# 线程锁 基础

锁的侧重点在于 资源的保护。

## 互斥锁 mutex

互斥锁 是用于互斥的。当资源的访问出现竞争的时候，则需考虑用 互斥锁。

### 互斥锁 的状态

互斥锁 有两种状态：

| **状态** | **描述** |
| --- | --- |
| **未锁定状态** |  |
| **锁定状态** | 只有一个线程可以占有 互斥锁 |

互斥锁 有两种操作:

| **操作** | **描述** |
| --- | --- |
| **加锁** | 加锁 |
| **解锁** | 解锁 |

互斥锁 的状态跳转表：

| **原状态** | **加锁** | **解锁** |
| --- | --- | --- |
| **未锁定状态** | 互斥锁 进入锁定状态 | 互斥锁 保持未锁定状态 |
| **锁定状态** | 本线程 进入阻塞等待 | 互斥锁 进入未锁定状态 |

### 互斥锁 静态初始化

POSIX定义了一个宏 PTHREAD\_MUTEX\_INITIALIZER 来静态初始化 互斥锁：

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

### 互斥锁 动态初始化

pthread\_mutex\_init() 用来初始化 互斥锁。

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

其中 attr 用于指定互斥锁属性，如果为 NULL 则使用缺省属性。

### 互斥锁 销毁

pthread\_mutex\_destroy() 用于销毁 互斥锁。

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

销毁互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。

### 互斥锁 加锁

pthread\_mutex\_lock() 用于 互斥锁 加锁。

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

**1. 如果 互斥锁 没有加锁。pthread\_mutex\_lock 对该 互斥锁 加锁。最后返回 0。**

**2. 如果 互斥锁 已经加锁。pthread\_mutex\_lock 则会阻塞当前线程。直到 互斥锁 释放。**

注意：

同一线程连续多次 pthread\_mutex\_lock ，会造成死锁。

pthread\_mutex\_trylock() 用于 互斥锁 加锁。但是不会阻塞。

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

**1. pthread\_mutex\_trylock 与 pthread\_mutex\_lock 相似。**

**2. 如果 互斥锁 已经加锁。pthread\_mutex\_trylock 则不会阻塞线程。而是立即返回 错误码。**

### 互斥锁 解锁

pthread\_mutex\_unlock 用于 互斥锁 解锁。

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

## 读写锁 read-write lock

读写锁，用于并发读取和独占写入。适合在读取的次数大于写入的场合。

### 读写锁 的状态

读写锁 有三种状态：

| **状态** | **描述** |
| --- | --- |
| **未锁定状态** |  |
| **读锁定状态** | 多个线程可以同时占有 读锁 |
| **写锁定状态** | 只有一个线程可以占有 写锁 |

读写锁 有三种操作：

| **操作** | **描述** |
| --- | --- |
| **读加锁** | 读加锁 |
| **写加锁** | 写加锁 |
| **解锁** | 读解锁 或者 写解锁 |

互斥锁 的状态跳转表：

| **原状态** | **读加锁** | **写加锁** | **解锁** |
| --- | --- | --- | --- |
| **未锁定状态** | 取得 读锁，读写锁 进入 读锁定状态 | 取得 写锁，读写锁 进入 进入写锁定状态 | 出现未定义问题 |
| **读锁定状态** | * 如果已经有 写锁 在阻塞等待，则 本线程阻塞； * 否则取得 读锁，读写锁 进入 读锁定状态 | 本线程阻塞，并且 读写锁 阻塞随后的 读加锁 操作 | * 如果这是最后的读锁，则 读写锁 进入 未锁定状态； * 否则 读写锁 进入 读锁定状态 |
| **写锁定状态** | 本线程阻塞 | 本线程阻塞 | 读写锁 进入 未锁定状态 |

### 读写锁 初始化

pthread\_rwlock\_init() 用于初始化 读写锁。正常初始化后 读写锁会在 未加锁的状态。

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock,

const pthread\_rwlockattr\_t \*restrict attr);

其中 attr 用于指定 读写锁 属性，如果为 NULL 则使用缺省属性。

使用 没有初始化 或者 多次初始化的 读写锁，都会出现不确定的问题。

### 读写锁 销毁

pthread\_rwlock\_destroy 用于销毁 读写锁。

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

**1. 使用销毁后的 读写锁 会出现不确定的问题。**

**2. 但是销毁后的 读写锁 可以 使用 pthread\_rwlock\_init 重新初始化。**

### 读写锁 读加锁

* pthread\_rwlock\_rdlock() 用于 读写锁 的 读加锁。

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);

