从 flock 引发的一个 bug 谈起（1）进程的文件描述符

引子

前两天我们 QA 发现了一个比较有意思的 bug，我细细分析一下，发现多个进程卡死在一个·配置文件上。简单的说，我们为了防止多个进程同时写同一个配置文件，将文件格式破坏，我们用了 flock，对于写打开，同时调用 flock 系统调用，LOCK_EX 方式。当然了由于持有锁，就必须临界区要小，写完之后，尽量释放，持有锁的期间不要有 time cost high 的操作，否则，会有其他进程获取不到文件锁，活活饿死。

这个 bug 比较有意思的地方是，大家都等锁的原因调用那个了一个 python 脚本，而这个脚本并不需要操作配置文件，仅仅是因为父进程 system 函数调用 python 脚本之前，没有关闭文件释放锁，导致 python 脚本很无辜的持有了这本锁，而 python 偏偏是个 time cost high 的操作，这就真是急中风偏偏遇到了慢郎中，外围一群进程焦急地等待这把锁，而 python 进程却占着毛坑不那啥，呵呵。

我们知道，linux 存在强制锁（mandatory lock）和劝告锁（advisory lock）。所谓强制锁，比较好理解，就是你家大门上的那把锁，最要命的是只有一把钥匙，只有一个进程可以操作。所谓劝告锁，本质是一种协议，你访问文件前，先检查锁，这时候锁才其作用，如果你不那么 kind，不管三七二十一，就要读写，那么劝告锁没有任何的作用。而遵守协议，读写前先检查锁的那些进程，叫做合作进程。我们代码用的是 flock 这种劝告锁。

Linux 实现了 POSIX 规定的基于 fcntl 系统调用文件加锁机制，同时 LINUX 还支持 BSD 变体的 flock 系统调用实现的劝告锁，当然 system V 变体的 lockf 也支持，大家可以自行查找手册。对于 fcntl 这个系统调用，大家可以阅读 Stevens 大神的 UNIX 网络编程卷 2 进程间通信，讲解的非常好。我的重点是 flock。

应用层

flock 的应用层接口如下

```
#include <sys/file.h>
int flock(int fd, int operation);
```

其中 fd 是系统调用 open 返回的文件描述符，operation 的选项有：

1. LOCK_SH ：共享锁
2. LOCK_EX ：排他锁或者独占锁
3. LOCK_UN :解锁。 事实上 Linux 内核也实现了 LOCK_MAND 选项，所然 manual 中没有提到。这种情况我们不讨论。 注意了，flock 系统调用实现的 FL_FLOCK 类型的锁，本质是一种劝告锁，只有多个进程之间遵循要读写，先调锁的协议，才会生效。遵循协议的进程叫合作进程。

下面看一段代码：

```
#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>

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

    int ret = flock(fd,LOCK_EX);
    if(ret)
    {
        fprintf(stderr,"flock 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;
   }

}
```

当然了，执行前，tmp.txt 是存在的。我们写打开了一个文件 tmp.txt，同时通过 flock 系统调用，申请了一把 FL_FLOCK 类型的锁然后 fork 了一个子进程。50 秒后，父进程退出，子进程变成孤儿，100 秒后，子进程退出。现在的问题是，父进程死去，子进程活着的期间，子进程是否持有这把锁？ 我们让事实说话。先启动一个./test 5 秒后启动另一个./test 这是第一个 test 所在的终端：

```
   root@manu:~/code/c/self/flock# ./test
   Wed Feb  6 23:53:29 2013
       I got the lock
   Wed Feb  6 23:53:29 2013
       I am the father process,pid is 5632
   Wed Feb  6 23:53:29 2013
     I am the son process,pid is 5633,ppid = 5632
   Wed Feb  6 23:54:19 2013
       father exit
   root@manu:~/code/c/self/flock# Wed Feb  6 23:55:09 2013
       son exit
```

这是第二个 test 所在的终端：

```
root@manu:~/code/c/self/flock#
root@manu:~/code/c/self/flock# ./test
Wed Feb  6 23:55:09 2013
   I got the lock
Wed Feb  6 23:55:09 2013
   I am the father process,pid is 5634
Wed Feb  6 23:55:09 2013
  I am the son process,pid is 5647,ppid = 5634
Wed Feb  6 23:55:59 2013
   father exit
root@manu:~/code/c/self/flock# Wed Feb  6 23:56:49 2013
   son exit
```

