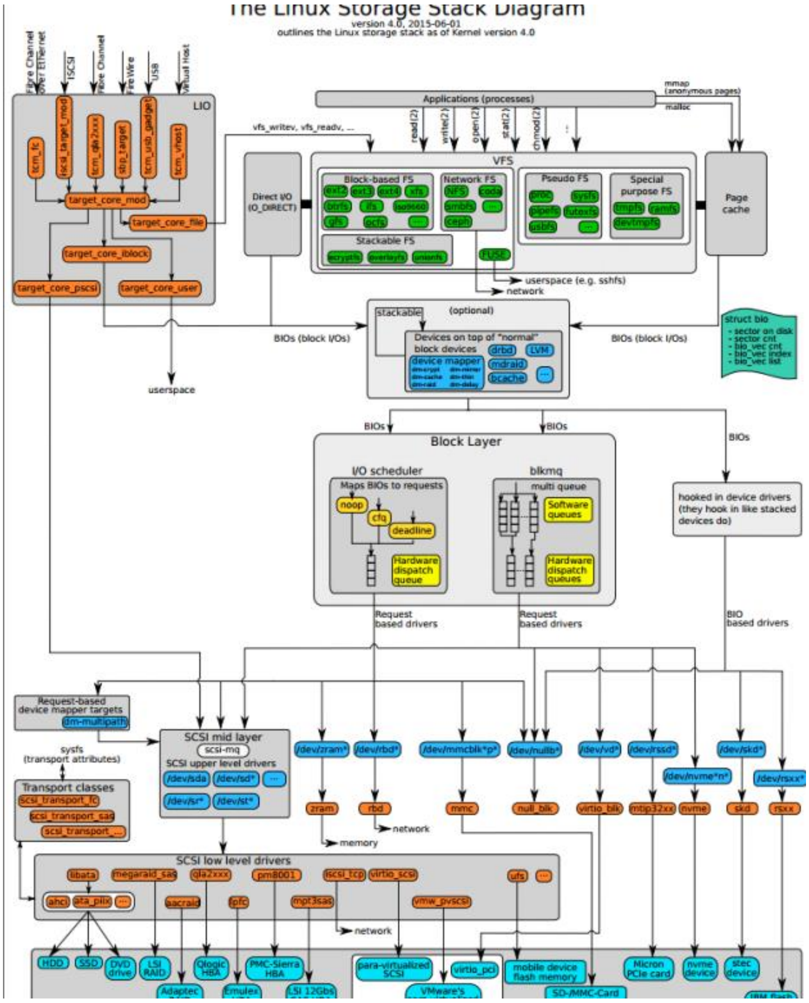
Linux ext4写流程代码分析

2018年5月22日　　19:07

什么都不说，先上图：



一次用户IO操作依次经过以下流程交给硬件处理：

　　Syscall ==》 vfs ==》 filesystem ==》 block layer ==》 device

本次流程分析主要分析到filesystem结束，以ext4写文件为例，梳理整个处理流程。

1. write系统调用

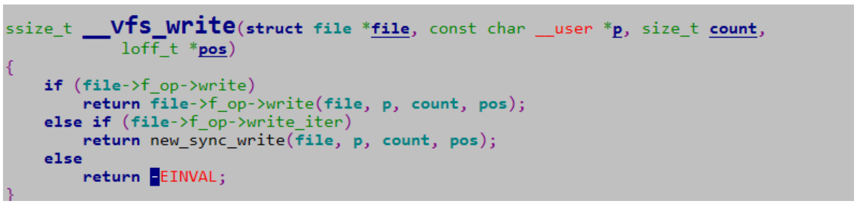
看定义：根据文件句柄，写count字节的buf内容到文件中。

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

2.内核对于系统调用的实现以宏定义方式SYSCALL_DEFINE3，其中3代表参数个数：

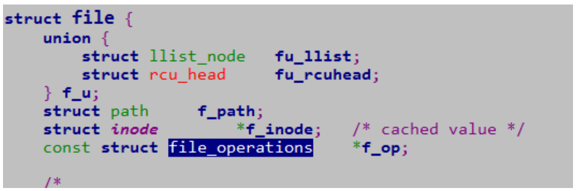
```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
		size_t, count)
{
	struct fd f = fdget_pos(fd);
	ssize_t ret = -EBADF;

	if (f.file) {
		loff_t pos = file_pos_read(f.file);
		ret = vfs_write(f.file, buf, count, &pos);
		if (ret >= 0)
			file_pos_write(f.file, pos);
		fdput_pos(f);
	}

	return ret;
}
```

可以看到内核只是将fd做了一次结构定义转换，增加了pos的输入，pos就是我们熟悉的offset。
我们一般默认写文件是从文件头开始，可以通过lseek修改pos位置，也可以采用append模式打开文件，这样pos会指向文件末尾。
接下来看一下vfs_write的实现，这一层就是vfs层了：



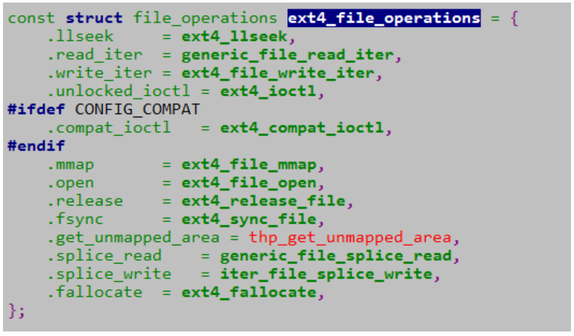
```
ssize_t __vfs_write(struct file *file, const char __user *p, size_t count,
		   loff_t *pos)
{
	if (file->f_op->write)
		return file->f_op->write(file, p, count, pos);
	else if (file->f_op->write_iter)
		return new_sync_write(file, p, count, pos);
	else
		return -EINVAL;
}
```