**1. 如果 读写锁 在 未锁定状态。则可以取得 读锁。读写锁 进入 读锁定状态。函数最后返回 0。**

**2. 如果 读写锁 在 读锁定状态:**

**2.1. 如果 已经有 写锁 在阻塞等待。则会 阻塞本线程，直到 写锁 释放。**

**2.2. 如果 没有 写锁 在阻塞等待。则可以取得 读锁。读写锁 进入 读锁定状态。函数最后返回 0。**

**3. 如果 读写锁 在 写锁状态。则会阻塞本线程，直到 写锁 释放。**

* pthread\_rwlock\_tryrdlock() 用于 读写锁 读加锁。但是不会阻塞。

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock);

**1. pthread\_rwlock\_tryrdlock 与 pthread\_rwlock\_rdlock 相似。**

**2. 只是在 无法取得 读锁 的时候，不会阻塞线程。而是返回一个错误码。**

### 读写锁 写加锁

* pthread\_rwlock\_wrlock 用于 读写锁 写加锁。

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

**1. 如果 读写锁 在 未锁定状态。则可以取得 写锁。读写锁 进入 写锁定状态。函数最后返回 0。**

**2. 如果 读写锁 在 读锁状态 或者 写锁状态。则会 阻塞线程，直到 读锁 或者 写锁 释放。**

注意：

同一线程连续多次 pthread\_rwlock\_wrlock ，会造成死锁。

* pthread\_rwlock\_trywrlock 用于 读写锁 写加锁。但是不会阻塞。

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock);

**1. pthread\_rwlock\_trywrlock 与 pthread\_rwlock\_wrlock 类似。**

**2. 只是在 无法取得 写锁 的时候，不会阻塞。而是返回一个错误码。**

### 读写锁 解锁

pthread\_rwlock\_unlock 用于 读写锁 解锁。

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

**1. 如果 读写锁 在 未锁定状态，会造成 未定义问题。**

**2. 如果 读写锁 在 读锁状态:**

**2.1. 如果 这是最后的读锁，则 读写锁 进入 未锁定状态。**

**2.2. 如果 另外有其他线程占有读锁，读写锁 进入 读锁定状态。**

**3. 如果 读写锁 在 写锁状态，则返回 未锁定状态。**

注意：

跨线程调用 pthread\_rwlock\_unlock 解锁会造成未定义的问题。

## 自旋锁 spin lock

自旋锁 与 互斥锁 类似。只是 当 加锁的时候，如果 遇到 锁定状态，则自旋等待。

自旋等待可以理解为 一个 while 循环等待，线程不会睡眠，所以免除了线程唤醒的时间，但是会比较耗费 CPU。

自旋锁 适用于 锁定 短时间的 操作。

由于 自旋锁 的函数接口与 互斥锁 类似，所以不再赘述。

### 自旋锁 初始化 和 销毁

int pthread\_spin\_init(pthread\_spinlock\_t \*lock, int pshared);

int pthread\_spin\_destroy(pthread\_spinlock\_t \*lock);

pthread\_spin\_init 中的 pshared 表示进程共享同步：

**1. pshared 为 PTHREAD\_PROCESS\_SHARED，可以用于进程间共享（通过共享内存）。**

**2. pshared 为 PTHREAD\_PROCESS\_PRIVATE ，只可以用于进程内部线程间访问。**

### 自旋锁 加锁

int pthread\_spin\_lock(pthread\_spinlock\_t \*lock);

int pthread\_spin\_trylock(pthread\_spinlock\_t \*lock);

### 自旋锁 解锁

int pthread\_spin\_unlock(pthread\_spinlock\_t \*lock);

# 线程同步 基础

线程同步的侧重点在于 消息的通知。

## 条件变量 condition variable

条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。避免线程的忙等。

条件变量需要 互斥锁 来配合使用。

条件变量使我们可以睡眠等待某种条件出现。

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：

**1. 一个线程等待 "条件变量的条件成立" 而挂起。**

**2. 另一个线程使 "条件成立"（给出条件成立信号）。**

### 条件变量 静态初始化

宏 PTHREAD\_COND\_INITIALIZER 用来静态初始化 条件变量：

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

### 条件变量 动态初始化

pthread\_cond\_init() 用来初始化 条件变量。

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,

const pthread\_condattr\_t \*restrict attr);

