# 线程同步

## 概述

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。

通过同步可以防止进程执行诸如同时更新一块共享内存或者同时更新文件的同一个数据块之类的操作。如果没有同步，那么这种同时更新的操作可能会导致应用程序产生错误的结果。

## 工具

UNIX系统提供了下列同步工具：

1. 信号量：一个信号量是一个由内核维护的整数，其值永远不会小于0。一个进程可以增加或减小一个信号量的值。如果一个进程试图将信号量的值减小到小于0，那么内核会阻塞该操作直至信号量的值增长到允许执行该操作的程度。信号量的含义是由应用程序来确定的。一个进程减小一个信号量（如从1到0）是为了预约对某些共享资源的独占访问，在完成了资源的使用之后可以增加信号量来释放共享资源以供其他进程使用。最常用的信号量是二元信号量——一个值只能是0或1的信号量，但处理一类共享资源拥有多个实例的应用程序需要使用最大值等于共享资源数量的信号量。
2. 文件锁：文件锁是设计用来协调操作同一个文件的多个进程的动作的一种同步方法。他也可以用来协调对其他共享资源的访问。文件锁分为两类：读（共享）锁和写（互斥）锁。任意进程都可以持有同一文件（或一个文件的某段区域）的读锁，但当一个进程持有了一个文件（或文件区域）的写锁之后，其他进程将无法获取该文件（或文件区域）上的读锁和写锁。Linux通过fork()和fcntl()系统调用来提供文件加锁工具。flock()系统调用提供了一种简单的加锁机制，允许进程将一个共享或互斥锁加到整个文件上。由于功能有限，现在已经很少使用flock()这个加锁工具了。fcntl()系统调用提供了记录加锁，允许进程在同一文件的不同区域加上多个读锁和写锁。
3. 互斥体和条件变量

在执行进程间同步时通常需要根据功能需求来选择工具。当协调对文件的访问时文件记录加锁通常是最佳的选择，而对于协调对其他共享资源的访问来讲，信号量通常是更加的选择。

# 原子变量

参考：<https://www.cnblogs.com/mmxingye/p/16183864.html>

## 概述

多线程间是通过互斥锁与条件变量来保证共享数据的同步的，互斥锁主要是针对过程加锁来实现对共享资源的排他性访问。很多时候，对共享资源的访问主要是对某一数据结构的读写操作，如果数据结构本身就带有排他性访问的特性，也就相当于该数据结构自带一个细粒度的锁，对该数据结构的并发访问就能更加简单高效，这就是C++11提供的原子数据类型< atomic >。下面解释两个概念：

原子操作：顾名思义就是不可分割的操作，该操作只存在未开始和已完成两种状态，不存在中间状态；

原子类型：原子库中定义的数据类型，对这些类型的所有操作都是原子的，包括通过原子类模板std::atomic< T >实例化的数据类型，也都是支持原子操作的。

C++11 提供了一个原子类型 std::atomic，通过这个原子类型管理的内部变量就可以称之为原子变量，我们可以给原子类型指定 bool、char、int、long、指针等类型作为模板参数（不支持浮点类型和复合类型）。

## 原理

### 一致性模型

### 内存顺序

原子操作中的内存访问模型

原子操作保证了对数据的访问只有未开始和已完成两种状态，不会访问到中间状态，但我们访问数据一般是需要特定顺序的，比如想读取写入后的最新数据，原子操作函数是支持控制读写顺序的，即带有一个数据同步内存模型参数std::memory\_order，用于对同一时间的读写操作进行排序。C++11定义的6种类型如下：

memory\_order\_relaxed: 宽松操作，没有同步或顺序制约，仅对此操作要求原子性；

memory\_order\_release & memory\_order\_acquire: 两个线程A&B，A线程Release后，B线程Acquire能保证一定读到的是最新被修改过的值；这种模型更强大的地方在于它能保证发生在A-Release前的所有写操作，在B-Acquire后都能读到最新值；

memory\_order\_release & memory\_order\_consume: 上一个模型的同步是针对所有对象的，这种模型只针对依赖于该操作涉及的对象：比如这个操作发生在变量a上，而s = a + b; 那s依赖于a，但b不依赖于a; 当然这里也有循环依赖的问题，例如：t = s + 1，因为s依赖于a，那t其实也是依赖于a的；

memory\_order\_seq\_cst: 顺序一致性模型，这是C++11原子操作的默认模型；大概行为为对每一个变量都进行Release-Acquire操作，当然这也是一个最慢的同步模型；