可以看到write其实调用的是个函数指针，这就是vfs的层核心结构，通过file_operations挂接到对应的不通文件系统中：

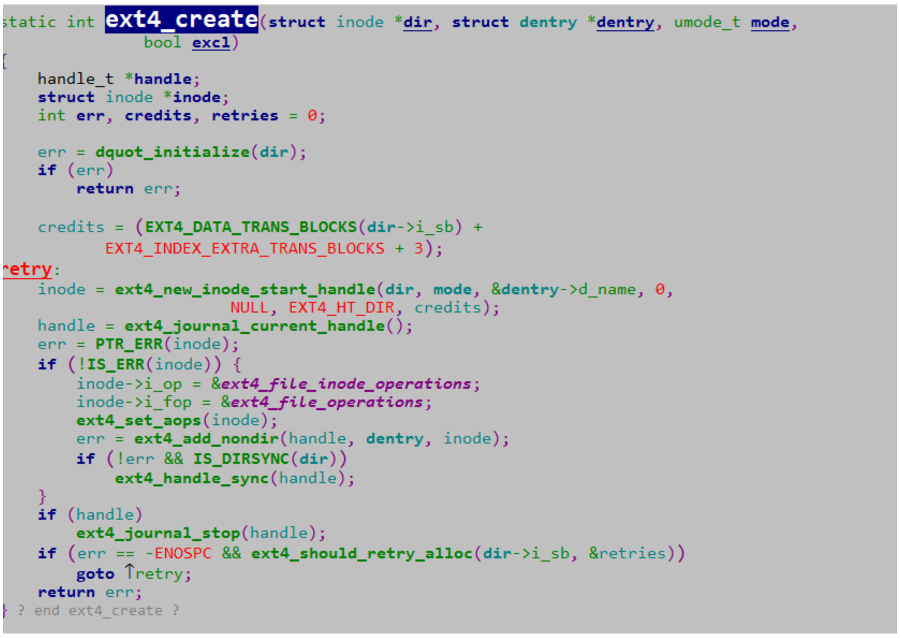


```
struct file {
    union {
        struct llist_node   fu_llist;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path     f_path;
    struct inode        *f_inode;   /* cached value */
    const struct file_operations    *f_op;

    /*
```

例如ext4文件系统创建inode时初始化函数指针：



```
const struct file_operations ext4_file_operations = {
    .llseek     = ext4_llseek,
    .read_iter  = generic_file_read_iter,
    .write_iter = ext4_file_write_iter,
    .unlocked_ioctl = ext4_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl   = ext4_compat_ioctl,
#endif
    .mmap       = ext4_file_mmap,
    .open       = ext4_file_open,
    .release    = ext4_release_file,
    .fsync      = ext4_sync_file,
    .get_unmapped_area = thp_get_unmapped_area,
    .splice_read    = generic_file_splice_read,
    .splice_write   = iter_file_splice_write,
    .fallocate  = ext4_fallocate,
};
```



```
static int ext4_create(struct inode *dir, struct dentry *dentry, umode_t mode,
               bool excl)
{
    handle_t *handle;
    struct inode *inode;
    int err, credits, retries = 0;

    err = dquot_initialize(dir);
    if (err)
        return err;

    credits = (EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
           EXT4_INDEX_EXTRA_TRANS_BLOCKS + 3);
retry:
    inode = ext4_new_inode_start_handle(dir, mode, &dentry->d_name, 0,
                        NULL, EXT4_HT_DIR, credits);
    handle = ext4_journal_current_handle();
    err = PTR_ERR(inode);
    if (!IS_ERR(inode)) {
        inode->i_op = &ext4_file_inode_operations;
        inode->i_fop = &ext4_file_operations;
        ext4_set_aops(inode);
        err = ext4_add_nondir(handle, dentry, inode);
        if (!err && IS_DIRSYNC(dir))
            ext4_handle_sync(handle);
    }
    if (handle)
        ext4_journal_stop(handle);
    if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
        goto ↑retry;
    return err;
} ? end ext4_create ?
```

