pthread的各种同步机制 - Casa Taloyum

pthread的各种同步机制

Date Thu 29 January 2015 Tags pthread

简述

pthread是POSIX标准的多线程库，UNIX、Linux上广泛使用，windows上也有对应的实现，所有的函数都是pthread打头，也就一百多个函数，不是
引入了一定的不可预知性，要控制这些不可预知性，就需要使用各种锁各种同步机制，不同的情况就应该使用不同的锁不同的机制。什么事情一旦
就像潘多拉魔盒，带来的好处不可胜数，然而工程师只要一不小心，就很容易让你的程序失去控制，所以你得用各种锁各种机制管住它。要解决好
合适的解决方案。

处理方案不对的话，那能不能正确跑完程序就只好看运气啦～

预备

1. 阅读这篇文章之前你最好有一些实际的pthread使用经验，因为这篇文章不是写给从零开始学习pthread的人的。
2. 想要5分钟内立刻搞定多线程同步机制的人，觉得文章太长不看的人，这篇文章就不是写给你们看的。Mac请点左上角关闭，KDE和GNOME
3. 如果你经常困惑于各种锁和同步机制的方案，或者你想寻找比现有代码更优雅的方案来处理你遇到的多线程问题，那这篇文章就是写给你的。
4. 如果你发现别人的多线程代码写得不对，但是勉强能跑，然后你找到他让他改的时候，跟他解释半天也不愿意改，那这篇文章就是写给他们

开始。

Mutex Lock 互斥锁

MUTual-EXclude Lock，互斥锁。 它是理解最容易，使用最广泛的一种同步机制。顾名思义，被这个锁保护的临界区就只允许一个线程进入，其它
它使用得非常广泛，以至于大多数人谈到锁就是mutex。mutex是互斥锁，pthread里面还有很多锁，mutex只是其中一种。

相关宏和函数

```
PTHREAD_MUTEX_INITIALIZER // 用于静态的mutex的初始化，采用默认的attr。比如: static pthread_mutex_t mutex = PTHREAD_M

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr); // 用于动态的

int pthread_mutex_destroy(pthread_mutex_t *mutex); // 把mutex锁干掉，并且释放所有它所占有的资源

int pthread_mutex_lock(pthread_mutex_t *mutex); // 请求锁，如果当前mutex已经被锁，那么这个线程就会卡在这儿，直到mutex被
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex); // 解锁

int pthread_mutex_trylock(pthread_mutex_t *mutex); // 尝试请求锁，如果当前mutex已经被锁或者不可用，这个函数就直接return
```

要注意的地方

关于mutex的初始化

mutex的初始化分两种，一种是用宏(PTHREAD_MUTEX_INITIALIZER)，一种是用函数(pthread_mutex_init)。 如果没有特殊的配置要求的话，使用宏
也就是说，凡是 `pthread_mutex_init(&mutex, NULL)` 的地方都可以使用 `PTHREAD_MUTEX_INITIALIZER` ，因为在 `pthread_mutex_init` 这个

```
//////////////////// pthread_src/include/pthread/pthread.h

#define PTHREAD_MUTEX_INITIALIZER __PTHREAD_MUTEX_INITIALIZER


//////////////////// pthread_src/sysdeps/generic/bits/mutex.h

#  define __PTHREAD_MUTEX_INITIALIZER \
    { __PTHREAD_SPIN_LOCK_INITIALIZER, __PTHREAD_SPIN_LOCK_INITIALIZER, 0, 0, 0, 0, 0, 0 } // mutex锁本质上是一个


//////////////////// pthread_src/sysdeps/generic/pt-mutex-init.c
int
_pthread_mutex_init (pthread_mutex_t *mutex,
             const pthread_mutexattr_t *attr)
{
  *mutex = (pthread_mutex_t) __PTHREAD_MUTEX_INITIALIZER; // 你看，这里其实用的也是宏。就这一句是初始化，下面都是在设置属

  if (! attr
      || memcmp (attr, &__pthread_default_mutexattr, sizeof (*attr) == 0))
    /* The default attributes.  */
    return 0;

  if (! mutex->attr
      || mutex->attr == __PTHREAD_ERRORCHECK_MUTEXATTR
      || mutex->attr == __PTHREAD_RECURSIVE_MUTEXATTR)
    mutex->attr = malloc (sizeof *attr);                //pthread_mutex_destroy释放的就是这里的资源

  if (! mutex->attr)
    return ENOMEM;

  *mutex->attr = *attr;
  return 0;
}
```

