write指令跟踪分析

1、比如一个write指令，从用户态到内核态的过程，它经过libc库，glibc中的'write'函数实现可能直接使用syscall函数来发起系统调用

```
ssize_t write(int fd, const void *buf, size_t count) {
    return syscall(SYS_write, fd, buf, count);
}
```

2、系统调用的触发

当write函数被调用时，它会触发一个系统调用，系统调用号为SYS_write。通过这个系统调用号会被传递到内核，内核会根据这个号来确定需要执行的系统调用函数。

linux-5.10.102/arch/x86/entry/syscalls/syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0   common  read            sys_read
1   common  write           sys_write
2   common  open            sys_open
3   common  close           sys_close
4   common  stat            sys_newstat
5   common  fstat           sys_newfstat
6   common  lstat           sys_newlstat
7   common  poll            sys_poll
8   common  lseek           sys_lseek
9   common  mmap            sys_mmap
10  common  mprotect        sys_mprotect
11  common  munmap          sys_munmap
```

注：这里只列举了一部分关于x86架构的系统调用号和系统调用函数

每一列的含义如下：

- 编号（第一列）：这是系统调用的编号，用户空间程序通过这个编号与内核交互调用相应的系统调用函数。
- 适用架构（第二列）：这一列指定了该系统调用适用于哪种架构，common表示适用于所有架构，64表示就是只适用于64位的架构。
- 系统调用名称（第三列）：这是用户态程序使用的系统调用名称。
- 实现函数（第四列）：这是在内核中实际实现该系统调用的函数名。

在编译过程中，syscall_64.tal 文件通过内核构建系统中的脚本生成最终的syscalls_64.h文件，该文件包含系统调用与处理函数的映射。

3、系统调用的入口代码

入口负责处理从用户空间到内核空间的转换。对于x86_64架构，入口代码通常在linux-5.10.102/arch/x86/entry/entry_64.S文件，它包含了系统调用入口的汇编代码。

```
SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

    /* Construct struct pt_regs on stack */
    pushq   $__USER_DS              /* pt_regs->ss */
    pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)   /* pt_regs->sp */
    pushq   %r11                    /* pt_regs->flags */
    pushq   $__USER_CS              /* pt_regs->cs */
    pushq   %rcx                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
    pushq   %rax                    /* pt_regs->orig_ax */

    PUSH_AND_CLEAR_REGS rax=$-ENOSYS

    /* IRQs are off. */
    movq    %rax, %rdi
    movq    %rsp, %rsi
    call    do_syscall_64       /* returns with IRQs disabled */
```

代码的总体作用：

- 构建struct pt_regs:这段代码的主要作用是构建struct pt_regs结构体，并将当前用户态的寄存器状态保存在栈中。struct pt_regs是一个保存进程上下文的结构体，包含所有重要的寄存器值，当发生系统调用、异常或中断时，内核需要保存这些状态以便在返回用户态时恢复。
- 准备并调用：do_syscall_64：在保存了所有寄存器状态后，代码将系统调用号和struct pt_regs的地址传递给do_syscall_64的函数。这个函数会从系统调用表中查找对应的处理函数并执行。
- 系统调用的执行：do_syscall_64会根据传入的系统调用号，查找并调用对应的系统调用处理函数。处理函数会执行相应的内核操作，之后返回结果。

4、这是在64位x86结构下处理系统调用的关键函数，linux-5.10.102/arch/x86/entry/common.c

```
#ifdef CONFIG_X86_64
__visible noinstr void do_syscall_64(unsigned long nr, struct pt_regs *regs)
{
    nr = syscall_enter_from_user_mode(regs, nr);

    instrumentation_begin();
    if (likely(nr < NR_syscalls)) {
        nr = array_index_nospec(nr, NR_syscalls);
        regs->ax = sys_call_table[nr](regs);
#ifdef CONFIG_X86_X32_ABI
    } else if (likely((nr & __X32_SYSCALL_BIT) &&
              (nr & ~__X32_SYSCALL_BIT) < X32_NR_syscalls)) {
        nr = array_index_nospec(nr & ~__X32_SYSCALL_BIT,
                    X32_NR_syscalls);
        regs->ax = x32_sys_call_table[nr](regs);
#endif
    }
    instrumentation_end();
    syscall_exit_to_user_mode(regs);
}
```

```
#endif
```

（1）do_syscall_64是处理64位系统调用的函数，接收两个参数：

- nr：系统调用号
- regs:指向包含系统调用相关寄存器状态得到pt_regs结构体的指针

（2）regs->ax = sys_call_table[nr] (regs):根据系统调用号，从sys_call_table表中获取处理函数，并调用它。sys_call_table是一个函数指针数组，包含了所有系统调用的处理函数。

linux-5.10.102/arch/x86/include/asm/syscall.h:这个文件声明了 sys_call_table 及其相关的表。它通常包含系统调用表的外部声明（extern）以及宏定义。

```
extern const sys_call_ptr_t sys_call_table[];
```

linux-5.10.102/arch/x86/entry/syscall_64.c

```
#define __SYSCALL_64(nr, sym) [nr] = __x64_##sym,
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    /*
     * Smells like a compiler bug -- it doesn't work
     * when the & below is removed.
     */
    [0 ... __NR_syscall_max] = &__x64_sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

详细解释：

- asmlinkage:是一个宏，用于确保函数参数的传递方式符合系统调用的约定。它通常用来标记系统调用函数和相关表的声明，使得编译器不会对参数的传递进行优化。
- sys_call_ptr_t:是一个类型定义，通常定义为 void (*)(void) 或类似类型，表示系统调用表中的每个元素是一个指向系统调用处理函数的指针。
- sys_call_table:是一个数组，其大小为 **NR_syscall_max + 1，其中** NR_syscall_max 是系统支持的最大系统调用号。这是一个函数指针数组，用于将系统调用号映射到相应的处理函数。
- *_SYSCALL_64宏的作用：是将sys_call_table数组中的索引'nr'(系统调用号)与处理函数 _x64##*sym关联起来。

例如：

```
__SYSCALL_64(0, sys_read)
```

展开后变为：

```
[0] = __x64_sys_read,
```

linux-5.10.102/arch/x86/um/shared/sysdep/syscalls_64.h：

```
/*
 * Copyright 2003 PathScale, Inc.
 *
 * Licensed under the GPL
 */
```

```
#ifndef __SYSDEP_X86_64_SYSCALLS_H__
#define __SYSDEP_X86_64_SYSCALLS_H__

#include <linux/msg.h>
#include <linux/shm.h>

typedef long syscall_handler_t(void);

extern syscall_handler_t *sys_call_table[];

#define EXECUTE_SYSCALL(syscall, regs) \
    (((long (*)(long, long, long, long, long, long)) \
      (*sys_call_table[syscall]))(UPT_SYSCALL_ARG1(&regs->regs), \
                      UPT_SYSCALL_ARG2(&regs->regs), \
                      UPT_SYSCALL_ARG3(&regs->regs), \
                      UPT_SYSCALL_ARG4(&regs->regs), \
                      UPT_SYSCALL_ARG5(&regs->regs), \
                      UPT_SYSCALL_ARG6(&regs->regs)))

extern long old_mmap(unsigned long addr, unsigned long len,
             unsigned long prot, unsigned long flags,
             unsigned long fd, unsigned long pgoff);
extern syscall_handler_t sys_modify_ldt;
extern syscall_handler_t sys_arch_prctl;

#endif
```

1、syscall参数：syscall是传递给宏的系统调用号，它用于从sys_call_table数组中索引到对应的系统调用处理函数。

2、syscall_table:是一个数组，数组的每个元素都是一个指向系统调用处理函数的指针。

syscall_call_table[syscall]：就是系统调用号对应的处理函数指针。

3、UPT_SYSCALL_ARG1(&regs->regs)等：这些宏（如"UPT_SYSCALL_ARG1"、"UPT_SYSCALL_ARG2"等）用于从"regs"结构体中提取系统调用的参数。"regs"代表系统调用的寄存器状态或上下文。

UPT_SYSCALL_ARG1(&regs->regs)：可能会返回系统调用第一个参数的值

UPT_SYSCALL_ARG2(&regs->regs)：返回第二个参数，以此类推。

也就是说这些参数，其实在用户态的时候就已经确定了。

x86_64架构：参数通过 `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9` 寄存器传递（前六个参数），更多参数通过栈传递。

linux-5.10.102/include/linux/sched.h:定义了struct task_struct(可以看到每个进程里面都有这个文件描述符表)

```
/* Filesystem information: */
    struct fs_struct        *fs; //进程挂载的文件系统

    /* Open file information: */
    struct files_struct     *files;  //打开的文件
```

linux-5.10.102/include/linux/fdtable.h： （ struct files_struct { } ）

```
struct files_struct {
  /*
   * read mostly part
   */
    atomic_t count;
    bool resize_in_progress;
    wait_queue_head_t resize_wait;

    struct fdtable __rcu *fdt;
    struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    unsigned int next_fd;
    unsigned long close_on_exec_init[1];
    unsigned long open_fds_init[1];
    unsigned long full_fds_bits_init[1];
    struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
```

fdt 和 fdtable 的关系：

- fdt是一个指向fdtable结构体的RCU指针，通常指向实际的文件描述符表。这个指针在files_struct中是为了提供对当前文件描述符的快速访问，而不需要操作fdtable。
- fdtab是一个嵌入在 files_struct中的fdtable结构体实例。这个字段通常用作一个备份或用于某些特定的目的，例如在调整文件描述符大小时。

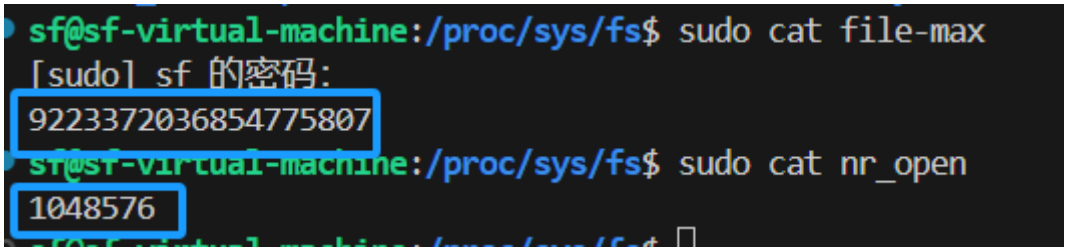
linux-5.10.102/include/linux/fdtable.h:(struct fdtable {} 进程的文件描述符表)

```
struct fdtable {
    unsigned int max_fds;  //表示文件描述符的最大数量
    struct file __rcu **fd;      /* current fd array */  //指向当前文件描述符数组的指
针
    unsigned long *close_on_exec; //表示在执行exec()系统调用时关闭对应的文件描述符
    unsigned long *open_fds;  //表示当前打开的文件描述符
    unsigned long *full_fds_bits; //表示所有可能的文件描述符状态
    struct rcu_head rcu; //RCU头部，用于RCU同步机制
};
```

文件描述符管理：

1、文件描述符的数量是由操作系统决定的，默认情况下通常是由0开始，直到某个最大值（如/proc/sys/fs/file-max所示的最大文件句柄数）。通常用的三个文件描述符是：

- 标准输入（stdin），文件描述符0；
- 标准输出（stdout）,文件描述符1；
- 标准错误输出（stderr），文件描述符2.



在/proc/sys/fs目录下查看file-max文件的内容时，输出显示为上图圈出来的部分，这个数字实际上是2^63-1,这是64位系统上文件描述符的最大值，这意味着在理论上，每个进程都可以有最多2^63-1个打开的文件描述符。

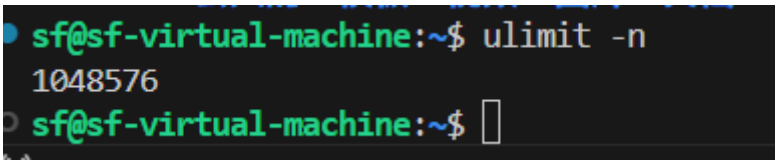
然而，实际上操作系统为了节省资源和提高性能，并不会允许进程打开如此多的进程描述符。通常每个人进程的文件描述符数量都会受到更严格的限制。

2、系统级限制：

```
/proc/sys/fs/file-max:表示的是整个系统可以支持的最大文件描述符数量
/proc/sys/fs/nr-open:表示当前系统实际可以打开的文件描述符数量。
```

