创建线程：

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void \*arg);

thread ： 线程ID，

attr： 线程的一些属性，这些属性包括很多东西，比如线程的调度策略，线程的优先级，线程栈的大小，…

void \*(\*start\_routine) (void \*arg)：它是线程执行函数的回调。

arg：传递给线程的参数

为什么需要使用多线程：

1. 多个线程，那就是意味着我的应用程序可能会在多个CPU上执行，做到了并发编程实现。
2. 如果有阻塞的情况发送，我们完全可以把这个阻塞放在线程里，让我们程序还可以继续往下执行，

T1：阻塞

T2：执行

main()

{

…阻塞 // 线程挂起来，

}

1. 计算密集型的任务，也可以放在一个线程里去执行。

SCHED\_FIFO, SCHED\_RR：可以设置线程的优先级。

默认是SCHED\_OTHER：此时不能设置线程的优先级

非分离属性：就是子线程和主线程还有关系，这个关系体现在子线程的销毁上，主线程一定要调用pthread\_join去销毁这个线程。

分离：线程创建的时候和主线程没有任何关系，线程退出，线程资源就被销毁。

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

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

CPU的亲缘性设置：

线程在CPU上执行的时候：CPU 缓存，经常访问到的资源（变量）缓存CPU缓存（L1，L2，L3）, 高并发网络IO，cash line ： 64k

int pthread\_setaffinity\_np(pthread\_t thread, size\_t cpusetsize,

const cpu\_set\_t \*cpuset);

int pthread\_getaffinity\_np(pthread\_t thread, size\_t cpusetsize,

cpu\_set\_t \*cpuset);

可以通过获取CPU的数量：sched\_getcpu

栈属性

默认的线程栈的大小只有1M。

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

void \*stackaddr, size\_t stacksize);

int pthread\_attr\_getstack(const pthread\_attr\_t \*attr,

void \*\*stackaddr, size\_t \*stacksize);

线程私有数据（TSD）:

i; key – value

进程是管理资源的单位，线程是CPU调度基本单位：

进程：创建了线程

进程和线程的区别：线程的前世今生

远古时代：只有一个CPU，对于用户，我们期望应用程序能够同时运行起来：CPU分时，假的同时执行应用程序的现象。

SMP : 多核，多个CPU，理论上就能同时执行程序。CPU1 CPU2 CPU3 …… fork() 进程他们的，他们有独立的地址空间：

能不能有一种办法？比较能够方便地工运行在不同的CPU上的进程共享资源？

LinuxThreads：clone： 创建一个进程去模拟一个线程：线程就是一个进程，pid

kill：

NTPL : Redhat IBM : NTGL

POSIX :

共享相同的资源 ： 进程里所有的资源在线程里可以任意访问

线程也有自己的内存空间：g\_number,

T1: g\_number,

T2: g\_number,

T3: g\_number,

1. 调度代价低；
2. 共享资源，便利与用户开发。
3. 进程的资源是独立的，如果出现了段错误或者其它的致命错误，线程的方式会导致整个进程退出，也就是多个线程都会退出；但是多进程实现，只会影响一个进程。

4）多线程处理并发相当困难。

并行编程：同步？： 多个线程共享了资源，那么势必需要按照某种顺序去访问和请求这些资源。

Lee哥去坐火车，遇到了苍老师： 门 ： 门上面会显示“关” “开” ， 有人 / 无人。

厕所：是公共资源，谁都可以用，

怎么解决这种共享资源的排它性：有顺序的访问，不会造成乱套。

flag ： true xxx

false ooo

flag = false;

T1 :

请求锁住()

if(flag)

{

xxx();

}

else

{

ooo();

}

释放锁()

T2 :

请求锁()

set (flag = true);

解锁();

同步：

一类相关的数据在更新时，并没有完全完成，但是此时有另外的线程开始使用这些更新的值，这就会造成数据的不一致。

数据库：t1 t2 t3

T1:

update\_db(t1); 10

update\_db(t2); 100

---

update\_db(t3); 1000

T2 :

get\_from\_db(t1);

get\_from\_db(t2);

get\_from\_db(t3);

