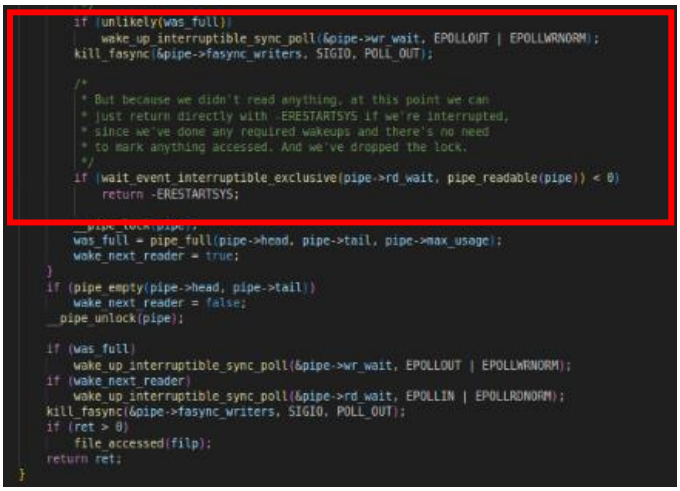
一、pip_read()

```
static ssize_t
pipe_read(struct kiocb *iocb, struct iov_iter *to)
{
    size_t total_len = iov_iter_count(to);
    struct file *filp = iocb->ki_filp;
    struct pipe_inode_info *pipe = filp->private_data;
    bool was_full, wake_next_reader = false;
    ssize_t ret;

    /* Null read succeeds. */
    if (unlikely(total_len == 0))
        return 0;

    ret = 0;
    __pipe_lock(pipe);

    /*
     * We only wake up writers if the pipe was full when we started
     * reading in order to avoid unnecessary wakeups.
     *
     * But when we do wake up writers, we do so using a sync wakeup
     * (WF_SYNC), because we want them to get going and generate more
     * data for us.
     */
    was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
    for (;;) {
        /* Read ->head with a barrier vs post_one_notification() */
        unsigned int head = smp_load_acquire(&pipe->head);
        unsigned int tail = pipe->tail;
        unsigned int mask = pipe->ring_size - 1;

#ifdef CONFIG_WATCH_QUEUE
        if (pipe->note_loss) {
            struct watch_notification n;

            if (total_len < 8) {
                if (ret == 0)
                    ret = -ENOBUFS;
                break;
            }

            n.type = WATCH_TYPE_META;
            n.subtype = WATCH_META_LOSS_NOTIFICATION;
            n.info = watch_sizeof(n);
            if (copy_to_iter(&n, sizeof(n), to) != sizeof(n)) {
                if (ret == 0)
                    ret = -EFAULT;
                break;
            }
            ret += sizeof(n);
            total_len -= sizeof(n);
            pipe->note_loss = false;
        }
#endif
```

```
        if (!pipe_empty(head, tail)) {
            struct pipe_buffer *buf = &pipe->bufs[tail & mask];
            size_t chars = buf->len;
            size_t written;
            int error;

            if (chars > total_len) {
                if (buf->flags & PIPE_BUF_FLAG_WHOLE) {
                    if (ret == 0)
                        ret = -ENOBUFS;
                    break;
                }
                chars = total_len;
            }

            error = pipe_buf_confirm(pipe, buf);
            if (error) {
                if (!ret)
                    ret = error;
                break;
            }

            written = copy_page_to_iter(buf->page, buf->offset, chars, to);
            if (unlikely(written < chars)) {
                if (!ret)
                    ret = -EFAULT;
                break;
            }
            ret += chars;
            buf->offset += chars;
            buf->len -= chars;

            /* Was it a packet buffer? Clean up and exit */
            if (buf->flags & PIPE_BUF_FLAG_PACKET) {
                total_len = chars;
                buf->len = 0;
            }

            if (!buf->len) {
                pipe_buf_release(pipe, buf);
                spin_lock_irq(&pipe->rd_wait.lock);
#ifdef CONFIG_WATCH_QUEUE
                if (buf->flags & PIPE_BUF_FLAG_LOSS)
                    pipe->note_loss = true;
#endif
                tail++;
                pipe->tail = tail;
                spin_unlock_irq(&pipe->rd_wait.lock);
            }
            total_len -= chars;
            if (!total_len)
                break;  /* common path: read succeeded */
            if (!pipe_empty(head, tail))    /* More to do? */
                continue;
        }

        if (!pipe->writers)
            break;
        if (ret)
            break;
        if (filp->f_flags & O_NONBLOCK) {
            ret = -EAGAIN;
            break;
        }
        __pipe_unlock(pipe);
```