3、进程级限制：

- 每个进程可以打开的文件描述符数量通常由操作系统内核维护，并且可以通过/proc//limits文件查看。
- 在linux中，还可以通过ulimit -n命令查看和设置当前shell会话中的最大打开文件数限制。



linux-5.10.102/include/linux/fs.h:在这里面有这个 （struct file {} 结构体，文件描述符指向的这个具体的文件对象）

```
struct file {
    union {
        struct llist_node    fu_llist;  // Used for file list operations
        struct rcu_head     fu_rcuhead; // Used for RCU (Read-Copy-Update)
synchronization
    } f_u;

    struct path           f_path;      // Contains the file's path information
    struct inode         *f_inode;     // Cached inode pointer for the file
    const struct file_operations *f_op; // Pointer to file operations structure

    spinlock_t            f_lock;      // Protects f_ep_links, f_flags; should
not be used in IRQ context
    enum rw_hint          f_write_hint; // Hints for the file write behavior
    atomic_long_t        f_count;      // Reference count for the file
    unsigned int         f_flags;      // File descriptor flags (e.g., O_RDONLY,
O_WRONLY)
    fmode_t              f_mode;       // File mode (e.g., read/write)
    struct mutex         f_pos_lock;   // Mutex for synchronizing file position
    loff_t               f_pos;        // File offset for reading/writing
    struct fown_struct   f_owner;      // Owner information (e.g., signal
handling)
```

```
    const struct cred    *f_cred;      // Credentials for access control

    struct file_ra_state f_ra;         // File read-ahead state

    u64                  f_version;    // Version number of the file (for caching
purposes)
#ifdef CONFIG_SECURITY
    void                 *f_security;  // Security-related information
#endif
    void                 *private_data; // Private data used by drivers or
filesystems

#ifdef CONFIG_EPOLL
    struct list_head     f_ep_links;   // Links for epoll events
    struct list_head     f_tfile_llink; // Links for tracking file descriptors
#endif /* CONFIG_EPOLL */

    struct address_space *f_mapping;   // Address space mapping for file memory
    errseq_t             f_wb_err;     // Error sequence for writeback
    errseq_t             f_sb_err;     // Error sequence for syncfs
} __randomize_layout __attribute__((aligned(4)));
```

举例：如果我们现在在shell下输入这个 echo "what 's your name?" > 123.txt | 现在来分析一下这个代码的执行过程。

1、解析命令

shell（如bash）解析这行命令，shell会识别出echo是一个外部命令，并且注意到重定向符号 > ,表示输出需要重定向到文件123.txt，而不是默认的标准输出（终端）。

shell将命令分为两部分：echo "what 's your name?" 是要执行后的命令，而 > 123.txt是重定向操作。

2、文件重定向准备

在执行命令前，shell会准备重定向：

- shell识别 '>' 符号意味要将标准输出（文件描述符 1）重定向到一个文件 123.txt
- shell会检查文件 123.txt 是否存在，如果存在，文件会被清空（ > 符号默认是覆盖写 模式）。如果文件不存在，shell会使用open系统调用来创建文件 123.txt。
- 在open()系统调用中，文件会以 O_WRONLY(只写)和O_CREATE（创建）模式打开。
- 此时，文件描述符 1 （标准输出）会被重定向到 123.txt，这意味着所有的输出将写入该文件。

  工作流程：

  1、open()系统调用：首先，shell会使用open（）打开或创建文件123.txt，得到一个新的文件描述符（假设是 fd = 3）

  ```
  int fd = open("123.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  ```

  2、dup2()系统调用：然后，shell使用dup2()将文件描述符3复制到标准输出（文件描述符1）上，这样后续的所有写入标准输出的操作都会写入 123.txt 文件。

  ```
  dup2(fd, 1);
  ```

这一步是关键，dup2()将fd复制到标准输出（即文件描述符1），使得标准输出指向文件123.txt。这样，当echo命令输出文本时，文件将写入文件123.txt 而不是输出到终端。

3、关闭文件描述符：完成重定向后，可以关闭原始文件描述符。

```
close(fd);
```

3、创建子进程、在子进程中执行命令

shell调用fork()创建一个新的子进程，子进程继承了父进程的资源（如文件描述符、环境变量等）。

子进程调用了exec()系统调用来执行echo命令。exec()会替换子进程的映像，使其开始执行echo命令的代码。这意味着子进程现在不再是shell进程，而是echo命令进程。

内部命令

内部命令确实是shell（如bash)代码的一部分，由shell直接解释和执行，而不创建子进程。这意味着内部命令的执行速度通常更快，因为不需要进行进程切换和资源复制。

- cd:改变当前工作目录。
- export：设置环境变量。
- source或 . :读取并执行脚本文件。
- eval：执行经过解释的命令。
- alias：创建命令别名。
- history：显示历史命令记录。
- builtin:执行内置命令。

外部命令

外部命令是指那些不是shell内置的一部分，而是独立的程序，通常存储在文件系统中的某个路径下（如/bin、/usr/bin等）。这些命令需要通过创建子进程来执行。

- ls:列出目录内容。
- echo:输出文本到标准输出。
- grep：搜索文本模式。
- cat：连接和打印文件内容。
- chmod：更改文件权限。

linux-5.10.102/include/linux/syscalls.h：定义了sys_fork的函数类型

```
asmlinkage long sys_fork(void);
```

linux-5.10.102/kernel/fork.c:这里面生成了这个fork系统调用的代码

```
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    struct kernel_clone_args args = {
        .exit_signal = SIGCHLD,
    };

    return kernel_clone(&args);
#else
    /* can not support in nommu mode */
```

```
    return -EINVAL;
#endif
}
#endif
```

```
pid_t kernel_clone(struct kernel_clone_args *args)
{
    u64 clone_flags = args->flags;
    struct completion vfork;
    struct pid *pid;
    struct task_struct *p;
    int trace = 0;
    pid_t nr;

    /*
     * For legacy clone() calls, CLONE_PIDFD uses the parent_tid argument
     * to return the pidfd. Hence, CLONE_PIDFD and CLONE_PARENT_SETTID are
     * mutually exclusive. With clone3() CLONE_PIDFD has grown a separate
     * field in struct clone_args and it still doesn't make sense to have
     * them both point at the same memory location. Performing this check
     * here has the advantage that we don't need to have a separate helper
     * to check for legacy clone().
     */
    if ((args->flags & CLONE_PIDFD) &&
        (args->flags & CLONE_PARENT_SETTID) &&
        (args->pidfd == args->parent_tid))
        return -EINVAL;

    /*
     * Determine whether and which event to report to ptracer.  When
     * called from kernel_thread or CLONE_UNTRACED is explicitly
     * requested, no event is reported; otherwise, report if the event
     * for the type of forking is enabled.
     */
    if (!(clone_flags & CLONE_UNTRACED)) {
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK;
        else if (args->exit_signal != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;

        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }

    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    add_latent_entropy();

    if (IS_ERR(p))
        return PTR_ERR(p);

    /*
     * Do this prior waking up the new thread - the thread pointer
     * might get invalid after that point, if the thread exits quickly.
```

```
     */
    trace_sched_process_fork(current, p);

    pid = get_task_pid(p, PIDTYPE_PID);
    nr = pid_vnr(pid);

    if (clone_flags & CLONE_PARENT_SETTID)
        put_user(nr, args->parent_tid);

    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
        get_task_struct(p);
    }

    wake_up_new_task(p);

    /* forking complete and child started to run, tell ptracer */
    if (unlikely(trace))
        ptrace_event_pid(trace, pid);

    if (clone_flags & CLONE_VFORK) {
        if (!wait_for_vfork_done(p, &vfork))
            ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
    }

    put_pid(pid);
    return nr;
}
```

```
static __latent_entropy struct task_struct *copy_process(
                    struct pid *pid,
                    int trace,
                    int node,
                    struct kernel_clone_args *args)
{
    int pidfd = -1, retval;
    struct task_struct *p;
    struct multiprocess_signals delayed;
    struct file *pidfile = NULL;
    u64 clone_flags = args->flags;
    struct nsproxy *nsp = current->nsproxy;

/*这里初始化了一些局部变量，包括pidfd、retval、p、delayed、pidfile等，并且获取了克隆标志
clone_flags和当前进程的命名空间代理nsp*/

......
......
......
    /* Perform scheduler related setup. Assign this task to a CPU. */(调度初始化，
为新进程分配CPU)
retval = sched_fork(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_policy;
    /*初始化共享内存*/
```

```
    shm_init_task(p);
```

函数签名

- struct task_struct *:返回一个新的 task_struct ， 即新创建的进程的结构体。
- struct pid *pid : 指定新进程的PID。
- int trace : 是否跟踪进程创建。
- int node： 指定进程将在哪个节点上运行。
- struct kernel_clone_args *args : 包含了克隆标志和其他参数的结构体。

linux-5.10.102/include/linux/syscalls.h：定义了sys_write的函数类型

sys_write()是系统调用的入口函数。

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

linux-5.10.102/fs/read_write.c:定义了sys_write函数原型：

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    return ksys_write(fd, buf, count);
}
```

SYSCALL_DEFINE3是一个宏，用于定义带有三个参数的系统调用，这个宏会展开成一个实际的系统调用函数，其实的参数和名称已经被宏替换掉了。

SYSCALL_DEFINE3展开后生成的函数原型是：

```
asmlinkage long sys_write(unsigned int fd, const char __user *buf, size_t count)
{
    return ksys_write(fd, buf, count);
}
```

linux-5.10.102/fs/write_read.c:分析ksys_write这个代码(ksys_write)

```
ssize_t ksys_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, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_write(f.file, buf, count, ppos);
        if (ret >= 0 && ppos)
            f.file->f_pos = pos;
        fdput_pos(f);
    }
```

```
    return ret;
}
```

linux-5.10.102/include/linux/file.h：(struct fd{})

```
struct fd {
    struct file *file;
    unsigned int flags;
};
```

linux-5.10.102/fs/fs.h:(stuct file{})

```
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;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t      f_lock;
    enum rw_hint        f_write_hint;
    atomic_long_t       f_count;
    unsigned int        f_flags;
    fmode_t         f_mode;
    struct mutex        f_pos_lock;
    loff_t          f_pos;
    struct fown_struct  f_owner;
    const struct cred   *f_cred;
    struct file_ra_state    f_ra;

    u64         f_version;
#ifdef CONFIG_SECURITY
    void            *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void            *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head    f_ep_links;
    struct list_head    f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space    *f_mapping;
    errseq_t        f_wb_err;
    errseq_t        f_sb_err; /* for syncfs */
} __randomize_layout
  __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

linux-5.10.102/fs/read_write.c:（vfs_write)

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

    if (!(file->f_mode & FMODE_WRITE))  //检查文件是否处于可写模式（FMODE_WRITE）
        return -EBADF;
    if (!(file->f_mode & FMODE_CAN_WRITE))  //检查文件是否允许写操作
(FMODE_CAN_WRITE)
        return -EINVAL;
    if (unlikely(!access_ok(buf, count))) //检查用户缓冲区是否可访问（access_ok）
        return -EFAULT;

    ret = rw_verify_area(WRITE, file, pos, count);
    if (ret)
        return ret;
    if (count > MAX_RW_COUNT)
        count =  MAX_RW_COUNT;
    file_start_write(file); //函数准备写操作
    if (file->f_op->write)
        ret = file->f_op->write(file, buf, count, pos);
    else if (file->f_op->write_iter)
        ret = new_sync_write(file, buf, count, pos);
    else
        ret = -EINVAL;
    if (ret > 0) {
        fsnotify_modify(file); //函数通知文件系统修改
        add_wchar(current, ret); //更新当前进程的写字符计数
    }
    inc_syscw(current); //增加系统调用写计数
    file_end_write(file);  //函数结束写操作
    return ret;
}
```

这里，file->f_op->write(file, buf, count, pos)和new_sync_write(file, buf, count, pos)都是同步操作，只是这个write_iter提供了更灵活的接口。它在这里通过new_sync_write(file, buf, count, pos)进行处理。因此，也被视为异步操作。

不对，这里需要注意一个情况就是，我们这里写的文档考虑的是写入，这个写入文件的时候，其实就是相当于已经有了一个具体的文件，然后给这个具体的文件里面去写入东西。

这里的这个f_op其实不用纠结，就是open系统调用的时候，它已经设置好了这个file文件对象，已经将这个f_op要指向的方法确定了，我们这里文件都是在ext4文件系统里面，所以这个ext4文件系统（底下有源码分析），得知，这个vfs_write函数里面最后肯定调用的是这个file->f_op->write_iter()再到这个new_sync_write(file, buf, count, pos);函数了。

因为我们的ext4文件系统确实是支持这个异步写入的。因此在创建一个文件的时候，就已经确定好了这个file对象里面的这个f_op，它其实是指向这个ext4文件系统里面的operations的。所以它基本都走的是这个write_iter方法了。

传统的这个写入是同步，而ext4的写入是异步写入。

new_sync_write(file, buf, count, pos)的实现：在sync_write(file, buf, count, pos)中，它会准备调用write_iter。通常如下所示：

```
ret = file->f_op->write_iter(file, &kiocb, &iter);
```

这里的file->f_op->write_iter实际上是指向ext4_file_write_iter的指针，因此，调用将跳转到ext4_file_write_iter函数，这是ext4文件系统实现的写入操作函数，负责处理具体的写入逻辑。（在下文有这个调用链的具体分析）

linux-5.10.102/include/linux/fs.h:（这个文件定义了 VFS（虚拟文件系统）层的各种数据结构和函数原型）

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
            loff_t, size_t, unsigned int);
    loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                   struct file *file_out, loff_t pos_out,
```

```
                    loff_t len, unsigned int remap_flags);
    int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

linux-5.10.102/fs/ext4/ext4.h：（`file_operations`结构体可能会在`fs/ext4/`目录中的源文件中被初始化。）

```
extern const struct file_operations ext4_file_operations;
```

linux-5.10.102/fs/ext4/file.c:（进行具体的初始化）

```
const struct file_operations ext4_file_operations = {
    .llseek     = ext4_llseek,
    .read_iter  = ext4_file_read_iter,
    .write_iter = ext4_file_write_iter,
    .iopoll     = iomap_dio_iopoll,
    .unlocked_ioctl = ext4_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl   = ext4_compat_ioctl,
#endif
    .mmap       = ext4_file_mmap,
    .mmap_supported_flags = MAP_SYNC,
    .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,
};
```

linux-5.10.102/fs/ext4/file.c:（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);

    if (unlikely(ext4_forced_shutdown(EXT4_SB(inode->i_sb))))
        return -EIO;