内存访问模型属于比较底层的控制接口，如果对编译原理和CPU指令执行过程不了解的话，容易引入bug。内存模型不是本章重点，这里不再展开介绍，后续的代码都使用默认的顺序一致性模型或比较稳妥的Release-Acquire模型，如果想了解更多，可以参考链接： C++11 Memory Order

## 原子变量与锁

由于原子操作是通过指令提供的支持，因此它的性能相比锁和消息传递会好很多。相比较于锁而言，原子类型不需要开发者处理加锁和释放锁的问题，同时支持修改，读取等操作，还具备较高的并发性能，几乎所有的语言都支持原子类型。

可以看出原子类型是无锁类型，但是无锁不代表无需等待，因为原子类型内部使用了CAS循环，当大量的冲突发生时，该等待还是得等待！但是总归比锁要好。

CAS全称是Compare and swap，它通过一条指令读取指定的内存地址，然后判断其中的值是否等于给定的前置值，如果相等，则将其修改为新的值TAS为 test and set。

### 使用原子类型替代互斥锁编程

为便于比较，直接基于前篇文章：线程同步之互斥锁中的示例程序进行修改，用原子库取代互斥库的代码如下：

//atomic1.cpp 使用原子库取代互斥库实现线程同步

#include <chrono>

#include <atomic>

#include <thread>

#include <iostream>

std::chrono::milliseconds interval(100);

std::atomic<bool> readyFlag(false); //原子布尔类型，取代互斥量

std::atomic<int> job\_shared(0); //两个线程都能修改'job\_shared',将该变量特化为原子类型

int job\_exclusive = 0; //只有一个线程能修改'job\_exclusive',不需要保护

//此线程只能修改 'job\_shared'

void job\_1()

{

std::this\_thread::sleep\_for(5 \* interval);

job\_shared.fetch\_add(1);

std::cout << "job\_1 shared (" << job\_shared.load() << ")\n";

readyFlag.store(true); //改变布尔标记状态为真

}

// 此线程能修改'job\_shared'和'job\_exclusive'

void job\_2()

{

while (true) {

//无限循环，直到可访问并修改'job\_shared'

if (readyFlag.load()) {

//判断布尔标记状态是否为真，为真则修改‘job\_shared’

job\_shared.fetch\_add(1);

std::cout << "job\_2 shared (" << job\_shared.load() << ")\n";

return;

} else {

//布尔标记为假,则修改'job\_exclusive'

++job\_exclusive;

std::cout << "job\_2 exclusive (" << job\_exclusive << ")\n";

std::this\_thread::sleep\_for(interval);

}

}

}

int main()

{

std::thread thread\_1(job\_1);

std::thread thread\_2(job\_2);

thread\_1.join();

thread\_2.join();

getchar();

return 0;

}

由示例程序可以看出，原子布尔类型可以实现互斥锁的部分功能，但在使用条件变量condition variable时，仍然需要mutex保护对condition variable的消费，即使condition variable是一个atomic object。

### 使用原子类型实现自旋锁

自旋锁（spinlock）与互斥锁(mutex)类似，在任一时刻最多只能有一个持有者，但如果资源已被占用，互斥锁会让资源申请者进入睡眠状态，而自旋锁不会引起调用者睡眠，会一直循环判断该锁是否成功获取。自旋锁是专为防止多处理器并发而引入的一种锁，它在内核中大量应用于中断处理等部分（对于单处理器来说，防止中断处理中的并发可简单采用关闭中断的方式，即在标志寄存器中关闭/打开中断标志位，不需要自旋锁）。

