Thread ID和ProcessID是不一样的。Process ID是全系统唯一的，而ThreadID是在其所属的进程内可以保证唯一。

对于pthread的函数调用都是直接判断其返回值，而不要依赖errno。

结束线程的三种方式，注意这三种方式不会影响到进程的退出。

1.从线程执行函数中退出，退出码就是线程的退出码

2.被其他线程取消[pthread\_cancel](http://bbs.chinaunix.net/thread-2134331-1-1.html)

3.自己调用pthread\_exit

注意如果线程调用exit，\_Exit, or \_exit, then the entire process terminates。而且当我们向线程发送signal时会导致所属进程退出。

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

假设线程A调用pthread\_join等待线程B退出。则线程A调用该函数后会阻塞。有三种方式会导致该函数退出阻塞

* 线程B自己调用pthread\_exit，此时rval\_ptr指向pthread\_exit被调用时传入的参数
* 线程B自然退出，此时rval\_ptr包含退出码
* 线程B被其他线程调用pthread\_cancel取消，则rval\_ptr的值为PTHREAD\_CANCELED

如果我们不想关心pthrad\_join的rval\_ptr，直接赋Null

pthread\_join(tid, NULL);

线程退出清除函数栈

有关线程的Join和Detach，可以参考

Joinable And Detached [Threads @ http://www.domaigne.com/blog/computing/joinable-and-detached-threads/](mailto:Threads@http://www.domaigne.com/blog/computing/joinable-and-detached-threads/)

所谓Detach的技术就是我们不需要对创建的线程都用join方法来回收，Detach有两种方法：

* 创建时就赋予detach属性pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);
* 线程被pthread\_detach，

这样我们可以不用考虑回收了。

如果多个线程只是要对一些共享资源进行无条件独占访问，那么用互斥量就足够了。就是说线程抢占到互斥量后是无条件执行的。但如果竞争存在一种**有条件执行**的情况，或者说即使抢占了互斥量，也不是无条件执行，那么就需要加上条件变量来辅助。

使用条件+互斥量的方式进行同步的方法适用于如下线程同步模型。

打个比方，类似银行叫号的情况。互斥访问的是窗口，条件是服务人员有空。（用生产者和消费者模型更好，比如一个生产者对多个消费者的例子。都可以看做竞争窗口的例子。）

柜台的那个小窗口就是一个需要互斥访问的变量。如果不使用系统提供的条件变量，只使用互斥量。那么程序可以自己定义一个bool变量condition。需要访问小窗口的用户（线程）就要自己使用一个while循环不断地抢占窗口并查询服务人员当前是否空闲用程序表示如下：

while (1){

pthread\_lock() //抢占窗口

if ( condition == true ){ //判断服务人员空闲

//do task

taskDone = true;

}

pthread\_unlock() //释放窗口

if ( taskDone) break;

}

使用条件+互斥量后，相当于改善了原来的流程，柜台采用了一种挂号叫号的方式，有了一种互动。用户来了之后，先尝试占住窗口，但只是挂个号，然后就退出了（按照pthread的说法就是condition上会维护一个等待线程队列，呼叫pthread\_cond\_wait后系统将该呼叫线程放入等待condition的队列后就释放互斥量但函数不退出，等待condition变为真）。等到服务人员的确空闲了，比如别的用户使用完毕，银行就发生叫号的动作，此时服务人员必定空闲（condition变为真），系统将pthread\_cond\_wait函数退出并将互斥量再次锁住，相当于用户（线程）被激活，并将窗口给该用户独占使用，此时用户就可以继续自己的操作了。

这么做明显效率得到提高，用户（线程）不会发生无谓的争抢窗口（过多地进行lock互斥量和unlock互斥量）的操作。银行（系统）出面推出叫号系统，用户只要尝试争抢窗口一次挂个号就可以**到一边呆着（释放锁）**和**休息（睡眠）**即可。

Code sample:

void \*decrement\_count(void \*arg) //等待线程1

{

pthread\_mutex\_lock(&count\_lock);

while(count == 0) //当条件不满足时进入等待

{

pthread\_cond\_wait(&count\_nonzero, &count\_lock);

} //为什么使用while呢？因为在pthread\_cond\_signal和pthread\_cond\_wait返回之间，有时间差（见下面红色标注部分注释），假设在这个时间差内，条件改变了，显然需要重新检查条件。也就是说在pthread\_cond\_wait被唤醒的时候可能该条件已经不成立。

count = count - 1;

pthread\_mutex\_unlock(&count\_lock);

}

void \*increment\_count(void \*arg) //激活线程2

{

pthread\_mutex\_lock(&count\_lock);

count = count + 1;

if (count > 0) //当条件满足时调用pthread\_cond\_signal来激活等待线程

{

pthread\_cond\_signal(&count\_nonzero);

}

pthread\_mutex\_unlock(&count\_lock);

//从时序上来说，等待线程的pthread\_cond\_wait只有在激活线程的pthread\_mutex\_unlock操做完成之后才有机会lock住互斥量并对共享数据count进行“减法”操做。所以可以预见的是如果还有别的线程3会改变count的值，那么当激活线程2退出pthread\_mutex\_unlock释放锁到等待线程1占用锁并退出pthread\_cond\_wait，这之间count会被线程3改变，这也是为何线程1中要增加while循环检测的原因。

}

[Linux 的多线程编程的高效开发经验--](http://blog.csdn.net/kaizi318/article/details/8201728) [多线程下linux和 windows开发应注意的区别](http://blog.csdn.net/jamesf1982/article/details/4370949)

<http://blog.csdn.net/robertzhouxh/article/details/8260439>