#ifdef CONFIG_FS_DAX
    if (IS_DAX(inode))
        return ext4_dax_write_iter(iocb, from);
#endif
    if (iocb->ki_flags & IOCB_DIRECT)
        return ext4_dio_write_iter(iocb, from);
    else
        return ext4_buffered_write_iter(iocb, from);
}
```

1、处理DAX（Direct Access）

如果启用了DAX，并且inode是DAX类型，调用 ext4_dax_write_iter进行写入，DAX允许文件直接映射到设备，绕过页缓存，提供更好的性能。

2、处理直接I/O

如果iocb标记了直接I/O，则调用ext4_dio_write_iter。直接I/O通常用于避免页缓存的干扰，直接将数据写入设备。

3、处理缓冲I/O

如果没有使用直接I/O，则调用ext4_buffered_write_iter，这会使用页缓存来进行写入。缓冲写入提供了更高的效率，因为它允许数据在后台异步写入。

linux-5.10.102/fs/ext4/file.c:（ext4_buffered_write_iter）

```
static ssize_t ext4_buffered_write_iter(struct kiocb *iocb,
                    struct iov_iter *from)
{
    ssize_t ret;
    struct inode *inode = file_inode(iocb->ki_filp);

    if (iocb->ki_flags & IOCB_NOWAIT)
        return -EOPNOTSUPP;

    ext4_fc_start_update(inode);
    inode_lock(inode);
    ret = ext4_write_checks(iocb, from);
    if (ret <= 0)
        goto out;

    current->backing_dev_info = inode_to_bdi(inode);
    ret = generic_perform_write(iocb->ki_filp, from, iocb->ki_pos);
    current->backing_dev_info = NULL;

out:
    inode_unlock(inode);
    ext4_fc_stop_update(inode);
    if (likely(ret > 0)) {
        iocb->ki_pos += ret;
        ret = generic_write_sync(iocb, ret);
    }

    return ret;
}
```

调用generic_perform_write执行真正的写入操作。将数据写入文件系统的缓冲区里面。

这里如果是同步写入的话，就调用的是generic_write_sync()函数。

linux-5.10.102/include/linux/fs.h:(generic_write_sync())

```
static inline ssize_t generic_write_sync(struct kiocb *iocb, ssize_t count)
{
    if (iocb->ki_flags & IOCB_DSYNC) {
        int ret = vfs_fsync_range(iocb->ki_filp,
                iocb->ki_pos - count, iocb->ki_pos - 1,
                (iocb->ki_flags & IOCB_SYNC) ? 0 : 1);
        if (ret)
            return ret;
    }

    return count;
}
```

- generic_write_sync负责在同步写入的情况下，将脏数据持久化到磁盘。
- 它通过调用vfs_fsync_range确保数据的同步性，确保数据在文件系统和存储设备之间的一致性。、
- 返回的count值表示写入成功的字节数，使得调用者能够确认写入操作的结果。

linux-5.10.102/fs/sync.c:(分析同步写入流程： vfs_fsync_range)

```
int vfs_fsync_range(struct file *file, loff_t start, loff_t end, int datasync)
{
    struct inode *inode = file->f_mapping->host;

    if (!file->f_op->fsync)
        return -EINVAL;
    if (!datasync && (inode->i_state & I_DIRTY_TIME))
        mark_inode_dirty_sync(inode);
    return file->f_op->fsync(file, start, end, datasync);
}
EXPORT_SYMBOL(vfs_fsync_range);
```

用于同步文件的特定范围到底层存储设备。它通常用于确保对文件特定部分所做的更改被写入磁盘并持久化，这对数据的完整性和可靠性至关重要。

然后它在这里调用file->f_op->fsync(file, start, end, datasync);对应调用的是ext4函数操作集里面的ext4_sync_file这个函数。

在打开一个文件时，操作系统会根据文件的路径和文件系统类型来确定使用哪种文件系统的实现。这个过程中，`struct file_operations` 结构体中的函数指针集合 (`f_op`) 就会被设置为对应文件系统的操作函数集合。

如果文件位于 ext4文件系统上，对应的 struct file_operations 结构体就会设置为 ext4 文件系统的操作函数集合。

/home/sf/linux-5.10.102/fs/ext4/fsync.c:(同步写入ext4_sync_file())

```
int ext4_sync_file(struct file *file, loff_t start, loff_t end, int datasync)
{
    int ret = 0, err;
    bool needs_barrier = false;
    struct inode *inode = file->f_mapping->host;
    struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);

    if (unlikely(ext4_forced_shutdown(sbi)))
```

```
        return -EIO;

    J_ASSERT(ext4_journal_current_handle() == NULL);

    trace_ext4_sync_file_enter(file, datasync);

    if (sb_rdonly(inode->i_sb)) {
        /* Make sure that we read updated s_mount_flags value */
        smp_rmb();
        if (ext4_test_mount_flag(inode->i_sb, EXT4_MF_FS_ABORTED))
            ret = -EROFS;
        goto out;
    }

    ret = file_write_and_wait_range(file, start, end);
    if (ret)
        goto out;

    /*
     * data=writeback,ordered:
     *  The caller's filemap_fdatawrite()/wait will sync the data.
     *  Metadata is in the journal, we wait for proper transaction to
     *  commit here.
     *
     * data=journal:
     *  filemap_fdatawrite won't do anything (the buffers are clean).
     *  ext4_force_commit will write the file data into the journal and
     *  will wait on that.
     *  filemap_fdatawait() will encounter a ton of newly-dirtied pages
     *  (they were dirtied by commit).  But that's OK - the blocks are
     *  safe in-journal, which is all fsync() needs to ensure.
     */
    if (!sbi->s_journal)
        ret = ext4_fsync_nojournal(inode, datasync, &needs_barrier);
    else if (ext4_should_journal_data(inode))
        ret = ext4_force_commit(inode->i_sb);
    else
        ret = ext4_fsync_journal(inode, datasync, &needs_barrier);

    if (needs_barrier) {
        err = blkdev_issue_flush(inode->i_sb->s_bdev, GFP_KERNEL);
        if (!ret)
            ret = err;
    }
out:
    err = file_check_and_advance_wb_err(file);
    if (ret == 0)
        ret = err;
    trace_ext4_sync_file_exit(inode, ret);
    return ret;
}
```

调用 ext4_fsync_journal 执行日志同步操作。

如果需要将数据真正写入到物理块设备（如磁盘）上，可能会调用块设备层面的接口，如 blkdev_issue_flush，来刷新缓冲区数据并确保数据被持久化。

linux-5.10.102/block/blk-flush.c:(blkdev_issue_flush)

```
int blkdev_issue_flush(struct block_device *bdev, gfp_t gfp_mask)
{
    struct bio *bio;
    int ret = 0;

    bio = bio_alloc(gfp_mask, 0);
    bio_set_dev(bio, bdev);
    bio->bi_opf = REQ_OP_WRITE | REQ_PREFLUSH;

    ret = submit_bio_wait(bio);
    bio_put(bio);
    return ret;
}
EXPORT_SYMBOL(blkdev_issue_flush);
```

用于向指定的块设备发出刷新请求，确保所有待写入的数据被持久化到物理存储介质上。

`submit_bio_wait(bio)` 函数将 `bio` 结构提交给块设备层面处理，以确保数据被写入到物理存储介质上。这可能涉及到块设备驱动程序的处理和硬件层面的操作。

使用 `submit_bio_wait` 函数提交 `bio` 结构，并同步等待操作完成，以确保数据被成功写入到物理设备上。

linux-5.10.102/block/bio.c:(submit_bio_wait())

```
int submit_bio_wait(struct bio *bio)
{
    DECLARE_COMPLETION_ONSTACK_MAP(done, bio->bi_disk->lockdep_map);
    unsigned long hang_check;

    bio->bi_private = &done;
    bio->bi_end_io = submit_bio_wait_endio;
    bio->bi_opf |= REQ_SYNC;
    submit_bio(bio);

    /* Prevent hang_check timer from firing at us during very long I/O */
    hang_check = sysctl_hung_task_timeout_secs;
    if (hang_check)
        while (!wait_for_completion_io_timeout(&done,
                    hang_check * (HZ/2)))
            ;
    else
        wait_for_completion_io(&done);

    return blk_status_to_errno(bio->bi_status);
}
EXPORT_SYMBOL(submit_bio_wait);
```

这段代码定义了 submit_bio_wait 函数，它的作用是提交一个 bio`结构的同步 I/O 请求，并等待该请求完成。

linux-5.10.102/fs/ext4/fsync.c:(ext4_fsync_journal())-这里是多余写了，这个是ext4的事务处理

```
static int ext4_fsync_journal(struct inode *inode, bool datasync,
                  bool *needs_barrier)
{
    struct ext4_inode_info *ei = EXT4_I(inode);
    journal_t *journal = EXT4_SB(inode->i_sb)->s_journal;
    tid_t commit_tid = datasync ? ei->i_datasync_tid : ei->i_sync_tid;

    if (journal->j_flags & JBD2_BARRIER &&
        !jbd2_trans_will_send_data_barrier(journal, commit_tid))
        *needs_barrier = true;

    return ext4_fc_commit(journal, commit_tid);
}
```

linux-5.10.102/mm/filemap.c:(generic_perform_write)

```
ssize_t generic_perform_write(struct file *file,
                struct iov_iter *i, loff_t pos)
{
    struct address_space *mapping = file->f_mapping;
    const struct address_space_operations *a_ops = mapping->a_ops;
    long status = 0;
    ssize_t written = 0;
    unsigned int flags = 0;

    do {
        struct page *page;
        unsigned long offset;   /* Offset into pagecache page */
        unsigned long bytes;    /* Bytes to write to page */
        size_t copied;      /* Bytes copied from user */
        void *fsdata;

        offset = (pos & (PAGE_SIZE - 1));
        bytes = min_t(unsigned long, PAGE_SIZE - offset,
                        iov_iter_count(i));

again:
        /*
         * Bring in the user page that we will copy from _first_.
         * Otherwise there's a nasty deadlock on copying from the
         * same page as we're writing to, without it being marked
         * up-to-date.
         *
         * Not only is this an optimisation, but it is also required
         * to check that the address is actually valid, when atomic
         * usercopies are used, below.
         */
        if (unlikely(iov_iter_fault_in_readable(i, bytes))) {
            status = -EFAULT;
            break;
        }

        if (fatal_signal_pending(current)) {
```

```
            status = -EINTR;
            break;
        }

        status = a_ops->write_begin(file, mapping, pos, bytes, flags,
                        &page, &fsdata);
        if (unlikely(status < 0))
            break;

        if (mapping_writably_mapped(mapping))
            flush_dcache_page(page);

        copied = iov_iter_copy_from_user_atomic(page, i, offset, bytes);
        flush_dcache_page(page);

        status = a_ops->write_end(file, mapping, pos, bytes, copied,
                        page, fsdata);
        if (unlikely(status < 0))
            break;
        copied = status;

        cond_resched();

        iov_iter_advance(i, copied);
        if (unlikely(copied == 0)) {
            /*
             * If we were unable to copy any data at all, we must
             * fall back to a single segment length write.
             *
             * If we didn't fallback here, we could livelock
             * because not all segments in the iov can be copied at
             * once without a pagefault.
             */
            bytes = min_t(unsigned long, PAGE_SIZE - offset,
                        iov_iter_single_seg_count(i));
            goto again;
        }
        pos += copied;
        written += copied;

        balance_dirty_pages_ratelimited(mapping); //脏页管理
    } while (iov_iter_count(i));

    return written ? written : status;  //如果写入成功，则返回写入的字节数written
}
EXPORT_SYMBOL(generic_perform_write);
```

它用于从用户空间缓冲区向内核文件系统的页缓存(page cache)写入数据，这个函数可以被多个文件系统使用，就比如ext4。处理与页缓存相关的文件写操作。

在generic_perform_write执行完后，数据从用户空间写入到内核的页缓存中，接下来的部分主要涉及将脏页数据（即修改后的页缓存数据）写回到磁盘。确保持久存储，并维持文件系统的一致性。

在 generic_perform_write() 中的写操作成功后，脏页就会生成。

1、页缓存中的脏页管理

脏页的管理通过balance_dirty_pages_ratelimited()函数完成。它会控制脏页的数量，确保脏页不会超出系统的阈值。如果脏页过多，内核会触发回写机制，将数据写回磁盘。

2、回写机制（Writeback)

系统通常不会立即将每次写操作同步写入磁盘。相反，数据会先存储在页缓存中，直到某个时刻系统决定将脏页回写到磁盘，这个过程称为回写。

有两种常见的回写触发机制：

- 定时回写：内核会定期通过后台守护进程（如pdflush或writeback内核线程）将脏页写回到磁盘。
- 同步回写：当某个进程执行fsync()或者O_SYNC标志的文件操作时，会立即触发回写操作，确保数据同步到磁盘。

linux-5.10.102/mm/page-writeback.c：(balance_dirty_pages_ratelimited{})