基本关系沥青了我们开始分析ext4写文件的入口函数：ext4_file_write_iter

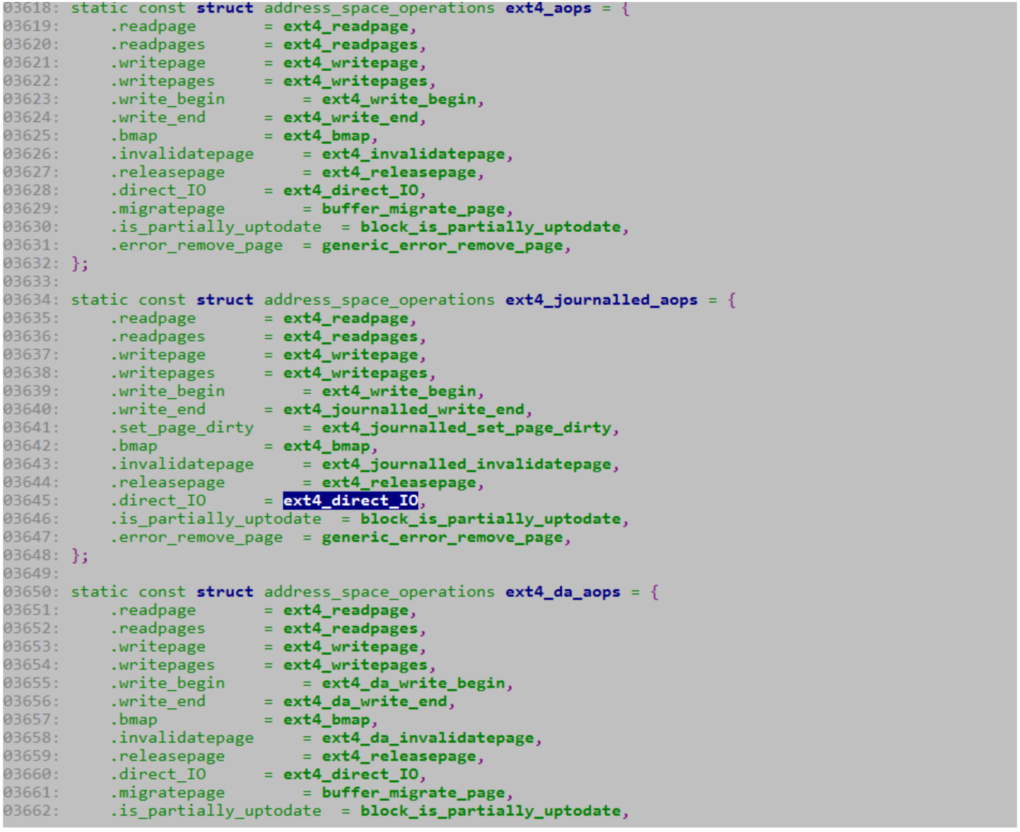
```
static ssize_t
ext4_file_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
        struct inode *inode = file_inode(iocb->ki_filp);    //找到文件对应的inode
        int o_direct = iocb->ki_flags & IOCB_DIRECT;   //判断是否是direct模式
        int unaligned_aio = 0;
        int overwrite = 0;
        ssize_t ret;

        inode_lock(inode);
        ret = generic_write_checks(iocb, from);
        if (ret <= 0)
                goto out;

        /*
         * Unaligned direct AIO must be serialized among each other as zeroing
         * of partial blocks of two competing unaligned AIOs can result in data
         * corruption.
         */
        if (o_direct && ext4_test_inode_flag(inode, EXT4_INODE_EXTENTS) &&
            !is_sync_kiocb(iocb) &&
            ext4_unaligned_aio(inode, from, iocb->ki_pos)) {
                unaligned_aio = 1;
                ext4_unwritten_wait(inode);
        }

        /*
         * If we have encountered a bitmap-format file, the size limit
         * is smaller than s_maxbytes, which is for extent-mapped files.    //处理非extent文件
         */
        if (!(ext4_test_inode_flag(inode, EXT4_INODE_EXTENTS))) {
                struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);

                if (iocb->ki_pos >= sbi->s_bitmap_maxbytes) {
                        ret = -EFBIG;
                        goto out;
                }
                iov_iter_truncate(from, sbi->s_bitmap_maxbytes - iocb->ki_pos);
        }

        iocb->private = &overwrite;
        if (o_direct) {
                size_t length = iov_iter_count(from);
                loff_t pos = iocb->ki_pos;
```

```
		/* check whether we do a DIO overwrite or not */
		if (ext4_should_dioread_nolock(inode) && !unaligned_aio &&
		    pos + length <= i_size_read(inode)) {
			struct ext4_map_blocks map;
			unsigned int blkbits = inode->i_blkbits;
			int err, len;

			map.m_lblk = pos >> blkbits;
			map.m_len = EXT4_MAX_BLOCKS(length, pos, blkbits);
			len = map.m_len;

			err = ext4_map_blocks(NULL, inode, &map, 0);
			/*
			 * 'err==len' means that all of blocks has
			 * been preallocated no matter they are
			 * initialized or not.  For excluding
			 * unwritten extents, we need to check
			 * m_flags.  There are two conditions that
			 * indicate for initialized extents.  1) If we
			 * hit extent cache, EXT4_MAP_MAPPED flag is
			 * returned; 2) If we do a real lookup,
			 * non-flags are returned.  So we should check
			 * these two conditions.
			 */
			if (err == len && (map.m_flags & EXT4_MAP_MAPPED))
				overwrite = 1;
		}
	}

	ret = __generic_file_write_iter(iocb, from);
	inode_unlock(inode);

	if (ret > 0)
		ret = generic_write_sync(iocb, ret);

	return ret;

out:
	inode_unlock(inode);
	return ret;
}
```