对于多核处理器来说，检测到锁可用与设置锁状态两个动作需要实现为一个原子操作，如果分为两个原子操作，则可能一个线程在获得锁后设置锁前被其余线程抢到该锁，导致执行错误。这就需要原子库提供对原子变量“读-修改-写(Read-Modify-Write)”的原子操作，上文原子类型支持的操作中就提供了RMW(Read-Modify-Write)原子操作，比如a.exchange(val)与a.compare\_exchange(expected,desired)。

标准库还专门提供了一个原子布尔类型std::atomic\_flag，不同于所有 std::atomic 的特化，它保证是免锁的，不提供load()与store(val)操作，但提供了test\_and\_set()与clear()操作，其中test\_and\_set()就是支持RMW的原子操作，可用std::atomic\_flag实现自旋锁的功能，代码如下：

//atomic2.cpp 使用原子布尔类型实现自旋锁的功能

#include <thread>

#include <vector>

#include <iostream>

#include <atomic>

std::atomic\_flag lock = ATOMIC\_FLAG\_INIT; //初始化原子布尔类型

void f(int n)

{

for (int cnt = 0; cnt < 100; ++cnt) {

while (lock.test\_and\_set(std::memory\_order\_acquire)) // 获得锁

; // 自旋

std::cout << n << " thread Output: " << cnt << '\n';

lock.clear(std::memory\_order\_release); // 释放锁

}

}

int main()

{

std::vector<std::thread> v; //实例化一个元素类型为std::thread的向量

for (int n = 0; n < 10; ++n) {

v.emplace\_back(f, n); //以参数(f,n)为初值的元素放到向量末尾，相当于启动新线程f(n)

}

for (auto& t : v) {

//遍历向量v中的元素，基于范围的for循环，auto&自动推导变量类型并引用指针指向的内容

t.join(); //阻塞主线程直至子线程执行完毕

}

getchar();

return 0;

}

自旋锁除了使用atomic\_flag的TAS(Test And Set)原子操作实现外，还可以使用普通的原子类型std::atomic实现：其中a.exchange(val)是支持TAS原子操作的，a.compare\_exchange(expected,desired)是支持CAS(Compare And Swap)原子操作的，感兴趣可以自己实现出来。其中CAS原子操作是无锁编程的主要实现手段。

## 无锁编程

### CAS原子操作实现无锁编程

CAS原子操作主要是通过函数a.compare\_exchange(expected,desired)实现的，其语义为“我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少”，CAS算法的实现伪码如下：

bool compare\_exchange\_strong(T& expected, T desired)

{

if( this->load() == expected ) {

this->store(desired);

return true;

} else {

expected = this->load();

return false;

}

}

下面尝试实现一个无锁栈，代码如下：

//atomic3.cpp 使用CAS操作实现一个无锁栈

#include <atomic>

#include <iostream>

template<typename T>

class lock\_free\_stack

{

private:

struct node

{

T data;

node\* next;

node(const T& data) : data(data), next(nullptr) {

}

};

std::atomic<node\*> head;

public:

lock\_free\_stack(): head(nullptr) {

}

void push(const T& data)

{

node\* new\_node = new node(data);

do{

new\_node->next = head.load(); //将 head 的当前值放入new\_node->next

}while(!head.compare\_exchange\_strong(new\_node->next, new\_node));

// 如果新元素new\_node的next和栈顶head一样，证明在你之前没人操作它，使用新元素替换栈顶退出即可；

// 如果不一样，证明在你之前已经有人操作它，栈顶已发生改变，该函数会自动更新新元素的next值为改变后的栈顶；

// 然后继续循环检测直到状态1成立退出；

}

T pop()

{

node\* node;

do{

node = head.load();

}while (node && !head.compare\_exchange\_strong(node, node->next));

if(node)

return node->data;

}

};

int main()

{

lock\_free\_stack<int> s;

s.push(1);

s.push(2);

s.push(3);

std::cout << s.pop() << std::endl;

std::cout << s.pop() << std::endl;

getchar();

return 0;

}

程序注释中已经解释的很清楚了，在将数据压栈前，先通过比较原子类型head与新元素的next指向对象是否相等来判断head是否已被其他线程修改，根据判断结果选择是继续操作还是更新期望，而这一切都是在一个原子操作中完成的，保证了在不使用锁的情况下实现共享数据的并发同步。

