昵称： Jessica程序猿
园龄： 5年10个月
粉丝： 544
关注： 27
+加关注

< 2020年4月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

常用链接

我的随笔

我的评论

我的参与

最新评论

我的标签

随笔分类 (988)

C++(79)

C++ template(5)

C++ 容器(28)

C++构造函数(6)

C++面向对象编程(7)

C++训练(71)

C++重载与类型转换(9)

careercup(87)

dubbo(2)

Effective C++(3)

ext2文件系统系列(6)

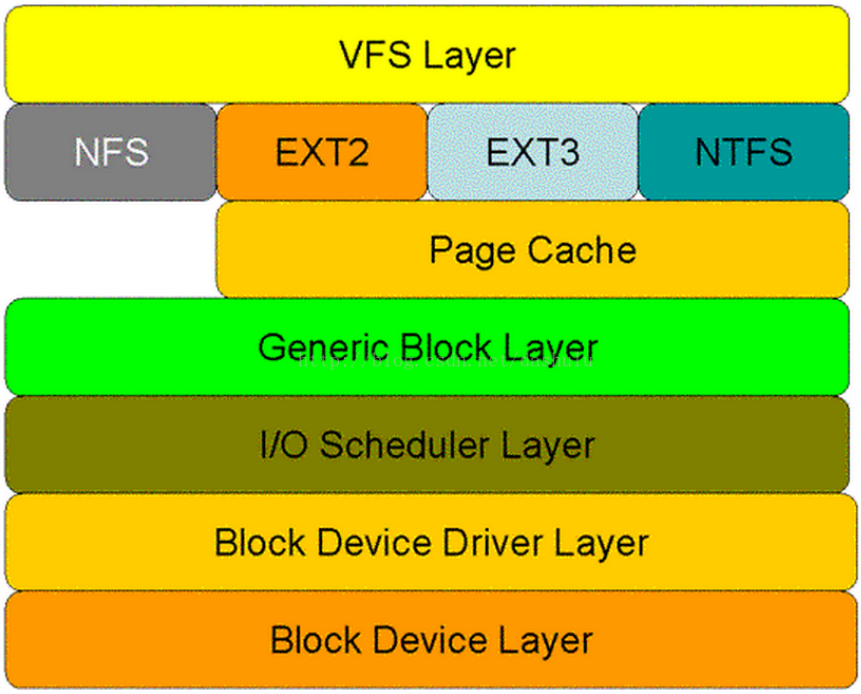
flashcache(2)

KVM虚拟机IO处理过程(一) ----Guest VM I/O 处理过程

虚拟化技术主要包含三部分内容:CPU虚拟化,内存虚拟化,设备虚拟化.本系列文章主要描述磁盘设备的虚拟化过程,包含了一个读操作的I/O请求如何从Guest Vm到其最终被处理的整个过程.本系列文章中引用到的linux内核代码版本为3.7.10,使用的虚拟化平台是KVM,qemu的版本是1.6.1.

用户程序想要访问IO设备需要调用操作系统提供的接口,即系统调用.当在用户程序中调用一个read操作时,系统先保存好read操作的参数,然后调用int 80命令(也可能是sysenter)进入内核空间,在内核空间中,读操作的逻辑由sys_read函数实现.

在讲sys_read的实现过程之前,我们先来看看read操作在内核空间需要经历的层次结构.从图中可以看出,read操作首先经过虚拟文件系统曾(vfs), 接下来是具体的文件系统层,Page cache层,通用块层(generic block layer),I/O调度层(I/O scheduler layer),块设备驱动层(block device driver layer),最后是块物理设备层(block device layer).