我们看到了，直到子进程退出，第二个启动的 test 的进程才申请到了这把 FL_FLOCK 锁。换言之，子进程会继承父进程的打开的所有文件，并且继承那把 FL_FLOCK 锁，哪怕他并不真正的操作这个文件。

BUT WHY ！！！

内核层之 fd 的分配

对于一个进程而言，我们知道有一个进程可以打开多个文件，ulimit -a 我们可以看到，默认最多打开 1024 个文件。 其中 STDIN，STDOUT，STDERR 是三个默认的，对应的文件描述符是 0 ，1 ，2.进程用 0 1 2 这种数字来表征对应的 FILE，当然他们是特殊的文件，对于打开的某真正的文件，那么可能对应的 fd 为 4 ，操作系统是如何根据这个 4 找到对应的文件的呢？这是我们这个小节需要解决的问题。

```
<sched.h>
struct task_struct {
   ...
   struct files_struct *files;
   ...
}
struct files_struct {
       atomic_t count;
       struct fdtable __rcu *fdt;
       struct fdtable fdtab;

       spinlock_t file_lock ____cacheline_aligned_in_smp;
       int next_fd;
       unsigned long close_on_exec_init[1];
       unsigned long open_fds_init[1];
       struct file __rcu * fd_array[NR_OPEN_DEFAULT];
```

```
};
struct fdtable {
        unsigned int max_fds;
        struct file __rcu **fd;      /* current fd array */
        unsigned long *close_on_exec;
        unsigned long *open_fds;
        struct rcu_head rcu;
        struct fdtable *next;
};
```

我给出了一陀数据结构，这些数据结构是进程和文件相关的数据结构，不要被吓倒，这部分的关系还是比较简单的，我们来看下，进程如何管理它打开的文件。真正的描述打开的文件信息的数据结构是:

```
struct file {
    ....
    atomic_long_t       f_count;
    unsigned int        f_flags;
    fmode_t         f_mode;
    loff_t          f_pos;
    ......
}
```

可以清楚的看到 struct file 才是正主，记录文件了 mode，当前读写的位置了，之类的信息。那么从进程，如何通过 fd，找到这个对应的 struct file 的呢？这就用到了我们前面提到的一坨数据结构了。fdtable 是距离 file 最近的数据结构，max_fds 是目前支持的最多文件数。fd 是一个 file 指针的指针，或者说 file 指针 数组的基地址，这个数组包含有 max_fds 个 file 指针。比如我们上面的 C 程序， tmp.txt 对应的文件描述符是 3,那么如何找到 3 对应的 struct file 呢？ 很简单

`fdtable->fd[3]`这个指针指向的就是 `tmp.txt` 对应的 `struct file`。

手握进程的 fdtable，就能找到某个数字对应的 struct file，那么如何从进程找到 fdtable 这个结构呢，也简单啊：

```
task_struct---->struct files_struct *files;  ---->struct fdtable __rcu *fdt;
```

task_struct 有一个 struct file_struct *类型的变量 files，而 file 又有一个 fdtable* 类型的成员变量 fdt，那么给个文件描述符的数字（即 open 的返回值如 3）我们就可以完成从 task_struct 找到对应的 struct file。

struct fdtable 里面的 close_on_exec 和 open_fds 是干啥的呢，这两个是位图，每个 bit 标记对应位置上的文件描述符有没有分配出去。比如我们打开 tmp.txt 的时候，就去 open_fds 里面去查找，发现 0 位置

出的 bit 为 1,表示文件描述符 0 已经分配出去了，（of course，这个 STDIN），1 位置出的 bit 值也是 1（of course，这是 STDOUT），一路找来，发现第一个不是 0 的 bit 是 3 位置处，OK，表示文件描述符 3 没有被占用，就将 3 作为 open 的返回值。

比较细心的看官可能要问了，这部所有的问题都解决了吗，为啥除了 struct files_struct 除了一个 struct fd_table 指针，还有一个 fd_table 实例呢？这不多余吗？，还有其中的 close_on_exec_init，open_fds_init 都是神马玩意儿啊，成员变量 fdt 不是已经把事情都办得妥妥当当的了吗？ 如下

```
struct files_struct {
       ...
       struct fdtable __rcu *fdt;
       struct fdtable fdtab;
       unsigned long close_on_exec_init[1];
       unsigned long open_fds_init[1];
       struct file __rcu * fd_array[NR_OPEN_DEFAULT];
}
```

