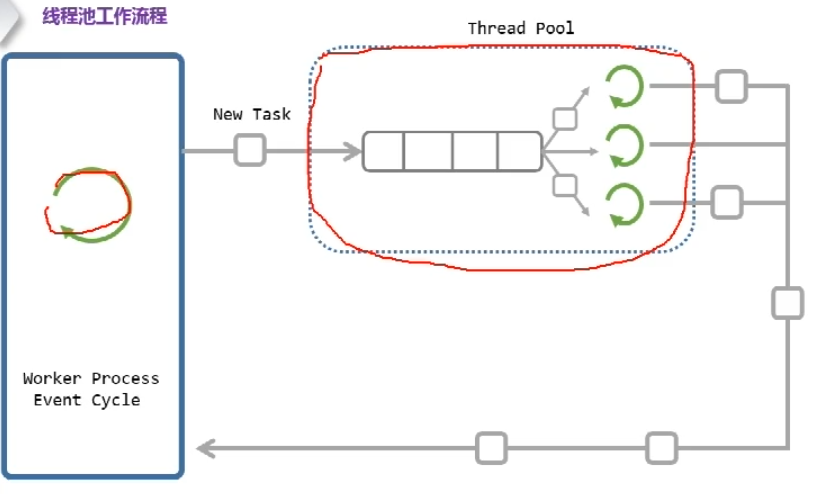
1. 线程池用来干什莫？

线程（相当于银行的柜员）

例如：Nginx（大型的web服务器）异步解耦

线程池如何做到异步解耦：



任务队列



2. 线程如何实现：

1.任务队列

2.执行队列

3.管理组件（管理任务队列和执行队列）

3．公示牌属于谁的属性：

他应该属于营业厅的属性（线程池），用来管理柜员和客户井然有序的进行工作

4．内存申请的区别

Malloc 和 calloc区别： 内核的kmalloc vmalloc jemalloc tcmalloc

Malloc 连续的一块块

Calloc 追加 calloc() 自动置0

5.线程的互斥锁：

互斥的概念：

在多线程编程中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。 每个对象都对应于一个可称为" 互斥锁" 的标记，这个标记用来保证在任一时刻， 只能有一个线程访问该对象

创建互斥锁：

在使用互斥锁之前，需要先创建一个互斥锁的对象。 互斥锁的类型是 pthread\_mutex\_t ，所以定义一个变量就是创建了一个互斥锁。

pthread\_mutex\_t mtx;

这就定义了一个互斥锁。但是如果想使用这个互斥锁还是不行的，我们还需要对这个互斥锁进行初始化， 使用函数 pthread\_mutex\_init() 对互斥锁进行初始化操作。

//第二个参数是 NULL 的话，互斥锁的属性会设置为默认属性

pthread\_mutex\_init(&mtx, NULL);

除了使用 pthread\_mutex\_init() 初始化一个互斥锁，我们还可以使用下面的方式定义一个互斥锁：

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

在头文件 /usr/include/pthread.h 中，对 PTHREAD\_MUTEX\_INITIALIZER 的声明如下

# define PTHREAD\_MUTEX\_INITIALIZER \

{ { 0, 0, 0, 0, 0, 0, { 0, 0 } } }

为什么可以这样初始化呢，因为互斥锁的类型 pthread\_mutex\_t 是一个联合体， 其声明在文件 /usr/include/bits/pthreadtypes.h 中，代码如下：

获取互斥锁

接下来是如何使用互斥锁进行互斥操作。在进行互斥操作的时候， 应该先"拿到锁"再执行需要互斥的操作，否则可能会导致多个线程都需要访问的数据结果不一致。 例如在一个线程在试图修改一个变量的时候，另一个线程也试图去修改这个变量， 那就很可能让后修改的这个线程把前面线程所做的修改覆盖了。

下面是获取锁的操作：

阻塞调用

pthread\_mutex\_lock(&mtx);

pthread\_mutex\_lock(&mtx);

这个操作是阻塞调用的，也就是说，如果这个锁此时正在被其它线程占用， 那么 pthread\_mutex\_lock() 调用会进入到这个锁的排队队列中，并会进入阻塞状态， 直到拿到锁之后才会返回。

非阻塞调用

如果不想阻塞，而是想尝试获取一下，如果锁被占用咱就不用，如果没被占用那就用， 这该怎么实现呢？可以使用 pthread\_mutex\_trylock() 函数。 这个函数和 pthread\_mutex\_lock() 用法一样，只不过当请求的锁正在被占用的时候， 不会进入阻塞状态，而是立刻返回，并返回一个错误代码 EBUSY，意思是说， 有其它线程正在使用这个锁。

超时调用

如果不想不断的调用 pthread\_mutex\_trylock() 来测试互斥锁是否可用， 而是想阻塞调用，但是增加一个超时时间呢，那么可以使用 pthread\_mutex\_timedlock() 来解决， 其调用方式如下：

struct timespec abs\_timeout;

abs\_timeout.tv\_sec = time(NULL) + 1;

abs\_timeout.tv\_nsec = 0;

int err = pthread\_mutex\_timedlock(&mtx, &abs\_timeout);

if(0 != err) {

if(ETIMEDOUT == err) {

//The mutex could not be locked before the specified timeout expired.