其中 attr 用于指定 条件变量 属性，如果为 NULL 则使用缺省属性。

### 条件变量 销毁

pthread\_cond\_destroy() 用于销毁 条件变量。

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

**1. 使用销毁后的 条件变量 会出现不确定的问题。**

**2. 销毁后的 条件变量 可以 使用 pthread\_cond\_init 重新初始化。**

### 条件变量 发送信号

pthread\_cond\_signal 用于 唤醒 （任意）一个在等待 条件变量 的线程。

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

pthread\_cond\_signal 用于 唤醒 所有在等待 条件变量 的线程。

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

### 条件变量 等待

* pthread\_cond\_wait 用于等待 条件变量的唤醒。

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

pthread\_mutex\_t \*restrict mutex);

**1. pthread\_cond\_wait 在调用之前，都需要 使用 mutex 来 加锁。否则会出现 未定义的问题。**

**2. pthread\_cond\_wait 会自动释放 mutex。 并且令到 线程进入阻塞等待，等待到 条件变量 被唤醒。**

**3. 当 pthread\_cond\_wait 返回时，会对 mutex 加锁。**

注意：

1. 为了防止竞争，条件变量总是和 互斥锁 配合一起使用。

2. pthread\_cond\_wait 需要 使用 while 来包裹。请参考后面的 “条件变量 例子”。

* pthread\_cond\_timedwait 用于等待 一个条件变量。

​

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex,

const struct timespec \*restrict abstime);

**1. pthread\_cond\_timedwait 与 pthread\_cond\_wait 类似。**

**2. pthread\_cond\_timedwait 在 在给定的时刻 abstime 前，如果都等待不到 pthread\_cond\_signal 或者 pthread\_cond\_broadcast 发起的唤起信号。则会返回错误码。**

**3. abstime 表示的是 绝对时刻。请 参考后记的 “绝对时刻”。**

### 条件变量 例子

#### 定义 和 初始化

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

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

pthread\_cond\_init(&cond, NULL);

#### 执行 signal 操作

pthread\_mutex\_lock(&mutex);

/\* 设置 <条件> 为真 \*/

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutex);

注意：

pthread\_cond\_signal 与 pthread\_mutex\_unlock的顺序：

1. 如果先 signal 后 unlock，则可以确定 signal 操作是由 lock 住 mutex 的线程调用的。

2. 如果先 unlock 后 signal，则任一线程都可调用 signal 操作。

3. 如果需要可预见的调度行为，最好先 signal 后 unlock，如上面所示。

4. 要先修改条件，再 signal。从源头上避免假唤醒 (spurios wakeup)。

#### 执行 wait 操作

pthread\_mutex\_lock(&mutex);

while (/\* 当 <条件> 为假 \*/) {

pthread\_cond\_wait(&cond, &mutex);

}

/\* 在 <条件> 为真 后，处理业务逻辑 \*/

pthread\_mutex\_unlock(&mutex);

注意：

1. pthread\_cond\_wait 必须放置 在 pthread\_mutex\_lock 的后面，否则会遗漏事件。

2. pthread\_cond\_wait 必须使用 while 来包裹（不能使用 if）。这是为了可以重复检测 <条件>，防止假唤醒 (spurios wakeup)。