但是业界有另一种说法是：早年的POSIX只支持在static变量上使用 `PTHREAD_MUTEX_INITIALIZER` ，所以 `PTHREAD_MUTEX_INITIALIZER` 尽量不
是比较老的。

mutex锁不是万能灵药

基本上所有的问题都可以用互斥的方案去解决，大不了就是慢点儿，但不要不管什么情况都用互斥，都能采用这种方案不代表都适合采用这种方案。
响的面比较大，本来不需要通过互斥就能让线程进入临界区，但用了互斥方案之后，就使这样的线程不得不等待互斥锁的释放，所以就慢了。甚至有
读多写少的场合，就比较适合读写锁(reader/writter lock)，如果临界区比较短，就适合空转锁(pin lock)...这些我在后面都会说的，你可以翻到下面去

提到这个的原因是：大多数人学pthread学到mutex就结束了，然后不管什么都用mutex。那是不对的！！！

预防死锁

如果要进入一段临界区需要多个mutex锁，那么就很容易导致死锁，单个mutex锁是不会引发死锁的。要解决这个问题也很简单，只要申请锁的时候代码有一定的要求，尤其是全局mutex锁的时候，更需要遵守一个约定。

如果是全局mutex锁，我习惯将它们写在同一个头文件里。一个模块的文件再多，都必须要有两个umbrella header file。一个是整个模块的伞，外界块的所有子模块去include，然后这个头文件里面就放一些公用的宏啊，配置啊啥的，全局mutex放在这里就最合适了。这两个文件不能是同一个，的，那现在就要改了。

然后我的mutex锁的命名规则就是：`作用_mutex_序号`，比如`LinkListMutex_mutex_1`,`OperationQueue_mutex_2`，后面的序号在每次有新锁mutex锁的，我就按照序号的顺序去进行加锁操作(pthread_mutex_lock)，这样就能够保证不会出现死锁了。

如果是属于某个struct内部的mutex锁，那么也一样，只不过序号可以不必跟全局锁挂钩，也可以从1开始数。

还有另一种方案也非常有效，就是用`pthread_mutex_trylock`函数来申请加锁，这个函数在mutex锁不可用时，不像`pthread_mutex_lock`那样错误: `EBUSY` (锁尚未解除)或者`EINVAL` (锁变量不可用)。一旦在trylock的时候有错误返回，那就把前面已经拿到的锁全部释放，然后过一段时间再来加锁，这个函数跟`pthread_mutex_trylock`类似，不同的是，你可以传入一个时间参数，在申请加锁失败之后会阻塞一段时间等解锁，超时之后

这两种方案我更多会使用第一种，原因如下：

- 一般情况下进入临界区需要加的锁数量不会太多，第一种方案能够hold住。如果多于2个，你就要考虑一下是否有些锁是可以合并的了。

第一种方案适合锁比较少的情况，因为这不会导致非常大的阻塞延时。但是当你要加的锁非常多，ABCDE，你加到D的时候阻塞了，然而其他线程才会采用第二种方案。如果要加的锁本身就不多，只有AB两个，那么阻塞一下也还可以。

- 第二种方案在面临阻塞的时候，要操作的事情太多。

当你把所有的锁都释放以后，你的当前线程的处理策略就会导致你的代码复杂度上升：当前线程总不能就此退出吧，你得找个地方把它放起来，让情况，你就需要一个线程池来存放这些等待解锁的线程。如果临界区是嵌套的，你在把这个线程挂起的时候，最好还要把外面的锁也释放掉，要不然情况。这里要做的事情太多，复杂度比较大，容易出错。

所以总而言之，设计的时候尽量减少同一临界区所需要mutex锁的数量，然后采用第一种方案。如果确实有需求导致那么多mutex锁，那么就只能采header file和按照序号命名mutex锁是个非常好的习惯，可以允许你随着软件的发展而灵活采取第一第二种方案。

但是到了semaphore情况下的死锁处理方案时，上面两种方案就都不顶用了，后面我会说。另外，还有一种死锁是自己把自己锁死了，这个我在后

Reader-Writter Lock 读写锁

前面mutex锁有个缺点，就是只要锁住了，不管其他线程要干什么，都不允许进入临界区。设想这样一种情况：临界区foo变量在被bar1线程读着，斥锁，那就不能读了。但事实情况是，读取数据不影响数据内容本身，所以即便被1个线程读着，另外一个线程也应该允许他去读。除非另外一个线结束了再写。

因此诞生了Reader-Writter Lock，有的地方也叫Shared-Exclusive Lock，共享锁。

> Reader-Writter Lock的特性是这样的，当一个线程加了读锁访问临界区，另外一个线程也想访问临界区读取数据能够成功进入临界区进行读操作了。此时读锁线程有两个。当第三个线程需要进行写操作时，它需要加一个写锁是等前两个读线程都释放读锁之后，第三个线程就能进去写了。总结一下就是，读写锁里，读锁能允许多个线程写。