可以看到申请block的入口是ext4_map_blocks,真正写文件的入口是__generic_file_write_iter

```
/**
 * __generic_file_write_iter - write data to a file
 * @iocb:	IO state structure (file, offset, etc.)
 * @from:	iov_iter with data to write
 *
 * This function does all the work needed for actually writing data to a
 * file. It does all basic checks, removes SUID from the file, updates
 * modification times and calls proper subroutines depending on whether we
 * do direct IO or a standard buffered write.
 *
 * It expects i_mutex to be grabbed unless we work on a block device or similar
 * object which does not need locking at all.
 *
 * This function does *not* take care of syncing data in case of O_SYNC write.
 * A caller has to handle it. This is mainly due to the fact that we want to
 * avoid syncing under i_mutex.
 */
ssize_t __generic_file_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
	struct file *file = iocb->ki_filp;
	struct address_space * mapping = file->f_mapping;
	struct inode	*inode = mapping->host;
	ssize_t		written = 0;
	ssize_t		err;
	ssize_t		status;

	/* We can write back this queue in page reclaim */
	current->backing_dev_info = inode_to_bdi(inode);
	err = file_remove_privs(file);
	if (err)
		goto out;

	err = file_update_time(file);//更新元数据信息中的时间信息
	if (err)
		goto out;

	if (iocb->ki_flags & IOCB_DIRECT) {   //处理direct io
		loff_t pos, endbyte;

		written = generic_file_direct_write(iocb, from);//这里调用 ext4_direct_IO
		/*
		 * If the write stopped short of completing, fall back to
		 * buffered writes.  Some filesystems do this for writes to
		 * holes, for example.  For DAX files, a buffered write will
		 * not succeed (even if it did, DAX does not handle dirty
		 * page-cache pages correctly).
		 */
		if (written < 0 || !iov_iter_count(from) || IS_DAX(inode))
			goto out;

		status = generic_perform_write(file, from, pos = iocb->ki_pos);
		/*
		 * If generic_perform_write() returned a synchronous error
		 * then we want to return the number of bytes which were
		 * direct-written, or the error code if that was zero.  Note
		 * that this differs from normal direct-io semantics, which
		 * will return -EFOO even if some bytes were written.
		 */
		if (unlikely(status < 0)) {
			err = status;
			goto out;
		}
		/*
		 * We need to ensure that the page cache pages are written to
		 * disk and invalidated to preserve the expected O_DIRECT
		 * semantics.
		 */
		endbyte = pos + status - 1;
		err = filemap_write_and_wait_range(mapping, pos, endbyte);
		if (err == 0) {
			iocb->ki_pos = endbyte + 1;
			written += status;
			invalidate_mapping_pages(mapping,
						 pos >> PAGE_SHIFT,
						 endbyte >> PAGE_SHIFT);
		} else {
			/*
			 * We don't know how much we wrote, so just return
			 * the number of bytes which were direct-written
			 */
		}
	} else {				//处理非direct io写，调用write_begin和write_end
		written = generic_perform_write(file, from, iocb->ki_pos);
		if (likely(written > 0))
			iocb->ki_pos += written;
	}
out:
	current->backing_dev_info = NULL;
	return written ? written : err;
}
```

以下是不通mount选项下ext4写实现的函数实现



```
static const struct address_space_operations ext4_aops = {
	.readpage		= ext4_readpage,
	.readpages		= ext4_readpages,
	.writepage		= ext4_writepage,
	.writepages		= ext4_writepages,
	.write_begin		= ext4_write_begin,
	.write_end		= ext4_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,
};

static const struct address_space_operations ext4_journalled_aops = {
	.readpage		= ext4_readpage,
	.readpages		= ext4_readpages,
	.writepage		= ext4_writepage,
	.writepages		= ext4_writepages,
	.write_begin		= ext4_write_begin,
	.write_end		= ext4_journalled_write_end,
	.set_page_dirty		= ext4_journalled_set_page_dirty,
	.bmap			= ext4_bmap,
	.invalidatepage		= ext4_journalled_invalidatepage,
	.releasepage		= ext4_releasepage,
	.direct_IO		= ext4_direct_IO,
	.is_partially_uptodate  = block_is_partially_uptodate,
	.error_remove_page	= generic_error_remove_page,
};

static const struct address_space_operations ext4_da_aops = {
	.readpage		= ext4_readpage,
	.readpages		= ext4_readpages,
	.writepage		= ext4_writepage,
	.writepages		= ext4_writepages,
	.write_begin		= ext4_da_write_begin,
	.write_end		= ext4_da_write_end,
	.bmap			= ext4_bmap,
	.invalidatepage		= ext4_da_invalidatepage,
	.releasepage		= ext4_releasepage,
	.direct_IO		= ext4_direct_IO,
	.migratepage		= buffer_migrate_page,
	.is_partially_uptodate  = block_is_partially_uptodate,
```

Ext4 direct io:

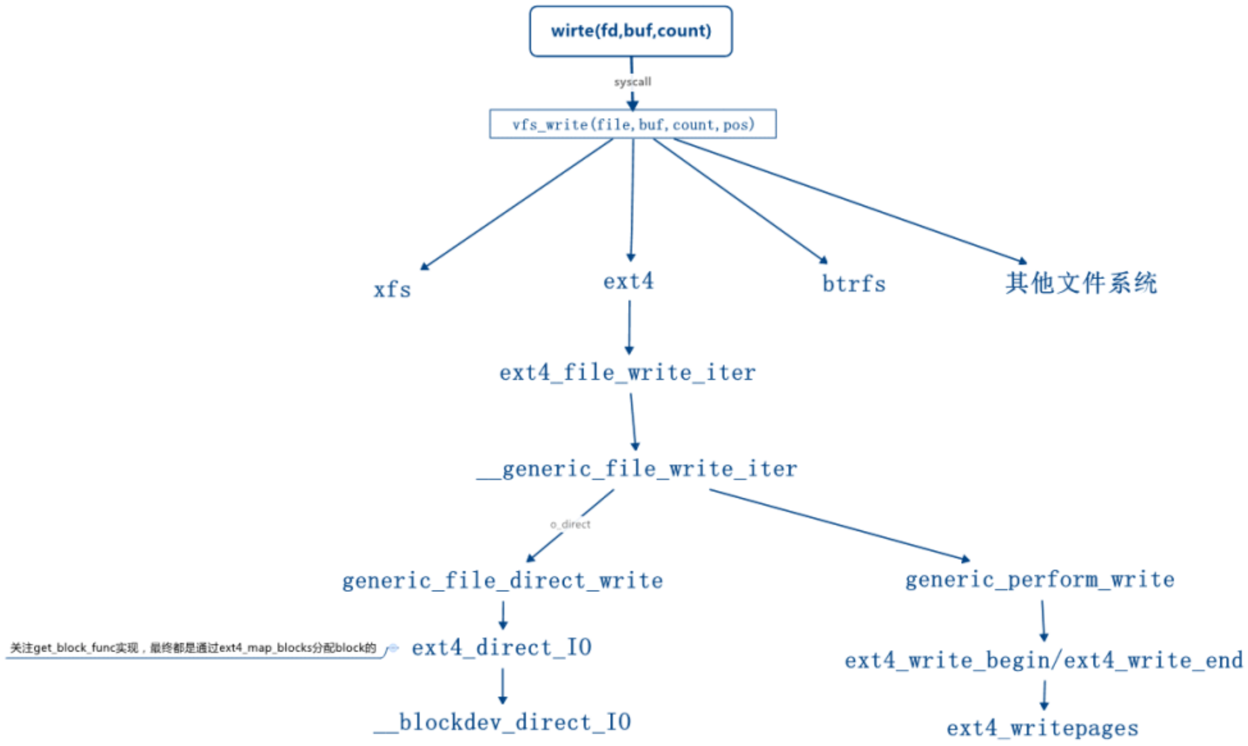
```
static ssize_t ext4_direct_IO(struct kiocb *iocb, struct iov_iter *iter)
{
	struct file *file = iocb->ki_filp;
	struct inode *inode = file->f_mapping->host;
	size_t count = iov_iter_count(iter);
	loff_t offset = iocb->ki_pos;
	ssize_t ret;

#ifdef CONFIG_EXT4_FS_ENCRYPTION
	if (ext4_encrypted_inode(inode) && S_ISREG(inode->i_mode))
		return 0;
#endif

	/*
	 * If we are doing data journalling we don't support O_DIRECT
	 */
	if (ext4_should_journal_data(inode))
		return 0;

	/* Let buffer I/O handle the inline data case. */
	if (ext4_has_inline_data(inode))
		return 0;

	trace_ext4_direct_IO_enter(inode, offset, count, iov_iter_rw(iter));
	if (iov_iter_rw(iter) == READ)
		ret = ext4_direct_IO_read(iocb, iter);//direct read
	else
		ret = ext4_direct_IO_write(iocb, iter);//direct write
	trace_ext4_direct_IO_exit(inode, offset, count, iov_iter_rw(iter), ret);
	return ret;
}
```