- 虚拟文件系统层:该层屏蔽了下层的具体操作,为上层提供统一的接口,如vfs_read,vfs_write等.vfs_read,vfs_write通过调用下层具体文件系统的接口来实现相应的功能.
- 具体文件系统层:该层针对每一类文件系统都有相应的操作和实现了,包含了具体文件系统的处理逻辑.
- page cache层:该层缓存了从块设备中获取的数据.引入该层的目的是避免频繁的块设备访问,如果在page cache中已经缓存了I/O请求的数据,则可以将数据直接返回,无需访问块设备.
- 通过块层:接收上层的I/O请求,并最终发出I/O请求.该层向上层屏蔽了下层设备的特性.
- I/O调度层： 接收通用块层发出的 IO 请求，缓存请求并试图合并相邻的请求（如果这两个请求的数据在磁盘上是相邻的）。并根据设置好的调度算法，回调驱动层提供的请求处理函数，以处理具体的 IO 请求
- 块设备驱动层:从上层取出请求,并根据参数,操作具体的设备.
- 块设备层:真正的物理设备.

了解了内核层次的结构,让我们来看一下read操作的代码实现.
sys_read函数声明在include/linux/syscalls.h文件中,

[cpp] view plain copy

```
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
```

其函数实现在fs/read_write.c文件中:

[cpp] view plain copy

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    struct fd f = fdget(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_read(f.file, buf, count, &pos); //调用vfs layer中的read操作
        file_pos_write(f.file, pos);//设置当前文件的位置
        fdput(f);
    }
    return ret;
}
```

vfs_read函数属于vfs layer,定义在fs/read_write.c, 其主要功能是调用具体文件系统中对应的read操作,如果具体文件系统没有提供read操作,则使用默认的do_sync_read函数.

[cpp] view plain copy

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos
)
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_READ))
        return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;

    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read) {
            ret = file->f_op->read(file, buf, count, pos); //该函数由具体的文件系统
指定
        } else
            ret = do_sync_read(file, buf, count, pos);  //内核默认的读文件操作
        if (ret > 0) {
            fsnotify_access(file);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }

    return ret;
}
```

file->f_op的类型为struct file_operations, 该类型定义了一系列涉及文件操作的函数指针,针对不同的文件系统,这些函数指针指向不同的实现.以ext4 文件系统为例子,该数据结构的初始化在fs/ext4/file.c,从该初始化可以知道,ext4的read操作调用了内核自带的do_sync_read()函数

[cpp] view plain copy

```
const struct file_operations ext4_file_operations = {
    .llseek     = ext4_llseek,
    .read       = do_sync_read,
    .write      = do_sync_write,
    .aio_read   = generic_file_aio_read,
    .aio_write  = ext4_file_write,
    .unlocked_ioctl = ext4_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl   = ext4_compat_ioctl,
```

软件安装(31)
设计模式(12)
深度探索C++对象模型(17)
深入理解Linux内核(1)
数据结构与算法(60)
搜索引擎(2)
算法 每日一练(7)
算法导论(27)
文件系统(20)
虚拟化(11)
杂项(23)

随笔档案 (974)
2020年1月(1)
2019年2月(2)
2019年1月(1)
2018年8月(1)
2018年7月(1)
2018年6月(6)
2018年5月(7)
2018年4月(2)
2018年2月(3)
2017年12月(3)
2017年11月(1)
2017年9月(2)
2017年5月(2)
2017年4月(2)
2017年3月(3)
2017年2月(3)
2016年5月(4)
2016年4月(12)
2016年3月(3)
2016年2月(2)
2016年1月(6)

```
#endif
    .mmap       = ext4_file_mmap,
    .open       = ext4_file_open,
    .release    = ext4_release_file,
    .fsync      = ext4_sync_file,
    .splice_read    = generic_file_splice_read,
    .splice_write   = generic_file_splice_write,
    .fallocate  = ext4_fallocate,
};
```