这样更精细的控制，就能减少mutex导致的阻塞延迟时间。虽然用mutex也能起作用，但这种场合，明显读写锁更好嘛！

相关宏和函数

```
    PTHREAD_RWLOCK_INITIALIZER

    int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
    int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

    int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
    int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

    int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
    int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

    int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

    int pthread_rwlock_timedrdlock_np(pthread_rwlock_t *rwlock, const struct timespec *deltatime); // 这个函数在Lin
找到
    int pthread_rwlock_timedwrlock_np(pthread_rwlock_t *rwlock, const struct timespec *deltatime); // 同上
```

要注意的地方

命名

跟上面提到的写muetx锁的约定一样，操作，类别，序号最好都要有。比如`OperationQueue_rwlock_1`。

认真区分使用场合，记得避免写线程饥饿

由于读写锁的性质，在默认情况下是很容易出现写线程饥饿的。因为它必须要等到所有读锁都释放之后，才能成功申请写锁。不过不同系统的实现
统默认读线程优先。

比如在写线程阻塞的时候，有很多读线程是可以一个接一个地在那儿插队的(在默认情况下，只要有读锁在，写锁就无法申请，然而读锁可以一直申
申请成功写锁了，然后它就饿死了。

为了控制写线程饥饿，必须要在创建读写锁的时候设置`PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE`，不要用`PTHREAD_RWLOCK_PREFER_W`
么知道的。。。

```
////////////////////////////// /usr/include/pthread.h

/* Read-write lock types.  */
#if defined __USE_UNIX98 || defined __USE_XOPEN2K
enum
{
  PTHREAD_RWLOCK_PREFER_READER_NP,
  PTHREAD_RWLOCK_PREFER_WRITER_NP, // 妈蛋，没用，一样reader优先
  PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP,
  PTHREAD_RWLOCK_DEFAULT_NP = PTHREAD_RWLOCK_PREFER_READER_NP
};
```

总的来说，这样的锁建立之后一定要设置优先级，不然就容易出现写线程饥饿。而且读写锁适合读多写少的情况，如果读、写一样多，那这时候还

spin lock 空转锁

上面在给出mutex锁的实现代码的时候提到了这个spin lock，空转锁。它是互斥锁、读写锁的基础。在其它同步机制里condition variable、barrier等

我先说一下其他锁申请加锁的过程，你就知道什么是spin lock了。

互斥锁和读写锁在申请加锁的时候，会使得线程阻塞，阻塞的过程又分两个阶段，第一阶段是会先空转，可以理解成跑一个while循环，不断地去申样，线程也分很多状态)，此时线程就不占用CPU资源了，等锁可用的时候，这个线程会被唤醒。

为什么会有这两个阶段呢？主要还是出于效率因素。

- 如果单纯在申请锁失败之后，立刻将线程状态挂起，会带来context切换的开销，但挂起之后就可以不占用CPU资源了，原属于这个线程的CP就又可用了，那么短时间内进行context切换的开销就显得很没效率。
- 如果单纯在申请锁失败之后，不断轮询申请加锁，那么可以在第一时间申请加锁成功，同时避免了context切换的开销，但是浪费了宝贵的CP在这么长时间里都被这个线程拿来轮询了，也显得很没效率。

于是就出现了两种方案结合的情况：在第一次申请加锁失败的时候，先不着急切换context，空转一段时间。如果锁在短时间内又可用了，那么就避发现还是不能申请加锁成功，那么就有很大概率在将来的不短的一段时间里面加锁也不成功，那么就把线程挂起，把轮询用的CPU时间释放出来给

所以spin lock就是这样的一个锁：`它在第一次申请加锁失败的时候，会不断轮询，直到申请加锁成功为止，期间不会进行线程context的切换。`互斥锁和已。

这里是spin lock申请加锁的实现：

```
////////////////////////////pthread_src/sysdeps/posix/pt-spin.c

/* Lock the spin lock object LOCK.  If the lock is held by another
    thread spin until it becomes available.  */
int
_pthread_spin_lock (__pthread_spinlock_t *lock)
{
  int i;

  while (1)
    {
      for (i = 0; i < __pthread_spin_count; i++)
    {
      if (__pthread_spin_trylock (lock) == 0)
        return 0;
    }

      __sched_yield ();
    }
}
```

相关宏和函数

我没在man里面找到spin lock相关的函数，但事实上外面还是能够使用的，下面是我在源代码里面挖到的原型：

```
////////////////////////////////pthread_src/pthread/pt-spin-inlines.c

/* Weak aliases for the spin lock functions.  Note that
   pthread_spin_lock is left out deliberately.  We already provide an
   implementation for it in pt-spin.c.  */
weak_alias (__pthread_spin_destroy, pthread_spin_destroy);
weak_alias (__pthread_spin_init, pthread_spin_init);
weak_alias (__pthread_spin_trylock, pthread_spin_trylock);
weak_alias (__pthread_spin_unlock, pthread_spin_unlock);

////////////////////////////////pthread_src/sysdeps/posix/pt-spin.c

weak_alias (_pthread_spin_lock, pthread_spin_lock);

/*--------------------------------------------------*/
```

```
    PTHREAD_SPINLOCK_INITIALIZER
    int pthread_spin_init (__pthread_spinlock_t *__lock, int __pshared);
    int pthread_spin_destroy (__pthread_spinlock_t *__lock);
    int pthread_spin_trylock (__pthread_spinlock_t *__lock);
    int pthread_spin_unlock (__pthread_spinlock_t *__lock);
    int pthread_spin_lock (__pthread_spinlock_t *__lock);

/*--------------------------------------------------*/
```