CAS 看起来很厉害，但也有缺点，最著名的就是 ABA 问题，假设一个变量 A ，修改为 B之后又修改为 A，CAS 的机制是无法察觉的，但实际上已经被修改过了。如果在基本类型上是没有问题的，但是如果是引用类型呢？这个对象中有多个变量，我怎么知道有没有被改过？聪明的你一定想到了，加个版本号啊。每次修改就检查版本号，如果版本号变了，说明改过，就算你还是 A，也不行。

上面的例子节点指针也属于引用类型，自然也存在ABA问题，比如在线程2执行pop操作，将A,B都删掉，然后创建一个新元素push进去，因为操作系统的内存分配机制会重复使用之前释放的内存，恰好push进去的内存地址和A一样，我们记为A’，这时候切换到线程1，CAS操作检查到A没变化成功将B设为栈顶，但B是一个已经被释放的内存块。该问题的解决方案就是上面说的通过打标签标识A和A’为不同的指针，具体实现代码读者可以尝试实现。

# 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。

## 互斥锁与读写锁

**互斥锁和读写锁区别：**

互斥锁：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。

读写锁：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。

注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

**互斥锁和读写锁的区别：**

1）读写锁区分读者和写者，而互斥锁不区分

2）互斥锁同一时间只允许一个线程访问该对象，无论读写；读写锁同一时间内只允许一个写者，但是允许多个读者同时读对象。

## 原理

## 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。

3、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。

## 原理

底层实现原理（手写代码）

## 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是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

## 原理

## 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

## 概述

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。

尽管文件加锁通常会与文件I/O一起使用，但也可以将其作为一项更通用的同步技术来使用。协作进程可以约定一个进程对整个文件或一个文件区域进行加锁表示对一些共享资源（如一个共享内存区域）而非文件本身的访问。

**混合使用加锁和stdio函数**

由于stdio库会在用户空间进行缓冲，因此在混合使用stdio函数与加锁技术时需要特别小心。这里的问题是一个输入缓冲器在被加锁之前可能会被填满或者一个输出缓冲器在锁被删除之后可能会被刷新。要避免这些问题则可以采用下面的方法：

1. 使用read()和write()（以及相关的系统调用）取代stdio库来执行文件I/O；
2. 在对文件加锁之后立即刷新stdio流，并且在释放锁之前立即再次刷新这个流；
3. 使用setbuf()（或类似函数）来禁止stdio缓冲，当然这可能会牺牲一些效率。

**劝告式和强制式加锁**

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上放置的锁。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I/O时需要在文件上放置一把锁。与之对应的是，强制式加锁系统会强制一个进程在执行I/O时需要遵从其他进程持有的锁。

## 原理

## 函数

flock()对整个文件加锁，fcntl()对一个文件区域加锁。

flock()系统调用源自BSD，而fcntl()则源自System V。

### flock

#### 背景

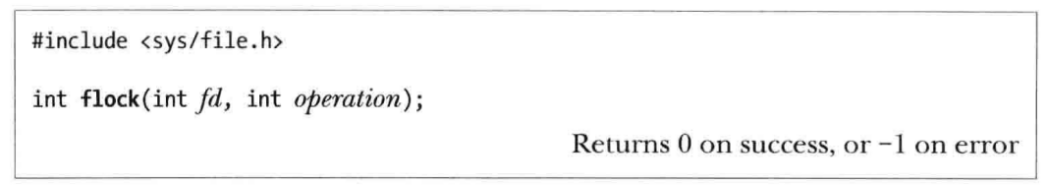
我们常常需要通过crontab部署某个脚本运行某些定时任务，但在实际的过程中，一旦处理不好可能导致在同一时刻出现脚本的多个运行副本，比如crontab的调度是每5 分钟运行一次脚本，如果运行的脚本能够在5分钟内处理完任务并退出，则没问题，同一时刻最多只有一个脚本在运行，但如果脚本如果无法在5分钟内处理完任务并退出，则意味着在下一个5分钟，脚本又会被crontab调起来，出现了同一时刻有2个甚至更多的脚本事例在运行，对应一些依赖关系，需要串行处理的脚本来说，可能会导致数据相互覆盖等问题。