do_sync_read()函数定义fs/read_write.c中,

[cpp] view plain copy

```
ssize_t do_sync_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos)
{
    struct iovec iov = { .iov_base = buf, .iov_len = len };
    struct kiocb kiocb;
    ssize_t ret;

    init_sync_kiocb(&kiocb, filp);//初始化kiocp,描述符kiocb是用来记录I/O操作的完成状态
    kiocb.ki_pos = *ppos;
    kiocb.ki_left = len;
    kiocb.ki_nbytes = len;

    for (;;) {
        ret = filp->f_op->aio_read(&kiocb, &iov, 1, kiocb.ki_pos);//调用真正做读操作的函数,ext4文件系统在fs/ext4/file.c中配置
        if (ret != -EIOCBRETRY)
            break;
        wait_on_retry_sync_kiocb(&kiocb);
    }

    if (-EIOCBQUEUED == ret)
        ret = wait_on_sync_kiocb(&kiocb);
    *ppos = kiocb.ki_pos;
    return ret;
}
```

在ext4文件系统中filp->f_op->aio_read函数指针只想generic_file_aio_read, 该函数定义于mm/filemap.c文件中,该函数有两个执行路径,如果是以O_DIRECT方式打开文件,则读操作跳过page cache直接去读取磁盘,否则调用do_generic_sync_read函数尝试从page cache中获取所需的数据.

[cpp] view plain copy

```
ssize_t
generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov,
        unsigned long nr_segs, loff_t pos)
{
    struct file *filp = iocb->ki_filp;
    ssize_t retval;
    unsigned long seg = 0;
    size_t count;
    loff_t *ppos = &iocb->ki_pos;

    count = 0;
    retval = generic_segment_checks(iov, &nr_segs, &count, VERIFY_WRITE);
    if (retval)
        return retval;

    /* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
    if (filp->f_flags & O_DIRECT) {
        loff_t size;
        struct address_space *mapping;
        struct inode *inode;

        struct timex txc;
        do_gettimeofday(&(txc.time));

        mapping = filp->f_mapping;
        inode = mapping->host;
        if (!count)
```

2015年11月(3)
2015年10月(7)
2015年9月(15)
2015年8月(10)
2015年7月(11)
2015年6月(2)
2015年5月(38)
2015年4月(60)
2015年3月(71)
2015年2月(3)
2015年1月(3)
2014年12月(119)
2014年11月(180)
2014年10月(69)
2014年9月(18)
2014年8月(109)
2014年7月(67)
2014年6月(85)
2014年5月(37)

文章分类 (0)

metaq
netty
UML

linux

阮一峰的网络日志
淘宝内核组
阿里核心系统团队博客

算法牛人

v_JULY_v
分布式文件系统测试
acm之家
Linux开发专注者

```
28.             goto out; /* skip atime */
29.         size = i_size_read(inode);
30.         if (pos < size) {
31.             retval = filemap_write_and_wait_range(mapping, pos,
32.                     pos + iov_length(iov, nr_segs) - 1);
33.             if (!retval) {
34.                 retval = mapping->a_ops->direct_IO(READ, iocb,
35.                             iov, pos, nr_segs);
36.             }
37.             if (retval > 0) {
38.                 *ppos = pos + retval;
39.                 count -= retval;
40.             }
41.
42.             /*
43.              * Btrfs can have a short DIO read if we encounter
44.              * compressed extents, so if there was an error, or if
45.              * we've already read everything we wanted to, or if
46.              * there was a short read because we hit EOF, go ahead
47.              * and return.  Otherwise fallthrough to buffered io for
48.              * the rest of the read.
49.              */
50.             if (retval < 0 || !count || *ppos >= size) {
51.                 file_accessed(filp);
52.                 goto out;
53.             }
54.         }
55.     }
56.
57.     count = retval;
58.     for (seg = 0; seg < nr_segs; seg++) {
59.         read_descriptor_t desc;
60.         loff_t offset = 0;
61.
62.         /*
63.          * If we did a short DIO read we need to skip the section of the
64.          * iov that we've already read data into.
65.          */
66.         if (count) {
67.             if (count > iov[seg].iov_len) {
68.                 count -= iov[seg].iov_len;
69.                 continue;
70.             }
71.             offset = count;
72.             count = 0;
73.         }
74.
75.         desc.written = 0;
76.         desc.arg.buf = iov[seg].iov_base + offset;
77.         desc.count = iov[seg].iov_len - offset;
78.         if (desc.count == 0)
79.             continue;
80.         desc.error = 0;
81.         do_generic_file_read(filp, ppos, &desc, file_read_actor);
82.         retval += desc.written;
83.         if (desc.error) {
84.             retval = retval ?: desc.error;
85.             break;
86.         }
87.         if (desc.count > 0)
88.             break;
89.     }
90. out:
91.     return retval;
92. }
```