注意事项

还是要分清楚使用场合

了解了空转锁的特性，我们就发现这个锁其实非常适合临界区非常短的场合，或者实时性要求比较高的场合。

由于临界区短，线程需要等待的时间也短，即便轮询浪费CPU资源，也浪费不了多少，还省了context切换的开销。 由于实时性要求比较高，来不及

不过说实话，大部分情况你都不会直接用到空转锁，其他锁在申请不到加锁时也是会空转一定时间的，如果连这段时间都无法满足你的请求，那要么

pthread_cleanup_push() & pthread_cleanup_pop()

线程是允许在退出的时候，调用一些回调方法的。如果你需要做类似的事情，那么就用以下这两种方法:

```
    void pthread_cleanup_push(void (*callback)(void *), void *arg);
    void pthread_cleanup_pop(int execute);
```

正如名字所暗示的，它背后有一个stack，你可以塞很多个callback函数进去，然后调用的时候按照先入后出的顺序调用这些callback。所以你在塞c

但是！你塞进去的callback只有在以下情况下才会被调用:

1. 线程通过pthread_exit()函数退出
2. 线程被pthread_cancel()取消
3. pthread_cleanup_pop(int execute)时，execute传了一个非0值

也就是说，如果你的线程函数是这么写的，那在线程结束的时候就不会调到你塞进去的那些callback了:

```
static void * thread_function(void *args)
{
    ...
    ...
    ...
    ...
    return 0; // 线程退出时没有调用pthread_exit()退出，而是直接return，此时是不会调用栈内callback的
}
```

用`exit()`行不行？尼玛一调用这个整个进程就挂掉了～只要在任意线程调用exit()，整个进程就结束了，不要瞎搞。 `pthread_cleanup_push`塞

太会在这里执行业务逻辑。在线程结束之后如果要执行业务逻辑，一般用下面提到的`pthread_join`。

注意事项

callback函数是可以传参数的

对的，在`pthread_cleanup_push`函数中，第二个参数的值会作为callback函数的第一个参数，不要浪费了，拿来打打日志也不错。举个例子：

```
void callback(void *callback_arg)
{
    printf("arg is : %s\n", (char *)callback_arg);
}

static void * thread_function(void *thread_arg)
{
    ...
    pthread_cleanup_push(callback, "this is a queue thread, and was terminated.");
    ...
    pthread_exit((void *) 0); // 这句不调用，线程结束就不会调用你塞进去的callback函数。
    return ((void *) 0);
}

int main ()
{
    ...
    ...
    error = pthread_create(&tid, NULL, thread_function, (void *)thread_arg)
    ...
    ...
    return 0;
}
```

你也发现了，callback函数的参数是在线程函数里面设置的，所以拿来做业务也是可以的，不过一般都是拿来做清理的事情，很少会把它放到业务里

要保持callback栈平衡

有的时候我们并不一定要在线程结束的时候调用这些callback，那怎么办？直接return不就好了么，return的话，不就不调用callback了？

如果你真是这么想的，请去撞墙5分钟。

callback的调用栈一定要保持平衡，如果你不保持平衡就退出了线程，后面的结果是undefine的，有的系统就core dump了(比如Mac)，有的系统还就
callback栈不平衡的结果是未定义的)。

所以遇到有时要调用有时又不需要的时候，这么写才是正确的姿势：

```
void callback1(void *callback_arg)
{
    printf("arg is : %s\n", (char *)callback_arg);
}

void callback2(void *callback_arg)
{
    printf("arg is : %s\n", (char *)callback_arg);
}

static void * thread_function(void *thread_arg)
{
    ...

    pthread_cleanup_push(callback1, "this is callback 1.");
    pthread_cleanup_push(callback2, "this is callback 2.");

    ...

    if (thread_arg->should_callback) {
```

```
        pthread_exit((void *) result);
    }

    pthread_cleanup_pop(0); // 传递参数0，在pop的时候就不会调用对应的callback，如果传递非0值，pop的时候就会调用对应callback了
    pthread_cleanup_pop(0); // push了两次就pop两次,你要是只pop一次也可以，因为下面也有pthread_exit。这样一来就只会调用callbac
去了。
    pthread_exit((void *) result);   // 所有的线程都应该用pthread_exit来结束，一方面是确保栈平衡，另一方面，也给别的线程join拨

    return ((void *) 0);
}

int main ()
{
    ...
    ...
    error = pthread_create(&tid, NULL, thread_function, (void *)thread_arg)
    ...
    ...
    return 0;
}
```

pthread_join()