```
void balance_dirty_pages_ratelimited(struct address_space *mapping)
{
    struct inode *inode = mapping->host;
    struct backing_dev_info *bdi = inode_to_bdi(inode);
    struct bdi_writeback *wb = NULL;
    int ratelimit;
    int *p;

    if (!(bdi->capabilities & BDI_CAP_WRITEBACK))
        return;

    if (inode_cgwb_enabled(inode))
        wb = wb_get_create_current(bdi, GFP_KERNEL);
    if (!wb)
        wb = &bdi->wb;

    ratelimit = current->nr_dirtied_pause;
    if (wb->dirty_exceeded)
        ratelimit = min(ratelimit, 32 >> (PAGE_SHIFT - 10));

    preempt_disable();
    /*
     * This prevents one CPU to accumulate too many dirtied pages without
     * calling into balance_dirty_pages(), which can happen when there are
     * 1000+ tasks, all of them start dirtying pages at exactly the same
     * time, hence all honoured too large initial task->nr_dirtied_pause.
     */
    p =  this_cpu_ptr(&bdp_ratelimits);
    if (unlikely(current->nr_dirtied >= ratelimit))
        *p = 0;
    else if (unlikely(*p >= ratelimit_pages)) {
        *p = 0;
        ratelimit = 0;
    }
    /*
     * Pick up the dirtied pages by the exited tasks. This avoids lots of
     * short-lived tasks (eg. gcc invocations in a kernel build) escaping
     * the dirty throttling and livelock other long-run dirtiers.
     */
    p = this_cpu_ptr(&dirty_throttle_leaks);
```

```
    if (*p > 0 && current->nr_dirtied < ratelimit) {
        unsigned long nr_pages_dirtied;
        nr_pages_dirtied = min(*p, ratelimit - current->nr_dirtied);
        *p -= nr_pages_dirtied;
        current->nr_dirtied += nr_pages_dirtied;
    }
    preempt_enable();

    if (unlikely(current->nr_dirtied >= ratelimit))
        balance_dirty_pages(wb, current->nr_dirtied);

    wb_put(wb);
}
EXPORT_SYMBOL(balance_dirty_pages_ratelimited);
```

后台写回线程（writeback内核线程）是由一系列机制来定期或者根据条件自动触发的。

1、脏页生成和触发条件

当应用程序执行写操作时，内核将数据写入页缓存，这些页被标记为脏页（dirty pages）。脏页不会立即写入磁盘，目的是通过延迟写入减少磁盘I/O，提升性能。但是，内核必须在一定条件下将脏页写回磁盘，避免脏页占用过多内存。

脏页写回的主要触发条件有：

- 脏页比例超过系统阈值：系统有一系列配置参数来控制脏页写回的时机，如dirty_ratio、dirty_background_ratio等。
- 脏页时间超时：脏页在内存中停留的时间超过一定的阈值（例如dirty_expire_centisecs）,会被后台线程选中进行写回。
- 显示调用fsync（）或类似的同步操作：当应用程序调用fsync()、sync()时，会强制将相关文件的脏页写回磁盘。

2、后台写回线程（writeback内核线程）

后台写回线程的核心作用是定期检查系统中的脏页，当检测到脏页比例超过阈值，或者脏页时间超时，writeback线程会自动触发，将脏页写回磁盘。

内核线程会一直运行，但并不是一直处于活跃状态，它的行为由特定的触发条件控制，只有在需要时才会实际执行写回操作，通常情况下，它处于休眠状态，并在满足一些条件时被唤醒。

linux-5.10.102/fs/fs-writeback.c：(wb_workfn{})

```
void wb_workfn(struct work_struct *work)
{
    struct bdi_writeback *wb = container_of(to_delayed_work(work),
                        struct bdi_writeback, dwork);
    long pages_written;

    set_worker_desc("flush-%s", bdi_dev_name(wb->bdi));
    current->flags |= PF_SWAPWRITE;

    if (likely(!current_is_workqueue_rescuer() ||
           !test_bit(WB_registered, &wb->state))) {
        /*
         * The normal path.  Keep writing back @wb until its
         * work_list is empty.  Note that this path is also taken
         * if @wb is shutting down even when we're running off the
```

```
         * rescuer as work_list needs to be drained.
         */
        do {
            pages_written = wb_do_writeback(wb);
            trace_writeback_pages_written(pages_written);
        } while (!list_empty(&wb->work_list));
    } else {
        /*
         * bdi_wq can't get enough workers and we're running off
         * the emergency worker.  Don't hog it.  Hopefully, 1024 is
         * enough for efficient IO.
         */
        pages_written = writeback_inodes_wb(wb, 1024,
                            WB_REASON_FORKER_THREAD);
        trace_writeback_pages_written(pages_written);
    }

    if (!list_empty(&wb->work_list))
        wb_wakeup(wb);
    else if (wb_has_dirty_io(wb) && dirty_writeback_interval)
        wb_wakeup_delayed(wb);

    current->flags &= ~PF_SWAPWRITE;
}
```

wb_workfn是linux内核中writeback线程的工作函数，它的职责是处理脏页写回到磁盘的操作。

核心逻辑：执行脏页写回

整个写回过程分为两个主要逻辑路径。

1、正常写回路径

```
if (likely(!current_is_workqueue_rescuer() || !test_bit(WB_registered, &wb->state))) {
    /* 正常路径，执行写回操作 */
    do {
        pages_written = wb_do_writeback(wb);
        trace_writeback_pages_written(pages_written);
    } while (!list_empty(&wb->work_list));
}
```

- wb_do_writeback(wb)是核心写回函数，它负责实际的脏页写回过程。
- trace_writeback_pages_written(pages_written)用于内核跟踪系统记录写回的页面数量。
- do...while循环保证在work_list非空的情况下持续处理写回操作。

2、紧急路径

```
else {
    /* 紧急情况下，使用有限的页数来进行写回 */
    pages_written = writeback_inodes_wb(wb, 1024, WB_REASON_FORKER_THREAD);
    trace_writeback_pages_written(pages_written);
}
```

- 如果内核工作队列中无法获得足够的工作线程，writeback 线程将使用紧急模式，只写回最多 1024 页。
- writeback_inodes_wb 是另一个负责具体写回操作的函数，只不过在此情境下它限额为 1024 页。

关键函数

wb_do_writeback(wb)：这是执行写回的核心函数，它负责处理从脏页缓存到写回磁盘的具体逻辑。

writeback_inodes_wb(wb, 1024, WB_REASON_FORKER_THREAD)：紧急情况下的写回操作函数，在紧急情况下，只写回一定数量的脏页（在这里是 1024 页）。

总结

wb_workfn()：是负责处理脏页写回磁盘的工作函数。

通过wb_do_writeback(wb)或writeback_inodes_wb(wb, 1024, WB_REASON_FORKER_THREAD)，将脏页从内存写回到磁盘。

根据系统负载或是否为紧急情况，采用不同的写回策略，并根据条件决定是否唤醒线程继续处理剩余的写回任务。

这个函数位于 fs/fs-writeback.c，并由 workqueue 调度来周期性处理脏页写回任务。

在 Linux 内核中，`workqueue` 是用于调度延迟任务的一种机制。`writeback` 线程实际上是通过 `workqueue` 来管理和调度的。而具体到 `writeback` 任务，`wb_workfn` 函数就是作为 `workqueue` 的一个工作函数被调度的。

linux-5.10.102/fs/fs-writeback.c： (wb_do_writeback(wb))

```
static long wb_do_writeback(struct bdi_writeback *wb)
{
    struct wb_writeback_work *work;
    long wrote = 0;

    set_bit(WB_writeback_running, &wb->state);
    while ((work = get_next_work_item(wb)) != NULL) {
        trace_writeback_exec(wb, work);
        wrote += wb_writeback(wb, work);
        finish_writeback_work(wb, work);
    }

    /*
     * Check for a flush-everything request
     */
    wrote += wb_check_start_all(wb);

    /*
     * Check for periodic writeback, kupdated() style
     */
    wrote += wb_check_old_data_flush(wb);
    wrote += wb_check_background_flush(wb);
    clear_bit(WB_writeback_running, &wb->state);

    return wrote;
}
```

wb_writeback(wb, work)：该函数是核心，执行脏页的实际写回操作，它遍历需要同步的数据页，将它们写入磁盘。wb_writeback() 内部会调用文件系统特定的 writepages() 方法，比如 ext4_writepages() 或 btrfs_writepages()，来完成实际的写回。

linux-5.10.102/fs/fs-writeback.c：(wb_writeback(wb, work))

```
static long wb_writeback(struct bdi_writeback *wb,
             struct wb_writeback_work *work)
{
    unsigned long wb_start = jiffies;
    long nr_pages = work->nr_pages;
    unsigned long dirtied_before = jiffies;
    struct inode *inode;
    long progress;
    struct blk_plug plug;

    blk_start_plug(&plug);
    spin_lock(&wb->list_lock);
    for (;;) {
        /*
         * Stop writeback when nr_pages has been consumed
         */
        if (work->nr_pages <= 0)
            break;

        /*
         * Background writeout and kupdate-style writeback may
         * run forever. Stop them if there is other work to do
         * so that e.g. sync can proceed. They'll be restarted
         * after the other works are all done.
         */
        if ((work->for_background || work->for_kupdate) &&
            !list_empty(&wb->work_list))
            break;

        /*
         * For background writeout, stop when we are below the
         * background dirty threshold
         */
        if (work->for_background && !wb_over_bg_thresh(wb))
            break;

        /*
         * Kupdate and background works are special and we want to
         * include all inodes that need writing. Livelock avoidance is
         * handled by these works yielding to any other work so we are
         * safe.
         */
        if (work->for_kupdate) {
            dirtied_before = jiffies -
                msecs_to_jiffies(dirty_expire_interval * 10);
        } else if (work->for_background)
            dirtied_before = jiffies;

        trace_writeback_start(wb, work);
```

```
        if (list_empty(&wb->b_io))
            queue_io(wb, work, dirtied_before);
        if (work->sb)
            progress = writeback_sb_inodes(work->sb, wb, work);
        else
            progress = __writeback_inodes_wb(wb, work);
        trace_writeback_written(wb, work);

        wb_update_bandwidth(wb, wb_start);

        /*
         * Did we write something? Try for more
         *
         * Dirty inodes are moved to b_io for writeback in batches.
         * The completion of the current batch does not necessarily
         * mean the overall work is done. So we keep looping as long
         * as made some progress on cleaning pages or inodes.
         */
        if (progress)
            continue;
        /*
         * No more inodes for IO, bail
         */
        if (list_empty(&wb->b_more_io))
            break;
        /*
         * Nothing written. Wait for some inode to
         * become available for writeback. Otherwise
         * we'll just busyloop.
         */
        trace_writeback_wait(wb, work);
        inode = wb_inode(wb->b_more_io.prev);
        spin_lock(&inode->i_lock);
        spin_unlock(&wb->list_lock);
        /* This function drops i_lock... */
        inode_sleep_on_writeback(inode);
        spin_lock(&wb->list_lock);
    }
    spin_unlock(&wb->list_lock);
    blk_finish_plug(&plug);

    return nr_pages - work->nr_pages;
}
```

在 `wb_writeback()` 函数中，根据 `work->sb` 是否为空，选择是调用 `writeback_sb_inodes()` 还是 `__writeback_inodes_wb()`，它们的区别在于处理的范围和上下文对象不同。具体来说：

1. `writeback_sb_inodes(work->sb, wb, work)`：
   当 `work->sb` 不为空时，该函数会被调用。这表示写回操作针对**特定的超级块**（`superblock`），也就是文件系统的整体管理结构。
   - **超级块**（`superblock`）是文件系统的元数据结构，管理整个文件系统的全局信息和状态。写回超级块的过程涉及将文件系统的脏 inode 结构中的数据写回磁盘。
   - **调用时机**：当你需要将某个特定文件系统中的所有脏页（通过 inode 关联）写回时，就会用到这个函数。这个函数会遍历该超级块管理的所有 inode，将对应的脏数据写回磁盘。

- **适用场景**：比如在对文件系统执行 `sync` 或者某个特定文件系统的批量写回任务时会用到。这确保了文件系统级别的写回任务可以有序进行。

2. `__writeback_inodes_wb(wb, work)`：
   当 `work->sb` 为空时，写回操作作用于整个 `bdi_writeback` 结构中的所有脏 inode，而不是某个特定文件系统的超级块。这表示对 `wb`（`bdi_writeback`）结构中的所有 inode 进行脏页写回。
   - `bdi_writeback` **结构**管理与存储设备（如块设备或虚拟设备）相关的脏页写回状态。它不局限于某个超级块，可能管理多个文件系统的脏页状态。
   - **调用时机**：当需要进行全局的后台脏页写回任务时，或者系统定期执行的写回操作（例如定期触发的 `background flush`），则会使用这个函数进行广泛的脏页写回。
   - **适用场景**：后台写回任务，定期刷新脏页，以及处理设备级别的写回请求时会用到。

两者的区别总结：

- **写回范围不同**：
  - `writeback_sb_inodes()` 处理的是特定文件系统（由超级块管理的文件系统）内的脏 inode。
  - `__writeback_inodes_wb()` 处理的是整个 `bdi_writeback` 结构中的脏 inode，可能涉及多个文件系统。
- **写回的粒度和作用对象不同**：
  - `writeback_sb_inodes()` 作用于某个文件系统内所有的 inode，确保文件系统的一致性和完整性。
  - `__writeback_inodes_wb()` 则是全局性的操作，作用于整个设备的写回上下文（`bdi_writeback`），处理背景任务或全局性的脏页刷新。

`fsync` 和 `sync` 与脏页写回机制有密切关系。它们都是为了确保数据的持久性和一致性，但在功能和作用范围上有所不同。

1. sync

- **功能**：`sync` 是一个系统调用，用于将所有文件系统中的脏数据（包括文件和元数据）写回到磁盘。它会触发所有挂载文件系统的写回操作。
- **作用范围**：`sync` 会遍历所有挂载的文件系统，确保所有脏页都被写回，保证文件系统的一致性。
- **触发写回**：在执行 `sync` 时，内核会调用 `writeback_sb_inodes()` 来将特定超级块的所有脏 inode 写回磁盘。

2. fsync

- **功能**：`fsync` 是一个系统调用，用于将指定文件的脏数据写回到磁盘，确保该文件的数据和元数据都是最新的。
- **作用范围**：`fsync` 仅影响单个文件，确保该文件的所有未写回的修改都被持久化到磁盘。
- **触发写回**：在执行 `fsync` 时，内核会调用 `__writeback_inodes_wb()` 或相关的写回函数，处理该文件对应的 inode，将其脏页写回。

关系总结

- **触发机制**：`sync` 和 `fsync` 都会触发内核的脏页写回机制，确保数据安全。它们通过不同的接口和范围来控制写回操作。
- **粒度不同**：`sync` 是全局性的操作，会影响所有挂载的文件系统，而 `fsync` 则是针对特定文件的局部操作。

- **性能影响**：由于 `sync` 会写回所有脏数据，因此可能会比 `fsync` 需要更长的时间和更多的资源，特别是在有大量脏页的情况下。

linux-5.10.102/fs/fs-writeback.c：(writeback_sb_inodes())

