write 文件一个字节后何时发起写磁盘 IO

原创 张彦飞allen 工作日志 2020/11/07 10:34 阅读数 212

在前文《read 文件一个字节实际会发生多大的磁盘 IO？》写完之后，本来想着偷个懒，只通过读操作来让大家了解下 Linux IO 栈的各个模块就行了。但很多同学表示再让我写一篇关于写操作的。既然不少人都有这个需求，那我就写一下吧。

Linux 内核真的是太复杂了，源代码的行数已经从 1.0 版本时的几万行，到现在已经是千万行的一个庞然大物了。直接钻进去的话，很容易在各种眼花缭乱的各种调用中迷失了自己，再也钻不出来了。我分享给大家一个我在琢磨内核的方法。一般我自己先想一个自己很想搞清楚的问题。不管在代码里咋跳来跳去，时刻都要记得自己的问题，无关的部分尽量少去发散，只要把自己的问题搞清楚了就行了。

现在我想搞明白的问题是，在最常用的方式下，不开 O_DIRECT、不开 O_SYNC（写文件的方法有很多，有 sync 模式、direct 模式、mmap 内存映射模式），write 是怎么写的。c 的代码示例如下:

```
#include <fcntl.h>
int main()
{
    char c = 'a';
    int out;

    out = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC);
    write(out,&c,1);
    ...
}
```