在线程结束的时候，我们能通过上面的`pthread_cleanup_push`塞入的callback方法知道，也能通过`pthread_join`这个方法知道。一般情况下，用`pthread_join`这个方法。

它适用这样的场景：

> 你有两个线程，B线程在做某些事情之前，必须要等待A线程把事情做完，然后才能接着做下去。这时候就可以用

原型:

```
    int pthread_join(pthread_t thread, void **value_ptr);
```

在B线程里调用这个方法，第一个参数传A线程的thread_id, 第二个参数你可以扔一个指针进去。当A线程调用`pthread_exit(void *value_ptr)`的`value_ptr`去，你可以理解成A把它计算出来的结果放到exit函数里面去，然后其他join的线程就能拿到这个数据了。

在B线程join了A线程之后，B线程会阻塞住，直到A线程跑完。A线程跑完之后，自动被detach，后续再要join的线程就会报`EINVAL`。

注意事项

新创建的线程默认是join属性，每一个join属性的线程都需要通过pthread_join来回收资源

- 如果A线程已经跑完，但没被join过，此时B线程要去join A线程的时候，`pthread_join`是会立刻正确返回的，之后A线程就被detach了，占
- 如果A线程已经跑完，后面没人join它，它占用的资源就会一直在哪儿，变成僵尸线程。

所以要么在创建线程的时候就把线程设置为detach的线程，这样线程跑完以后不用join，占用的资源自动回收。

要么不要忘记去join一下，把资源回收了，不要留僵尸。

注意传递的参数的内存生命周期

虽然线程和进程共享同一个进程资源，但如果在`pthread_exit()`里面你传递的指针指向的是栈内存，那么在结束之后，这片内存还是会被回收的

还有就是，一定要在获得`value_ptr`之后，检查一下`value_ptr`是否`PTHREAD_CANCELED`，因为如果你要等待的线程被cancel掉了，你拿到的就

多个线程join同一个线程

`pthread_join`是允许多个线程等待同一个线程的结束的。如果要一个线程等待多个线程的结束，那就需要用下面提到的`条件变量`了，或者`barr`

但是多个线程join同一个线程的时候，情况就比较多。多而已，不复杂。我们先建立一个约定：`A线程是要被join的线程，BCDEF是要等待A线程结束的`

- A线程正在运行，BCDEF线程发起对A的join，发起join结束后，A仍然在运行中

此时BCDEF线程都会被阻塞，等待A线程的结束。A线程结束之后，BCDEF都被唤醒，能够正常获得A线程通过`pthread_exit()`返回的数据。

- A线程正在运行，BCDEF发起对A的join，BCD发起join成功后，A线程结束，然后EF发起join

此时BCD线程能够正常被唤醒，并完成任务，由于被join后A线程被detach，资源释放，后续EF再要发起join，就会`EINVAL`。

- A线程正在运行，且运行结束。此时BCDEF发起对A的join。

此时谁先调用成功，谁就能完成任务，后续再要join的就都会`EINVAL`。一旦有一个线程join成功，A立刻被detach，资源释放，然后后面其他的线程

总的来说，只要线程运行结束，并且被detach了，后面再join就不行了，只要线程还在运行中，就能join。如果运行结束了，第一次被join之后，线程
属性的线程，那任何时候都无法被join。

Condition Variables 条件变量

`pthread_join`解决的是多个线程等待同一个线程的结束。条件变量能在合适的时候唤醒正在等待的线程。具体什么时候合适由你自己决定。它必

> 场景：B线程和A线程之间有合作关系，当A线程完成某件事情之前，B线程会等待。当A线程完成某件事情之后，
> 醒，然后继续做自己要做的事情。

如果不用条件变量的话，也行。那就是搞个volatile变量，然后让其他线程不断轮询，一旦这个变量到了某个值，你就可以让线程继续了。如果有多

但是！！！这做法太特么愚蠢了，还特别浪费CPU时间，所以还在用volatile变量标记线程状态的你们也真是够了！！！

大致的实现原理是：一个条件变量背后有一个池子，所有需要wait这个变量的线程都会进入这个池子。当有线程扔出这个条件变量的signal，系统就

相关宏和函数

```
PTHREAD_COND_INITIALIZER
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timesp
```

补充一下，原则上 `pthread_cond_signal` 是只通知一个线程， `pthread_cond_broadcast` 是用于通知很多线程。但POSIX标准也允许让 `pthrea` 程。具体看各系统的实现。一般我都是用 `pthread_cond_broadcast` 。

