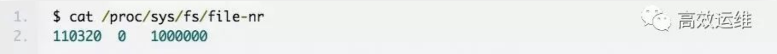
# Linux 文件句柄的这些技术内幕，只有 1% 的人知道

Original 王子勇 高效运维 2018-07-24

> 王子勇
>
> 腾讯高级业务运维工程师，有10年研发与运维工作经验。崇尚开源，喜欢钻研系统技术。微信号: jacuro

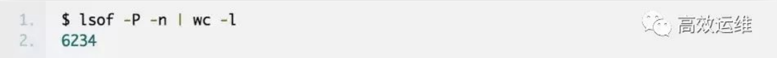
## 1. 缘起

某个月朗风清的晚上，正在公司对面的深大操场跑步，突然接到同事发来的消息，他发现某机器上的文件句柄使用量有十一万多个（下面输出中的第一个字段）



```
$ cat /proc/sys/fs/file-nr
110320  0   1000000
```

但是通过运维常用的lsof命令算了下，相差甚远。

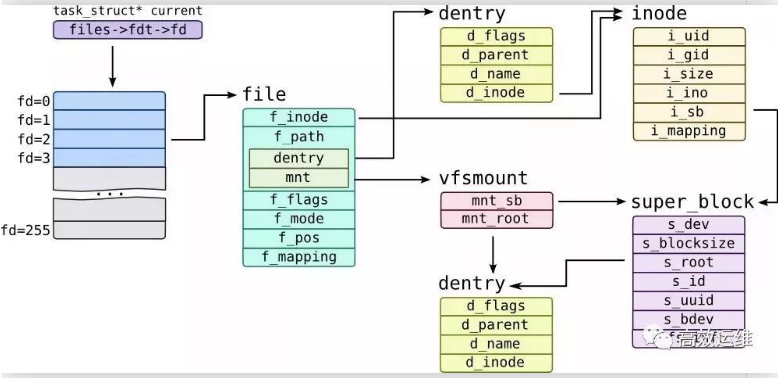


```
$ lsof -P -n | wc -l
6234
```

似乎很不科学，这里看到的数据不到1万个，剩下10多万的文件句柄哪里去了呢（系统完整性检查已排除黑客入侵可能性）

## 2. 文件描述符和文件句柄的故事

先看一张著名的图吧



这里我们先区分好两个概念：**文件描述符和文件句柄**

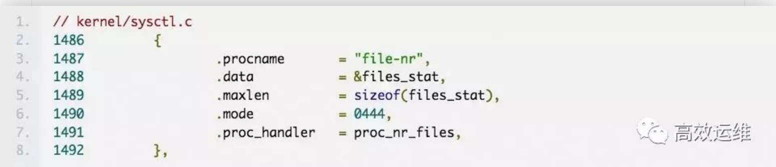
简单来说，每个进程都有一个打开的文件表（fdtable)。表中的每一项是struct file类型，包含了打开文件的一些属性比如偏移量，读写访问模式等，这是真正意义上的文件句柄。

文件描述符是一个整数。代表fdtable中的

索引位置（下标），指向具体的struct file（文件句柄）。

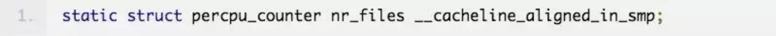
## 3. file-nr 文件里的值是文件描述符还是文件句柄?

