从 flock 引起的一个 bug 谈起之 flock 与 fcntl lock 的差异

提到了 flock，不提 fcntl 这个锁有点不想话，毕竟 fcntl 这个锁才是更常见的一把锁。咱也不能拈轻怕重，逮着软柿子可劲捏，今天我们比较下这两种类型锁的异同，并从 kernel 实现的层面，来讲讲为啥表现不同，准备好了没，LET GO！

上一篇博文讲到了 flock 系统调用那把锁是 FL_FLOCK 类型的锁，而 fcntl 创建的锁是遵循 POSIX 标准的，所以称为 FL_POSIX 类型的锁。上一篇博文做了一个实验，进程 A 首先申请 FL_FLOCK 类型的锁一把，然后 fork 出来子进程 B，此时在启动同一个可执行程序，启动进程 C，C 也会首先申请 FL_FLOCK 锁，当然了，都是对同一个文件加排他锁。我们发现，在 A 进程推出后，C 进程依然申请不到这把锁，直到 B 进程推出，C 进程才持有了这把锁。我们得到结论，fork 出来的子进程，不但拷贝所有父进程的所有打开的文件（当然了同一个 struct file，struct file 引用计数+1），同时也持有了父进程申请的 FL_FLOCK 类型的锁。这就是上篇博文的结论，当然我们没有从代码层面分析这种锁的继承性的缘由。没关系，这是我们这篇博文涉及的东西。

应用层 fcntl

首先说，我不太喜欢 fcntl 这个函数，因为这个函数有点瑞士军刀的意思，方便是方便了，但是这厮干的事儿有点多，不符合一个接口只干一件事，并把事情干好的 UNIX 哲学。不喜欢归不喜欢，但是咱也得从了。西游记说，世界尚不完美，经书怎能苛求完美。是啊，世界尚不完美，我们也没办法苛求太多。

flock 系统调用本质是给文件上锁，它比较死心眼，一锁就是整个文件，要求 flock 系统调用给某文件前 40 个字节上锁，不好意思，flock 他老人家太老了，这么细的活儿干不了。但是 fcntl 不同了，它属于江湖晚辈，做的就比较细致了，他能够精确打击，让它给文件的某一个字节加锁，他都能办得到。OK ，闲言少叙看接口。

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );

struct flock {
   ...
   short l_type;    /* Type of lock: F_RDLCK,
```

```
                    F_WRLCK, F_UNLCK */
        short l_whence;  /* How to interpret l_start:
                    SEEK_SET, SEEK_CUR, SEEK_END */
        off_t l_start;   /* Starting offset for lock */
        off_t l_len;     /* Number of bytes to lock */
        pid_t l_pid;     /* PID of process blocking our lock
                    (F_GETLK only) */
        ...
    };
```

文件记录加锁相关的 cmd 分三种（fcntl 这厮还有其他于加锁无关的 cmd）：

1. F_SETLK

   申请锁（读锁 F_RDLCK，写锁 F_WRLCK）或者释放所（F_UNLCK），但是如果 kernel 无法将锁授予本进程（被其他进程抢了先，占了锁），不傻等，返回 error

2. F_SETLKW

   和 F_SETLK 几乎一样，唯一的区别，这厮是个死心眼的主儿，申请不到，就傻等。

3. F_GETLK

   这个接口是获取锁的相关信息： 这个接口会修改我们传入的 struct flock。

   如果探测了一番，发现根本就没有进程对该文件指定数据段加锁，那么了 `l_type` 会被修改成 `F_UNLCK`
   如果有进程持有了锁，那么了 `l_pid` 会返回持锁进程的 `PID`

   参考 UNIX 网络编程卷 2 进程间通信，将这个接口封装了下，让接口变得好用些。

```
#include <unistd.h>
#include <fcntl.h>

static int lock_reg(int fd,int cmd,int type,off_t offset,int
whence,off_t len)
{
    struct flock lock;
    lock.l_type = type;
    lock.l_start = offset;
    lock.l_whence = whence;
    lock.l_len = len;

    return (fcntl(fd,cmd,&lock));
}
```

```
static pid_t lock_test(int fd,int type,off_t offset,int whence,off_t
len)
{
    struct flock lock;

    lock.l_type = type;
    lock.l_start = offset;
    lock.l_whence = whence;
    lock.l_len = len;

    if(fcntl(fd,F_GETLK,&lock) == -1)
    {
        return -1;
    }
    if(lock.l_type = F_UNLCK)
        return 0;
    return lock.l_pid;
}

int read_lock(int fd,off_t offset,int whence,off_t len)
{
    return lock_reg(fd,F_SETLKW,F_RDLCK,offset,whence,len);
}

int read_lock_try(int fd,off_t offset,int whence,off_t len)
{
    return lock_reg(fd,F_SETLK,F_RDLCK,offset,whence,len);
}

int write_lock(int fd,off_t offset,int whence,off_t len)
{
    return lock_reg(fd,F_SETLKW,F_WRLCK,offset,whence,len);
}

int write_lock_try(int fd,off_t offset,int whence,off_t len)
{
    return lock_reg(fd,F_SETLK,F_WRLCK,offset,whence,len);
}

int unlock(int fd,off_t offset, int whence,off_t len)
{
    return lock_reg(fd,F_SETLK,F_UNLCK,offset,whence,len);
}

int is_read_lockable(int fd, off_t offset,int whence,off_t len)
```

```
{
    return !lock_test(fd,F_RDLCK,offset,whence,len);
}

int is_write_lockable(int fd, off_t offset,int whence,off_t len)
{
    return !lock_test(fd,F_WRLCK,offset,whence,len);
}
```

下面是头文件 rwlock.h