分析direct写：

```
/*
 * Handling of direct IO writes.
 *
 * For ext4 extent files, ext4 will do direct-io write even to holes,
 * preallocated extents, and those write extend the file, no need to
 * fall back to buffered IO.
 *
 * For holes, we fallocate those blocks, mark them as unwritten
 * If those blocks were preallocated, we mark sure they are split, but
 * still keep the range to write as unwritten.
 *
 * The unwritten extents will be converted to written when DIO is completed.
 * For async direct IO, since the IO may still pending when return, we
 * set up an end_io call back function, which will do the conversion
 * when async direct IO completed.
 *
 * If the O_DIRECT write will extend the file then add this inode to the
 * orphan list.  So recovery will truncate it back to the original size
 * if the machine crashes during the write.
 *
 */
static ssize_t ext4_direct_IO_write(struct kiocb *iocb, struct iov_iter *iter)
{
        struct file *file = iocb->ki_filp;
        struct inode *inode = file->f_mapping->host;
        struct ext4_inode_info *ei = EXT4_I(inode);
        ssize_t ret;
        loff_t offset = iocb->ki_pos;
        size_t count = iov_iter_count(iter);
        int overwrite = 0;
        get_block_t *get_block_func = NULL;
        int dio_flags = 0;
        loff_t final_size = offset + count;
        int orphan = 0;
        handle_t *handle;

        if (final_size > inode->i_size) {
                /* Credits for sb + inode write */
                handle = ext4_journal_start(inode, EXT4_HT_INODE, 2);
                if (IS_ERR(handle)) {
                        ret = PTR_ERR(handle);
                        goto out;
                }
                ret = ext4_orphan_add(handle, inode);
                if (ret) {
                        ext4_journal_stop(handle);
                        goto out;
                }
                orphan = 1;
                ei->i_disksize = inode->i_size;
                ext4_journal_stop(handle);
        }

        BUG_ON(iocb->private == NULL);

        /*
         * Make all waiters for direct IO properly wait also for extent
         * conversion. This also disallows race between truncate() and
         * overwrite DIO as i_dio_count needs to be incremented under i_mutex.
         */
        inode_dio_begin(inode);

        /* If we do a overwrite dio, i_mutex locking can be released */
        overwrite = *((int *)iocb->private);

        if (overwrite)
                inode_unlock(inode);

        /*
         * For extent mapped files we could direct write to holes and fallocate.
         *
         * Allocated blocks to fill the hole are marked as unwritten to prevent
         * parallel buffered read to expose the stale data before DIO complete
         * the data IO.
         *
         * As to previously fallocated extents, ext4 get_block will just simply
         * mark the buffer mapped but still keep the extents unwritten.
         *
         * For non AIO case, we will convert those unwritten extents to written
         * after return back from blockdev_direct_IO. That way we save us from
         * allocating io_end structure and also the overhead of offloading
         * the extent convertion to a workqueue.
         *
         * For async DIO, the conversion needs to be deferred when the
         * IO is completed. The ext4 end_io callback function will be
         * called to take care of the conversion work.  Here for async
         * case, we allocate an io_end structure to hook to the iocb.
         */
        iocb->private = NULL;
        if (overwrite)
                get_block_func = ext4_dio_get_block_overwrite;//几种获取block num的函数，核心是
                ext4_map_blocks
        else if (IS_DAX(inode)) {
                /*
                 * We can avoid zeroing for aligned DAX writes beyond EOF. Other
                 * writes need zeroing either because they can race with page
                 * faults or because they use partial blocks.
                 */
                if (round_down(offset, 1<<inode->i_blkbits) >= inode->i_size &&
                    ext4_aligned_io(inode, offset, count))
                        get_block_func = ext4_dio_get_block;
                else
                        get_block_func = ext4_dax_get_block;
                dio_flags = DIO_LOCKING;
        } else if (!ext4_test_inode_flag(inode, EXT4_INODE_EXTENTS) ||
                   round_down(offset, 1 << inode->i_blkbits) >= inode->i_size) {
                get_block_func = ext4_dio_get_block;
                dio_flags = DIO_LOCKING | DIO_SKIP_HOLES;
        } else if (is_sync_kiocb(iocb)) {
                get_block_func = ext4_dio_get_block_unwritten_sync;
                dio_flags = DIO_LOCKING;
        } else {
                get_block_func = ext4_dio_get_block_unwritten_async;
                dio_flags = DIO_LOCKING;
        }
#ifdef CONFIG_EXT4_FS_ENCRYPTION
        BUG_ON(ext4_encrypted_inode(inode) && S_ISREG(inode->i_mode));
#endif
```

```
    if (IS_DAX(inode)) {
        ret = dax_do_io(iocb, inode, iter, get_block_func,
                ext4_end_io_dio, dio_flags);
    } else
        ret = __blockdev_direct_IO(iocb, inode,//提交到block layer
                     inode->i_sb->s_bdev, iter,
                     get_block_func,
                     ext4_end_io_dio, NULL, dio_flags);

    if (ret > 0 && !overwrite && ext4_test_inode_state(inode,
                        EXT4_STATE_DIO_UNWRITTEN)) {
        int err;
        /*
         * for non AIO case, since the IO is already
         * completed, we could do the conversion right here
         */
        err = ext4_convert_unwritten_extents(NULL, inode,
                         offset, ret);
        if (err < 0)
            ret = err;
        ext4_clear_inode_state(inode, EXT4_STATE_DIO_UNWRITTEN);
    }

    inode_dio_end(inode);
    /* take i_mutex locking again if we do a ovewrite dio */
    if (overwrite)
        inode_lock(inode);

    if (ret < 0 && final_size > inode->i_size)
        ext4_truncate_failed_write(inode);

    /* Handle extending of i_size after direct IO write */
    if (orphan) {
        int err;

        /* Credits for sb + inode write */
        handle = ext4_journal_start(inode, EXT4_HT_INODE, 2);
        if (IS_ERR(handle)) {
            /* This is really bad luck. We've written the data
             * but cannot extend i_size. Bail out and pretend
             * the write failed... */
            ret = PTR_ERR(handle);
            if (inode->i_nlink)
                ext4_orphan_del(NULL, inode);

            goto out;
        }
        if (inode->i_nlink)
            ext4_orphan_del(handle, inode);
        if (ret > 0) {
            loff_t end = offset + ret;
            if (end > inode->i_size) {
                ei->i_disksize = end;
                i_size_write(inode, end);
                /*
                 * We're going to return a positive `ret'
                 * here due to non-zero-length I/O, so there's
                 * no way of reporting error returns from
                 * ext4_mark_inode_dirty() to userspace.  So
                 * ignore it.
                 */
                ext4_mark_inode_dirty(handle, inode);
            }
        }
        err = ext4_journal_stop(handle);
        if (ret == 0)
            ret = err;
    }
out:
    return ret;
}
```

最后梳理一下整个的调用流程：


wirte(fd,buf,count)
syscall
vfs_write(file, buf, count, pos)
xfs
ext4
btrfs
其他文件系统
ext4_file_write_iter
__generic_file_write_iter
o_direct
generic_file_direct_write
generic_perform_write
关注get_block_func实现，最终都是通过ext4_map_blocks分配block的
ext4_direct_IO
ext4_write_begin/ext4_write_end
__blockdev_direct_IO
ext4_writepages

