这是我从网上搜集筛选过来作为自己的笔记的内容，一直认为笔记的目标对象我，所以我不能保证大家看我笔记的效果（收获）。。。不过我想对应看那个zip里面的源码应该没问题。

41.互斥量Mutex： 只能0/1变换的信号量，但是信号量还可以是负数

在windows下使用mutex可能会有问题，需要注意：

hBuf2Mutex = CreateMutex(NULL, FALSE, NULL);

printf("%d", WaitForSingleObject(hBuf2Mutex, INFINITE));看似会发生上面，但是问题在于不会阻塞，如果不阻塞，那函数成功运行，返回0。CreateMutex中第二个参数是FALSE（我是在VS中编写C语言代码，所以可使用FALSE），那么这个Mutex创建以后就没有被拥有，直到有一个线程使用对其使用了Wait..()系列函数才会拥有，一旦没有任何线程拥有mutex，这个mutex便处于激发状态，因此，如果没有任何线程拥有那个mutex，Wait...()便会成功。然后就会变成未激发的状态，其他线程使用Wait...会被阻塞。我在测试中发现在Windows下，拥有Mutex对象的线程与创建Mutex的线程（也就是调用CreateMutex的线程）都不会被这个Mutex所阻塞，网上面的人似乎称这个现象较“重入”。所以windows下不要用互斥量，还是用Semaphore来代替好了。

HANDLE CreateMutex(LPSECURITY\_ATTRIBUTES lpMutexAttributes,

BOOL bInitialOwner,

LPCTSTR lpName)；

参数说明：

lpMutexAttributes—取值NULL。

bInitialOwner—指示当前线程是否马上拥有该互斥量(即马上加锁)。

lpName—互斥量名称。lpName: The name is limited to MAX\_PATH characters.If lpName matches the name of an existing named mutex object, this function requests the MUTEX\_ALL\_ACCESS access right.

如果函数调用之前，有名互斥对象已存在，那么函数给已存在的对象返回一个句柄。

BOOL WIANPI ReleaseMutex(HANDLE hMutex);

参数说明：

hMutex—互斥对象句柄。

返回值：TRUE表示成功，FALSE表示失败。

Linux Mutex：

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

参数说明：

mutex——互斥信号量指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。如果该互斥锁已被另一个线程锁定和拥有，则调用该线程将阻塞，直到该互斥锁变为可用为止

pthread\_mutex\_unlock

功能：给互斥量解锁。

格式：

int pthread\_mutex\_unock(pthread\_mutex\_t \*mutex);

参数说明：

mutex——需要解锁的互斥信号量指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。

42.信号量Semaphore：

[PV操作](https://baike.baidu.com/item/PV%E6%93%8D%E4%BD%9C" \t "/home/vega/Documents\\x/_blank)与信号量的处理相关，P表示通过的意思，V表示释放的意思。

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,

LONG lInitialCount,

LONG lMaximumCount,

LPCTSTR lpName);

If the lpSemaphoreAttributes parameter is not **NULL**, it determines whether the handle can be inherited, but its security-descriptor member is ignored

参数说明：

lpSemaphoreAttributes——用于定义信号量的安全特性。

lInitialCount——设置信号量的初始计数。（没有开始增加之前能有多少个减少，一开始能进来多少个）

lMaximumCount——设置信号量的最大计数。

lpName——指定信号量对象的名称。

BOOL ReleaseSemaphore(

HANDLE hSemaphore,

　　 LONG lReleaseCount,

　　 LPLONG lpPreviousCount);

参数说明：

hSemaphore——所要释放的信号量句柄。

lReleaseCount——所要释放信号量的数目。 信号量就是这样对数字的操控你可以“随意”

lpPreviousCount——指向返回信号量上次值的变量的指针，如果不需要信号量上次的值，那么这个参数可以设置为NULL。就是用来看值么。。。

Linux semaphore: <semaphore.h>

sem\_t binSem; 创建一个信号量变量

struct semaphore {

spinlock\_t lock; /\* 自旋锁结构体变量 \*/

unsigned int count; /\* 用于计录资料数量 \*/

struct list\_head wait\_list; /\* 内部链表结构体变量 \*/

};

从此结构体中可以看从，semaphore机制也就是自旋锁的包装

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

sem\_init函数是Posix信号量操作中的函数。用来初始化信号量。sem\_init() 成功时返回 0；错误时，返回 -1，并把 errno 设置为合适的值。如果 pshared 是非零值，那么信号量将在进程之间共享

int sem\_post(sem\_t \*sem); 使信号量加1，是个原子操作

int sem\_wait(sem\_t \*sem); 使信号量减1，是个原子操作

参数说明：

sem——信号量资源指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);阻塞等待若干时间直到信号量>0或者时间结束。第二个参数是时间戳。

int sem\_destroy(sem\_t \* sem);