------

1. 排它性
2. 可重入性

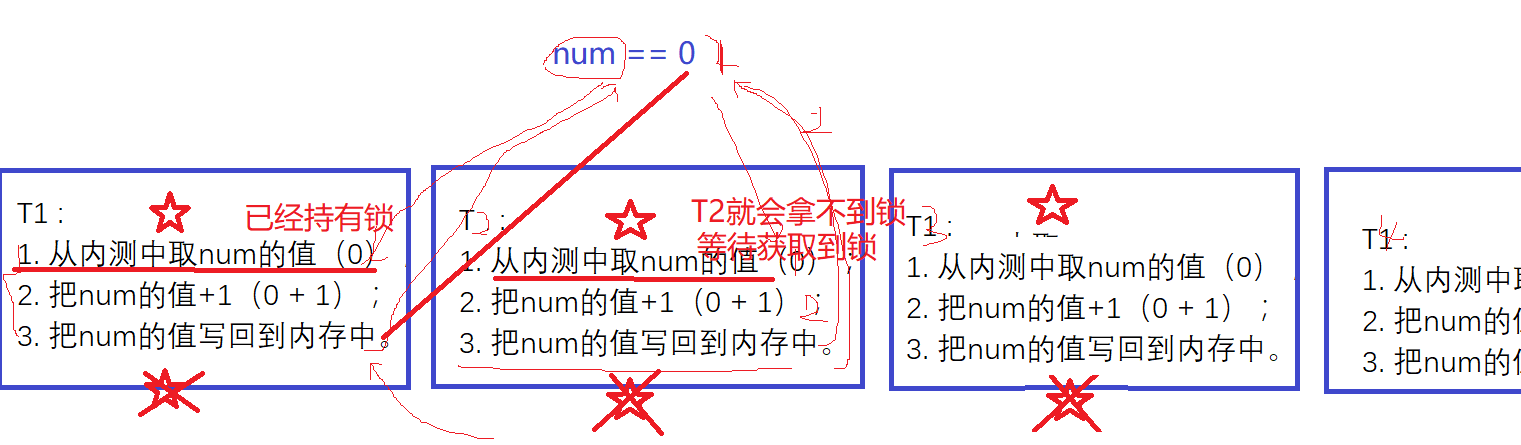
同步的手段有哪些：

1. Mutex :
2. pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

b．pthread\_mutex\_lock

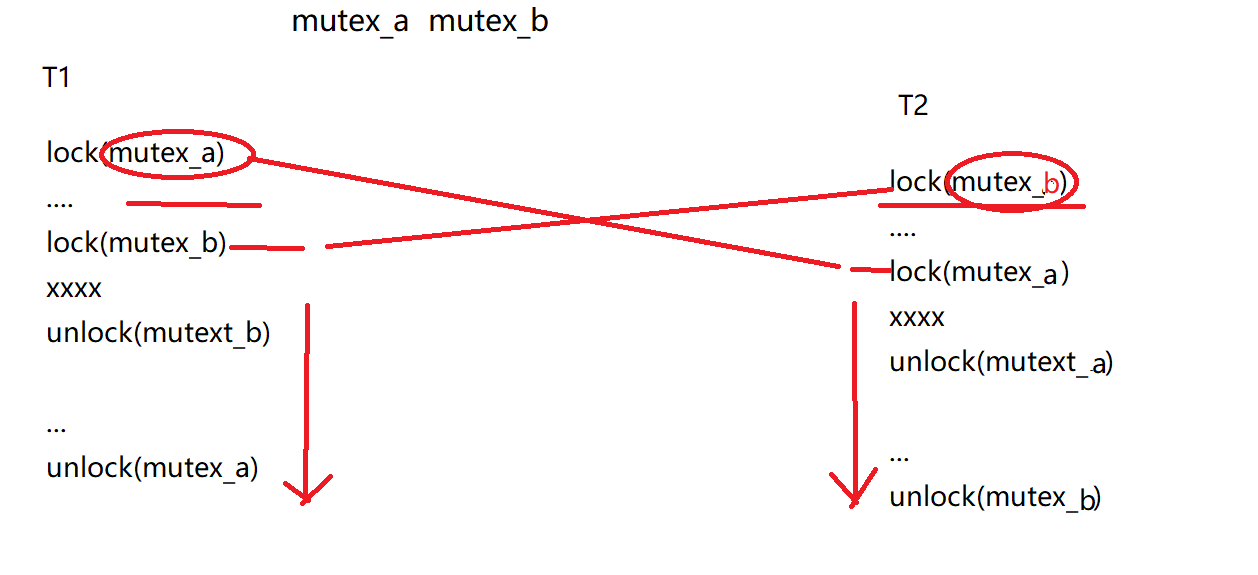
c. xxx ooo

d. pthread\_mutex\_unlock



Mutex用法不当：重复加锁的会导致死锁：

两个线程各自持有一把锁，都在相互等对方释放所持有的锁，此时，两个线程都没有办法执行下去：



并行开发：活锁 锁的护航 ……

Mutex的原理：<http://47.106.79.26:9090/2018/11/28/kernel_mutex/>

等待队列：如果没有请求到锁，那么就把该线程放入等待队列里，挂起整个线程的执行。

2）如果持有锁的线程执行完且释放锁，则会唤醒等待队列里的某个线程去执行，那么它就能获取到锁。

线程会出现挂起的状态。

1. 自旋锁：一直处于一种忙的状态，相对于Mutex

state == 0; 没有锁住的状态 == 1，是锁住的状态

while(state != 0)

{

}

while(!lock())

{

continue;

}

1. CAS：原子操作： 在任意时刻有且仅能有一个线程对某个变量的操作是成功的（合法的），其余的线程都会返回失败（只能继续等待）。

原子操作的实现原理？？？：

1. CPU --- 内存之间通信总线，如果有一个CPU对某个变量进行操作，那么它在操作之前会尝试锁住这个总线（LOCK）。
2. CPU他会把一些常用的变量缓存在CPU的缓存里，锁住这些缓存。
3. 条件变量（等待和通知）：
4. 信号量
5. barrier，内存屏障

pthread\_barrier\_wait : Java 发令枪

++num / num ++;

其实是分成三个步骤在执行：

T1 :

1. 从内测中取num的值（0）；
2. 把num的值+1（0 + 1）；
3. 把num的值写回到内存中。

如何设置CPU的亲缘性和查看设置是否生效：<http://47.106.79.26:9090/2018/12/06/cpu-affinity/>

并行编程：

他们共享了资源：同步，