Qemu之rfifolock（可重入FIFO锁）

我们知道，C语言编程下的大多数锁都是不可重入的，通俗的说就说无法递归获取，这类锁也俗称非递归锁。对非递归锁，在多进程（线程）的使用下，要十分谨慎。稍有不慎，就会造成死锁，同一个进程（线程）在获取锁的情况下，再次尝试获取该锁，便会产生死锁现象。

然而，在现实场景中，可重入锁在多线程同步编程中具有十分重要的作用，这也是很多高级编程语言都提供了“可重入锁”的原因，例如Java的ReentrantLock。

Qemu在实现AIO上下文时，就需要这样一把可重入锁，而且需要一把具有FIFO特性的可重入锁。所谓FIFO锁，这里的含义就是当多线程竞争该锁时，按照“先到先得”的顺序获取该锁，当然前提是该锁没有任何其它线程持有。至于为什么需要这样的锁，以后在AIOContext中介绍。

C语言中，没有现成的可重入锁，更不具有FIFO特性的锁，Qemu就索性利用pthread的mutex和条件变量造了这样一个轮子。其原理主要基于

（1）互斥特性：对不同线程的加锁采用条件变量进行同步。

（2）可重入机制：采用引用计数原理（也是大多数可重入锁的原理），对同一线程，第一次获取该锁时，引用计数为1。后面每一次获取该锁，都递加引用计数。

（3）FIFO：采用ticket机制，每个尝试获取锁的线程都会分得一个ticket id，无法获取锁的线程“持票”等待。ticket id是递加的，按照ticket id的“自小到大”的顺序，依次分配锁资源。

下面是锁结构体的设计：

typedef struct {

QemuMutex lock; /\* protects all fields \*/

/\* FIFO order \*/

unsigned int head; /\* active ticket number \*/

unsigned int tail; /\* waiting ticket number \*/

QemuCond cond; /\* used to wait for our ticket number \*/

/\* Nesting \*/

QemuThread owner\_thread; /\* thread that currently has ownership \*/

unsigned int nesting; /\* amount of nesting levels \*/

/\* Contention callback \*/

void (\*cb)(void \*); /\* called when thread must wait, with ->lock

\* held so it may not recursively lock/unlock

\*/

void \*cb\_opaque;

} RFifoLock;

* 其中lock用于互斥多线程对加锁函数和解锁函数的访问，由于加锁和解锁主要操作了锁内部结构体数据，因而，实际起到了保护结构体内数据的作用。
* nesting对同一线程多次获取该锁进行引用计数。
* head和tail分别用于ticket发放和持票准入，每次发放ticket后tail都会递增（对于统一线程，只有第一次进入时递增）。当线程让出锁资源后，head递增，开始“叫号”通知其它线程竞争从等待中醒来。当然，只允许ticket id与head相同的线程获取锁资源。

Rfifiolock的操作函数如下：

void rfifolock\_init(RFifoLock \*r, void (\*cb)(void \*), void \*opaque);

void rfifolock\_destroy(RFifoLock \*r);

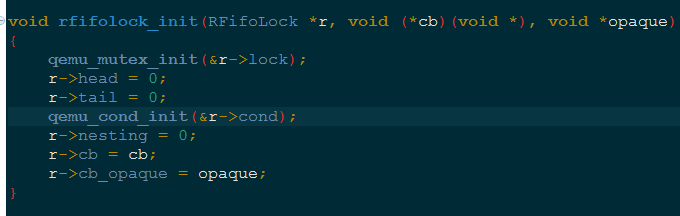
void rfifolock\_lock(RFifoLock \*r);

void rfifolock\_unlock(RFifoLock \*r);

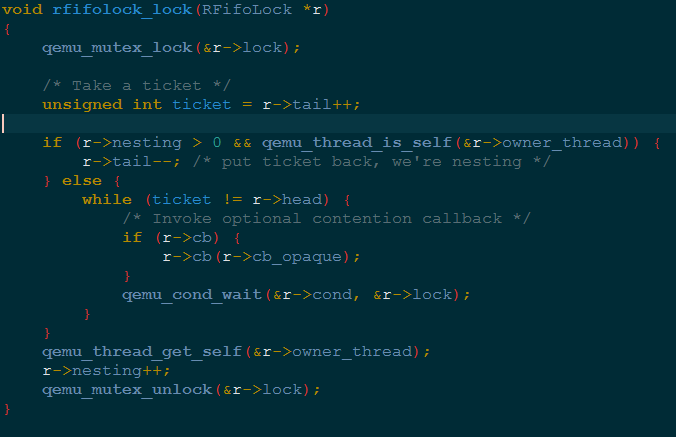
锁的实现也相对比较简单：

1. 锁初始化

锁初始化相对比较简单，初始化head、tail、nesting均为0。

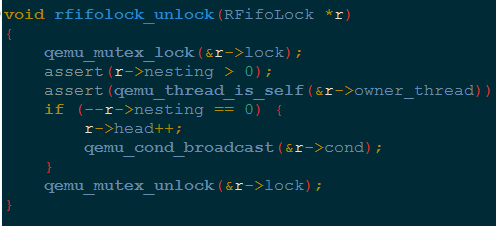


1. 加锁



可以看到，一旦线程进入加锁函数，就会分配一个ticket，同时递加tail。当发现锁被持有且是当前线程时，就会回退tail（递减）。一旦获取锁成果，递加引用计数。如果锁被其它线程持有，这里就会“持票”等待，直到自己的ticket排上号。在进入条件等待之前，rfifolock允许该线程执行一个初始化锁资源时注册的回调。

1. 解锁



解锁过程，首先递减应用计数，当计数为0时，递加head，开始“叫号”，并通知其它线程。其它线程中被“叫”到的线程成功获取锁资源。