其中get_block_func要实现的就是文件的offset和block的映射关系，其中会涉及block的分配，
之前有提到过核心函数就是
ext4_map_blocks：

```
static int _ext4_get_block(struct inode *inode, sector_t iblock,
                           struct buffer_head *bh, int flags)
{
        struct ext4_map_blocks map;
        int ret = 0;

        if (ext4_has_inline_data(inode))
                return -ERANGE;

        map.m_lblk = iblock;
        map.m_len = bh->b_size >> inode->i_blkbits;

        ret = ext4_map_blocks(ext4_journal_current_handle(), inode, &map,
                              flags);
        if (ret > 0) {
                map_bh(bh, inode->i_sb, map.m_pblk);
                ext4_update_bh_state(bh, map.m_flags);
                bh->b_size = inode->i_sb->s_blocksize * map.m_len;
                ret = 0;
        }
        return ret;
}

/*
 * The ext4_map_blocks() function tries to look up the requested blocks,
 * and returns if the blocks are already mapped.
 *
 * Otherwise it takes the write lock of the i_data_sem and allocate blocks
 * and store the allocated blocks in the result buffer head and mark it
 * mapped.
 *
 * If file type is extents based, it will call ext4_ext_map_blocks(),
 * Otherwise, call with ext4_ind_map_blocks() to handle indirect mapping
 * based files
 *
 * On success, it returns the number of blocks being mapped or allocated.  if
 * create==0 and the blocks are pre-allocated and unwritten, the resulting @map
 * is marked as unwritten. If the create == 1, it will mark @map as mapped.
 *
 * It returns 0 if plain look up failed (blocks have not been allocated), in
 * that case, @map is returned as unmapped but we still do fill map->m_len to
 * indicate the length of a hole starting at map->m_lblk.
 *
 * It returns the error in case of allocation failure.
 */
int ext4_map_blocks(handle_t *handle, struct inode *inode,
                    struct ext4_map_blocks *map, int flags)
{
        struct extent_status es;
        int retval;
        int ret = 0;
#ifdef ES_AGGRESSIVE_TEST
        struct ext4_map_blocks orig_map;

        memcpy(&orig_map, map, sizeof(*map));
#endif

        map->m_flags = 0;
        ext_debug("ext4_map_blocks(): inode %lu, flag %d, max_blocks %u,"
                  "logical block %lu\n", inode->i_ino, flags, map->m_len,
                  (unsigned long) map->m_lblk);
	/*
	 * ext4_map_blocks returns an int, and m_len is an unsigned int
	 */
	if (unlikely(map->m_len > INT_MAX))//处理len异常
		map->m_len = INT_MAX;

	/* We can handle the block number less than EXT_MAX_BLOCKS */
	if (unlikely(map->m_lblk >= EXT_MAX_BLOCKS))//处理block number异常
		return -EFSCORRUPTED;

	/* Lookup extent status tree firstly */
	if (ext4_es_lookup_extent(inode, map->m_lblk, &es)) {
		if (ext4_es_is_written(&es) || ext4_es_is_unwritten(&es)) {
			map->m_pblk = ext4_es_pblock(&es) +
					map->m_lblk - es.es_lblk;
			map->m_flags |= ext4_es_is_written(&es) ?
					EXT4_MAP_MAPPED : EXT4_MAP_UNWRITTEN;
			retval = es.es_len - (map->m_lblk - es.es_lblk);
			if (retval > map->m_len)
				retval = map->m_len;
			map->m_len = retval;
		} else if (ext4_es_is_delayed(&es) || ext4_es_is_hole(&es)) {
			map->m_pblk = 0;
			retval = es.es_len - (map->m_lblk - es.es_lblk);
			if (retval > map->m_len)
				retval = map->m_len;
			map->m_len = retval;
			retval = 0;
		} else {
			BUG_ON(1);
		}
#ifdef ES_AGGRESSIVE_TEST
		ext4_map_blocks_es_recheck(handle, inode, map,
					   &orig_map, flags);
#endif
		goto found;
	}

	/*
	 * Try to see if we can get the block without requesting a new
	 * file system block.
	 */
	down_read(&EXT4_I(inode)->i_data_sem);
	if (ext4_test_inode_flag(inode, EXT4_INODE_EXTENTS)) {
		retval = ext4_ext_map_blocks(handle, inode, map, flags &
					     EXT4_GET_BLOCKS_KEEP_SIZE);
	} else {
		retval = ext4_ind_map_blocks(handle, inode, map, flags &
					     EXT4_GET_BLOCKS_KEEP_SIZE);
	}
	if (retval > 0) {
		unsigned int status;

		if (unlikely(retval != map->m_len)) {
			ext4_warning(inode->i_sb,
				     "ES len assertion failed for inode "
				     "%lu: retval %d != map->m_len %d",
				     inode->i_ino, retval, map->m_len);
			WARN_ON(1);
		}

		status = map->m_flags & EXT4_MAP_UNWRITTEN ?
				EXTENT_STATUS_UNWRITTEN : EXTENT_STATUS_WRITTEN;
		if (!(flags & EXT4_GET_BLOCKS_DELALLOC_RESERVE) &&
		    !(status & EXTENT_STATUS_WRITTEN) &&
		    ext4_find_delalloc_range(inode, map->m_lblk,
					     map->m_lblk + map->m_len - 1))
			status |= EXTENT_STATUS_DELAYED;
		ret = ext4_es_insert_extent(inode, map->m_lblk,
					    map->m_len, map->m_pblk, status);
		if (ret < 0)
			retval = ret;
	}
	up_read((&EXT4_I(inode)->i_data_sem));

found:
	if (retval > 0 && map->m_flags & EXT4_MAP_MAPPED) {
		ret = check_block_validity(inode, map);
		if (ret != 0)
			return ret;
	}

	/* If it is only a block(s) look up */
	if ((flags & EXT4_GET_BLOCKS_CREATE) == 0)
		return retval;

	/*
	 * Returns if the blocks have already allocated
	 *
	 * Note that if blocks have been preallocated
	 * ext4_ext_get_block() returns the create = 0
	 * with buffer head unmapped.
	 */
	if (retval > 0 && map->m_flags & EXT4_MAP_MAPPED)
		/*
		 * If we need to convert extent to unwritten
		 * we continue and do the actual work in
		 * ext4_ext_map_blocks()
		 */
		if (!(flags & EXT4_GET_BLOCKS_CONVERT_UNWRITTEN))
			return retval;

	/*
	 * Here we clear m_flags because after allocating an new extent,
	 * it will be set again.
	 */