顺着内核代码找一下：



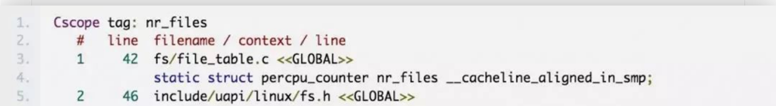
```
// kernel/sysctl.c
1486        {
1487                .procname       = "file-nr",
1488                .data           = &files_stat,
1489                .maxlen         = sizeof(files_stat),
1490                .mode           = 0444,
1491                .proc_handler   = proc_nr_files,
1492        },
```

可以看出file-nr指标是由proc_nr_files函数处理，该函数最终其实是读取了nr_files全局变量



```
static struct percpu_counter nr_files __cacheline_aligned_in_smp;
```

找下什么地方增加了这个值：



```
Cscope tag: nr_files
   #   line  filename / context / line
   1     42  fs/file_table.c <<GLOBAL>>
             static struct percpu_counter nr_files __cacheline_aligned_in_smp;
   2     46  include/uapi/linux/fs.h <<GLOBAL>>
```

```
          unsigned long nr_files;
3    54   fs/file_table.c <<file_free>>
          percpu_counter_dec(&nr_files);
4    64   fs/file_table.c <<get_nr_files>>
          return percpu_counter_read_positive(&nr_files);
5    83   fs/file_table.c <<proc_nr_files>>
          files_stat.nr_files = get_nr_files();
6   119   fs/file_table.c <<get_empty_filp>>
          if (percpu_counter_sum_positive(&nr_files) >= files_stat.max_files)
7   127   fs/file_table.c <<get_empty_filp>>
          percpu_counter_inc(&nr_files);
```

可以看到fs/file_table.c文件中第127行的get_empty_filp函数会增加这个值。那么这个函数是干什么的呢?

内核里面有注释“Find an unused file structure and return a pointer to it.”， 其实就是用来分配struct file的。

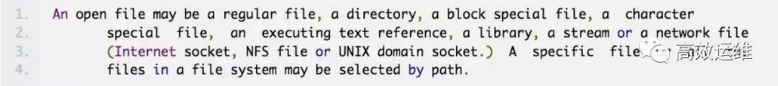
到此，相信你已经知道答案了。**file-nr文件里面的第一个字段代表的是内核分配的struct file的个数，也就是文件句柄个数，而不是文件描述符。**

## 4. 哪些地方会分配文件句柄?

知道文件句柄最终是通过get_empty_filp函数从filp cache中分配的之后，我们顺着函数调用链路简单梳理下，就能知道有哪些地方会分配文件句柄了：

- open系统调用打开文件（path_openat内核函数)
- 打开一个目录（dentry_open函数)
- 共享内存attach （do_shmat函数）
- socket套接字（sock_alloc_file函数）
- 管道（create_pipe_files函数）
- epoll/inotify/signalfd等功能用到的匿名inode文件系统（anon_inode_getfile函数)

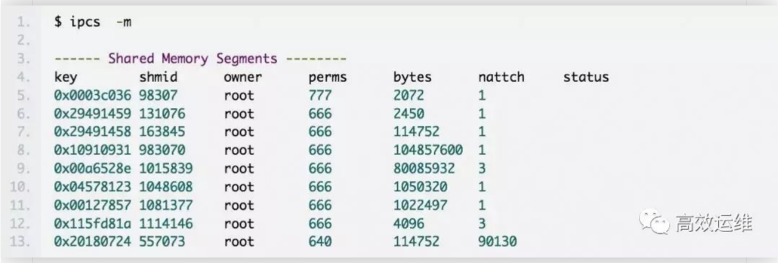
实际上，lsof的手册页也有部分描述：



```
An open file may be a regular file, a directory, a block special file, a  character
        special  file,  an  executing text reference, a library, a stream or a network file
        (Internet socket, NFS file or UNIX domain socket.)  A  specific  file
        files in a file system may be selected by path.
```

## 5. 找出元凶

有了上面的知识，我们除了看lsof的输出之外，再通过ipcs命令看一下共享内存的情况：



```
$ ipcs  -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x0003c036 98307      root       777        2072       1
0x29491459 131076     root       666        2450       1
0x29491458 163845     root       666        114752     1
0x10910931 983070     root       666        104857600  1
0x00a6528e 1015839    root       666        80085932   3
0x04578123 1048608    root       666        1050320    1
0x00127857 1081377    root       666        1022497    1
0x115fd81a 1114146    root       666        4096       3
0x20180724 557073     root       640        114752     90130
```