```
#ifndef __RWLOCK_H__
#define __RWLOCK_H__

int read_lock(int fd,off_t offset,int whence,off_t len);
int read_lock_try(int fd,off_t offset,int whence,off_t len);
int write_lock(int fd,off_t offset,int whence,off_t len);
int write_lock_try(int fd,off_t offset,int whence,off_t len);
int unlock(int fd,off_t offset, int whence,off_t len);
int is_read_lockable(int fd, off_t offset,int whence,off_t len);
int is_write_lockable(int fd, off_t offset,int whence,off_t len);

#endif
```

现在万事具备了，我们可以写我们的测试程序了。实验内容同 flock 系统调用一样，A 进程申请锁，然后 fork 出 B 进程，然后 C 进程申请锁。过一会 A 进程死去，B 仍然活着，看下 C 能否申请到锁。

FL_POSIX 锁父子进程继承性实验

测试程序和上一篇一样，只不过使用我们上面提到的 write_lock,而不是 flock 函数。

```
#include<stdio.h>
       #include <stdlib.h>
       #include <sys/types.h>
       #include <unistd.h>
       #include <sys/file.h>
       #include <errno.h>
       #include <string.h>
       #include <time.h>
       #include <fcntl.h>
       #include "rwlock.h"
```

```
int main()
{
    char buf[128];
    time_t ltime;
    int fd = open("./tmp.txt",O_RDWR|O_APPEND);
    if(fd < 0)
    {
        fprintf(stderr,"open failed %s\n",strerror(errno));
        return -1;
    }

    int ret = write_lock(fd,0,SEEK_SET,0);
    if(ret)
    {
        fprintf(stderr,"fcntl  failed for father\n");
        return -2;
    }
    else
    {
        time(&ltime);
        fprintf(stderr,"%s    I got the lock\n",ctime_r(&ltime,buf));
    }

    ret = fork();
    if(ret == 0)
    {
        time(&ltime);
        fprintf(stdout,"%s  I am the son process,pid is %d,ppid =
%d\n",ctime_r(&ltime,buf),getpid(),getppid());
        write(fd,"write by son\n",32);
        sleep(100);
        time(&ltime);
        fprintf(stdout,"%s    son exit\n",ctime_r(&ltime,buf));
    }
    else if(ret > 0)
    {
        time(&ltime);
        fprintf(stdout,"%s    I am the father process,pid is
%d\n",ctime_r(&ltime,buf),getpid());
        write(fd,"write by father\n",32);
        sleep(50);
        close(fd);
        time(&ltime);
        fprintf(stdout, "%s    father exit\n",ctime_r(&ltime,buf));
        return 0;
```

```
        }
        else
        {
            fprintf(stderr, "error happened in fork\n");
            return -3;
        }

    }
```

A 进程持有锁后，持续 50 秒，B 进程作为子进程持续 100s，C 进程在 A 推出前创建，我们观察 A 死去后，C 能否立刻获取 FL_POSIX 类型的锁 如果可以，表明锁没有继承性，子进程 B 并不持有锁。 如果不可以，非要等到 B 死去后才能申请到，那么说明父进程的锁，被继承到了子进程。

其实细心的筒子看到 struct flock 的 l_pid 大概就能猜到，锁记录了进程 ID，精确归某进程所有，就不会被继承到子进程，我们验证之。

```
 pid_t l_pid;     /* PID of process blocking our lock
                        (F_GETLK only) */
```

看下输出结果：

```
root@manu:~/code/c/self/flock# ./fcntl_test
Sun Feb 10 16:14:45 2013
   I got the lock
Sun Feb 10 16:14:45 2013
   I am the father process,pid is 6475
Sun Feb 10 16:14:45 2013
  I am the son process,pid is 6476,ppid = 6475
Sun Feb 10 16:15:35 2013
   father exit
root@manu:~/code/c/self/flock# Sun Feb 10 16:16:25 2013
   son exit

root@manu:~/code/c/self/flock#
root@manu:~/code/c/self/flock# ./fcntl_test
Sun Feb 10 16:15:35 2013
   I got the lock
Sun Feb 10 16:15:35 2013
   I am the father process,pid is 6477
Sun Feb 10 16:15:35 2013
  I am the son process,pid is 6482,ppid = 6477
Sun Feb 10 16:16:25 2013
   father exit
```

