通知状态的改变: POSIX 条件变量

作用

- 在 POSIX 线程中，互斥量用于防止多个线程同时访问同一共享变量。条件变量允许一个线程就某个共享变量的状态变化通知其它线程，并让其它线程等待（阻塞）这一通知
- 阻塞式消息队列是对条件变量的一个非常经典的应用，在 Java 以及 Python 标准库中的双端阻塞队列均使用条件变量实现

为什么需要条件变量?

- 我们考虑这样一个场景: 消费者从队列中取出数据，当队列为空时，我们该怎么做?
- Solution 1: while 循环 + sleep(0.5)，的确能用，但是如果生产者生产数据的速度不高，比如每天 10 条，那么就会出现大量无用的 CPU 消耗，白白浪费系统资源
- Solution 2: 当我们发现队列为空时，主动的让出 CPU，使线程阻塞在此处。当生产者生产的新的消息时，传递一个信号唤醒当前线程，线程继续执行
- 可以看到，Solution 2 对资源的消耗更低，因为性能更高。线程阻塞以及外部唤醒可由条件变量实现

使用

- 条件变量必须配合互斥量使用。条件变量针对于共享变量的状态改变发出通知，而互斥量则提供对该共享变量的互斥的访问
- 释义（为什么需要互斥量）
  - 消费者从队列中取出数据，那么我们必须要判断队列是否为空。这相当于对共享资源的并发访问，也就是说，必须使用互斥量对其进行保护
  - 可以看到，互斥量是为了保护共享数据的

```
pthread_mutex_t mtx =PTHREAD_MUTEX_INITIALIZER;     // 静态初始化互斥量

pthread_mutex_lock(&mtx);                           // 加锁

if(get_size(queue) != 0) {/* 消费数据 */}            // 队列不为空，消费数据
else {/* 阻塞在此处，等待唤醒 */}                    // 队列为空，阻塞在此处

pthread_mutex_unlock(&mtx);                         // 解锁
```

通知和等待条件变量

- 初始化条件变量
  - 静态初始化
    - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - 动态初始化
    - int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr); // 初始化
      int pthread_cond_destory(pthread_cond_t *cond); // 必须调用该函数进行销毁
- 等待
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - 主动让出 CPU，使当前线程阻塞在此处
- 通知
  - int pthread_cond_signal(pthread_cond_t *cond);
    - 唤醒至少一条阻塞线程
    - 常用于多个线程执行相同的任务，那么此时唤醒一个线程即可，也会减少锁的争抢
  - int pthread_cond_broadcast(pthread_cond_t *cond);
    - 唤醒全部阻塞线程
    - 常用于多个线程执行不同的任务
  - pthread_cond_signal 和 pthread_cond_broadcast 可多次调用，不会有副作用

All return 0 on success, or a positive err number on error

pthread_cond_wait() 做了什么?

pthread_cond_wait 主要执行了三个操作

- 解锁互斥量 mutex
  - 当我们发现条件不满足程序运行的要求时（例如队列为空），那么线程应该在进入休眠之前解锁互斥量
  - 这样一来其它的线程才能够访问共享变量，否则互斥量将一直被当前线程所持有，多线程将退化成单线程
- 阻塞调用线程，直至另一个线程就条件变量 cond 发出信号
- 重新锁定 mutex
  - 当线程因为条件变量的通知而再度被唤醒时，必须对互斥量再次加锁，因为在典型情况下，线程会立即访问共享变量

测试条件变量的判断条件（predict）

- 使用 codition 改写消费者代码

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

// 我们需要定义两个条件变量
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

while (true) {
   pthread_mytex_lock(&mtx);

   while (get_size(queue) == 0)
      pthread_cond_wait(&not_empty, &mutex);

   /* 从队列中取出数据并消费 */

   pthread_cond_signal(&not_full);      // 通知阻塞在 not_full 条件变量的线程数据已被消费

   pthread_mutex_unlock(&mtx);
}
```

- ps: 篇幅有限，故上述代码未做错误判断
- Warning
  - 尽可能地使得消费者代码简短，如果处理数据耗时较长，也不要提前将互斥量释放
  - 也就是说，当我们从队列中取出一条数据之后就将互斥量释放，这么做固然能提高效率，但是消费者会有永久阻塞的风险
- 我们对条件的判断必须使用一个 while 循环来控制，而不是 if 语句
- 当线程从 pthread_cond_wait() 调用返回时，我们不能够保证队列一定不为空，也就是说，可能由其它线程获得了互斥量并消费了数据
- 这和"双重校验"的原理基本一样，当线程被唤醒时，所做的第一件事情就是重新锁定互斥量，该互斥量可能由其它线程取得并执行了消费动作。那么此时该线程获得互斥量之后队列可能仍然为空
- 我们用一段 Python 代码来演示双重校验

```python
class Singleton(object):
    _instance_lock = threading.Lock()

    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, "_single_ins"):
            with cls._instance_lock:
                if not hasattr(cls, "_single_ins"):
                    cls._single_ins = object.__new__(cls)
        return cls._single_ins
```