# 线程同步

线程同步，是协调步调，按预定先后次序执行，解决与时间相关的错误。

线程不同步，产生的现象就是数据混乱：

1. 资源共享（系统层面）
2. 调度随机（系统层面）
3. 缺乏必要的同步机制（用户层面，我们可以在这里做限制）

多个控制流访问同一共享资源时，必须同步。

# 互斥锁/互斥量

## 概述

当多个线程并发的访问同一个共享资源的时候，可能导致数据异常。可以通过互斥锁确保一个共享资源每次只能被一个线程访问。

互斥锁属于重量级锁，因为存在用户态和内核态的切换。

**互斥锁的特点：**

1. 建议锁；
2. 不会限制资源访问；
3. 线程不按规则访问数据依然成功，会出现数据混乱。

互斥量通常用于有一个访问资源，多个访问流的时候。

## 初始化锁

有三种方法初始化互斥锁：

1. 动态初始化，用完后销毁

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

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

1. 静态初始化

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

注：锁一定是全局变量锁

## 函数

### pthread\_mutex\_init

POSIX互斥锁被声明为具有pthread\_mutex\_t数据类型的变量。

**参数：**

互斥锁mutex（传出）

互斥锁属性attr（传入），默认属性NULL

restrict关键字，默认修改该指针指向内存中内容的操作，只能通过本指针完成

**功能：**初始化一把互斥锁，将互斥量结构体初始化为1。

1、静态初始化：如果互斥锁变量时静态分配的，那么我们可以把它树池化成常值PTHREAD\_MUTEX\_INITIALIZER，例如：

static pthread\_mutex\_t lock= PTHREAD\_MUTEX\_INITILIZER；

2、动态初始化：如果互斥锁是动态分配的，或者分配在共享内存区中，那么我们必须在运行之时通过调用pthread\_mutex\_init函数来初始化它。

### pthread\_mutex\_lock

功能：加锁（即将互斥量-1）

如果尝试给一个已由另外某个线程锁住的互斥锁上锁，那么pthread\_mutex\_lock将阻塞到该互斥锁解锁为止。Pthread\_mutex\_trylock是对应的非阻塞函数，如果该互斥锁已锁住，它就返回一个EBUSY错误。

如果多个线程阻塞在等待同一个互斥锁上，在解锁时同步函数（互斥锁、读写锁、信号量）将唤醒优先级最高的被阻塞线程（不同线程可被赋予不同的优先级）。

### pthread\_mutex\_trylock

功能：非阻塞加锁。

### pthread\_mutex\_timelock

### pthread\_mutex\_unlock

功能：解锁（可以理解为mutex--操作）

说明：同时会将阻塞在该锁上的所有线程全部唤醒。

### pthread\_mutex\_destroy

## 使用

锁的“粒度”（临界区）：

1. 锁粒度越小越好；
2. 在访问共享资源前加锁，访问结束后立即解锁。

## 死锁

产生原因：

1. 对同一互斥量重复加锁
2. 持有锁A的线程1请求锁B，持有锁B的线程请求锁A

避免方法：

1. 保证资源的获取顺序，要求每个线程获取资源的顺序一致；
2. 当得不到所有所需要的资源时，放弃已经获得的资源，等待。

# 读写锁

## 概述

pthread读写锁把对共享资源的访问者分为读者和写者，读者只对共享资源进行读访问，写者只对共享资源进行写操作。在互斥机制，读者和写者都需要独立独占互斥量以独占共享资源，在读写锁机制下，允许同时有多个读者访问共享资源，只有写者才需要独占资源（**读共享，写独占**）。

相比互斥锁，读写机制由于允许多个读者同时访问共享资源，进一步提高了多线程的并发度。

读写锁机制：

写者：写者使用写锁，如果当前没有读者，也没有其他写者，写者立即获取写锁；否则写者将等待，直到没有读者和写者。

读者：读者使用读锁时，如果当前没有写者，读者立即获取读锁；否则读者等待，直到没有写者。

读写锁特性：

1. **同一时刻只有一个线程可以获得写锁（锁只有一把）**，同一时刻可以有多个线程获得读锁。
2. 读写锁处于写锁状态时，所有试图对读写锁加锁的线程，不管是读者试图加读锁，还是写者试图加写锁，都会被阻塞。

即，写模式加锁，在解锁前，所有对该锁加锁的线程都会被阻塞。

1. 读写锁处于读锁状态时，有写者试图加写锁时，之后的其他线程的读锁请求会被阻塞，以避免写者长时间的不写锁。

即，读模式加锁，如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞；既有试图以写模式加锁的线程，也有试图以读模式加锁的线程，优先满足写锁。

## 记录上锁

记录上锁是读写锁的一种拓展，可以用于有亲缘关系或者无亲缘关系的进程之间共享某个文件的读写。被锁住的文件通过其文件描述符访问，执行上锁操作的函数时fcntl，这种类型的锁通常在内核中维护，其属主是由属主的进程ID标识的。这意味着这些锁用于不同进程间的上锁，而不是用于同一进程内不同线程的上锁。