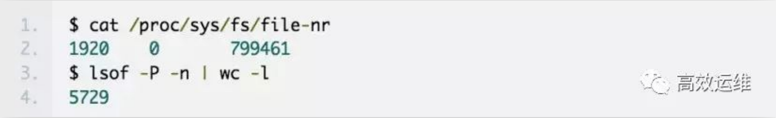
居然有部分共享内存段被attach了9多万次，数量级对得上，毫无疑问就是它了！

可能有些同学会有疑问，同一个进程居然可以重复attach同一段共享内存那么次？答案是可以的，内核无限制。显然，这样做逻辑

上是有问题的，对共享内存的正常操作，要么是attach后一直用，要么是attach用完之后就detach。

## 6. 排除了共享内存等的情况，我看到的file-nr值和lsof输出还是有很大差异?

上面的案例算是比较典型，然而，即便在一个比较“正常”的系统上，我们可以看到file-nr和lsof的输出还是有不小的差距的：



```
$ cat /proc/sys/fs/file-nr
1920    0       799461
$ lsof -P -n | wc -l
5729
```

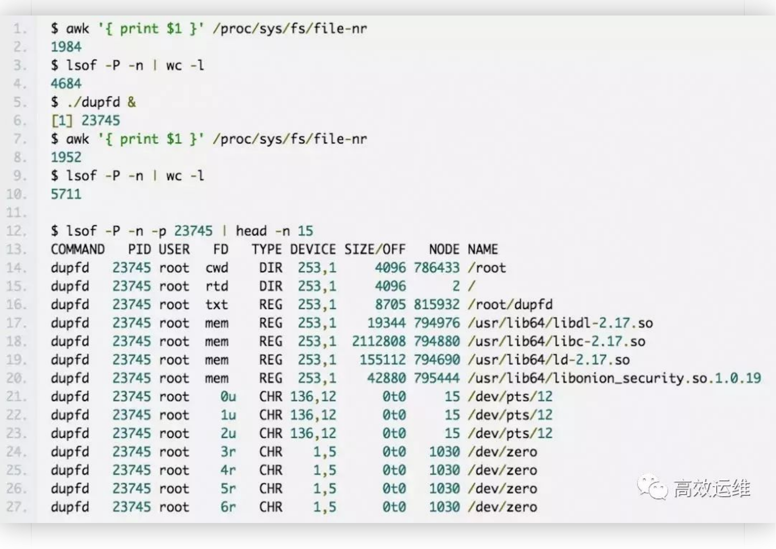
这里本质上是因为文件描述符和文件句柄是两个不同的东西：lsof在用户空间，主要还是从文件描述符的角度来看文件句柄。

我们来做一个实验：只打开一次文件，然后复制1000次文件描述符。测试代码如下：

```
// dupfd.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
   do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
    int i;
    int fd;
    fd = open("/dev/zero", O_RDONLY);
    if (fd < 0 )
        handle_error("open");
    for( i = 0; i < 1000; i ++)
        dup(fd);
    pause();
}
```