为了解决这个问题，脚本必须能够自我保护，即在启动之前判断是否已经有脚本启动了，若当前已经有脚本启动了，则虽然crontab调度到该脚本，但脚本自身会马上退出。

解决问题的思路很多，其中一种笔者觉得比较简便的是：使用文件锁。

#### 概述

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。

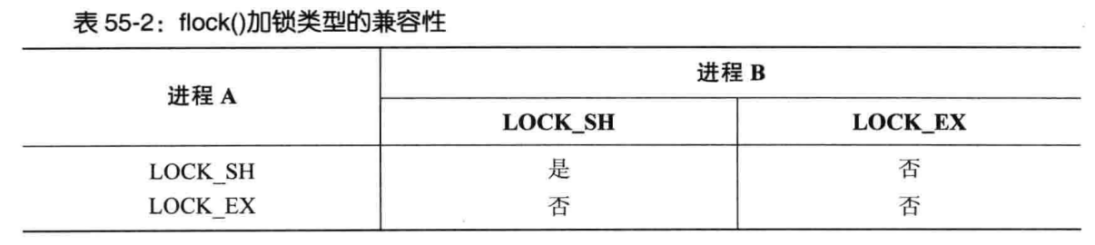


flock系统调用是在整个文件中加锁，通过对传入的fd所指向的文件进行操作，然后再通过operation参数所设置的值来确定做什么样的操作，operation的值如下：

|  |  |
| --- | --- |
| 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）。



不管一个进程在文件上的访问模式是什么（读、写、或读写），它都可以在文件上放置一把共享锁或互斥锁。

通过再次调用flock()并在operation参数中指定恰当的值可以将一个既有共享锁转换成一个互斥锁（反之亦然）。将一个共享锁转换成一个互斥锁，在另一个进程持有了文件上的共享锁时会被阻塞，除非同时指定了LOCK\_NB标记。

#### 限制

**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。

#### 命令

参考：

<https://blog.csdn.net/tenfyguo/article/details/51012527>

在linux中，提供了flock命令，具体格式如下：

1、flock [-sxon] [-w timeout] lockfile [-c]command...

2、flock [-sxun] [-w timeout] fd

参数说明如下：

-s为共享锁，在定向为某文件的FD上设置共享锁而未释放锁的时间内，其他进程试图在定向为此文件的FD上设置独占锁的请求失败，而其他进程试图在定向为此文件的FD上设置共享锁的请求会成功。

-e为独占或排他锁，在定向为某文件的FD上设置独占锁而未释放锁的时间内，其他进程试图在定向为此文件的FD上设置共享锁或独占锁都会失败。只要未设置-s参数，此参数默认被设置。

-u手动解锁，一般情况不必须，当FD关闭时，系统会自动解锁，此参数用于脚本命令一部分需要异步执行，一部分可以同步执行的情况。

-n为非阻塞模式，当试图设置锁失败，采用非阻塞模式，直接返回1，并继续执行下面语句。

-w设置阻塞超时，当超过设置的秒数，就跳出阻塞，返回1，并继续执行下面语句。

-o必须是使用第一种格式时才可用，表示当执行command前关闭设置锁的FD，以使command的子进程不保持锁。

-c执行其后的comand。

flock是对于整个文件的建议性锁。也就是说，如果一个进程在一个文件（inode）上放了锁，那么其它进程是可以知道的。它的第一个参数是文件描述符，在此文件描述符关闭时，锁会自动释放。而当进程终止时，所有的文件描述符均会被关闭。于是，很多时候就不用考虑解锁的事情。

#### 应用

flock命令最大的用途就是实现对crontab任务的串行化。在crontab任务中，有可能出现某个任务的执行时间超过了crontab中为此任务设定的执行周期，这就导致了当前的任务实例还未执行完成，crontab又启动了同一任务的另外一个实例，这通常不是用户所期望的行为。极端情况下，如果某个任务执行异常一直未返回，crontab不会处理这种情形，会继续启动新的实例，而新的实例很可能又会异常，这样就导致crontab对同一任务不断的启动新的实例，最终导致系统内存被耗尽，影响到整个操作系统的运行。**为了防止crontab任务出现多实例的情况，可以使用flock命令将crontab中任务的周期性执行串行化**。