```
root@manu:~/code/c/self/flock#
```

结论： 父进程 A 退出后，进程 C 就获取到了 FL_POSIX 锁，所以子进程不会继承 FL_POSIX 类型的锁。这和 FL_FLOCK 类型的锁是不同的。 WHY！！！

kernel 分析原因

实验到了这个份上，我们就需要从内核代码分析原因了。所有的代码都在 fs/locks.c,大家感兴趣可以细细參详，我只讲继承性差异的原因，为啥 FL_FLOCK 锁可以被继承，但是 FL_POSIX 只精确的属于某进程，不会被子进程继承。

注意了我们都没有主动 UN_LOCK，flock 我们没有调用 LOCK_UN，fcntl 没有调用 F_UNLCK，锁的释放在 close 的时候去释放。 先说 flock：flock 在内核调用 locks_delete_flock 来释放锁，同时唤醒沉睡在这把锁上的其他进程。 close--->filp_close--------->fput 注意 fput：

```
void fput(struct file *file)
{
    if (atomic_long_dec_and_test(&file->f_count)) {
        struct task_struct *task = current;
        file_sb_list_del(file);
        if (unlikely(in_interrupt() || task->flags & PF_KTHREAD)) {
            unsigned long flags;
            spin_lock_irqsave(&delayed_fput_lock, flags);
            list_add(&file->f_u.fu_list, &delayed_fput_list);
            schedule_work(&delayed_fput_work);
            spin_unlock_irqrestore(&delayed_fput_lock, flags);
            return;
        }
        init_task_work(&file->f_u.fu_rcuhead, ____fput);
        task_work_add(task, &file->f_u.fu_rcuhead, true);
    }
}
```

注意了，条件 atomic_long_dec_and_test(&file->f_count)，由于父子进程，那么父进程退出引用计数减 1,仍然不会调用到里面的内容，而我们释放 FL_FLOCK 类型锁是在____fput，脉络如下：

```
____fput----->  __fput----->locks_remove_flock---------->locks_delete_flock
```

那么大家也就明白了，正是因为引用计数并没有减少到 1,所以父进程的退出，并不会调用 locks_delete_flock 来唤醒等待这把锁的进程。

对于 fcntl 实现的 FL_POSIX 类型的锁，则不同，最终的释放会走到__posix_lock_file,当然了，调用 F_UNLCK 最终也会调到此处。当进程推出，尝试关闭进程打开的文件的时候，遵循这样的脉络

```
   close----->filp_close----->locks_remove_posix---->vfs_lock_file-----
>posix_lock_file----->__posix_lock_file
```

当然走的是解锁的分支。这条路径上，没有什么条件阻止走到真正解锁的地方，所以，当进程推出的时候，FL_POSIX 类型的锁就被释放了。

观察 tool

我们如何观测文件锁的状况呢？比如，我们知道某文件被锁，如何知道是那个进程锁的这个文件呢？procfs 提供了信息：

```
root@manu:~/code/c/self/flock# ./test
Sun Feb 10 20:51:06 2013
   I got the lock
Sun Feb 10 20:51:06 2013
   I am the father process,pid is 9941
Sun Feb 10 20:51:06 2013
  I am the son process,pid is 9942,ppid = 9941

root@manu:~/code/c/self/flock# ./fcntl_test
Sun Feb 10 20:51:14 2013
   I got the lock
Sun Feb 10 20:51:14 2013
   I am the father process,pid is 9943
Sun Feb 10 20:51:14 2013
  I am the son process,pid is 9944,ppid = 9943

  root@manu:~/code/c/classical/linux-3.6.7/fs# cat /proc/locks
1: POSIX  ADVISORY  WRITE 9943 08:06:2359759 0 EOF
2: FLOCK  ADVISORY  WRITE 9941 08:06:2359759 0 EOF
```

我们可以看到/proc/locks 下面有锁的信息：我现在分别叙述下含义：

1. POSIX FLOCK 这个比较明确，就是哪个类型的锁。flock 系统调用产生的是 FLOCK，fcntl 调用 F_SETLK，F_SETLKW 产生的是 POSIX 类型

2. ADVISORY 表明是劝告锁

3. WRITE 顾名思义，是写锁，还有读锁

4. 9943 是持有锁的进程 ID。当然对于 flock 这种类型的锁，会出现进程已经退出的状况。

5. 08：06：2359759 表示的对应磁盘文件的所在设备的主设备好，次设备号，还有文件对应的 inode number。

6. 0 表示的是所的其实位置

7. EOF 表示的是结束位置。 这两个字段对 fcntl 类型比较有用，对 flock 来是总是 0 和 EOF。

   看下/home 所在的分区主设备号就是 8,次设备号就是 6，而我们操作的文件的 inode，就是 2359759

```
/dev/sda6      77993572 47528652 26558672   65% /home

  8        6   78125000 sda6

root@manu:~/code/c/self/flock# ls -li tmp.txt
2359759 -rw-r--r-- 1 manu root 2689  2月 10 20:51 tmp.txt
```

参考文献

1. 深入理解 linux 内核

2. linux 设备驱动程序（如何将锁的信息 show 出来，代码用了 seq_file，这个又能写一篇博文，唉太多了）
3. Manual