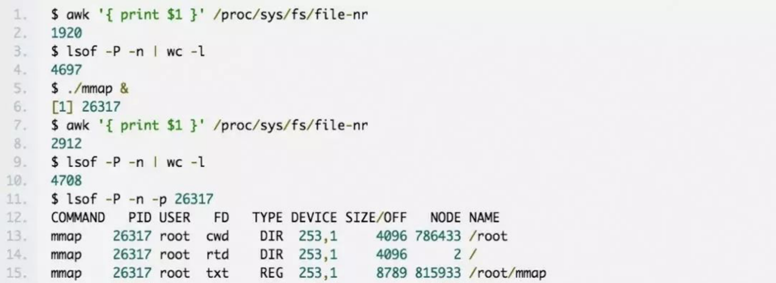
```
$ awk '{ print $1 }' /proc/sys/fs/file-nr
1984
$ lsof -P -n | wc -l
4684
$ ./dupfd &
[1] 23745
$ awk '{ print $1 }' /proc/sys/fs/file-nr
1952
$ lsof -P -n | wc -l
5711

$ lsof -P -n -p 23745 | head -n 15
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
dupfd   23745 root  cwd    DIR  253,1     4096 786433 /root
dupfd   23745 root  rtd    DIR  253,1     4096      2 /
dupfd   23745 root  txt    REG  253,1     8705 815932 /root/dupfd
dupfd   23745 root  mem    REG  253,1    19344 794976 /usr/lib64/libdl-2.17.so
dupfd   23745 root  mem    REG  253,1  2112808 794880 /usr/lib64/libc-2.17.so
dupfd   23745 root  mem    REG  253,1   155112 794690 /usr/lib64/ld-2.17.so
dupfd   23745 root  mem    REG  253,1    42880 795444 /usr/lib64/libonion_security.so.1.0.19
dupfd   23745 root    0u   CHR 136,12      0t0     15 /dev/pts/12
dupfd   23745 root    1u   CHR 136,12      0t0     15 /dev/pts/12
dupfd   23745 root    2u   CHR 136,12      0t0     15 /dev/pts/12
dupfd   23745 root    3r   CHR    1,5      0t0   1030 /dev/zero
dupfd   23745 root    4r   CHR    1,5      0t0   1030 /dev/zero
dupfd   23745 root    5r   CHR    1,5      0t0   1030 /dev/zero
dupfd   23745 root    6r   CHR    1,5      0t0   1030 /dev/zero
```

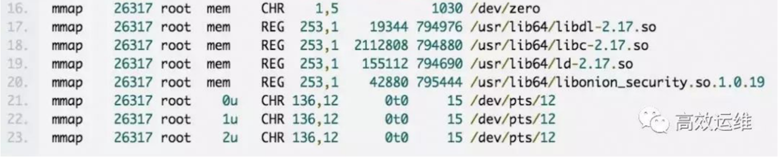
我们启动dupfd进程打开了一次/dev/zero文件，复制了1000次文件描述符。file-nr中的文件句柄数只是个位数的变化，而lsof看到的结果涨了1000多。

