# 互斥锁

用于控制多个线程对他们之间共享资源互斥访问的一个信号量。为了避免多个线程在某一时刻同时操作一个共享资源。

在某一时刻，只有一个线程可以获取互斥锁，在释放互斥锁之前其他线程都不能获取该互斥锁。如果其他线程想要获取这个互斥锁，那么这个线程只能以阻塞方式进行等待。

#include <pthread.h>

//动态方式创建锁，相当于new动态创建一个对象

pthread\_mutex\_init( pthread\_mutex\_t \* mutex, const pthread\_mutexattr\_t \* mutexattr);

//释放互斥锁，相当于delete

pthread\_mutex\_destory(pthread\_mutex\_t \* mutex);

//以静态方式创建锁

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

//以阻塞方式运行。如果之前mutex被加锁了，那么程序会阻塞在这里

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

pthread\_mutex\_unlock(pthread\_mutex\_t \* mutex);

//会尝试对mutex加锁，如果mutex之前已经被锁定，返回非0；如果mutex没有被锁定，则函数返回并锁定mutex

//以非阻塞方式运行。程序继续往下执行

int pthread\_mutex\_\_trylock(pthread\_mutex\_t \* mutex);

# 条件锁

条件锁就是所谓的条件变量，某一个线程因为某个条件为满足时可以使用条件变量使该程序处于阻塞状态。一旦条件满足，以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。

最为常见的就是在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。这个过程中就使用到了条件变量pthread\_cond\_t.

#include <pthread.h>

//对条件变量进行动态初始化，相当于new创建对象

pthread\_cond\_init( pthread\_cond\_t \* condition, const pthread\_condattr\_ t \* condattr );

//释放动态申请的条件变量，相当于delete释放对象

pthread\_cond\_destory( pthread\_cond\_t \* condition );

//静态初始化条件变量

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

//该函数以阻塞方式执行。如果某个线程中的程序执行了该函数，那么这个线程就会以阻塞方式等待

//直到收到pthread\_cond\_signal或者pthread\_cond\_broadcast函数发来的信号而被唤醒。

pthread\_cond\_wait( pthread\_cond\_t \* cond, pthread\_mutex\_t \* mutex);

pthread\_cond\_wait用法示例：

//pthread\_cond\_wait函数的语义相当于：首先解锁互斥锁，然后以阻塞方式等待条件变量的信号，收到信号后又会对互斥锁加锁

//为了防止“虚假唤醒”，该函数一般放在while循环体中

pthread\_mutex\_lock(mutex);//加互斥锁

while(条件不成立)//当前线程中条件变量不成立

{

pthread\_cond\_wait( cond, mutex );

}

…//对进程之间的共享资源进行操作

pthread\_mutex\_unlock(mutex);//释放互斥锁

//在另外一个线程中改变线程，条件满足发送信号。唤醒一个等待的线程（可能有多个线程处于阻塞状态），唤醒哪个线程由具体的线程调度策略决定

pthread\_cond\_signal( pthread\_cond\_t \*cond );

//以广播形式唤醒所有因为该条件变量而阻塞的所有线程，唤醒哪个线程由具体的线程调度策略决定

pthread\_cond\_broadcast( pthread\_cond\_t \* cond);

//以阻塞方式等待，如果时间time到了条件还没有满足还是会结束

pthread\_cond\_timewait( pthread\_cond\_t \* cond, pthread\_mutex\_t \* mutex, struct timespec \* time );

# 自旋锁

假设我们有一个两个处理器core1和core2计算机。现在，在这台计算机上运行的程序中有两个线程：T1和T2分别在处理器core1和core2上运行，两个线程之间共享着一个资源。

互斥锁工作原理：一种sleep-waiting的锁。假设线程T1获取互斥锁并且正在core1上运行时，此时线程T2也想要获取互斥锁（pthread\_mutex\_lock）,但是由于T1正在使用互斥锁使得T2被阻塞。当T2处于阻塞状态时，T2被放入到等待队列中去，处理器core2会去处理其他任务而不必一直等待（忙等）。

自旋锁工作原理：一种busy-waiting的锁。如果T1正在使用自旋锁，而T2也去申请这个自旋锁，此时T2肯定得不到这个自旋锁。与互斥锁相反的是，此时运行T2的处理器core2会一致不断地循环检查锁是否可用（自旋锁请求），直到获取到这个自旋锁为止。

当发生阻塞时：

* 互斥锁可以让CPU去处理其他的任务
* 自旋锁让CPU一直不断循环请求获取这个锁

#include <linux/spinlock.h>

//初始化

spin\_lock\_init(spinlock\_t \* x);

//只有在获得锁的情况下才返回，否则一致“自旋”

spin\_lock(x);

//如立即获得锁则返回真，否则立即返回假

spin\_trylock(x);

//释放锁

spin\_unlock(x);

//该宏用于判断自旋锁x是否已经被某执行单元保持（即被锁），如果是，返回真；否则返回假

spin\_is\_locked(x)

注意：自旋锁适合于短时间的轻量级的加锁机制。

# 读写锁

计算机中某些数据被多个进程共享，对数据库的操作有两种：一种是读操作，就是从数据库中读取数据，不会修改数据库中内容；另外一种是写操作，会修改数据库中存放的数据。

允许在数据库上同时执行多个“读”操作，但是某一时刻只能在数据库上有一个“写”操作在更新数据（读者——写者模型）。