```
static long writeback_sb_inodes(struct super_block *sb,
                struct bdi_writeback *wb,
                struct wb_writeback_work *work)
{
    struct writeback_control wbc = {
        .sync_mode      = work->sync_mode,
        .tagged_writepages  = work->tagged_writepages,
        .for_kupdate        = work->for_kupdate,
        .for_background     = work->for_background,
        .for_sync       = work->for_sync,
        .range_cyclic       = work->range_cyclic,
        .range_start        = 0,
        .range_end      = LLONG_MAX,
    };
    unsigned long start_time = jiffies;
    long write_chunk;
    long wrote = 0;  /* count both pages and inodes */

    while (!list_empty(&wb->b_io)) {
        struct inode *inode = wb_inode(wb->b_io.prev);
        struct bdi_writeback *tmp_wb;

        if (inode->i_sb != sb) {
            if (work->sb) {
                /*
                 * We only want to write back data for this
                 * superblock, move all inodes not belonging
                 * to it back onto the dirty list.
                 */
                redirty_tail(inode, wb);
                continue;
            }

            /*
             * The inode belongs to a different superblock.
             * Bounce back to the caller to unpin this and
             * pin the next superblock.
             */
            break;
        }

        /*
         * Don't bother with new inodes or inodes being freed, first
         * kind does not need periodic writeout yet, and for the latter
         * kind writeout is handled by the freer.
         */
        spin_lock(&inode->i_lock);
        if (inode->i_state & (I_NEW | I_FREEING | I_WILL_FREE)) {
            redirty_tail_locked(inode, wb);
            spin_unlock(&inode->i_lock);
```

```
            continue;
        }
        if ((inode->i_state & I_SYNC) && wbc.sync_mode != WB_SYNC_ALL) {
            /*
             * If this inode is locked for writeback and we are not
             * doing writeback-for-data-integrity, move it to
             * b_more_io so that writeback can proceed with the
             * other inodes on s_io.
             *
             * We'll have another go at writing back this inode
             * when we completed a full scan of b_io.
             */
            spin_unlock(&inode->i_lock);
            requeue_io(inode, wb);
            trace_writeback_sb_inodes_requeue(inode);
            continue;
        }
        spin_unlock(&wb->list_lock);

        /*
         * We already requeued the inode if it had I_SYNC set and we
         * are doing WB_SYNC_NONE writeback. So this catches only the
         * WB_SYNC_ALL case.
         */
        if (inode->i_state & I_SYNC) {
            /* Wait for I_SYNC. This function drops i_lock... */
            inode_sleep_on_writeback(inode);
            /* Inode may be gone, start again */
            spin_lock(&wb->list_lock);
            continue;
        }
        inode->i_state |= I_SYNC;
        wbc_attach_and_unlock_inode(&wbc, inode);

        write_chunk = writeback_chunk_size(wb, work);
        wbc.nr_to_write = write_chunk;
        wbc.pages_skipped = 0;

        /*
         * We use I_SYNC to pin the inode in memory. While it is set
         * evict_inode() will wait so the inode cannot be freed.
         */
        __writeback_single_inode(inode, &wbc);

        wbc_detach_inode(&wbc);
        work->nr_pages -= write_chunk - wbc.nr_to_write;
        wrote += write_chunk - wbc.nr_to_write;

        if (need_resched()) {
            /*
             * We're trying to balance between building up a nice
             * long list of IOs to improve our merge rate, and
             * getting those IOs out quickly for anyone throttling
             * in balance_dirty_pages().  cond_resched() doesn't
             * unplug, so get our IOs out the door before we
```

```
             * give up the CPU.
             */
            blk_flush_plug(current);
            cond_resched();
        }

        /*
         * Requeue @inode if still dirty.  Be careful as @inode may
         * have been switched to another wb in the meantime.
         */
        tmp_wb = inode_to_wb_and_lock_list(inode);
        spin_lock(&inode->i_lock);
        if (!(inode->i_state & I_DIRTY_ALL))
            wrote++;
        requeue_inode(inode, tmp_wb, &wbc);
        inode_sync_complete(inode);
        spin_unlock(&inode->i_lock);

        if (unlikely(tmp_wb != wb)) {
            spin_unlock(&tmp_wb->list_lock);
            spin_lock(&wb->list_lock);
        }

        /*
         * bail out to wb_writeback() often enough to check
         * background threshold and other termination conditions.
         */
        if (wrote) {
            if (time_is_before_jiffies(start_time + HZ / 10UL))
                break;
            if (work->nr_pages <= 0)
                break;
        }
    }
    return wrote;
}
```

`writeback_sb_inodes()` 函数负责将指定超级块中的脏 inode 写回到磁盘。它通过遍历与超级块关联的 inode 列表，将脏页数据提交到块设备。函数中重要的逻辑包括：

遍历脏 inode：使用 `wb->b_io` 列表来获取待写回的 inode。

状态检查：在写回之前，检查 inode 的状态，确保它不是新创建或正在释放的 inode。

写回单个 inode：通过 `__writeback_single_inode()` 函数将脏页数据写回到磁盘。

处理写回进度：在写回过程中更新写回计数，并在需要时重新排队仍然脏的 inode。

linux-5.10.102/fs/fs-writeback.c：(__writeback_single_inode())

```
static int
__writeback_single_inode(struct inode *inode, struct writeback_control *wbc)
{
    struct address_space *mapping = inode->i_mapping;
    long nr_to_write = wbc->nr_to_write;
    unsigned dirty;
```

```
    int ret;

    WARN_ON(!(inode->i_state & I_SYNC));

    trace_writeback_single_inode_start(inode, wbc, nr_to_write);

    ret = do_writepages(mapping, wbc);

    /*
     * Make sure to wait on the data before writing out the metadata.
     * This is important for filesystems that modify metadata on data
     * I/O completion. We don't do it for sync(2) writeback because it has a
     * separate, external IO completion path and ->sync_fs for guaranteeing
     * inode metadata is written back correctly.
     */
    if (wbc->sync_mode == WB_SYNC_ALL && !wbc->for_sync) {
        int err = filemap_fdatawait(mapping);
        if (ret == 0)
            ret = err;
    }

    /*
     * If the inode has dirty timestamps and we need to write them, call
     * mark_inode_dirty_sync() to notify the filesystem about it and to
     * change I_DIRTY_TIME into I_DIRTY_SYNC.
     */
    if ((inode->i_state & I_DIRTY_TIME) &&
        (wbc->sync_mode == WB_SYNC_ALL || wbc->for_sync ||
         time_after(jiffies, inode->dirtied_time_when +
            dirtytime_expire_interval * HZ))) {
        trace_writeback_lazytime(inode);
        mark_inode_dirty_sync(inode);
    }

    /*
     * Some filesystems may redirty the inode during the writeback
     * due to delalloc, clear dirty metadata flags right before
     * write_inode()
     */
    spin_lock(&inode->i_lock);
    dirty = inode->i_state & I_DIRTY;
    inode->i_state &= ~dirty;

    /*
     * Paired with smp_mb() in __mark_inode_dirty().  This allows
     * __mark_inode_dirty() to test i_state without grabbing i_lock -
     * either they see the I_DIRTY bits cleared or we see the dirtied
     * inode.
     *
     * I_DIRTY_PAGES is always cleared together above even if @mapping
     * still has dirty pages.  The flag is reinstated after smp_mb() if
     * necessary.  This guarantees that either __mark_inode_dirty()
     * sees clear I_DIRTY_PAGES or we see PAGECACHE_TAG_DIRTY.
     */
    smp_mb();
```

```
    if (mapping_tagged(mapping, PAGECACHE_TAG_DIRTY))
        inode->i_state |= I_DIRTY_PAGES;

    spin_unlock(&inode->i_lock);

    /* Don't write the inode if only I_DIRTY_PAGES was set */
    if (dirty & ~I_DIRTY_PAGES) {
        int err = write_inode(inode, wbc);
        if (ret == 0)
            ret = err;
    }
    trace_writeback_single_inode(inode, wbc, nr_to_write);
    return ret;
}
```

在 __writeback_single_inode()函数中，内存数据写入块层的过程主要通过 do_writepages() 函数实现。以下是该函数的关键步骤：

1、写入页面

do_writepages(mapping, wbc)会调用文件系统的写回页面函数，将脏页数据从内存中写入到块设备。

linux-5.10.102/mm/page-writeback.c： （do_writepages(mapping, wbc)）

```
int do_writepages(struct address_space *mapping, struct writeback_control *wbc)
{
    int ret;

    if (wbc->nr_to_write <= 0)
        return 0;
    while (1) {
        if (mapping->a_ops->writepages)
            ret = mapping->a_ops->writepages(mapping, wbc);
        else
            ret = generic_writepages(mapping, wbc);
        if ((ret != -ENOMEM) || (wbc->sync_mode != WB_SYNC_ALL))
            break;
        cond_resched();
        congestion_wait(BLK_RW_ASYNC, HZ/50);
    }
    return ret;
}
```

do_writepages函数实际上负责调用文件系统特定的writepages方法，或者在没有定义时调用通用的generic_writepages。这个函数会根据writeback_control的状态执行写回操作。

在具体的文件系统中，比如ext4，如果定义了writepages方法（即mapping->a_ops->writepages），那么它将调用该方法进行写回。

linux-5.10.102/fs/ext4/inode.c： （ext4_writepages（））

```
static int ext4_writepages(struct address_space *mapping,
                struct writeback_control *wbc)
{
    pgoff_t writeback_index = 0;
```

```
    long nr_to_write = wbc->nr_to_write;
    int range_whole = 0;
    int cycled = 1;
    handle_t *handle = NULL;
    struct mpage_da_data mpd;
    struct inode *inode = mapping->host;
    int needed_blocks, rsv_blocks = 0, ret = 0;
    struct ext4_sb_info *sbi = EXT4_SB(mapping->host->i_sb);
    struct blk_plug plug;
    bool give_up_on_write = false;

    if (unlikely(ext4_forced_shutdown(EXT4_SB(inode->i_sb))))
        return -EIO;

    percpu_down_read(&sbi->s_writepages_rwsem);
    trace_ext4_writepages(inode, wbc);

    /*
     * No pages to write? This is mainly a kludge to avoid starting
     * a transaction for special inodes like journal inode on last iput()
     * because that could violate lock ordering on umount
     */
    if (!mapping->nrpages || !mapping_tagged(mapping, PAGECACHE_TAG_DIRTY))
        goto out_writepages;

    if (ext4_should_journal_data(inode)) {
        ret = generic_writepages(mapping, wbc);
        goto out_writepages;
    }

    /*
     * If the filesystem has aborted, it is read-only, so return
     * right away instead of dumping stack traces later on that
     * will obscure the real source of the problem.  We test
     * EXT4_MF_FS_ABORTED instead of sb->s_flag's SB_RDONLY because
     * the latter could be true if the filesystem is mounted
     * read-only, and in that case, ext4_writepages should
     * *never* be called, so if that ever happens, we would want
     * the stack trace.
     */
    if (unlikely(ext4_forced_shutdown(EXT4_SB(mapping->host->i_sb)) ||
                 ext4_test_mount_flag(inode->i_sb, EXT4_MF_FS_ABORTED))) {
        ret = -EROFS;
        goto out_writepages;
    }

    /*
     * If we have inline data and arrive here, it means that
     * we will soon create the block for the 1st page, so
     * we'd better clear the inline data here.
     */
    if (ext4_has_inline_data(inode)) {
        /* Just inode will be modified... */
        handle = ext4_journal_start(inode, EXT4_HT_INODE, 1);
        if (IS_ERR(handle)) {
```