## 信号量 semaphore

信号量 是一个原子的计数器，可以记录资源的使用情况。

当 计数器 为零的时候，表示资源耗尽，当进一步请求资源，将会阻塞线程。

注意：

这里只讨论了 线程间的无名信号量。另外两种是 线程间的 有名信号量 和 进程间的 信号量。

有名信号量请查看 sem\_open(manpage class 3)。

进程间信号量请参看 semget(manpage class 2)。

### 信号量 初始化

sem\_init 用于初始化 无名信号量。

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

**1. 如果 pshared 为零。信号量 可以在 线程间使用。信号量 需要的定义为 全局变量，或者 从 堆 中动态产生。**

**2. 如果 pshared 为非零。信号量 可以在 进程间使用。信号量 需要的定义在 共享内存 中。**

**3. 参数 value 为 信号量 计数器的初始值。**

注意：

重复 初始化 信号量，会导致 未知的问题。

### 信号量 解锁

sem\_post 用于 增加 信号量的值（解锁）。

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

**1. sem\_post 后，如果 信号量的值 大于 0，则 有由 sem\_wait 所 阻塞的 进程 或者 线程 将会被唤醒。**

**2. sem\_post 成功 增加 信号量的值后，sem\_post 会返回 0。**

**3. sem\_post 如果 出错。则 信号量的值 会保持不变。sem\_post 会返回 -1。**

### 信号量 等待

* sem\_wait 用于 减少 信号量的值（加锁）。

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

**1. 如果 当前 信号量的值 大于 0，信号量的值 减一，sem\_wait 立即返回 0。**

**2. 如果 当前 信号量的值 等于 0，线程阻塞等待。直到 信号量的值 大于 0 或者 线程 接收到 signal 中断。**

* sem\_trywait 用于 减少 信号量的值（加锁）。但是不会阻塞。

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

**1. sem\_trywait 与 sem\_wait 相似。**

**2. 如果 当前 信号量的值 等于 0，sem\_trywait 不会 阻塞线程，而是返回 错误码。**

* sem\_timedwait 用于 减少 信号量的值（加锁）。但提供超时等待。

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);

**1. sem\_timedwait 与 sem\_wait 相似。**

**2. sem\_timedwait 在给定的时刻 abs\_timeout (绝对时刻)前，如果都等待不到 信号量的值 大于 0。则会返回错误码。**

**3. abs\_timeout 表示的是 绝对时刻。请 参考后记的 “绝对时刻”。**

### 读取 信号量 的值

sem\_getvalue 用于读取 信号量的值。

int sem\_getvalue(sem\_t \*sem, int \*sval);

**1. sval 是 返回参数的 指针。指针所指向的类型是有符号整数。**

**2. 当 \*sval 为零，表示有线程在阻塞等待。**

## 屏障 barrier

屏障让每个合作线程等待，直到所有合作线程到达一点后，然后从该点继续执行。

### 屏障 初始化

pthread\_barrier\_init 用于 屏障 的初始化。

int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier,

const pthread\_barrierattr\_t \*restrict attr, unsigned count);

**1. 其中 attr 用于指定 屏障 属性，如果为 NULL 则使用缺省属性。**

**2. 其中 count 是 屏障 的计数器，表示 需要等待的合作线程的个数。count 的数量一定要大于零。**

### 屏障 销毁

pthread\_barrier\_destroy 用于 屏障 的销毁。

int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

**1. 使用销毁后的 屏障 会出现不确定的问题。**

**2. 但是销毁后的 屏障 可以 使用 pthread\_barrier\_init 重新初始化。**

**3. 如果有线程因为 屏障 而在阻塞等待中，此时 销毁屏障 会出现不确定的问题。**

### 屏障 等待

pthread\_barrier\_wait 用于 屏障 的等待。

int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

**1. 线程会阻塞等待，直到调用 pthread\_barrier\_wait() 线程的个数 到达 屏障计数器 的指定值。**

**2. pthread\_barrier\_wait() 的返回值：**

**2.1. 线程的个数 到达 屏障计数器 的指定值后: 屏障 会在合作线程中抽选一个，返回 PTHREAD\_BARRIER\_SERIAL\_THREAD；而其余的合作线程则返回 0；**

**2.2. 如果 pthread\_barrier\_wait() 出错，将返回错误码。**

**3. 当 屏障 到达指定值后，则需要 使用 pthread\_barrier\_init() 重新初始化，方可重新使用。**

注意：

1. 判断 pthread\_barrier\_wait() 的返回值是否为 PTHREAD\_BARRIER\_SERIAL\_THREAD。则可以将该线程作为后续的业务逻辑执行线程。

# 后记

## 绝对时刻

struct timespec 是绝对时刻的数据结构。表示的是 自从 1970-01-01 00:00:00 +0000 (UTC) 所经过的时间。

struct timespec {

time\_t tv\_sec; /\* Seconds \*/

long tv\_nsec; /\* Nanoseconds [0 .. 999999999] \*/

};

当前的时刻可以使用 clock\_gettime() 配合 CLOCK\_REALTIME 来读取。

int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp);