另外，在调用 `pthread_cond_wait` 之前，必须要申请互斥锁，当线程通过 `pthread_cond_wait` 进入waiting状态时，会释放传入的互斥锁。

下面我先给一个条件变量的使用例子，然后再讲需要注意的点。

```
void thread_waiting_for_condition_signal ()
{
    pthread_mutex_lock(&mutex);
    while (operation_queue == NULL) {
        pthread_cond_wait(&condition_variable_signal, &mutex);
    }

    /*********************************/
    /* 做一些关于operation_queue的事 */
    /*********************************/

    pthread_mutex_unlock(&mutex);
}

void thread_prepare_queue ()
{
    pthread_mutex_lock(&mutex);

    /*********************************/
    /* 做一些关于operation_queue的事 */
    /*********************************/

    pthread_cond_signal(&condition_variable_signal); // 事情做完了之后要扔信号给等待的线程告诉他们做完了
    pthread_mutex_unlock(&mutex);


    /*************************/
    /* 这里可以做一些别的事情 */
    /*************************/

    ...

    pthread_exit((void *) 0);
}
```

要注意的地方

一定要跟**mutex**配合使用

```
    void thread_function_1 ()
    {
        done = 1;
        pthread_cond_signal(&condition_variable_signal);
    }

    void thread_function_2 ()
    {
        while (done == 0) {
            pthread_cond_wait(&condition_variable_signal, NULL);
        }
    }
```

这样行不行？当然不行。为什么不行？

这里涉及一个非常精巧的情况：在 `thread_function_2` 发现done = 0的时候，准备要进行下一步的wait操作。在具体开始下一步的wait操作之前，嗯， `thread_function_2` 还没来得及waiting呢， `thread_function_1` 就把信号发出去了，也没人接收这信号， `thread_function_2` 继续执行

一定要检测你要操作的内容

```
void thread_function_1 ()
{
    pthread_mutex_lock(&mutex);

    ...
    operation_queue = create_operation_queue();
    ...

    pthread_cond_signal(&condition_variable_signal);
    pthread_mutex_unlock(&mutex);
}

void thread_function_2 ()
{
    pthread_mutex_lock(&mutex);
    ...
    pthread_cond_wait(&condition_variable_signal, &mutex);
    ...
    pthread_mutex_unlock(&mutex);
}
```

这样行不行？当然不行。为什么不行？

比如 `thread_function_1` 一下子就跑完了，operation_queue也初始化好了，信号也扔出去了。这时候 `thread_function_2` 刚刚启动，由于它没而事实是operation_queue早已搞定，再也不会有人扔 `我已经搞定operation_queue啦` 的信号， `thread_function_2` 也不知道operation_queue已

一定要用**while**来检测你要操作的内容而不是**if**

```
void thread_function_1 ()
{
    pthread_mutex_lock(&mutex);
    done = 1;
    pthread_cond_signal(&condition_variable_signal);
    pthread_mutex_unlock(&mutex);
}

void thread_function_2 ()
{
    pthread_mutex_lock(&mutex);
    if (done == 0) {
        pthread_cond_wait(&condition_variable_signal, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}
```

这样行不行？大多数情况行，但是用while更加安全。

如果有别人写一个线程去把这个done搞成0了，期间没有申请mutex锁。

那么这时用if去判断的话，由于线程已经从wait状态唤醒，它会直接做下面的事情，而全然不知done的值已经变了。

如果这时用while去判断的话，在 `pthread_cond_wait` 解除wait状态之后，会再去while那边判断一次done的值，只有这次done的值对了，才不会这一次判断就帮了你一把，能继续进入waiting。