```
            ret = PTR_ERR(handle);
            goto out_writepages;
        }
        BUG_ON(ext4_test_inode_state(inode,
                EXT4_STATE_MAY_INLINE_DATA));
        ext4_destroy_inline_data(handle, inode);
        ext4_journal_stop(handle);
    }

    if (ext4_should_dioread_nolock(inode)) {
        /*
         * We may need to convert up to one extent per block in
         * the page and we may dirty the inode.
         */
        rsv_blocks = 1 + ext4_chunk_trans_blocks(inode,
                        PAGE_SIZE >> inode->i_blkbits);
    }

    if (wbc->range_start == 0 && wbc->range_end == LLONG_MAX)
        range_whole = 1;

    if (wbc->range_cyclic) {
        writeback_index = mapping->writeback_index;
        if (writeback_index)
            cycled = 0;
        mpd.first_page = writeback_index;
        mpd.last_page = -1;
    } else {
        mpd.first_page = wbc->range_start >> PAGE_SHIFT;
        mpd.last_page = wbc->range_end >> PAGE_SHIFT;
    }

    mpd.inode = inode;
    mpd.wbc = wbc;
    ext4_io_submit_init(&mpd.io_submit, wbc);
retry:
    if (wbc->sync_mode == WB_SYNC_ALL || wbc->tagged_writepages)
        tag_pages_for_writeback(mapping, mpd.first_page, mpd.last_page);
    blk_start_plug(&plug);

    /*
     * First writeback pages that don't need mapping - we can avoid
     * starting a transaction unnecessarily and also avoid being blocked
     * in the block layer on device congestion while having transaction
     * started.
     */
    mpd.do_map = 0;
    mpd.scanned_until_end = 0;
    mpd.io_submit.io_end = ext4_init_io_end(inode, GFP_KERNEL);
    if (!mpd.io_submit.io_end) {
        ret = -ENOMEM;
        goto unplug;
    }
    ret = mpage_prepare_extent_to_map(&mpd);
    /* Unlock pages we didn't use */
```

```
    mpage_release_unused_pages(&mpd, false);
    /* Submit prepared bio */
    ext4_io_submit(&mpd.io_submit);
    ext4_put_io_end_defer(mpd.io_submit.io_end);
    mpd.io_submit.io_end = NULL;
    if (ret < 0)
        goto unplug;

    while (!mpd.scanned_until_end && wbc->nr_to_write > 0) {
        /* For each extent of pages we use new io_end */
        mpd.io_submit.io_end = ext4_init_io_end(inode, GFP_KERNEL);
        if (!mpd.io_submit.io_end) {
            ret = -ENOMEM;
            break;
        }

        /*
         * We have two constraints: We find one extent to map and we
         * must always write out whole page (makes a difference when
         * blocksize < pagesize) so that we don't block on IO when we
         * try to write out the rest of the page. Journalled mode is
         * not supported by delalloc.
         */
        BUG_ON(ext4_should_journal_data(inode));
        needed_blocks = ext4_da_writepages_trans_blocks(inode);

        /* start a new transaction */
        handle = ext4_journal_start_with_reserve(inode,
                EXT4_HT_WRITE_PAGE, needed_blocks, rsv_blocks);
        if (IS_ERR(handle)) {
            ret = PTR_ERR(handle);
            ext4_msg(inode->i_sb, KERN_CRIT, "%s: jbd2_start: "
                   "%ld pages, ino %lu; err %d", __func__,
                wbc->nr_to_write, inode->i_ino, ret);
            /* Release allocated io_end */
            ext4_put_io_end(mpd.io_submit.io_end);
            mpd.io_submit.io_end = NULL;
            break;
        }
        mpd.do_map = 1;

        trace_ext4_da_write_pages(inode, mpd.first_page, mpd.wbc);
        ret = mpage_prepare_extent_to_map(&mpd);
        if (!ret && mpd.map.m_len)
            ret = mpage_map_and_submit_extent(handle, &mpd,
                    &give_up_on_write);
        /*
         * Caution: If the handle is synchronous,
         * ext4_journal_stop() can wait for transaction commit
         * to finish which may depend on writeback of pages to
         * complete or on page lock to be released.  In that
         * case, we have to wait until after we have
         * submitted all the IO, released page locks we hold,
         * and dropped io_end reference (for extent conversion
         * to be able to complete) before stopping the handle.
```

```
         */
        if (!ext4_handle_valid(handle) || handle->h_sync == 0) {
            ext4_journal_stop(handle);
            handle = NULL;
            mpd.do_map = 0;
        }
        /* Unlock pages we didn't use */
        mpage_release_unused_pages(&mpd, give_up_on_write);
        /* Submit prepared bio */
        ext4_io_submit(&mpd.io_submit);

        /*
         * Drop our io_end reference we got from init. We have
         * to be careful and use deferred io_end finishing if
         * we are still holding the transaction as we can
         * release the last reference to io_end which may end
         * up doing unwritten extent conversion.
         */
        if (handle) {
            ext4_put_io_end_defer(mpd.io_submit.io_end);
            ext4_journal_stop(handle);
        } else
            ext4_put_io_end(mpd.io_submit.io_end);
        mpd.io_submit.io_end = NULL;

        if (ret == -ENOSPC && sbi->s_journal) {
            /*
             * Commit the transaction which would
             * free blocks released in the transaction
             * and try again
             */
            jbd2_journal_force_commit_nested(sbi->s_journal);
            ret = 0;
            continue;
        }
        /* Fatal error - ENOMEM, EIO... */
        if (ret)
            break;
    }
unplug:
    blk_finish_plug(&plug);
    if (!ret && !cycled && wbc->nr_to_write > 0) {
        cycled = 1;
        mpd.last_page = writeback_index - 1;
        mpd.first_page = 0;
        goto retry;
    }

    /* Update index */
    if (wbc->range_cyclic || (range_whole && wbc->nr_to_write > 0))
        /*
         * Set the writeback_index so that range_cyclic
         * mode will write it back later
         */
        mapping->writeback_index = mpd.first_page;
```

```
out_writepages:
    trace_ext4_writepages_result(inode, wbc, ret,
                     nr_to_write - wbc->nr_to_write);
    percpu_up_read(&sbi->s_writepages_rwsem);
    return ret;
}
```

通过ext4_io_submit提交准备好的I/O请求

linux-5.10.102/fs/ext4/ext4.h： （ext4_io_submit{}）

```
struct ext4_io_submit {
    struct writeback_control *io_wbc;
    struct bio      *io_bio;
    ext4_io_end_t       *io_end;
    sector_t        io_next_block;
};
```

用于在 EXT4 文件系统中处理 I/O 提交的结构体。

这个结构体在 EXT4 的 I/O 提交过程中起着关键作用，用于封装和管理每次 I/O 操作的相关信息。具体的 I/O 提交过程会使用这个结构体来处理块设备的写入请求

linux-5.10.102/fs/ext4/page-io.c： （ext4_io_submit（））

```
void ext4_io_submit(struct ext4_io_submit *io)
{
    struct bio *bio = io->io_bio;

    if (bio) {
        int io_op_flags = io->io_wbc->sync_mode == WB_SYNC_ALL ?
                  REQ_SYNC : 0;
        io->io_bio->bi_write_hint = io->io_end->inode->i_write_hint;
        bio_set_op_attrs(io->io_bio, REQ_OP_WRITE, io_op_flags);
        submit_bio(io->io_bio);
    }
    io->io_bio = NULL;
}
```

提交 I/O: 使用 submit_bio 提交生物 bio。这段代码负责将要写入的数据块提交到块设备，确保在写回时能够按预期进行 I/O 操作。

linux-5.10.102/block/blk-core.c:(submit_bio(struct bio *bio))--------同步机制走到这里也是这样，调用的是这个submit_bio（）函数。

```
blk_qc_t submit_bio(struct bio *bio)
{
    if (blkcg_punt_bio_submit(bio))
        return BLK_QC_T_NONE;

    /*
     * If it's a regular read/write or a barrier with data attached,
     * go through the normal accounting stuff before submission.
     */
```

```
    if (bio_has_data(bio)) {
        unsigned int count;

        if (unlikely(bio_op(bio) == REQ_OP_WRITE_SAME))
            count = queue_logical_block_size(bio->bi_disk->queue) >> 9;
        else
            count = bio_sectors(bio);

        if (op_is_write(bio_op(bio))) {
            count_vm_events(PGPGOUT, count);
        } else {
            task_io_account_read(bio->bi_iter.bi_size);
            count_vm_events(PGPGIN, count);
        }

        if (unlikely(block_dump)) {
            char b[BDEVNAME_SIZE];
            printk(KERN_DEBUG "%s(%d): %s block %Lu on %s (%u sectors)\n",
            current->comm, task_pid_nr(current),
                op_is_write(bio_op(bio)) ? "WRITE" : "READ",
                (unsigned long long)bio->bi_iter.bi_sector,
                bio_devname(bio, b), count);
        }
    }

    /*
     * If we're reading data that is part of the userspace workingset, count
     * submission time as memory stall.  When the device is congested, or
     * the submitting cgroup IO-throttled, submission can be a significant
     * part of overall IO time.
     */
    if (unlikely(bio_op(bio) == REQ_OP_READ &&
        bio_flagged(bio, BIO_WORKINGSET))) {
        unsigned long pflags;
        blk_qc_t ret;

        psi_memstall_enter(&pflags);
        ret = submit_bio_noacct(bio);
        psi_memstall_leave(&pflags);

        return ret;
    }

    return submit_bio_noacct(bio);
}
EXPORT_SYMBOL(submit_bio);
```

这个函数调用 submit_bio_noacct来真正提交 bio，而不是直接提交。这样做的目的是在提交前进行必要的处理和记录。

linux-5.10.102/block/blk-core.c:(submit_bio_noacct(bio))

```
blk_qc_t submit_bio_noacct(struct bio *bio)
{
    if (!submit_bio_checks(bio))
```

```
        return BLK_QC_T_NONE;

    /*
     * We only want one ->submit_bio to be active at a time, else stack
     * usage with stacked devices could be a problem.  Use current->bio_list
     * to collect a list of requests submited by a ->submit_bio method while
     * it is active, and then process them after it returned.
     */
    if (current->bio_list) {
        bio_list_add(&current->bio_list[0], bio);
        return BLK_QC_T_NONE;
    }

    if (!bio->bi_disk->fops->submit_bio)
        return __submit_bio_noacct_mq(bio);
    return __submit_bio_noacct(bio);
}
EXPORT_SYMBOL(submit_bio_noacct);
```

这段代码是 submit_bio_noacct 的实现，它主要负责提交 bio 结构而不进行 I/O 会计。

根据 bio->bi_disk->fops->submit_bio 的存在与否，决定使用 **submit_bio_noacct_mq或** submit_bio_noacct 来提交 bio。这两个函数分别处理多队列和单队列的情况。

submit_bio_noacct处理了 I/O 提交的前期准备工作，确保了在提交过程中不会造成栈使用问题。

__submit_bio_noacct_mq 用于多队列块设备，这种设备允许多个 I/O 队列并行处理 I/O 请求。

那如果是多队列块设备这种并行处理I/O请求的方式，在高并发的场景下。

如果是同步写入的话，这里调用的就是这个__submit_bio_noacct(bio)函数。

linux-5.10.102/block/blk-core.c:(__submit_bio_noacct(bio))----同步写入

```
static blk_qc_t __submit_bio_noacct(struct bio *bio)
{
    struct bio_list bio_list_on_stack[2];
    blk_qc_t ret = BLK_QC_T_NONE;

    BUG_ON(bio->bi_next);

    bio_list_init(&bio_list_on_stack[0]);
    current->bio_list = bio_list_on_stack;

    do {
        struct request_queue *q = bio->bi_disk->queue;
        struct bio_list lower, same;

        if (unlikely(bio_queue_enter(bio) != 0))
            continue;

        /*
         * Create a fresh bio_list for all subordinate requests.
         */
        bio_list_on_stack[1] = bio_list_on_stack[0];
        bio_list_init(&bio_list_on_stack[0]);
```

```
        ret = __submit_bio(bio);

        /*
         * Sort new bios into those for a lower level and those for the
         * same level.
         */
        bio_list_init(&lower);
        bio_list_init(&same);
        while ((bio = bio_list_pop(&bio_list_on_stack[0])) != NULL)
            if (q == bio->bi_disk->queue)
                bio_list_add(&same, bio);
            else
                bio_list_add(&lower, bio);

        /*
         * Now assemble so we handle the lowest level first.
         */
        bio_list_merge(&bio_list_on_stack[0], &lower);
        bio_list_merge(&bio_list_on_stack[0], &same);
        bio_list_merge(&bio_list_on_stack[0], &bio_list_on_stack[1]);
    } while ((bio = bio_list_pop(&bio_list_on_stack[0])));

    current->bio_list = NULL;
    return ret;
}
```

- 调用 `__submit_bio(bio)` 函数来实际提交 `bio` 请求。

linux-5.10.102/block/blk-core.c:（__submit_bio()）---------同步写入