clk\_id 表示时钟类型，常用的有如下4种：

**1. CLOCK\_REALTIME 统当前时间，从 1970-01-01 00:00:00 +0000 (UTC)算起。**

**2. CLOCK\_MONOTONIC 系统的启动时间，不能被设置。**

**3. CLOCK\_PROCESS\_CPUTIME\_ID 本进程运行时间。**

**4. CLOCK\_THREAD\_CPUTIME\_ID 本线程运行时间。**

为了得到未来的时刻。则需要先取得当前的时刻，然后在此基础上加上一定的时间间隔。

## 互斥量 和 信号量 的比较

**1. 互斥量用于线程的互斥，信号量用于线程的同步。**

这是互斥量和信号量的根本区别，也就是互斥和同步之间的区别。

互斥：

是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。

但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

同步：

是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。

在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。

少数情况是指可以允许多个访问者同时访问资源。

**2. 互斥量值只能为0/1，信号量值可以为非负整数。**

也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。

信号量可以实现多个同类资源的多线程互斥和同步。

当信号量为单值信号量时，也可以完成一个资源的互斥访问。

理论上 互斥锁 可以理解为取值为 0/1 的 信号量，但是实现上会有所不同。

**3. 互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。**

信号量用在多线程多任务同步的，一个线程完成了某一个动作就通过信号量告诉别的线程，别的线程再进行某些动作（大家都在semtake的时候，就阻塞在那里）。

而互斥锁是用在多线程多任务互斥的，一个线程占用了某一个资源，那么别的线程就无法访问，直到这个线程 unlock，其他的线程才开始可以利用这个资源。

比如对全局变量的访问，有时要加锁，操作完了，再解锁。

有的时候锁和信号量会同时使用的

也就是说，信号量不一定是锁定某一个资源，而是流程上的概念，比如：

有 A,B 两个线程，B线程要等A线程完成某一任务以后再进行自己下面的步骤，这个任务 并不一定是锁定某一资源，还可以是进行一些计算或者数据处理之类。

而线程互斥量则是 “锁住某一资源” 的概念，在锁定期间内，其他线程无法对被保护的数据进 行操作。在有些情况下两者可以互换。

两者之间的区别:

* 作用域
* 信号量:

进程间或线程间 (linux 仅线程间的无名信号量 pthread semaphore)

* 互斥锁:

线程间

* 加锁时
* 信号量:

只要信号量的value大于 0，其他线程就可以 sem\_wait成功，成功后信号量的value减一。

若value值不大于0，则 sem\_wait 使得线程阻塞，直到sem\_post释放后 value值加一, 但是 sem\_wait 返回之前还是会将此value值减一

* 互斥锁:

只要被锁住，其他任何线程都不可以访问被保护的资源

## 条件变量 和 信号量 的比较

信号量和 条件变量 都基于 互斥锁，都用来同步线程对于共享资源的访问。两者的使用场景相似。

### 条件变量 的例子

条件变量 一般是用于：

避免线程忙等资源。

例如 线程 需要等待 queue 为空的后，才可以继续业务。

为了检测 queue 是否为空。代码可以写为：

/\* 等待 queue 为空 \*/

while(!queue\_is\_empty(&queue)) {

sleep(1);

}

/\* queue 为空的后的业务逻辑 \*/

上面的代码，主要的问题是会耗费 CPU。因为在一直循环的检测条件是否满足。

如果使用 条件变量。代码可以改进为：

/\* 等待 queue 为空 \*/

pthread\_mutex\_lock(&mutex);

while(!queue\_is\_empty(&queue)) {

pthread\_cond\_wait(&cond, &mutex);

}

/\* queue 为空的后的业务逻辑 \*/

pthread\_mutex\_unlock(&mutex);

当 queue 为空后，控制线程就可以使用 pthread\_cond\_signal 来唤起一个等待线程。或者用 pthread\_cond\_broadcast 唤起所有等待线程。

### 信号量 的例子

信号量 最好用于：

使用一个整数计数器来表示资源状况。

例如 有一个 多 producer 多 consumers 的 自动贩卖机。

其中 的商品库存量，则可以用 信号量 来表示。

consumer 线程可以执行 sem\_wait()，表示购买商品。

如果 商品有库存。sem\_wait() 则会令 库存量 减一，并顺利返回。

如果 商品库存为零。sem\_wait() 则会 一直等待。

producer 线程可以 执行 sem\_post()，表示补充商品。

当 库存量（信号量）大于零。则可以唤醒等待的 consumer 线程。

条件变量 和 信号量 都基于 互斥，但是都添加了额外的功能，来处理线程同步。

到底要选择哪一个，往往取决于你的场景。

更有甚者，两者可以同时采用。

例如 上面的 自动贩卖机 的例子中。可以使用 条件变量 来通知 producer 线程，一旦商品库存量跌入安全阈值，则补充商品。

# 参考资料

APUE

https://www.jianshu.com/p/32068ffa6136

https://www.ibm.com/developerworks/cn/linux/thread/posix\_threadapi/part3/

https://stackoverflow.com/questions/3513045/conditional-variable-vs-semaphore