在将corntab中任务串行化时，flock通过对一个中间文件加文件锁来间接实现同一时刻某个任务只有一个实例运行的目标。对应的 crontab 中任务的描述形式如下：

\* \* \* \* \* flock -xn /tmp/mytest.lock -c 'php /home/fdipzone/php/test.php'

这里的定时任务是每分钟执行一次，但是任务中并未直接执行目标命令‘php /home/fdipzone/php/test.php’，而是将命令作为flock的-c选项的参数。flock命令中，-x 表示对文件加上排他锁，-n表示文件使用非阻塞模式，-c选项指明加锁成功后要执行的命令。因而上面flock命令的整体含义就是：如果对/tmp/mytest.lock文件（如果文件不存在， flock 命令会自动创建）加锁成功就执行后面的命令，否则不执行。

假如上面php命令要执行2分钟，而crontab任务每分钟就会执行一次，如果当前php命令正在执行，说明flock已经锁定了文件/tmp/mytest.lock，crontab到了再次执行任务的时间时，会发现文件已经被加了锁。由于设置的是非阻塞模式的文件锁，flock会在加锁失败时直接返回，并不执行php命令，这样就使php命令得以顺序执行，crontab任务就不会出现同时有两个实例运行的情况了，达到了串行化目的。

### fcntl

使用fcntl()能够在一个文件的任意部分放置一把锁，这个文件部分既可以是一个字节，也可以是整个文件。这种形式的加锁通常被称为记录加锁，但这种称谓是不恰当的，因为UNIX系统的文件是一个字节序列，并不存在记录边界的概念，文件记录的概念只存在应用程序中。

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。

#### 死锁

#### 锁的限制和性能

#### 锁继承和释放的语义

#### 锁定饿死和排队加锁请求

当多个进程必须要等待以便能够在当前被锁住的区域上放置一把锁时，一系列问题就出现了。

一个进程是否能够等待以便在由一系列进程放置读锁的同一块区域上放置一把写锁并因此可能会导致饿死？在Linux上，一系列的读锁确实能够导致一个被阻塞的写锁饿死，甚至会被无限地饿死。

当两个或多个进程等待放置一把锁时，是否存在一些规则来确定在锁可用时哪个进程会获取锁？例如，锁清秋是否满足FIFO顺序，规则跟每个进程请求的锁的类型是否有关系？在Linux上的规则如下：

1. 排队的锁清秋被准予的顺序是不确定的。如果多个进程正在等待加锁，那么它们被满足的顺序取决于进程的调度；
2. 写者并不比读者拥有更高的优先权，反之亦然。

### 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。

### 强制加锁

目前为止介绍的都是劝告式锁，这意味着一个进程可以自由地忽略fcntl()（或flock()）的使用或者简单地在文件上执行I/O。内核不会阻止进程的这种行为。在使用劝告式锁的时候，应用程序的设计者需要：

1. 为文件设置合适的所有权（或组所有权）以及权限以防止非协作进程执行文件I/O；
2. 通过在执行I/O之前获取恰当的锁来确保构成应用程序的进程相互协作。

与其他很多UNIX实现一样，Linux也允许fcntl()记录锁是强制式的。这表示需要对每个文件I/O操作进行检查以判断其他进程在执行I/O所在的文件区域上是否持有任何不兼容的锁。

劝告式模式加锁有时候被称为自由加锁（discretionary locking），而强制式加锁有时候则被称为强制模式加锁（enforcement-mode locking）。

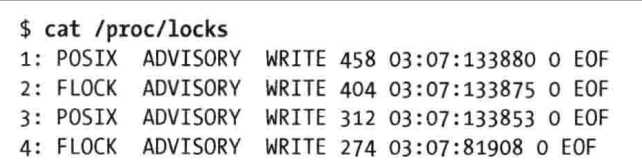
为了在Linux上使用强制式加锁就必须要在包含待加锁文件的文件系统以及每个待加锁的文件上启用这一项功能。通过在挂载文件系统时使用（Linux特有的）-o mand选项能够在该文件系统上启用强制式加锁。