```
static blk_qc_t __submit_bio(struct bio *bio)
{
    struct gendisk *disk = bio->bi_disk;
    blk_qc_t ret = BLK_QC_T_NONE;

    if (blk_crypto_bio_prep(&bio)) {
        if (!disk->fops->submit_bio)
            return blk_mq_submit_bio(bio);
        ret = disk->fops->submit_bio(bio);
    }
    blk_queue_exit(disk->queue);
    return ret;
}
```

这段代码的主要作用是根据 `bio` 请求所需的处理类型（是否需要加密等），选择合适的提交方式，并通过块设备操作函数指针将 `bio` 请求提交给底层块设备驱动程序进行处理。

`if (!disk->fops->submit_bio)`：检查块设备的操作函数指针 `submit_bio` 是否为空。如果为空，说明使用的是多队列（multiqueue）机制，会调用 `blk_mq_submit_bio` 来提交 `bio` 请求，该函数会负责处理多队列相关的提交逻辑。

linux-5.10.102/block/blk-core.c:(__submit_bio_noacct_mq)

```
static blk_qc_t __submit_bio_noacct_mq(struct bio *bio)
{
```

```
    struct bio_list bio_list[2] = { };
    blk_qc_t ret = BLK_QC_T_NONE;

    current->bio_list = bio_list;

    do {
        struct gendisk *disk = bio->bi_disk;

        if (unlikely(bio_queue_enter(bio) != 0))
            continue;

        if (!blk_crypto_bio_prep(&bio)) {
            blk_queue_exit(disk->queue);
            ret = BLK_QC_T_NONE;
            continue;
        }

        ret = blk_mq_submit_bio(bio);
    } while ((bio = bio_list_pop(&bio_list[0])));

    current->bio_list = NULL;
    return ret;
}
```

调用 blk_mq_submit_bio 实际提交 bio到对应的队列。

__submit_bio_noacct_mq 函数通过调用 blk_mq_submit_bio 将 bio 提交到对应的多队列中。每个队列负责处理特定的块设备或存储设备。

linux-5.10.102/block/blk-mq.c:(blk_mq_submit_bio())-------(同步写入也是这个函数)

```
blk_qc_t blk_mq_submit_bio(struct bio *bio)
{
    struct request_queue *q = bio->bi_disk->queue;
    const int is_sync = op_is_sync(bio->bi_opf);
    const int is_flush_fua = op_is_flush(bio->bi_opf);
    struct blk_mq_alloc_data data = {
        .q      = q,
    };
    struct request *rq;
    struct blk_plug *plug;
    struct request *same_queue_rq = NULL;
    unsigned int nr_segs;
    blk_qc_t cookie;
    blk_status_t ret;

    blk_queue_bounce(q, &bio);
    __blk_queue_split(&bio, &nr_segs);

    if (!bio_integrity_prep(bio))
        goto queue_exit;

    if (!is_flush_fua && !blk_queue_nomerges(q) &&
        blk_attempt_plug_merge(q, bio, nr_segs, &same_queue_rq))
        goto queue_exit;
```

```
if (blk_mq_sched_bio_merge(q, bio, nr_segs))
    goto queue_exit;

rq_qos_throttle(q, bio);

data.cmd_flags = bio->bi_opf;
rq = __blk_mq_alloc_request(&data);
if (unlikely(!rq)) {
    rq_qos_cleanup(q, bio);
    if (bio->bi_opf & REQ_NOWAIT)
        bio_wouldblock_error(bio);
    goto queue_exit;
}

trace_block_getrq(q, bio, bio->bi_opf);

rq_qos_track(q, rq, bio);

cookie = request_to_qc_t(data.hctx, rq);

blk_mq_bio_to_request(rq, bio, nr_segs);

ret = blk_crypto_init_request(rq);
if (ret != BLK_STS_OK) {
    bio->bi_status = ret;
    bio_endio(bio);
    blk_mq_free_request(rq);
    return BLK_QC_T_NONE;
}

plug = blk_mq_plug(q, bio);
if (unlikely(is_flush_fua)) {
    /* Bypass scheduler for flush requests */
    blk_insert_flush(rq);
    blk_mq_run_hw_queue(data.hctx, true);
} else if (plug && (q->nr_hw_queues == 1 ||
       blk_mq_is_sbitmap_shared(rq->mq_hctx->flags) ||
       q->mq_ops->commit_rqs || !blk_queue_nonrot(q))) {
    /*
     * Use plugging if we have a ->commit_rqs() hook as well, as
     * we know the driver uses bd->last in a smart fashion.
     *
     * Use normal plugging if this disk is slow HDD, as sequential
     * IO may benefit a lot from plug merging.
     */
    unsigned int request_count = plug->rq_count;
    struct request *last = NULL;

    if (!request_count)
        trace_block_plug(q);
    else
        last = list_entry_rq(plug->mq_list.prev);

    if (request_count >= blk_plug_max_rq_count(plug) || (last &&
```

```
            blk_rq_bytes(last) >= BLK_PLUG_FLUSH_SIZE)) {
            blk_flush_plug_list(plug, false);
            trace_block_plug(q);
        }

        blk_add_rq_to_plug(plug, rq);
    } else if (q->elevator) {
        /* Insert the request at the IO scheduler queue */
        blk_mq_sched_insert_request(rq, false, true, true);
    } else if (plug && !blk_queue_nomerges(q)) {
        /*
         * We do limited plugging. If the bio can be merged, do that.
         * Otherwise the existing request in the plug list will be
         * issued. So the plug list will have one request at most
         * The plug list might get flushed before this. If that happens,
         * the plug list is empty, and same_queue_rq is invalid.
         */
        if (list_empty(&plug->mq_list))
            same_queue_rq = NULL;
        if (same_queue_rq) {
            list_del_init(&same_queue_rq->queuelist);
            plug->rq_count--;
        }
        blk_add_rq_to_plug(plug, rq);
        trace_block_plug(q);

        if (same_queue_rq) {
            data.hctx = same_queue_rq->mq_hctx;
            trace_block_unplug(q, 1, true);
            blk_mq_try_issue_directly(data.hctx, same_queue_rq,
                    &cookie);
        }
    } else if ((q->nr_hw_queues > 1 && is_sync) ||
            !data.hctx->dispatch_busy) {
        /*
         * There is no scheduler and we can try to send directly
         * to the hardware.
         */
        blk_mq_try_issue_directly(data.hctx, rq, &cookie);
    } else {
        /* Default case. */
        blk_mq_sched_insert_request(rq, false, true, true);
    }

    return cookie;
queue_exit:
    blk_queue_exit(q);
    return BLK_QC_T_NONE;
}
```

blk_mq_submit_bio函数的实现负责将 bio提交到块设备的多队列系统中。

blk_qc_t blk_mq_submit_bio(struct bio *bio)这里，这个函数我看就是已经把bio请求，准备发送请求调度了

```
blk_mq_submit_bio(bio)
    └─> __blk_mq_alloc_request(data)
        └─> blk_mq_bio_to_request(rq, bio, nr_segs)
            └─> blk_crypto_init_request(rq)
                ├─> (如果需要加密处理)
                └─> (返回请求的初始化结果)
            └─> (根据情况选择调度)
                ├─> blk_mq_try_issue_directly(rq)
                └─> blk_mq_sched_insert_request(rq)
```

如图是它这个函数的调用链。再这个blk_mq_submit_bio(bio)函数里面的函数调用过程。

我们现在默认的就是我们的这个操作系统都是有调度器的。所以我们现在要执行的函数应该是：blk_mq_sched_insert_request(rq)

根据我们之前的分析和代码流程，无论是同步写入还是异步写入，在最终提交 `bio` 请求到块设备的过程中，都会通过 `blk_mq_submit_bio` 函数来处理。这个函数负责管理和调度 `bio` 请求的提交到块设备的请求队列中，确保它们按照适当的顺序和方式被处理。因此，无论是同步操作（需要等待请求完成）还是异步操作（可以继续执行其他任务），最终都会经过这个函数来进行请求的实际提交。

接下来的处理流程和异步写入的流程类似，因为在blk_mq_submit_bio函数中，无论是同步写入还是异步写入，最终都会提交bio请求到

/home/sf/linux-5.10.102/block/blk-mq-sched.c： (blk_mq_sched_insert_request(rq))

```
void blk_mq_sched_insert_request(struct request *rq, bool at_head,
                 bool run_queue, bool async)
{
    struct request_queue *q = rq->q;
    struct elevator_queue *e = q->elevator;
    struct blk_mq_ctx *ctx = rq->mq_ctx;
    struct blk_mq_hw_ctx *hctx = rq->mq_hctx;

    WARN_ON(e && (rq->tag != BLK_MQ_NO_TAG));

    if (blk_mq_sched_bypass_insert(hctx, !!e, rq)) {
        /*
         * Firstly normal IO request is inserted to scheduler queue or
         * sw queue, meantime we add flush request to dispatch queue(
         * hctx->dispatch) directly and there is at most one in-flight
         * flush request for each hw queue, so it doesn't matter to add
         * flush request to tail or front of the dispatch queue.
         *
         * Secondly in case of NCQ, flush request belongs to non-NCQ
         * command, and queueing it will fail when there is any
         * in-flight normal IO request(NCQ command). When adding flush
         * rq to the front of hctx->dispatch, it is easier to introduce
         * extra time to flush rq's latency because of S_SCHED_RESTART
         * compared with adding to the tail of dispatch queue, then
         * chance of flush merge is increased, and less flush requests
         * will be issued to controller. It is observed that ~10% time
         * is saved in blktests block/004 on disk attached to AHCI/NCQ
         * drive when adding flush rq to the front of hctx->dispatch.
         *
         * Simply queue flush rq to the front of hctx->dispatch so that
```

```
         * intensive flush workloads can benefit in case of NCQ HW.
         */
        at_head = (rq->rq_flags & RQF_FLUSH_SEQ) ? true : at_head;
        blk_mq_request_bypass_insert(rq, at_head, false);
        goto run;
    }

    if (e && e->type->ops.insert_requests) {
        LIST_HEAD(list);

        list_add(&rq->queuelist, &list);
        e->type->ops.insert_requests(hctx, &list, at_head);
    } else {
        spin_lock(&ctx->lock);
        __blk_mq_insert_request(hctx, rq, at_head);
        spin_unlock(&ctx->lock);
    }

run:
    if (run_queue)
        blk_mq_run_hw_queue(hctx, async);
}
```

如果 run_queue 参数为 true，则调用 blk_mq_run_hw_queue来尝试处理硬件队列中的请求，可能会立即发起 I/O 操作。

blk_mq_run_hw_queue:这个函数负责启动硬件队列并执行在队列中的请求。它会最终调用到驱动程序中负责具体 I/O 操作的函数。

linux-5.10.102/block/blk-mq.c:(blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx, bool async))

```
void blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx, bool async)
{
    int srcu_idx;
    bool need_run;

    /*
     * When queue is quiesced, we may be switching io scheduler, or
     * updating nr_hw_queues, or other things, and we can't run queue
     * any more, even __blk_mq_hctx_has_pending() can't be called safely.
     *
     * And queue will be rerun in blk_mq_unquiesce_queue() if it is
     * quiesced.
     */
    hctx_lock(hctx, &srcu_idx);
    need_run = !blk_queue_quiesced(hctx->queue) &&
        blk_mq_hctx_has_pending(hctx);
    hctx_unlock(hctx, srcu_idx);

    if (need_run)
        __blk_mq_delay_run_hw_queue(hctx, async, 0);
}
EXPORT_SYMBOL(blk_mq_run_hw_queue);
```

在 blk_mq_run_hw_queue 函数中，如果队列没有被挂起且有待处理的请求，它会调用 __blk_mq_delay_run_hw_queue 来实际运行硬件队列。

linux-5.10.102/block/blk-mq.c:(void __blk_mq_delay_run_hw_queue)

```
static void __blk_mq_delay_run_hw_queue(struct blk_mq_hw_ctx *hctx, bool async,
                    unsigned long msecs)
{
    if (unlikely(blk_mq_hctx_stopped(hctx)))
        return;

    if (!async && !(hctx->flags & BLK_MQ_F_BLOCKING)) {
        int cpu = get_cpu();
        if (cpumask_test_cpu(cpu, hctx->cpumask)) {
            __blk_mq_run_hw_queue(hctx);
            put_cpu();
            return;
        }

        put_cpu();
    }

    kblockd_mod_delayed_work_on(blk_mq_hctx_next_cpu(hctx), &hctx->run_work,
                    msecs_to_jiffies(msecs));
}
```

处理 CPU 亲和性：如果不是异步请求，并且该硬件上下文不是阻塞的，会检查当前 CPU 是否在适用的 CPU 集合中：

如果是，直接调用 __blk_mq_run_hw_queue，将请求发送给硬件处理。

```
static void __blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx)
{
    int srcu_idx;

    /*
     * We should be running this queue from one of the CPUs that
     * are mapped to it.
     *
     * There are at least two related races now between setting
     * hctx->next_cpu from blk_mq_hctx_next_cpu() and running
     * __blk_mq_run_hw_queue():
     *
     * - hctx->next_cpu is found offline in blk_mq_hctx_next_cpu(),
     *   but later it becomes online, then this warning is harmless
     *   at all
     *
     * - hctx->next_cpu is found online in blk_mq_hctx_next_cpu(),
     *   but later it becomes offline, then the warning can't be
     *   triggered, and we depend on blk-mq timeout handler to
     *   handle dispatched requests to this hctx
     */
    if (!cpumask_test_cpu(raw_smp_processor_id(), hctx->cpumask) &&
        cpu_online(hctx->next_cpu)) {
        printk(KERN_WARNING "run queue from wrong CPU %d, hctx %s\n",
```