do_generic_file_read定义在mm/filemap.c文件中,该函数调用page cache层中相关的函数.如果所需数据存在与page cache中,并且数据不是dirty的,则从page cache中直接获取数据返回.如果数据在page cache中不存在,或者数据是dirty的,则page cache会引发读磁盘的操作.该函数的读磁盘并不是简单的只读取所需数据的所在的block,而是会有一定的预读机制来提高cache的命中率,减少磁盘访问的次数.

page cache层中真正读磁盘的操作为readpage系列,readpage系列函数具体指向的函数实现在fs/ext4/inode.c文件中定义,该文件中有很多个struct address_space_operation对象来对应与不同

酷壳

C++11 中值得关注的几大变化（详解）

iTech's Blog

Not Only Algorithm，不仅仅是算法，关注数学、算法、数据结构、程序员笔试面试以及一切涉及计算机编程之美的内容 。

最新评论

1. Re:编写一个程序，从标准输入中读取若干string对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词

输入how cow，两个不同的单词，统计会出现问题。结果应该是没有连续出现的单词。

--想撸串的红杉树

2. Re:linux内核内存管理(zone_dma zone_normal zone_highmem)

你好，我想咨询您一个问题，32位系统中，用户进程可以访问物理内存的大小为3G，那么这3G有具体的空间范围吗？是物理内存的0-3G还是什么，因为我看0-896M是被内核线性映射，所以肯定表示0-3G，所...

--CV学习者

3. Re:spring中bean配置和bean注入

tql

--那么小晨呐

4. Re:如何使用jstack分析线程状态

thanks

--andyFly2016

5. Re:迪杰斯特拉算法介绍

不过有图挺好的，但我看到那里看不懂了。

--何所倚

阅读排行榜

1. spring中bean配置和bean注入(138538)

日志机制,我们选择linux默认的ordered模式的日志机制来描述I/O的整个流程, ordered模式对应的readpage系列函数如下所示.

[cpp] view plain copy

```cpp
static const struct address_space_operations ext4_ordered_aops = {
    .readpage       = ext4_readpage,
    .readpages      = ext4_readpages,
    .writepage      = ext4_writepage,
    .write_begin        = ext4_write_begin,
    .write_end      = ext4_ordered_write_end,
    .bmap           = ext4_bmap,
    .invalidatepage     = ext4_invalidatepage,
    .releasepage        = ext4_releasepage,
    .direct_IO      = ext4_direct_IO,
    .migratepage        = buffer_migrate_page,
    .is_partially_uptodate  = block_is_partially_uptodate,
    .error_remove_page  = generic_error_remove_page,
};
```

为简化流程,我们选取最简单的ext4_readpage函数来说明,该函数实现位于fs/ext4/inode.c中,函数很简单,只是调用了mpage_readpage函数.mpage_readpage位于fs/mpage.c文件中,该函数生成一个IO请求,并提交给Generic block layer.

[cpp] view plain copy

```cpp
int mpage_readpage(struct page *page, get_block_t get_block)
{
    struct bio *bio = NULL;
    sector_t last_block_in_bio = 0;
    struct buffer_head map_bh;
    unsigned long first_logical_block = 0;

    map_bh.b_state = 0;
    map_bh.b_size = 0;
    bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
            &map_bh, &first_logical_block, get_block);
    if (bio)
        mpage_bio_submit(READ, bio);
    return 0;
}
```

Generic block layer会将该请求分发到具体设备的IO队列中,由I/O Scheduler去调用具体的driver接口获取所需的数据.