如果我们把前面的代码换成open 1000次，就可以看到file-nr和lsof的输出几乎都涨了1000。

lsof看到的是文件描述符不能代表文件句柄，还有一个有趣的例子。下面的mmap程序运行后。 文件句柄增加了将近1000， 而lsof看到的文件描述符才个位数：

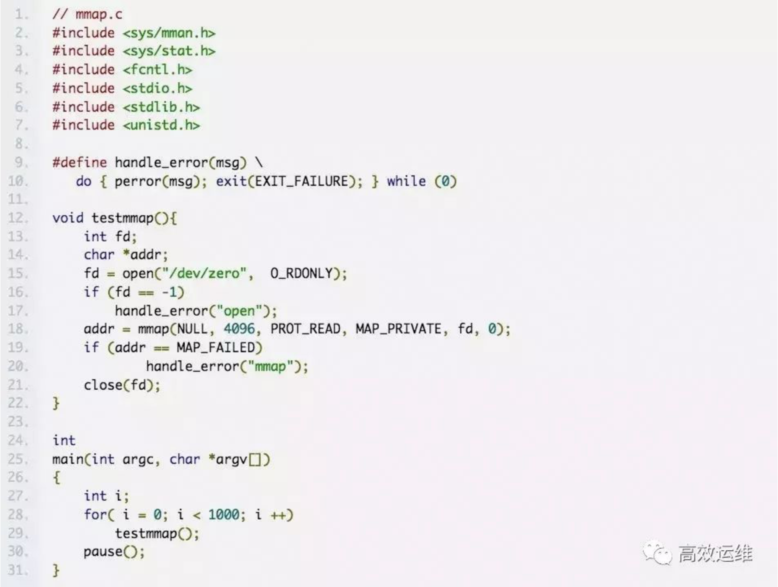


```
$ awk '{ print $1 }' /proc/sys/fs/file-nr
1920
$ lsof -P -n | wc -l
4697
$ ./mmap &
[1] 26317
$ awk '{ print $1 }' /proc/sys/fs/file-nr
2912
$ lsof -P -n | wc -l
4708
$ lsof -P -n -p 26317
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
mmap    26317 root  cwd    DIR  253,1     4096 786433 /root
mmap    26317 root  rtd    DIR  253,1     4096      2 /
mmap    26317 root  txt    REG  253,1     8789 815933 /root/mmap
```



```
mmap    26317 root  mem    CHR    1,5            1030 /dev/zero
mmap    26317 root  mem    REG  253,1    19344 794976 /usr/lib64/libdl-2.17.so
mmap    26317 root  mem    REG  253,1  2112808 794880 /usr/lib64/libc-2.17.so
mmap    26317 root  mem    REG  253,1   155112 794690 /usr/lib64/ld-2.17.so
mmap    26317 root  mem    REG  253,1    42880 795444 /usr/lib64/libonion_security.so.1.0.19
mmap    26317 root   0u    CHR 136,12      0t0     15 /dev/pts/12
mmap    26317 root   1u    CHR 136,12      0t0     15 /dev/pts/12
mmap    26317 root   2u    CHR 136,12      0t0     15 /dev/pts/12
```

我们来看一下测试的mmap程序代码：



```c
// mmap.c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
   do { perror(msg); exit(EXIT_FAILURE); } while (0)

void testmmap(){
    int fd;
    char *addr;
    fd = open("/dev/zero",  O_RDONLY);
    if (fd == -1)
        handle_error("open");
    addr = mmap(NULL, 4096, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED)
            handle_error("mmap");
    close(fd);
}

int
main(int argc, char *argv[])
{
    int i;
    for( i = 0; i < 1000; i ++)
        testmmap();
    pause();
}
```

代码中，我们循环1000次打开/dev/zero文件，之后mmap映射到进程地址空间，然后把这些打开的文件描述符都关掉。很显然，

打开的描述符都被close掉了，不会有什么变化。 那为什么文件句柄数还是增加了1000个左右呢?

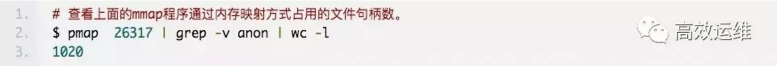
原来，linux内核中很多对象都是有引用计数的。 虽然文件句柄是由open先打开的，但mmap之后，引用计数被加1，尽管我们接着把文件描述符close掉了，但是底层指向的struct file由于引用数大于0，不会被回收。

通过上面两个例子，你应该知道lsof的输出和实际的文件句柄数有差距的原因了。

## 7. 如何找出内存映射间接占用的文件句柄?

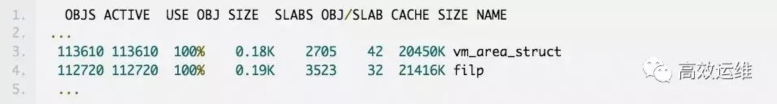
实际上，不管是mmap映射文件，还是通过

shmat连共享内存，最终都会在进程地址空间中分配一片内存区。 通过pmap命令可以看出一些端倪：



```
# 查看上面的mmap程序通过内存映射方式占用的文件句柄数。
$ pmap  26317 | grep -v anon | wc -l
1020
```

回到故事的开头。那个使用了11万文件句柄的机器，在内核slab cache中，除了文件句柄(struct file对象）对应的filp cache对象多之外，对应的内存区对象vm_area_struct占用也是超多的.
下面是slabtop的部分输出：

