性:

1.将cpu寄存器上的值修改为0，然后与内存中计数器进行数据交换(意味着这时候计数器变程了0,谁来访问，都是-种不可访问状态，别人都进不去，这时候，寄存器就可以慢慢判断是都可以访问了)
2.若寄存器交换后数据为0，则表示当前不可访问，则将pcb状态置为阻塞状态，线程将被挂起等待，若寄存器交换后数据为1,则表示当前可以访问,则加锁操作直接返回,表示加锁成功--继续可以访问资源
3.访问完数据之后，要进行解锁操作(将内存中计数器的值再修改回来)




1.互斥锁是一个计数器， 本身的计数操作是原子性(如何保证自己的原子性)

2.互斥锁如何实现互斥，通过在访问临界资源之前先加锁访问互斥锁，来进行状态判断是否可加锁

互斥锁的代码操作流程:

1.定义互斥锁变量pthread_ mutex _t mutex;

2.初始化互斥锁

mutex=PTHREAD_ MUTEX INITIALIZER

int pthread_ mutex_ init(pthread_ mutex. t *mutex, pthread_ mutexattr_ t *attr)

mutex:互斥锁变量首地址

attr:互斥锁属性---通常置NULL,

3.在对临界资源访问之前，先加锁(访问锁，判断是否可以访问) --- 保护对临界资源访问的过程

int pthread_ mutex_ lock(pthread mutex_ t *mutex);阻塞加锁---如果不能加锁，则一直等待

int pthread_ mutex_ trylock(pthread_ mutex_ t *mutex);非阻塞加锁---如果不能加锁，则立即报错返回,若可以加锁，则加锁后返回

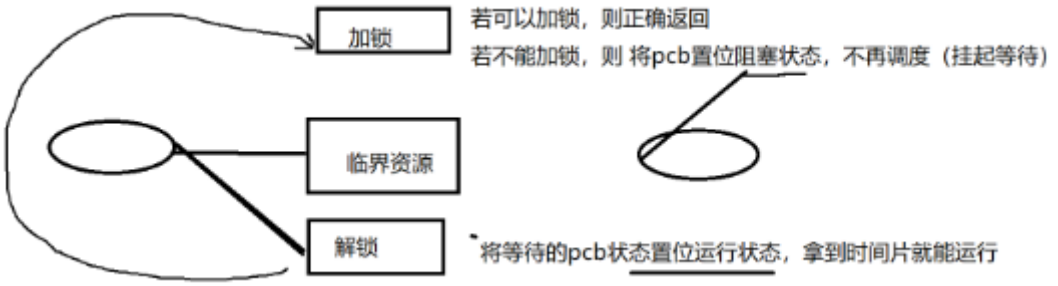
4.在对临界资源访问完毕之后，记得解锁(把状态标记为可访问)

int pthread_ mutex_ unlock(pthread_ mutex_ t *mutex);

5.不使用锁了,最终要释放资源，销毁互斥锁

int pthread_ mutex_ destroy(pthread_ mutex_ t *mutex);

在黄牛抢票的例子中，加锁后，一个黄牛抢完了所有的票没有票了,为什么大家不退出



互斥锁的操作流程中，需要注意的:

1.加锁后，在任意有可能退出线程的地方记得解锁

2.锁的初始化一定要在创建线程之前，

3.锁的销毁一定是保证没有人使用互斥锁的时候