不过这解决不了根本问题哈，如果那个不长眼的线程在while的第二次判断之后改了done，那还是要悲剧。根本方案还是要在改done的时候加mutex

总而言之，用if也可以，毕竟不太容易出现不长眼的线程改done变量不申请加mutex锁的。用while的话就多了一次判断，安全了一点，即便有不长眼

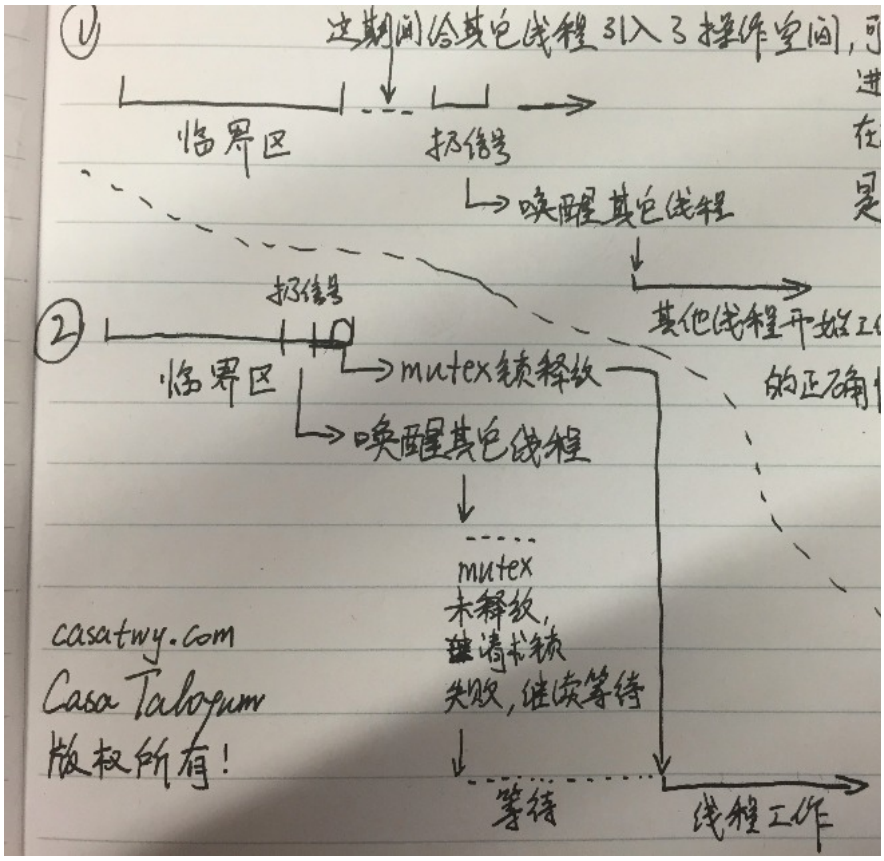
扔信号的时候，在临界区里面扔，不要在临界区外扔

```
void thread_function_1 ()
{
    pthread_mutex_lock(&mutex);
    done = 1;
    pthread_mutex_unlock(&mutex);

    pthread_cond_signal(&condition_variable_signal);
}

void thread_function_2 ()
{
    pthread_mutex_lock(&mutex);
    if (done == 0) {
        pthread_cond_wait(&condition_variable_signal, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}
```

这样行不行？当然不行。为什么不行？《Advanced Programming in the UNIX Enviroment 3 Edtion》这本书里也把扔信号的事儿放在临界区外面了

> 不行就是不行，哪怕是圣经上这么写，也不行。哼。

就应该永远都在临界区里面扔条件信号，我画了一个高清图来解释这个事情，图比较大，可能要加载一段时间：



看到了吧，1的情况就是在临界区外扔信号的坏处。由于在临界区外，其他线程要申请加mutex锁是可以成功的，然后这个线程要是改了你的敏感数

semaphore 信号量

pthread库里面藏了一个semaphore，man手册里面似乎也找不到semaphore相关的函数。

semaphore事实上就是我们学《操作系统》的时候所说的PV操作。 你也可以把它理解成带有数量控制的互斥锁，当`sem_init(&sem, 0, 1);`时，

> 场景：比如有3台打印机，有5个线程要使用打印机，那么semaphore就会先记录好有3台，每成功被申请一次，

它也可以用mutex和条件变量来实现，但实际上还是用semaphore比较方便。

相关函数

```
int sem_destroy(sem_t *sem);
int sem_init(sem_t *sem, int pshared, unsigned int value); // 里的第二个参数是表示这个semaphore是否跨进程可见，Linux

int sem_wait(sem_t *sem); // 如果sempahore的数值还够，那就semaphore数值减1，然后进入临界区。也就是传说中的P操作。
int sem_post(sem_t *sem); // 这个函数会给semphore的值加1，也就是V操作。

int sem_getvalue(sem_t *sem, int *valp); // 注意了，它把semaphore的值通过你传进去的指针告诉你，而不是用这个函数的返回值告
```

要注意的地方

semaphore下的死锁

mutex下的死锁比较好处理，因为mutex只会锁一个资源(当semaphore的值为1时，就是个mutex锁)，按照顺序来申请mutex锁就好了。但是到了se

要想避免死锁，即便采用前面提到的方案：按照顺序加锁，一旦出现加锁失败，就释放所有资源。这招也行不通。假设这样一个情况，当前系统剩

```
剩余资源：        全部系统资源
A:3               A:10
B:2               B:10
C:4               C:10
```

此时有两个线程：t1, t2。

```
t1需要3个A，4个B，1个C
t2需要2个A，2个B，2个C   根据当前剩余资源列表来看，t2可以得到执行，不会出现死锁。
```

假设我们采用旧方案：顺序申请加锁，加锁失败就释放。我们按照CPU时间来推演一个：

```
1. t1申请3个A -> 成功
2. t2申请2个A -> 失败，等待
3. t1申请4个B -> 失败，等待，并释放3个A
4. t1申请3个A -> 成功
5. t2申请2个A -> 失败，等待
6. t1申请4个B -> 失败，等待，并释放3个A
7. t1申请3个A -> 成功
8. t2申请2个A -> 失败，等待
9. t1申请4个B -> 失败，等待，并释放3个A
...
```

