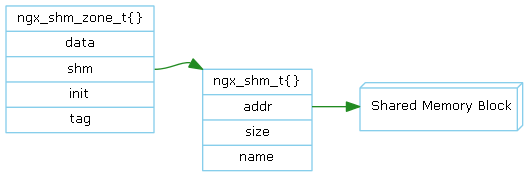
Nginx InterProcess Communication

# 概述

暂无

# Share Memory



共享内存是最高效的进程内通信方式。

## ngx\_shm\_zone\_t

管理一块共享内存。

* data

用户自定义数据。

* init

共享内存初始化函数。

* tag

通常是用户的标枳。

## ngx\_shm\_t

管理由mmap()或shmget()创建的共享内存空间。

* addr

共享内存的起始地址。

* size

共享内存大少。

* name

共享内存名称。

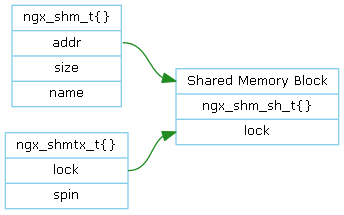
# Lock

Nginx的进程间锁分为原子锁和文件锁这二种实现。编译时根据系统特性选择其中一种。

在用户空间进程间锁实现的原理很简单，通过控制各个进程都有访问的对象来实现进程的互斥。

原子锁控制的对象是共享内存；文件锁控制的对象是文件句柄。

## 原子锁

ngx\_shmtx\_t实现了一个原子锁，其中lock指向一个共享内存中的一个int变量。各个进程操作这个共享变量lock来实现加锁和解锁等操作。

* ngx\_shmtx\_trylock：尝试加锁，如果加锁成功则返回true，否则返回false而不等待。
* ngx\_shmtx\_lock：加锁，如果已被其它进程加锁，则等待。
* ngx\_shmtx\_unlock：解锁，只可解由本进程加锁的锁。
* ngx\_shmtx\_force\_unlock：强制解锁，即可以解由其它进程加锁的锁。

### 非原子

我们先实现一个ngx\_shmtx\_trylock：

1. bool ngx\_shmt\_trylock(ngx\_shmtx\_t \*tx) {
2. if (\*tx->lock == 0) {
3. \*tx->lock = my\_pid;
4. return true;
5. }
6. return false;
7. }

这个实现是有问题的。假设初始状态该锁没有被任何进程加锁，即lock为0。

1. 进程A执行完2)在执行3)之前被切换到进程B。
2. 进程B执行2)，此时lock仍然为0，所以接着执行3)将lock设为pid(B)，接着返回true。
3. 进程B加锁成功。切换进程到进程A。
4. 进程A执行3)将lock设为pid(A)，接着返回true。
5. 进程A加锁成功。

在上述的情况下，进程A和进程B都成功加锁。出错的原因是lock判断2)和赋值3)这二个操作之间插入其它操作，导致赋值的假设条件被破坏。

解决方案是使判断和赋值这二个操作不可打断的，原子性的。

### 原子

bool ngx\_atomic\_cmp\_set(\*lock, old, set) 的实现或者是利用系统库的实现，或者是利用cmpxchgl汇编指令实现。

* ngx\_shmtx\_trylock

1. bool ngx\_shmt\_trylock(ngx\_shmtx\_t \*tx) {
2. return \*tx == 0 && ngx\_atomic\_cmp\_set(tx->lock, 0, my\_pid);
3. }

* ngx\_shmtx\_lock

1. void ngx\_shmt\_lock(ngx\_shmtx\_t \*tx) {
2. while (true) {
3. if (ngx\_shmt\_trylock(tx)) return;
4. if (ncpu > 1) { // 单CPU不能自旋
5. for (n = 1, 2, 4 , 8 , 16, …, 2048) { // 自旋由短到长
6. for (i = 1, 2, 3, 4, 5, …, n) { // 自旋操作
7. ngx\_cpu\_pause();
8. }
9. if (ngx\_shmt\_trylock(tx)) return;
10. }
11. }
12. yield(); // 自动放弃CPU，挂入进程执行队列的末尾
13. }
14. }

* ngx\_shmtx\_unlock

1. void ngx\_shmt\_unlock(ngx\_shmtx\_t \*tx) {
2. ngx\_atomic\_cmp\_set(tx->lock, my\_pid, 0);
3. }

* ngx\_shmtx\_force\_unlock

1. void ngx\_shmt\_force\_unlock(ngx\_shmtx\_t \*tx, any\_pid) {
2. ngx\_atomic\_cmp\_set(tx->lock, any\_pid, 0);
3. }

### 信号量

当长时间自旋后，仍然加锁不成功。上述的策略是自动放弃CPU，并挂入进程执行队列的末尾。

这里的策略是加入信号量等待队列，并从进程执行队列移除。当其它进程解锁时，释放一个信号，并唤醒所有等待的进程。

## 文件锁

利用系统调用fcntl()实现锁操作。

ngx\_trylock\_fd()、ngx\_lock\_fd()、ngx\_unlock\_fd()

## 死锁恢复

当某个Worker对成功加锁后，在没有解锁之前，Worker进程异常退出。这种情况就会造成死锁。Nginx是如何处理这种情况的呢？

Worker异常退出，Master会收到SIGCHLD信号，在SIGCHLD信号处理函数中，调用waitpid()得到Worker的pid，用这个pid调用

ngx\_shmtx\_force\_unlock()解锁。死锁得以恢复。

# Socketpair

TODO

# Prototype

暂无

# Latest revision

https://github.com/lingjf/nginx\_analyse/blob/master/doc/