记录上锁作为读写锁的拓展，是进程间而非线程间的锁。

## 函数

### pthread\_rwlock\_init

### pthread\_ rwlock \_lock

### pthread\_ rwlock \_timelock

### pthread\_ rwlock \_rdlock

### pthread\_ rwlock \_wrlock

### pthread\_ rwlock \_unlock

### pthread\_ rwlock \_destroy

## 使用

适用于读操作远大于写操作的场景。

# 条件锁/条件变量

## 概述

条件锁就是所谓的条件变量，某一个线程因为某个条件未满足时可以使用条件变量使该程序处于阻塞状态（主要用于访问公共资源）。一旦条件满足以“信号量”的方式唤醒一个因为条件而被阻塞的线程。

**最为常见的就是在线程池中，起初没有任何时刻任务队列为空，此时线程池中的线程以为“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。

与互斥锁相比较，互斥锁用于上锁，条件变量（条件锁）用于等待。

与读写锁相比较，更加先进，因为可以使用条件变量判断这个共享区域是否存在数据是否可以访问，不需要读写锁再去判断了，减少不必要的竞争。

条件变量的特性：

1. 条件变量不是锁
2. 可以造成线程阻塞
3. 与mutex配合使用

## 函数

### pthread\_cond\_t

pthread\_cond\_t condition = PTHREAD\_COND\_INITILIZER；//静态初始化条件变量

### pthread\_cond\_init

动态初始化条件变量（相当于new创建对象）

### pthread\_cond\_destroy

### pthread\_cond\_wait

功能：

1. 阻塞等待条件变量cond
2. 释放已经掌握的互斥量mutex
3. 被唤醒时重新申请获取互斥锁

注：1、2步为原子操作。

以阻塞的方式执行，如果某个线程的程序执行了该函数，那么这个线程就会以阻塞的方式等待，直到收到pthread\_cond\_signal或者pthread\_cond\_broadcase函数发来的信号而被唤醒。

其语义相当于：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。

while(当前线程中条件不成立){

pthread\_cond\_wait(cond, mutex);//解锁，其他线程使条件成立发送信号，加锁

}

//对进程之间的共享资源进行操作

pthread\_cond\_unlock();//释放互斥锁

### pthread\_cond\_timedwait

以阻塞方式等待，如果时间到了条件还没有满足还是会结束。

### pthread\_cond\_broadcast

唤醒全部阻塞在条件变量上的线程（一般不推荐使用，使用pthread\_cond\_signal）。

### pthread\_cond\_signal

在另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。

# 文件加锁

应用程序经常需要读取文件中的数据，修改数据，然后回写数据，如果某一时刻只有一

个进程执行文件操作不存在任何问题，但是如果同时多个进程执行操作就会出现问题。这就需要多进程之间实现同步，可以使用信号量来完成所需的同步，但通常文件锁更好一些，因为内核能够将锁和文件关联起来。

## flock

flock[file lock]，建议性锁，不具备强制性。一个进程使用flock将文件锁住，另一个进程可以直接操作正在被锁的文件，修改文件中的数据。原因在于flock只是用于检测文件是否被加锁，针对文件已经被加锁，另一个进程写入数据的情况，内核不会阻止这个进程的写入操作，也就是建议性锁的内核处理策略。

flock系统调用是在整个文件中加锁，通过对传入的fd所指向的文件进行操作，然后再通过operation参数所设置的值来确定做什么样的操作，operation的值如下：

|  |  |
| --- | --- |
| 值 | 描述 |
| LOCK\_SH  LOCK\_EX | 在fd引用的文件上放置一把共享锁  在fd引用的文件上放置一把互斥锁 |
| LOCK\_UN  LOCK\_NB | 解锁fd引用的文件  发起一个非阻塞的请求 |

在默认情况下，如果另外一个进程已经持有了文件上的一个不兼容的锁，那么flock会阻塞。如果需要防止这种情况的出现，可以在operation参数中对这些值取OR（|）操作。在这种情况下，如果一个进程已经有了一个文件上的一个不兼容锁，那么flock就会阻塞。相反，它会返回-1，并将errno设置成EWOULDBLOCK。

任意数量的进程可同时持有一个文件上的共享锁，但是任意时刻只能有一个进程能够持有一个文件上的互斥锁（这有点类似读写锁）。

flock放置的锁有如下限制：

1. 只能对整个文件进行加锁。这种粗粒度的加锁会限制协作进程间的并发，假如存在多个进程，其中各个进程都想同时访问同一个文件的不同部分。
2. 通过flock只能放置劝告式锁。
3. 很多NFS实现不识别flock放置的锁。

注：在默认情况下，文件锁是劝告式的，这表示一个进程可以简单地忽略另一个进程在文件中放置的锁。要使得劝告式加锁模型能够正常工作，所有访问文件的进程都必须要配合，即在执行文件IO之前先放置一把锁。

## fcntl

原型：int flock(int fd, int operation);

fcntl函数覆盖了flock的功能，且提供了比flock更加强大的功能，但是在某些应用中仍然使用flock函数，并且在继承和释放锁方面的一些语义中flock与fcntl还是有所不同的。

## ftruncate

原型：int ftruncate(int fd, off\_t length);

功能：改变文件大小

说明：ftruncate会将参数fd指定的文件大小改为参数length指定的大小。如果原来的文件大小比参数length大，则超过的部分被删除。

# 信号量