参数说明：

sem——信号量资源指针。

返回值：在成功完成之后会返回零。其他任何返回值都表示

出现了错误。

43. windows操作系统里面的临界区：

临界区结构CRITICAL\_SECTION

VOID InitializeCriticalSection(LPCRITICAL\_SECTION lpCriticalSection)//初始化临界区

VOID EnterCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);//进入临界区

VOID LeaveCriticalSection( LPCRITICAL\_SECTION lpCriticalSection)//离开临界区

VOID DeleteCriticalSection( LPCRITICAL\_SECTION lpCriticalSection) //临界区肯定也是要占用一些变量的，既然要自己调用函数来创建or初始化那么就要自己来删除。

临界区是防止进入同样的代码段中，所以一般还需要配合条件变量，这样就成了monitor

44.管程monitor：

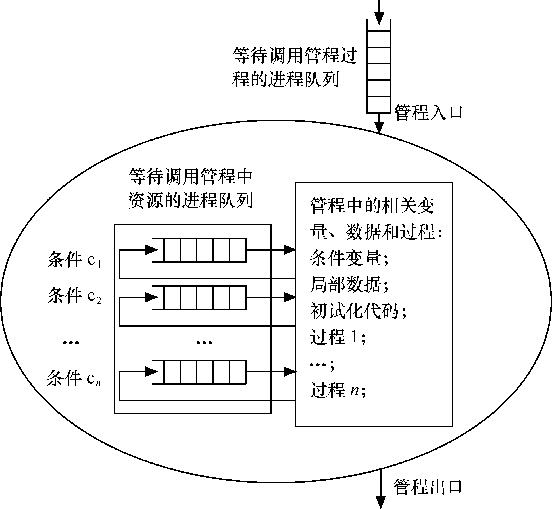
管程把共享变量上的操作集中起来，而临界区（访问共享变量的程序段）分散在每个进程中

用管程解决同步问题比用信号量解决同步问题更加简单，所以在实现上也就更加安全。管程管程封装了同步操作，对进程隐蔽了同步细节，简化了同步功能的调用界面。用户编写并发程序如同编写顺序(串行)程序。

**管程由4部分组成：就像是一个class封装**

1. **管程的名称**
2. **局部于管程内部的共享数据结构说明**
3. **对该数据结构进行操作的一组过程**
4. **对局部于管程内部的共享数据设置初始的语句**

管程只有一个入口和一个出口，并发进程在管程外等待调用管程中的过程，如果进程在管程中不能访问临界资源，则在管程中阻塞等待访问完临界资源的进程将其唤醒 。



管程之中用于维持线程互斥叫做条件变量，每一个条件变量都代表一种等待原因，也对应一个等待队列。

Windows Monitor:

condition variables是微软从vista和2008以后引入的技术，xp和2003的系统不支持。

CONDITION\_VARIABLE ConditionVar;

WakeConditionVariable           唤醒一个等待条件变量的线程

WakeAllConditionVariable      唤醒所有等待条件变量的线程；

SleepConditionVariableCS       释放临界区锁和等待条件变量作为原子性操作

SleepConditionVariableSRW   释放SRW锁和等待条件变量作为原子性操作.

使用方法如下：windows管程需要配合临界区使用

Thread1：

   EnterCriticalSection(&CritSection);

   while( TestPredicate() == FALSE ){

      SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);

   }

   DoSth();

   LeaveCriticalSection(&CritSection);

Thread2：

   EnterCriticalSection(&CritSection);

      WakeConditionVariable (ConditionVar);

   LeaveCriticalSection(&CritSection);

C++11：编译器很可能不支持这个C++11特性

std::condition\_variable cv;

std::mutex mtx;

std::unique\_lock lck(mtx);

cv.wait(lck) //需要独立锁

cv.notify\_one();//随便唤醒一个

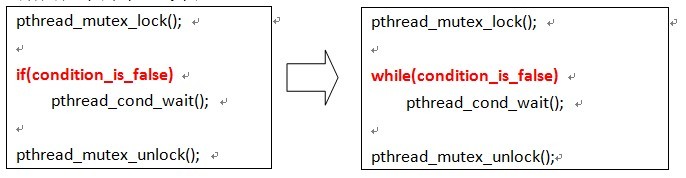
cv.notify\_all();

Linux Monitor： Linux管程配合条件变量使用，那就还要保证原子性

使用了Ubuntu 16.04系统，才能成功支持pthread库。之前试过了kali以及Ubuntu 12，但是都只是能编译成功,在kali里面我gdb调试是到了create\_pthread就会说找不到什么什么然后抛出异常产生段错误。