}

}

上面代码的意思是，阻塞等待线程锁，但是只等1秒钟，一秒钟后如果还没拿到锁的话， 那就返回，并返回一个错误代码 ETIMEDOUT，意思是超时了。

其中 timespec 定义在头文件 time.h 中，其定义如下

struct timespec

{

\_\_time\_t tv\_sec; /\* Seconds. \*/

long int tv\_nsec; /\* Nanoseconds. \*/

};

还需要注意的是，这个函数里面的时间，是绝对时间，所以这里用 time() 函数返回的时间增加了 1 秒。

释放互斥锁

用完了互斥锁，一定要记得释放，不然下一个想要获得这个锁的线程， 就只能去等着了，如果那个线程很不幸的使用了阻塞等待，那就悲催了。

释放互斥锁比较简单，使用 pthread\_mutex\_unlock() 即可：

pthread\_mutex\_unlock(&mtx);

销毁线程锁

通过 man pthread\_mutex\_destroy 命令可以看到 pthread\_mutex\_destroy() 函数的说明， 在使用此函数销毁一个线程锁后，线程锁的状态变为"未定义"。有的 pthread\_mutex\_destroy 实现方式，会使线程锁变为一个不可用的值。一个被销毁的线程锁可以被 pthread\_mutex\_init() 再次初始化。对被销毁的线程锁进行其它操作，其结果是未定义的。

对一个处于已初始化但未锁定状态的线程锁进行销毁是安全的。尽量避免对一个处于锁定状态的线程锁进行销毁操作。

销毁线程锁的操作如下：

pthread\_mutex\_destroy(&mtx)

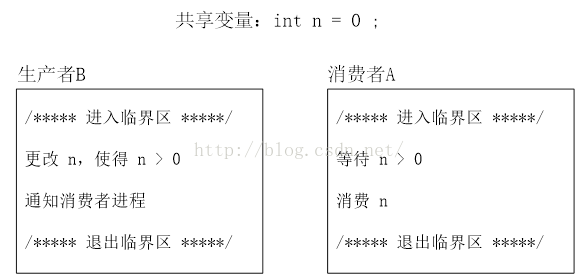
pthread\_mutex\_t 和 pthread\_cond\_t 配合使用的简要分析

假设有两个线程同时访问一个全局变量 n，这个全局变量的初始值等于0。

Int n = 0 ;

消费者线程 A 进入临界区，访问 n，A 必须等到 n 大于 0 才能接着往下执行，如果 n== 0，那么 A 将一直等待。

还有一个生产者线程 B，B 进入临界区，修改 n 的值，使得 n >0，当 n > 0 时，B 通知等待 n > 0 的消费者线程A。A 被 B 通知之后就可以接着往下执行了。



以上情况造成死锁：

当 A 进入临界区时，其他线程不能进入临界区，意味着 B 没有机会去修改 n， n 的值一直为 0，不满足A 继续执行的条件（n > 0），A 只能一直等待。

消费者进程拿到互斥锁 --> 进入临界区 --> 发现共享资源 n 不满足继续执行的条件（n> 0） --> 等待 n > 0

消费者进程占有互斥锁 --> 生产者进程无法进入临界区 --> 无法修改 n 的值 --> 生产者等待消费者释放互斥锁

解决死锁的方案就是采用条件变量。

通常情况下，对共享资源（比如 n）保护要用到锁操作，当一个进程进入临界区时会拿到互斥锁（lock 操作），然后其他进程拿不到互斥锁，也就无法进入临界区，因此当进程进入临界区，发现共享资源不满足继续向下执行的条件（n > 0）时，就应该释放锁，让其他进程修改共享资源，以满足自己所需的执行条件。

消费者进入临界区 --> 共享变量不满足继续向下执行的条件 --> 消费者等待在条件变量 --> 释放互斥锁 --> 生产者进入临界区 --> 修改条件变量 --> 生产者通知消费者：现在有多的资源了，快来使用 --> 消费者再次拿互斥锁 --> 消费资源 --> 释放互斥锁。如果有多个消费者进程等待在条件变量上，就可以形成等待队列。

2.使用方法

条件变量的使用主要有以下五个函数：

/\* 初始化一个条件变量 \*/

int pthread\_cond\_init (pthread\_cond\_t\* cond, pthread\_condattr\_t \*cond\_attr);

/\* 销毁一个条件变量 \*/

int pthread\_cond\_destroy(pthread\_cond\_t\* cond);

/\* 令一个消费者等待在条件变量上 \*/

int pthread\_cond\_destroy(pthread\_cond\_t\* cond);

/\* 生产者通知等待在条件变量上的消费者 \*/

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

/\* 生产者向消费者广播消息 \*/

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

消费者等待条件的伪代码：

pthread\_mutex\_lock(&mutex); // 拿到互斥锁，进入临界区

while( 条件为假)

pthread\_cond\_wait(cond, mutex); // 令进程等待在条件变量上

修改条件

pthread\_mutex\_unlock(&mutex); // 释放互斥锁

生产者通知消费者的伪代码：

pthread\_mutex\_lock(&mutex); // 拿到互斥锁，进入临界区

设置条件为真

pthread\_cond\_signal(cond); // 通知等待在条件变量上的消费者

pthread\_mutex\_unlock(&mutex); // 释放互斥锁