其实 fdt 这个指针，一开始指向的是 fdtable 这个实例，fdt->open_fds 指向的是 open_fds_init，同理 fdt->fd_array 指向的就是 files_struct 中的 fd_array。简单的说就是我家里有 32 个酒杯，如果来的客人少，那么直接用家里的 32 个酒杯就行了，很不幸，过一会第 33 个客人来了，那么家里的酒杯就不够了，我就给玄武饭店打了个电话，请帮我预留 1024 个酒杯，我马上过去喝酒，然后我将所有的 32 个客人+新来的客人一起带到玄武饭店，用那里准备好的 1024 个酒杯，当然暂时用不了这么多，但是我已经预先占下了。既然已经换了喝酒的地点，为了防止后来的客人找不到，必须将地点改为玄武饭店，就好像 fdt 不再指向 files_struct 自带的 fdtable，而指向新分配的数据结构。

进程创建之初，总是指向自家的那 32 个酒杯。代码中如何体现呢？

```
do_fork---->copy_process---->copy_files---->dup_fd
```

在 dup_fd 中如如下代码：

```
newf->next_fd = 0;
new_fdt = &newf->fdtab;
new_fdt->max_fds = NR_OPEN_DEFAULT;
new_fdt->close_on_exec = newf->close_on_exec_init;
new_fdt->open_fds = newf->open_fds_init;
new_fdt->fd = &newf->fd_array[0];
new_fdt->next = NULL;
```

对于 fork 出来的子进程来说，要拷贝父进程打开的所有文件，就好像子进程也打开了文件一样：可以用

lsof 验证之：

```
root@manu:~/code/c/self/flock# ./test &
[1] 6226
root@manu:~/code/c/self/flock# Fri Feb  8 00:03:29 2013
   I got the lock
Fri Feb  8 00:03:29 2013
   I am the father process,pid is 6226
Fri Feb  8 00:03:29 2013
  I am the son process,pid is 6227,ppid = 6226
root@manu:~/code/c/self/flock# lsof -p 6226
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
...
test    6226 root    0u   CHR  136,2      0t0       5 /dev/pts/2
test    6226 root    1u   CHR  136,2      0t0       5 /dev/pts/2
test    6226 root    2u   CHR  136,2      0t0       5 /dev/pts/2
test    6226 root    3uW  REG    8,6      321 2359759
/home/manu/code/c/self/flock/tmp.txt

root@manu:~/code/c/self/flock# lsof -p 6226
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
...
test    6227 root    0u   CHR  136,2      0t0       5 /dev/pts/2
test    6227 root    1u   CHR  136,2      0t0       5 /dev/pts/2
test    6227 root    2u   CHR  136,2      0t0       5 /dev/pts/2
test    6227 root    3u   REG    8,6      321 2359759
/home/manu/code/c/self/flock/tmp.txt
```

对于父进程宴请的宾客个数（打开的文件）比较多，超过了 32 个，那么子进程会判断父进程准备的最大酒杯数，如果超过了 32,得，刚才白忙乎了，还的去申请新的酒杯。注意，只是 struct file 的指针被拷贝，父进程的 bitmap 被拷贝，真正的 struct file 这个比较大的结构体并没有被拷贝一份。

```
            old_fds = old_fdt->fd;
            new_fds = new_fdt->fd;

                                        /*拷贝位图信息*/
            memcpy(new_fdt->open_fds, old_fdt->open_fds, open_files / 8);
            memcpy(new_fdt->close_on_exec, old_fdt->close_on_exec,
open_files / 8);
            /*拷贝打开的file对应的struct file指针*/
            for (i = open_files; i != 0; i--) {
                struct file *f = *old_fds++;
                if (f) {
                    get_file(f);/*增加文件的引用计数，多了一个进程持有该strcut file
*/
```

```
            } else {
                /*
                 * The fd may be claimed in the fd bitmap but not yet
                 * instantiated in the files array if a sibling thread
                 * is partway through open().  So make sure that this
                 * fd is available to the new process.
                 */
                __clear_open_fd(open_files - i, new_fdt);
            }
            rcu_assign_pointer(*new_fds++, f);
        }
```

参考文献

1. 深入理解 Linux 内核
2. 深入 linux 内核架构
3. Linux kernel code 3.6.7