**强制式加锁对文件I/O操作的影响**

**强制式加锁警告**

## /proc/locks文件

通过检查Linux特有的/proc/locks文件中的内容能够查看系统中当前存在的锁。



/proc/locks文件显示了使用flock()和fcntl()创建的锁的相关信息。每把锁的8个字段的含义如下（从左向右）：

1. 锁在该文件上所有锁中的序号
2. 锁的类型，其中FLOCK表示flock()创建的锁，POSIX表示fcntl()创建的锁
3. 锁的模式，其值是ADVISORY或MANDATORY
4. 锁的类型，其值是READ或WRITE（对应fcntl()的共享锁和互斥锁）
5. 持有锁的进程的进程ID
6. 三个用冒号分割的数字，它们标识出了锁所属的文件，这些数字是文件所处的文件系统的主要和次要设备号，后面跟着文件的inode号。
7. 锁的起始字节。对于flock()锁来讲，其值永远是0。
8. 锁的结尾字节，其中EOF表示锁延伸到文件的结尾（即对于fcntl()创建的锁来讲是将l\_len指定为0）。对于flock()锁来讲，这一列的值永远是EOF。

使用/proc/locks中的信息能够找出哪个进程持有了哪个文件上的锁。

## 老式加锁技术

# 信号量

## 概述

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。

一个信号量是一个由内核维护的整数，其值被限制为大于或等于0。在一个信号量上可以执行各种操作（即系统调用），包括：

1. 将信号量设置成一个绝对值；
2. 在信号量当前值的基础上加上一个数量；
3. 在信号量当前值的基础上减去一个数量；
4. 等待信号量的值等于0。

上面的操作中后面两个可能会导致调用进程阻塞。当减小一个信号量的值时，内核会将所有试图将信号量降低到0之下的操作阻塞。类似的，如果信号量的当前值不为0，那么等待信号量的值等于0的调用进程将会发生阻塞。不管是何种情况，调用进程会一直保持阻塞直到其他一些进程将信号量的值修改为一个允许这些操作继续向前的值，在那个时刻内核会唤醒被阻塞的进程。

从信号量的描述来看，其功能类似互斥锁，但是二者存在诸多区别：

**互斥量和信号量的区别：**

1、**互斥量用于线程的互斥，信号量用于线程的同步**。

2、互斥量值只能为0/1，信号量值可以为非负整数。

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**。

**注：**相同线程使用互斥锁没问题，但是多线程中不可以，需要互斥锁+条件变量或者使用信号量。

## System V信号量

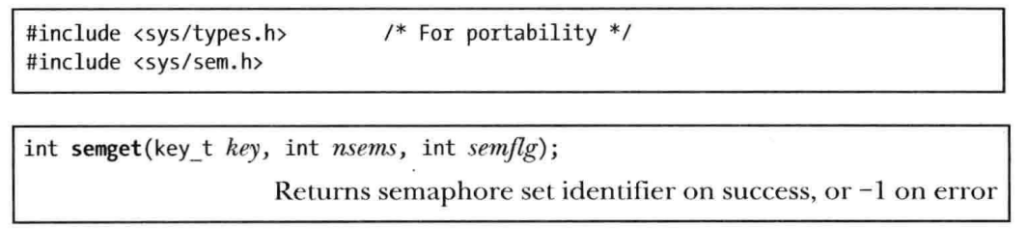
System V信号量并不是用来在进程间传输数据的，相反，它们用来同步进程的动作。信号量的一个常见用途是同步对一块共享内存的访问以防止出现一个进程在访问共享内存的同时另一个进程更新这块内存的情况。