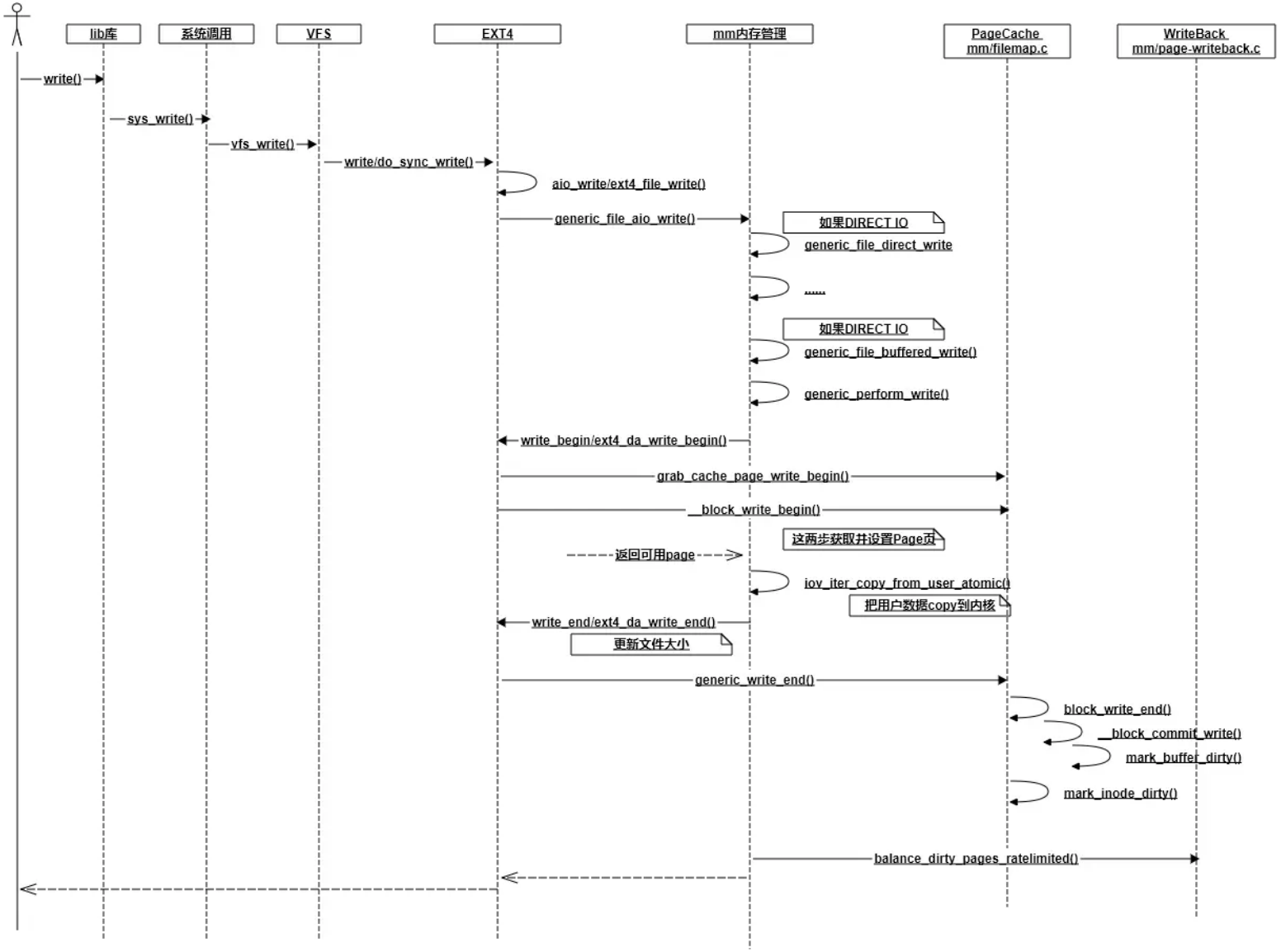
进一步细化我的问题，我们对打开的问题写入一个字节后

- write 函数在内核里是怎么执行的?
- 数据在什么时机真正能写入到磁盘上?

我们在讨论的过程中不可避免地要涉及到内核代码，我使用的内核版本是 3.10.1。如果有需要，你可以到这里来下载。
https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/。

write 函数实现剖析

我花了不短的时候跟踪 write 写到 ext4 文件系统时的各种调用和返回，大致理出来了一个交互图。当然为了突出重点，我抛弃了不少细节，比如 DIRECT IO、ext4 日志记录啥的都没有体现出来，只抽取出来了一些我认为关键的调用。



在上面的流程图里，所有的写操作最终到哪儿了呢？在最后面的__block_commit_write 中，只是 make dirty。然后大部分情况下你的函数调用就返回了（稍后再说 balance_dirty_pages_ratelimited）。数据现在还在内存中的 PageCache 里，并没有真正写到硬盘。

为什么要这样实现，不直接写硬盘呢？原因就在于硬盘尤其是机械硬盘，性能是在是太慢了。一块服务器级别的万转盘，最坏随机访问平均延迟都是毫秒级别的，换算成 IOPS 只有 100 多不到 200。设想一下，假如你的后端接口里每个用户来访问都需要一次随机磁盘 IO，不管你多牛的服务器，每秒 200 的 qps 都将直接打爆你的硬盘，相信作为为百万 / 千万 / 过亿用户提供接口的你，这个是你绝对不能忍的。

Linux 这么搞也是有副作用的，如果接下来服务器发生掉电，内存里东西全丢。所以 Linux 还有另外一个 “补丁”- 延迟写，帮我们缓解这个问题。注意下，我说的是缓解，并没有彻底解决。

再说下 balance_dirty_pages_ratelimited，虽然绝大部分情况下，都是直接写到 Page Cache 里就返回了。但在一种情况下，用户进程必须得等待写入完成才可以返回，那就是对 balance_dirty_pages_ratelimited 的判断如果超出限制了。该函数判断当前脏页是否已经超过脏页上限 dirty_bytes、dirty_ratio，超过了就必须得等待。这两个参数只有一个会生效，另外 1 个是 0。拿 dirty_ratio 来说，如果设置的是 30，就说明如果脏页比例超过内存的 30%，则 write 函数调用就必须等待写入完成才能返回。可以在你的机器下的 /proc/sys/vm/ 目录来查看这两个配置。

```
# cat /proc/sys/vm/dirty_bytes
0
# cat /proc/sys/vm/dirty_ratio
30
```

内核延迟写

内核是什么时候真正把数据写到硬盘中呢？为了快速摸清楚全貌，我想到的办法是用 systemtap 工具，找到内核写 IO 过程中的一个关键函数，然后在其中把函数调用堆栈打出来。查了半天资料以后，我决定用 do_writepages 这个函数。

```
#!/usr/bin/stap
probe kernel.function("do_writepages")
{
    printf("-----------------------------------------------------------\n");
    print_backtrace();
    printf("-----------------------------------------------------------\n");
}
```

systemtab 跟踪以后，打印信息如下:

```
0xffffffff8118efe0 : do_writepages+0x0/0x40 [kernel]
0xffffffff8122d7d0 : __writeback_single_inode+0x40/0x220 [kernel]
0xffffffff8122e414 : writeback_sb_inodes+0x1c4/0x490 [kernel]
0xffffffff8122e77f : __writeback_inodes_wb+0x9f/0xd0 [kernel]
0xffffffff8122efb3 : wb_writeback+0x263/0x2f0 [kernel]
0xffffffff8122f35c : bdi_writeback_workfn+0x1cc/0x460 [kernel]
0xffffffff810a881a : process_one_work+0x17a/0x440 [kernel]
0xffffffff810a94e6 : worker_thread+0x126/0x3c0 [kernel]
0xffffffff810b098f : kthread+0xcf/0xe0 [kernel]
0xffffffff816b4f18 : ret_from_fork+0x58/0x90 [kernel]
```