至此,在Guest vm中整个I/O的流程已经介绍完了,后续的文章会介绍I/O操作如何从Guest vm跳转到kvm及如何在qemu中模拟I/O设备.

参考资料:
1. read系统调用剖析:http://www.ibm.com/developerworks/cn/linux/l-cn-read/
转载：http://blog.csdn.net/dashulu/article/details/16820281

分类: 虚拟化

Jessica程序猿
关注 - 27
粉丝 - 544
+加关注

« 上一篇： KVM的初始化过程
» 下一篇： KVM虚拟机IO处理过程(二) ----QEMU/KVM I/O 处理过程

posted @ 2015-07-30 16:21 Jessica程序猿 阅读(869) 评论(9) 编辑 收藏

评论列表

2. 如何使用jstack分析线程状态(89973)

3. maven快照版本和发布版本(29243)

4. C++ stringstream介绍，使用方法与例子(28518)

5. Linux用户空间与内核空间（理解高端内存）(27348)

评论排行榜

1. KVM虚拟机IO处理过程(一) ----Guest VM I/O 处理过程(9)

2. careercup-递归和动态规划 9.10(7)

3. 如何使用jstack分析线程状态(6)

4. spring中bean配置和bean注入(6)

5. 蜻蜓fm面试(6)

推荐排行榜

1. 如何使用jstack分析线程状态(20)

2. spring中bean配置和bean注入(19)

3. 数据库索引原理及优化(12)

4. maven快照版本和发布版本(8)

5. linux下的僵尸进程处理SIGCHLD信号(8)

#1楼 2015-07-30 19:34 外禅内定，程序人生

Hi ～Jessica！没猜错你在牛客网做过题目吧，今天查东西偶然翻到你的博客，点个赞～！

支持(0) 反对(0)

#2楼 [楼主] 2015-07-30 19:37 Jessica程序猿

@ 外禅内定，程序人生
是的。。这样就被发现了？？？

支持(0) 反对(0)

#3楼 2015-07-30 21:04 外禅内定，程序人生

@ Jessica程序猿
是呀！你用的同一个id不被发现都难。。。。

支持(0) 反对(0)

#4楼 [楼主] 2015-07-30 21:05 Jessica程序猿

@ 外禅内定，程序人生
额，好吧~~

支持(0) 反对(0)

#5楼 2015-07-30 21:09 外禅内定，程序人生

@ Jessica程序猿
回复的好快，问一下，你大几了，当时我记得你好像在牛客2天做了56道算法题，当时真是吓到我了

支持(0) 反对(0)

#6楼 [楼主] 2015-07-30 21:10 Jessica程序猿

@ 外禅内定，程序人生
是因为有邮件提醒。。我研二了，当时无聊就刷题了，现在就没时间做了。。。

支持(0) 反对(0)

#7楼 2015-07-30 21:14 外禅内定，程序人生

@ Jessica程序猿
研二的学姐哈～！好厉害！我看你发的博客， 你是做linux服务器开发方面的么

支持(0) 反对(0)

#8楼 2015-08-18 10:54 LizSep

貌似我也在牛客见过这个ID，同为16年毕业为秋招做准备的嘛?

支持(0) 反对(0)

#9楼 [楼主] 2015-08-19 13:35 Jessica程序猿

@ ModeApril
是的。

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

相关博文：
· LinuxKernel文件系统写I/O流程代码分析（一）
· linux IO子系统和文件系统读写流程
· 转 -- linux IO子系统和文件系统读写流程
· Linux块设备IO子系统(二) _页高速缓存
· 文件IO中的direct和sync标志位——O_DIRECT和O_SYNC详析
» 更多推荐...