### 概述

### 操作

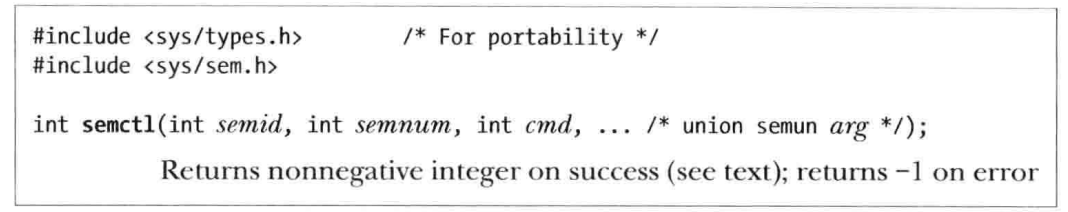
#### semget

semget()系统调用创建一个新信号集或打开/获取一个既有集合的标识符。



#### semctl

semctl()系统调用在一个信号量集或集合中的单个信号量上执行各种控制操作。



#### 信号量关联数据结构

#### 信号量初始化

#### 信号量操作

##### semop

##### semtimedop

#### 多个阻塞信号量操作的处理

#### 信号量撤销值

### 实现一个二元信号量协议

### 信号量限制

### 缺点

System V信号量存在很多缺点与消息队列的缺点是一样的，包括：

1. 信号量是通过标识符而不是大多数UNIX I/O和IPC所采用的的文件描述符来引用的。这使得执行诸如同时等待一个信号量和文件描述符的输入之类的操作就会变得比较困难。
2. 使用键而不是文件名来标识信号量增加了额外的编程复杂度。
3. 创建和初始化信号量需要使用单独的系统调用意味着在一些情况下必须要做一些额外的编程工作开防止在初始化一个信号量时出现竞争条件。
4. 内核不会维护引用一个信号集的进程数量。这就给确定何时删除一个信号集增加了难度并且难以确保一个不再使用的信号集会被删除。
5. System V提供的编程接口过于复杂。在通常情况下，一个程序只会操作一个信号量。同时操作集合中多个信号量的能力有时候是多余的。
6. 信号量的操作存在诸多限制。这些限制是可以配置的，但是如果一个应用程序超出了默认限制的范围，那么在安装应用程序时就需要完成额外的工作了。

## Posix信号量

### 概述

### 命名信号量

### 操作

**semget**

创建信号量集

#incldue <sys/sem.h>

int semget(key\_t key,int nsems,int flag);

返回值：成功返回信号量集ID，出错返回-1。

flag

IPC\_CREAT,IPC\_EXCL,权限组合

**semctl**

信号集控制

#include <sys/sem.h>

int semctl(int semid,int semnum,int cmd,.../\*union semun arg\*/)

union semum{

int val;

struct semid\_ds buf;

usigned short \*array;

}

通过cmd参数设定对信号量要执行的操作:

IPC\_STAT 获取信号量集的属性

IPC\_SET 设置信号量集的属性

IPC\_RMID 删除信号量集

GETVAL 返回semnum信号量的值

SETVAL 设置semnum信号量的值

GETTALL 获取所有信号量的值

SETALL 设置所有信号量的初始值

**semop**

信号量集的操作

#include <sys/sem.h>

int semop(int semid,struct sembuf \*semop,size\_t nops);

返回：成功返回0，出错返回-1。

struct sembuf{

unsigned short sem\_num; /\*member in set\*/

short sem\_op; /\*operation(negative,0,positive)\*/

short sem\_flg; /\*IPC\_NOWAIT,SEM\_UNDO\*/

}

基于基本信号量的up和down操作(PV操作)

可用于进程间的互斥和同步

Posix定义：

<https://blog.csdn.net/anonymalias/article/details/9219945>

**sem\_init**

初始化也叫建立（create），原型如下：

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

**sem\_wait**

等信号也叫挂起（suspend），原型如下：

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

**sem\_trywait**

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

**sem\_timedwait**

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

**sem\_post**

给信号（signal）或者发信号（psot），原型如下：

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

**sem\_close**

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

**sem\_unlink**

int sem\_unlink(const char \*name);

**sem\_destroy**

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

清理。

### 未命名信号量

### 对比

#### Posix信号量与System V信号量

#### Posix信号量与Pthreads互斥体

### 信号量限制