	map->m_flags &= ~EXT4_MAP_FLAGS;

	/*
	 * New blocks allocate and/or writing to unwritten extent
```

```
         * will possibly result in updating i_data, so we take
         * the write lock of i_data_sem, and call get_block()
         * with create == 1 flag.
         */
        down_write(&EXT4_I(inode)->i_data_sem);

        /*
         * We need to check for EXT4 here because migrate
         * could have changed the inode type in between
         */
        if (ext4_test_inode_flag(inode, EXT4_INODE_EXTENTS)) {
                retval = ext4_ext_map_blocks(handle, inode, map, flags);
        } else {
                retval = ext4_ind_map_blocks(handle, inode, map, flags);

                if (retval > 0 && map->m_flags & EXT4_MAP_NEW) {
                        /*
                         * We allocated new blocks which will result in
                         * i_data's format changing.  Force the migrate
                         * to fail by clearing migrate flags
                         */
                        ext4_clear_inode_state(inode, EXT4_STATE_EXT_MIGRATE);
                }

                /*
                 * Update reserved blocks/metadata blocks after successful
                 * block allocation which had been deferred till now. We don't
                 * support fallocate for non extent files. So we can update
                 * reserve space here.
                 */
                if ((retval > 0) &&
                        (flags & EXT4_GET_BLOCKS_DELALLOC_RESERVE))
                        ext4_da_update_reserve_space(inode, retval, 1);
        }

        if (retval > 0) {
                unsigned int status;

                if (unlikely(retval != map->m_len)) {
                        ext4_warning(inode->i_sb,
                                     "ES len assertion failed for inode "
                                     "%lu: retval %d != map->m_len %d",
                                     inode->i_ino, retval, map->m_len);
                        WARN_ON(1);
                }

                /*
                 * We have to zeroout blocks before inserting them into extent
                 * status tree. Otherwise someone could look them up there and
                 * use them before they are really zeroed. We also have to
                 * unmap metadata before zeroing as otherwise writeback can
                 * overwrite zeros with stale data from block device.
                 */
                if (flags & EXT4_GET_BLOCKS_ZERO &&
                    map->m_flags & EXT4_MAP_MAPPED &&
                    map->m_flags & EXT4_MAP_NEW) {
                        ext4_lblk_t i;

                        for (i = 0; i < map->m_len; i++) {
                                unmap_underlying_metadata(inode->i_sb->s_bdev,
                                                         map->m_pblk + i);
                        }
                        ret = ext4_issue_zeroout(inode, map->m_lblk,
                                                 map->m_pblk, map->m_len);
                        if (ret) {
                                retval = ret;
                                goto out_sem;
                        }
                }

                /*
                 * If the extent has been zeroed out, we don't need to update
                 * extent status tree.
                 */
                if ((flags & EXT4_GET_BLOCKS_PRE_IO) &&
                    ext4_es_lookup_extent(inode, map->m_lblk, &es)) {
                        if (ext4_es_is_written(&es))
                                goto out_sem;
                }
                status = map->m_flags & EXT4_MAP_UNWRITTEN ?
                                EXTENT_STATUS_UNWRITTEN : EXTENT_STATUS_WRITTEN;
                if (!(flags & EXT4_GET_BLOCKS_DELALLOC_RESERVE) &&
                    !(status & EXTENT_STATUS_WRITTEN) &&
                    ext4_find_delalloc_range(inode, map->m_lblk,
                                             map->m_lblk + map->m_len - 1))
                        status |= EXTENT_STATUS_DELAYED;
                ret = ext4_es_insert_extent(inode, map->m_lblk, map->m_len,
                                            map->m_pblk, status);
                if (ret < 0) {
                        retval = ret;
                        goto out_sem;
                }
        }

out_sem:
        up_write((&EXT4_I(inode)->i_data_sem));
        if (retval > 0 && map->m_flags & EXT4_MAP_MAPPED) {
                ret = check_block_validity(inode, map);
                if (ret != 0)
                        return ret;

                /*
                 * Inodes with freshly allocated blocks where contents will be
                 * visible after transaction commit must be on transaction's
                 * ordered data list.
                 */
                if (map->m_flags & EXT4_MAP_NEW &&
                    !(map->m_flags & EXT4_MAP_UNWRITTEN) &&
                    !(flags & EXT4_GET_BLOCKS_ZERO) &&
                    !IS_NOQUOTA(inode) &&
                    ext4_should_order_data(inode)) {
                        if (flags & EXT4_GET_BLOCKS_IO_SUBMIT)
                ret = ext4_jbd2_inode_add_wait(handle, inode);
            else
                ret = ext4_jbd2_inode_add_write(handle, inode);
            if (ret)
                return ret;
        }
    }
    return retval;
}
```