发现没有，这时候t1和t2都得不到执行，但实际上系统的剩余资源是满足t2的要求的，但由于运气不好，抢资源抢不过t1，在有新的资源被释放之前

要解决这样的问题，就需要采用银行家算法，银行家算法描述起来很简单：获取所有候选线程的需求，随机取一个假设资源分配出去，看是否能够
满足需求为止。如果都不能满足，那就挂起所有线程，等待新的资源释放。

也就是可以理解成很多个人去贷款，银行家先假设你们都能按期还得起钱，按照你们的需求给你们派钱，不过这不是真的派出去了，只是先写在纸
式，直到没有资金缺口为止。

说白了，你需要在你的程序里面建立一个资源分配者的角色，所有待分配资源的线程都去一个池子里排队，然后这个资源分配者一次只允许一个线
被满足需求，那就集体挂起，然后等有新的资源来了，就再把这些线程一个一个叫过来进行资源分配。

Barriers

`Barrier`可以理解成一个mile stone。当一个线程率先跑到mile stone的时候，就先等待。当其他线程都到位之后，再从等待状态唤醒，继续做后面

> 场景：超大数组排序的时候，可以采用多线程的方案来排序。比如开10个线程分别排这个超大数组的10个部分。
> 行后续的归并操作。先完成的线程会挂起等待，直到所有线程都完成之后，才唤醒所有等待的线程。

前面有提到过 条件变量 和 `pthread_join`，前者是在做完某件事情通知其他线程，后者是在线程结束之后让其他线程能够获得执行结果。如果有多
外，用 `semaphore` 也可以实现 `Barrier` 的功能。

但是我们已经有 `Barrier` 了好吗！你们能不要把代码搞那么复杂吗！

相关宏和函数

```
int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *restrict attr, unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

`pthread_barrier_wait` 在唤醒之后，会有两种返回值： `0` 或者 `PTHREAD_BARRIER_SERIAL_THREAD` ，在众多线程中只会有一个线程在唤醒时得
拿到 `PTHREAD_BARRIER_SERIAL_THREAD` 返回的，表示这是上天选择的主线程～

要注意的地方

其实 `Barrier` 很少被错用，因为本来也没几个函数。更多的情况是有人不知道有 `Barrier` ，然后用其他的方式实现了类似 `Barrier` 的功能。知道

关于attr

thread

创建thread的时候是可以设置attr的： `detachstate` 、 `guardsize` 、 `stackaddr` 、 `stacksize` 。一般情况下我都是采取默认的设置。只有在我非
把 `detachstate` 设置为 `PTHREAD_CREATE_DETACHED` 。

mutex

创建mutex的时候也是可以设置attr的： `process-shared` 、 `robust` 、 `type` 。一般情况下尽量不要出现跨进程的锁共享，万一有个相关进程被酒
无解。 `process-shared` 和 `robust` 就是跟跨进程有关。

关于 `type` ，我强烈建议显式设置为 `PTHREAD_MUTEX_ERRORCHECK` 。在Linux下，默认的 `type` 是 `PTHREAD_MUTEX_NORMAL` 。这在下面这种情况下

```
void thread_function()
{
    pthread_mutex_lock(&mutex);
    foo();
    pthread_mutex_unlock(&mutex);
}

void foo()
{
    pthread_mutex_lock(&mutex);
    pthread_mutex_unlock(&mutex);
}
```

上面的代码看着很正常是吧？但由于在调用 `foo` 之前， `mutex` 已经被锁住了，于是 `foo` 就停在那边等待 `thread_function` 释放 `mutex` 。但是！
住了。。。

如果 `type` 设置为 `PTHREAD_MUTEX_ERRORCHECK` ，那在 `foo` 里面的 `pthread_mutex_lock` 就会返回 `EDEADLK` 。如果你要求执行 `foo` 的时候一定

如果 `type` 设置为 `PTHREAD_MUTEX_RECURSIVE` ，也不会产生死锁，但不建议用这个。 `PTHREAD_MUTEX_RECURSIVE` 使用的场景其实很少，我一时

嗯，其他应该没什么了吧。

总结

这篇文章主要讲了pthread的各种同步机制相关的东西：mutex、reader-writter、spin、cleanup callbacks、join、condition variable、semaphore、拿这个作为同步机制的一部分写在程序中，这是不对的！所以我才写了一下这个。

文章很长，相信你们看到这里也不容易，看完了这篇文章，你对多线程编程的同步机制应该可以说比较了解了。但我还要说的是，多线程编程的复杂进程后的协作和控制、多线程和I/O之间的协作和控制、函数的可重入性等，我看我什么时候有时间再写这些内容了。