从上面的输出我们可以看出，真正的写文件过程操作是由 worker 内核线程发出来的（和我们自己的应用程序进程没有半毛钱关系，此时我们的应用程序的 write 函数调用早就返回了）。这个 worker 线程写回是周期性执行的，它的周期取决于内核参数 dirty_writeback_centisecs 的设置，根据参数名也大概能看出来，它的单位是百分之一秒。

```
# cat /proc/sys/vm/dirty_writeback_centisecs
500
```

我查看到我的配置是 500，就是说每 5 秒会周期性地来执行一遍。回顾我们的问题，我们最关心的问题的啥时候写入的，围绕这个思路不过多发散。于是沿着这个调用栈不断地跟踪，跳转，终于找到了下面的代码。如下代码里我们看到，如果是 for_background 模式，且 `over_bground_thresh` 判断成功，就会开始回写了。

```
static long wb_writeback(struct bdi_writeback *wb,
                         struct wb_writeback_work *work)
{
        work->older_than_this = &oldest_jif;
    ...
    if (work->for_background && !over_bground_thresh(wb->bdi))
        break;
        ...

    if (work->for_kupdate) {
        oldest_jif = jiffies -
                msecs_to_jiffies(dirty_expire_interval * 10);
    } else ...
}
static long wb_check_background_flush(struct bdi_writeback *wb)
{
    if (over_bground_thresh(wb->bdi)) {
```

```
                ...
        return wb_writeback(wb, &work);
    }
}
```

那么 `over_bground_thresh` 函数判断的是啥呢？其实就是判断当前的脏页是不是超过内核参数里 dirty_background_ratio 或 dirty_background_bytes 的配置，没超过的话就不写了（代码位于 fs/fs-writeback.c：1440，限于篇幅我就不贴了）。这两个参数只有一个会真正生效，其中 dirty_background_ratio 配置的是比例、dirty_background_bytes 配置的是字节。

在我的机器上的这两个参数配置如下，表示脏页比例超过 10% 就开始回写。

```
# cat /proc/sys/vm/dirty_background_bytes
0
# cat /proc/sys/vm/dirty_background_ratio
10
```

那如果脏页一直都不超过这个比例怎么办呢，就不写了吗？ 不是的。在上面的 wb_writeback 函数中我们看到了，如果是 for_kupdate 模式，会记录一个过期标记到 work->older_than_this，再往后面的代码中把符合这个条件的页面也写回了。dirty_expire_interval 这个变量是从哪儿来的呢？ 在 kernel/sysctl.c 里，我们发现了蛛丝马迹。哦，原来它是来自 /proc/sys/vm/dirty_expire_centisecs 这个配置。

```
1158        {
1159                .procname       = "dirty_expire_centisecs",
1160                .data           = &dirty_expire_interval,
1161                .maxlen         = sizeof(dirty_expire_interval),
1162                .mode           = 0644,
1163                .proc_handler   = proc_dointvec_minmax,
1164                .extra1         = &zero,
1165        },
```

在我的机器上，它的值是 3000。单位是百分之一秒，所以就是脏页过了 30 秒就会被内核线程认为需要写回到磁盘了。

```
# cat /proc/sys/vm/dirty_expire_centisecs
3000
```

结论

我们 demo 代码中的写入，其实绝大部分情况都是写入到 PageCache 中就返回了，这时并没有真正写入磁盘。我们的数据会在如下三个时机下被真正发起写磁盘 IO 请求：

- 第一种情况，如果 write 系统调用时，如果发现 PageCache 中脏页占比太多，超过了 dirty_ratio 或 dirty_bytes，write 就必须等待了。
- 第二种情况，write 写到 PageCache 就已经返回了。worker 内核线程异步运行的时候，再次判断脏页占比，如果超过了 dirty_background_ratio 或 dirty_background_bytes，也发起写回请求。
- 第三种情况，这时同样 write 调用已经返回了。worker 内核线程异步运行的时候，虽然系统内脏页一直没有超过 dirty_background_ratio 或 dirty_background_bytes，但是脏页在内存中呆的时间超过 dirty_expire_centisecs 了，也会发起会写。

如果对以上配置不满意，你可以自己通过修改 /etc/sysctl.conf 来调整，修改完了别忘了执行 sysctl -p。

最后我们要认识到，这套 write pagecache + 回写的机制第一目标是性能，不是保证不丢失我们写入的数据的。如果这时候掉电，脏页时间未超过 dirty_expire_centisecs 的就真的丢了。如果你做的是和钱相关非常重要的业务，必须保证落盘完成才能返回，那么你就可能需要考虑使用 fsync。

开发内功修炼之硬盘篇专辑：