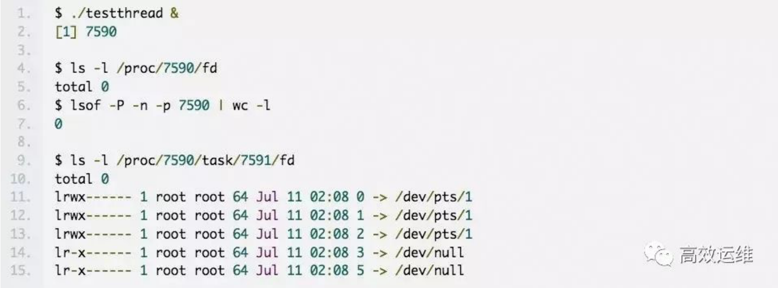


```
  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
...
 113610 113610  100%    0.18K    2705     42  20450K vm_area_struct
 112720 112720  100%    0.19K    3523     32  21416K filp
 ...
```

## 8. 还有其他lsof漏掉的情况吗?

当然有了，lsof是通过查看进程的内存映射

和文件描述符表来枚举打开文件的， 如果是一个多线程的服务。主线程先退出了，子线程还活着， 那么进程的fd表看起来就是空的。



```
$ ./testthread &
[1] 7590

$ ls -l /proc/7590/fd
total 0
$ lsof -P -n -p 7590 | wc -l
0

$ ls -l /proc/7590/task/7591/fd
total 0
lrwx------ 1 root root 64 Jul 11 02:08 0 -> /dev/pts/1
lrwx------ 1 root root 64 Jul 11 02:08 1 -> /dev/pts/1
lrwx------ 1 root root 64 Jul 11 02:08 2 -> /dev/pts/1
lr-x------ 1 root root 64 Jul 11 02:08 3 -> /dev/null
lr-x------ 1 root root 64 Jul 11 02:08 5 -> /dev/null
```

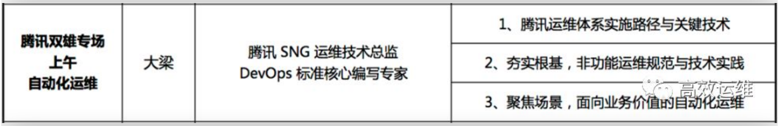
## 9. 总结

Linux内核暴露出来的指标对系统监控很有意义，认识这些指标背后隐含的对象以及增长原因，能够帮助我们在异常时找出问题所在。

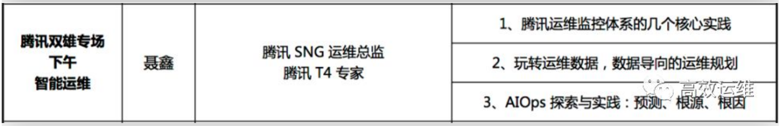
**9 月的上海，运维人有个必去的地方，第10届 GOPS ，回到起点，回到初心！**

# AIOps 风向标！ GOPS 全球运维大会 2018 · 上海站震撼来袭！

GOPS 2018 上海站 亮点之一：腾讯运维双雄：聂鑫、大梁带现身 GOPS 上海站 带您嗨聊一整天！



| | | | |
|---|---|---|---|
| 腾讯双雄专场<br>上午<br>自动化运维 | 大梁 | 腾讯 SNG 运维技术总监<br>DevOps 标准核心编写专家 | 1、腾讯运维体系实施路径与关键技术 |
| | | | 2、夯实根基，非功能运维规范与技术实践 |
| | | | 3、聚焦场景，面向业务价值的自动化运维 |



| | | | |
|---|---|---|---|
| 腾讯双雄专场<br>下午<br>智能运维 | 聂鑫 | 腾讯 SNG 运维总监<br>腾讯 T4 专家 | 1、腾讯运维监控体系的几个核心实践 |
| | | | 2、玩转运维数据，数据导向的运维规划 |
| | | | 3、AIOps 探索与实践：预测、根源、根因 |

点击**阅读原文**，了解更多精彩

Read more