```
            raw_smp_processor_id(),
            cpumask_empty(hctx->cpumask) ? "inactive": "active");
        dump_stack();
    }

    /*
     * We can't run the queue inline with ints disabled. Ensure that
     * we catch bad users of this early.
     */
    WARN_ON_ONCE(in_interrupt());

    might_sleep_if(hctx->flags & BLK_MQ_F_BLOCKING);

    hctx_lock(hctx, &srcu_idx);
    blk_mq_sched_dispatch_requests(hctx);
    hctx_unlock(hctx, srcu_idx);
}
```

锁定硬件上下文：使用 hctx_lock来获取对硬件上下文的锁，然后调用 blk_mq_sched_dispatch_requests。

blk_mq_sched_dispatch_requests 函数将处理调度请求的实际逻辑。这个函数负责从调度器中提取请求并将它们传递给底层的设备驱动，最终实现写入磁盘的操作。

linux-5.10.102/block/blk-mq-sched.c:(blk_mq_sched_dispatch_requests)

```
void blk_mq_sched_dispatch_requests(struct blk_mq_hw_ctx *hctx)
{
    struct request_queue *q = hctx->queue;

    /* RCU or SRCU read lock is needed before checking quiesced flag */
    if (unlikely(blk_mq_hctx_stopped(hctx) || blk_queue_quiesced(q)))
        return;

    hctx->run++;

    /*
     * A return of -EAGAIN is an indication that hctx->dispatch is not
     * empty and we must run again in order to avoid starving flushes.
     */
    if (__blk_mq_sched_dispatch_requests(hctx) == -EAGAIN) {
        if (__blk_mq_sched_dispatch_requests(hctx) == -EAGAIN)
            blk_mq_run_hw_queue(hctx, true);
    }
}
```

调度请求：调用 __blk_mq_sched_dispatch_requests(hctx)，这个函数负责从调度器中提取请求并将它们准备好，以便在下一个步骤中进行处理。如果返回值为 -EAGAIN，则说明有请求未处理，需要再次运行调度器。

linux-5.10.102/block/blk-mq-sched.c:(__blk_mq_sched_dispatch_requests)

```
static int __blk_mq_sched_dispatch_requests(struct blk_mq_hw_ctx *hctx)
{
    struct request_queue *q = hctx->queue;
```

```
    struct elevator_queue *e = q->elevator;
    const bool has_sched_dispatch = e && e->type->ops.dispatch_request;
    int ret = 0;
    LIST_HEAD(rq_list);

    /*
     * If we have previous entries on our dispatch list, grab them first for
     * more fair dispatch.
     */
    if (!list_empty_careful(&hctx->dispatch)) {
        spin_lock(&hctx->lock);
        if (!list_empty(&hctx->dispatch))
            list_splice_init(&hctx->dispatch, &rq_list);
        spin_unlock(&hctx->lock);
    }

    /*
     * Only ask the scheduler for requests, if we didn't have residual
     * requests from the dispatch list. This is to avoid the case where
     * we only ever dispatch a fraction of the requests available because
     * of low device queue depth. Once we pull requests out of the IO
     * scheduler, we can no longer merge or sort them. So it's best to
     * leave them there for as long as we can. Mark the hw queue as
     * needing a restart in that case.
     *
     * We want to dispatch from the scheduler if there was nothing
     * on the dispatch list or we were able to dispatch from the
     * dispatch list.
     */
    if (!list_empty(&rq_list)) {
        blk_mq_sched_mark_restart_hctx(hctx);
        if (blk_mq_dispatch_rq_list(hctx, &rq_list, 0)) {
            if (has_sched_dispatch)
                ret = blk_mq_do_dispatch_sched(hctx);
            else
                ret = blk_mq_do_dispatch_ctx(hctx);
        }
    } else if (has_sched_dispatch) {
        ret = blk_mq_do_dispatch_sched(hctx);
    } else if (hctx->dispatch_busy) {
        /* dequeue request one by one from sw queue if queue is busy */
        ret = blk_mq_do_dispatch_ctx(hctx);
    } else {
        blk_mq_flush_busy_ctxs(hctx, &rq_list);
        blk_mq_dispatch_rq_list(hctx, &rq_list, 0);
    }

    return ret;
}
```

如果 `rq_list` 不为空，则调用 `blk_mq_dispatch_rq_list(hctx, &rq_list, 0)` 来调度请求。如果成功，进一步调用 `blk_mq_do_dispatch_sched(hctx)` 或 `blk_mq_do_dispatch_ctx(hctx)` 进行处理

`blk_mq_do_dispatch_sched(hctx)` 和 `blk_mq_do_dispatch_ctx(hctx)` 函数的调用通常会将请求传递给具体的设备驱动程序。

但是通常会选择blk_mq_do_dispatch_sched(hctx)这个调度函数。

linux-5.10.102/block/blk-mq-sched.c:(blk_mq_do_dispatch_sched())

```
static int blk_mq_do_dispatch_sched(struct blk_mq_hw_ctx *hctx)
{
    int ret;

    do {
        ret = __blk_mq_do_dispatch_sched(hctx);
    } while (ret == 1);

    return ret;
}
```

linux-5.10.102/block/blk-mq-sched.c:(__blk_mq_do_dispatch_sched())

```
static int __blk_mq_do_dispatch_sched(struct blk_mq_hw_ctx *hctx)
{
    struct request_queue *q = hctx->queue;
    struct elevator_queue *e = q->elevator;
    bool multi_hctxs = false, run_queue = false;
    bool dispatched = false, busy = false;
    unsigned int max_dispatch;
    LIST_HEAD(rq_list);
    int count = 0;

    if (hctx->dispatch_busy)
        max_dispatch = 1;
    else
        max_dispatch = hctx->queue->nr_requests;

    do {
        struct request *rq;

        if (e->type->ops.has_work && !e->type->ops.has_work(hctx))
            break;

        if (!list_empty_careful(&hctx->dispatch)) {
            busy = true;
            break;
        }

        if (!blk_mq_get_dispatch_budget(q))
            break;

        rq = e->type->ops.dispatch_request(hctx);
        if (!rq) {
            blk_mq_put_dispatch_budget(q);
            /*
             * We're releasing without dispatching. Holding the
             * budget could have blocked any "hctx"s with the
```

```
             * same queue and if we didn't dispatch then there's
             * no guarantee anyone will kick the queue.  Kick it
             * ourselves.
             */
            run_queue = true;
            break;
        }

        /*
         * Now this rq owns the budget which has to be released
         * if this rq won't be queued to driver via .queue_rq()
         * in blk_mq_dispatch_rq_list().
         */
        list_add_tail(&rq->queuelist, &rq_list);
        if (rq->mq_hctx != hctx)
            multi_hctxs = true;
    } while (++count < max_dispatch);

    if (!count) {
        if (run_queue)
            blk_mq_delay_run_hw_queues(q, BLK_MQ_BUDGET_DELAY);
    } else if (multi_hctxs) {
        /*
         * Requests from different hctx may be dequeued from some
         * schedulers, such as bfq and deadline.
         *
         * Sort the requests in the list according to their hctx,
         * dispatch batching requests from same hctx at a time.
         */
        list_sort(NULL, &rq_list, sched_rq_cmp);
        do {
            dispatched |= blk_mq_dispatch_hctx_list(&rq_list);
        } while (!list_empty(&rq_list));
    } else {
        dispatched = blk_mq_dispatch_rq_list(hctx, &rq_list, count);
    }

    if (busy)
        return -EAGAIN;
    return !!dispatched;
}
```

进入驱动层：`blk_mq_dispatch_rq_list` 会遍历请求列表，并为每个请求调用 `blk_mq_dispatch_request`，后者最终会调用与具体硬件相关的驱动函数，比如 `blk_execute_rq` 或 `blk_mq_ops` 中的 `queue_rq`。

linux-5.10.102/block/blk-mq.c:(blk_mq_dispatch_rq_list())

```
bool blk_mq_dispatch_rq_list(struct blk_mq_hw_ctx *hctx, struct list_head *list,
                 unsigned int nr_budgets)
{
    enum prep_dispatch prep;
    struct request_queue *q = hctx->queue;
    struct request *rq, *nxt;
    int errors, queued;
```

```
    blk_status_t ret = BLK_STS_OK;
    LIST_HEAD(zone_list);
    bool needs_resource = false;

    if (list_empty(list))
        return false;

    /*
     * Now process all the entries, sending them to the driver.
     */
    errors = queued = 0;
    do {
        struct blk_mq_queue_data bd;

        rq = list_first_entry(list, struct request, queuelist);

        WARN_ON_ONCE(hctx != rq->mq_hctx);
        prep = blk_mq_prep_dispatch_rq(rq, !nr_budgets);
        if (prep != PREP_DISPATCH_OK)
            break;

        list_del_init(&rq->queuelist);

        bd.rq = rq;

        /*
         * Flag last if we have no more requests, or if we have more
         * but can't assign a driver tag to it.
         */
        if (list_empty(list))
            bd.last = true;
        else {
            nxt = list_first_entry(list, struct request, queuelist);
            bd.last = !blk_mq_get_driver_tag(nxt);
        }

        /*
         * once the request is queued to lld, no need to cover the
         * budget any more
         */
        if (nr_budgets)
            nr_budgets--;
        ret = q->mq_ops->queue_rq(hctx, &bd);
        switch (ret) {
        case BLK_STS_OK:
            queued++;
            break;
        case BLK_STS_RESOURCE:
            needs_resource = true;
            fallthrough;
        case BLK_STS_DEV_RESOURCE:
            blk_mq_handle_dev_resource(rq, list);
            goto out;
        case BLK_STS_ZONE_RESOURCE:
            /*
```

```
             * Move the request to zone_list and keep going through
             * the dispatch list to find more requests the drive can
             * accept.
             */
            blk_mq_handle_zone_resource(rq, &zone_list);
            needs_resource = true;
            break;
        default:
            errors++;
            blk_mq_end_request(rq, BLK_STS_IOERR);
        }
    } while (!list_empty(list));
out:
    if (!list_empty(&zone_list))
        list_splice_tail_init(&zone_list, list);

    hctx->dispatched[queued_to_index(queued)]++;

    /* If we didn't flush the entire list, we could have told the driver
     * there was more coming, but that turned out to be a lie.
     */
    if ((!list_empty(list) || errors) && q->mq_ops->commit_rqs && queued)
        q->mq_ops->commit_rqs(hctx);
    /*
     * Any items that need requeuing? Stuff them into hctx->dispatch,
     * that is where we will continue on next queue run.
     */
    if (!list_empty(list)) {
        bool needs_restart;
        /* For non-shared tags, the RESTART check will suffice */
        bool no_tag = prep == PREP_DISPATCH_NO_TAG &&
            (hctx->flags & BLK_MQ_F_TAG_QUEUE_SHARED);

        blk_mq_release_budgets(q, nr_budgets);

        spin_lock(&hctx->lock);
        list_splice_tail_init(list, &hctx->dispatch);
        spin_unlock(&hctx->lock);

        /*
         * Order adding requests to hctx->dispatch and checking
         * SCHED_RESTART flag. The pair of this smp_mb() is the one
         * in blk_mq_sched_restart(). Avoid restart code path to
         * miss the new added requests to hctx->dispatch, meantime
         * SCHED_RESTART is observed here.
         */
        smp_mb();

        /*
         * If SCHED_RESTART was set by the caller of this function and
         * it is no longer set that means that it was cleared by another
         * thread and hence that a queue rerun is needed.
         *
         * If 'no_tag' is set, that means that we failed getting
         * a driver tag with an I/O scheduler attached. If our dispatch
```

```
         * waitqueue is no longer active, ensure that we run the queue
         * AFTER adding our entries back to the list.
         *
         * If no I/O scheduler has been configured it is possible that
         * the hardware queue got stopped and restarted before requests
         * were pushed back onto the dispatch list. Rerun the queue to
         * avoid starvation. Notes:
         * - blk_mq_run_hw_queue() checks whether or not a queue has
         *   been stopped before rerunning a queue.
         * - Some but not all block drivers stop a queue before
         *   returning BLK_STS_RESOURCE. Two exceptions are scsi-mq
         *   and dm-rq.
         *
         * If driver returns BLK_STS_RESOURCE and SCHED_RESTART
         * bit is set, run queue after a delay to avoid IO stalls
         * that could otherwise occur if the queue is idle.  We'll do
         * similar if we couldn't get budget or couldn't lock a zone
         * and SCHED_RESTART is set.
         */
        needs_restart = blk_mq_sched_needs_restart(hctx);
        if (prep == PREP_DISPATCH_NO_BUDGET)
            needs_resource = true;
        if (!needs_restart ||
            (no_tag && list_empty_careful(&hctx->dispatch_wait.entry)))
            blk_mq_run_hw_queue(hctx, true);
        else if (needs_restart && needs_resource)
            blk_mq_delay_run_hw_queue(hctx, BLK_MQ_RESOURCE_DELAY);

        blk_mq_update_dispatch_busy(hctx, true);
        return false;
    } else
        blk_mq_update_dispatch_busy(hctx, false);

    return (queued + errors) != 0;
}
```

调用驱动层：核心部分在于调用 q->mq_ops->queue_rq(hctx, &bd)，这里的 mq_ops 是指向块设备驱动中实现的操作函数指针。这个函数会将请求发送给具体的驱动层进行处理。