Linux中帮助中提到的：在多核处理器下，pthread\_cond\_signal可能会激活多于一个线程（阻塞在条件变量上的线程）。结果是，当一个线程调用pthread\_cond\_signal()后，多个调用pthread\_cond\_wait()或pthread\_cond\_timedwait()的线程返回。这种效应成为”虚假唤醒”(spurious wakeup)

“虚假唤醒”解决办法：



pthread\_cond\_t notempty = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t notfull = PTHREAD\_COND\_INITIALIZER;

int pthread\_cond\_wait(pthread\_cond\_t \*cond,

pthread\_mutex\_t \*mutex);

参数说明：

cond——条件变量指针。

mutex——互斥信号量指针。

返回值：成功则返回0，失败则返回出错编号。

这个函数不同于：

pthread\_mutex\_unlock(mtx);

pthread\_cond\_just\_wait(cv);

pthread\_mutex\_lock(mtx);

拆分掉以后就会失去原子性，然后你的unlock唤醒的可能就是其他线程

先将线程附加到等待队列（这一步是加锁的），释放mutex，进入等待，确保unlock之后唤醒的是本线程。

pthread\_cond\_signal

功能：释放条件变量，唤醒一个等待在指定条件变量上的线程。

格式：int pthread\_cond\_signal(pthread\_cond\_t \*cond);

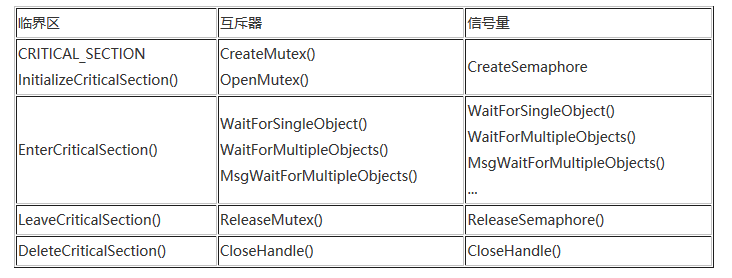
参数说明：

cond——将要释放的条件变量指针。

返回值：成功则返回0，失败则返回出错编号。

45. 在Posix.1基本原理一文声称，有了互斥锁和条件变量还提供信号量的原因是：“本标准提供信号量的而主要目的是提供一种进程间同步的方式；这些进程可能共享也可能不共享内存区。互斥锁和条件变量是作为线程间的同步机制说明的

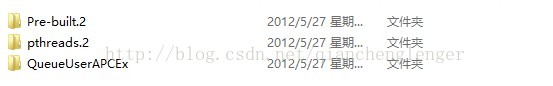
46.windows下： 这里是对上面没有提到的部分进行补充



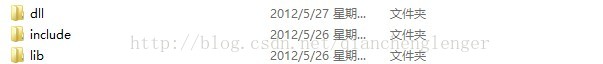
顺便希望你能注意，没想道这些函数首字母就开始大写了。

626.在VS中配置linux中的pthread.h就需要；

下载pthread包pthreads-w32-2-9-1，不知道是专门放在windows下还是别人从linux中抠出来了。



我们用到的主要是“Pre-built.2”这个文件夹下的三个文件夹，分别是动态链接库、头文件、静态链接库



之后就是需要把这几个文件放到对应的目录下面：

include文件夹中的三个文件直接拷贝到Visual Studio安装目录下VC->include文件夹下，例如我将include中文件拷贝到的位置是E:\Program Files\Microsoft Visual Studio 11.0\VC\include

 同样的办法与原因，我们也可以把lib文件夹下的内容拷贝到Visual Studio安装目录下默认的lib寻找路径中，即VC->lib中，例如我将lib文件夹下的x64与x86两个文件直接拷贝到E:\Program Files\Microsoft Visual Studio 11.0\VC\lib的下面。但是其实使用的就是x86文件夹下面的文件，所以可以直接将x86文件夹下面的文件放到lib中。

下面dll的存放我一直觉得可能反了，但是我还是照这个来了

把dll下的x64文件夹下的两个文件，即pthreadGC2.dll与pthreadVC2.dll拷贝到C:\Windows\System32下（用于64位程序的运行）

把dll下的x86文件夹下的五个文件，拷贝到C:\Windows\SysWOW64下（用于32位程序的运行），注意一下，千万不能将这些文件拷贝反位置，否则，程序运行时会提示说找不到对应的dll文件。这些在网上的很多文章中都被忽略掉了，所以我们特别提出。

最后在VS中加上#pragma comment(lib,"x86/pthreadVC2.lib")，如果上面没有放到x86文件夹下而是拿了出来，那么就不需在路径中加上x86。

库的使用方式要么是设置好环境变量要么就是直接丢到IDE的搜索路径之中。之后就需要配置ide，让它自己来找或者你在代码中引入lib文件。