1.pipe_read()傳入變數:

a) kiocb 是 Kernel I/O Control Block，I/O 操作的所有資訊打包成一個 struct kiocb
b) struct iov_iter *to (目的地): 資料讀出來後 ，搬去指定使用者空間的 Buffer
c) struct kiocb *iocb : 指向該 pipe 位置，包含: ki_filp (File Pointer)、ki_pos (Position)、ki_complete (Completion Callback)、ki_flags。

```
static ssize_t
pipe_read(struct kiocb *iocb, struct iov_iter *to)
{
```

2. 取得相關變數:

filp / pipe：

filp 是這個 file descriptor 對應的 struct file。

pipe = filp->private_data 存 pipe 內部狀態 (struct pipe_inode_info)，裡面有 head/tail/bufs/wait queue

3. full flag、and wake flag:

was_full：記錄➔一開始讀的時候這個 pipe 是不是滿的。因為只有「原本是滿 → 讀完就不滿了」這種情況，才需要去喚醒 writer，避免無謂 wakeup。

wake_next_reader：block 過後，為了 fairness 可能會把下一個 reader 一起叫醒。

```
size_t total_len = iov_iter_count(to);
struct file *filp = iocb->ki_filp;
struct pipe_inode_info *pipe = filp->private_data;
bool was_full, wake_next_reader = false;
ssize_t ret;
```

3. 讀取 ring buffer:

Pipe data:

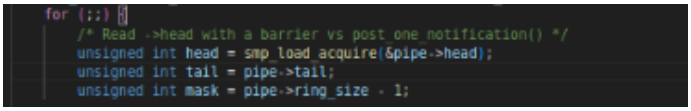
a. head：下一個要寫入的 slot（producer index）
b. tail：下一個要讀出的 slot（consumer index）
c. Wrap-around 機制: 程式碼註解提到 "We use head and tail indices that aren't masked off... allowed to wrap naturally"。這表示 head 和 tail 會一直增加，但在存取陣列時會使用 mask (即 pipe->ring_size - 1) 來取餘數定位。

判斷是否有資料可讀!pipe_empty(head, tail)

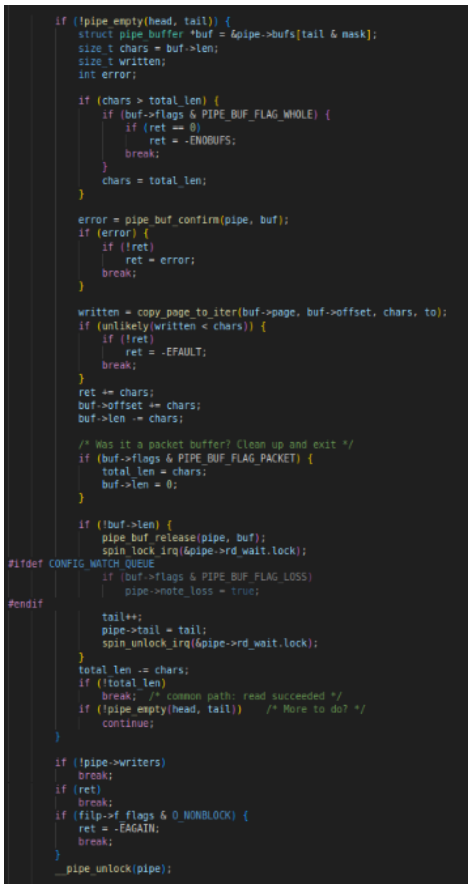
a. pipe->bufs[]：ring buffer 的每一格
b. buf = &pipe->bufs[tail & mask];→ 取出目前「最舊的一格」資料來讀

資料傳遞 user space

a. copy_page_to_iter(...)：等價於「從 kernel page 拷貝 chars bytes 到 user 的 iov_iter」。若小於 chars，表示 user buffer 出問題（通常是無效指標）→ -EFAULT。
b. 更新 ret: 整個 pipe_read() 最後要回傳的總 bytes。
c. 更新 buf->offset / buf->len：代表這格資料已經被吃掉一部分。



```
for (;;) {
    /* Read ->head with a barrier vs post_one_notification() */
    unsigned int head = smp_load_acquire(&pipe->head);
    unsigned int tail = pipe->tail;
    unsigned int mask = pipe->ring_size - 1;
```



```
        if (!pipe_empty(head, tail)) {
            struct pipe_buffer *buf = &pipe->bufs[tail & mask];
            size_t chars = buf->len;
            size_t written;
            int error;

            if (chars > total_len) {
                if (buf->flags & PIPE_BUF_FLAG_WHOLE) {
                    if (ret == 0)
                        ret = -ENOBUFS;
                    break;
                }
                chars = total_len;
            }

            error = pipe_buf_confirm(pipe, buf);
            if (error) {
                if (!ret)
                    ret = error;
                break;
            }

            written = copy_page_to_iter(buf->page, buf->offset, chars, to);
            if (unlikely(written < chars)) {
                if (!ret)
                    ret = -EFAULT;
                break;
            }
            ret += chars;
            buf->offset += chars;
            buf->len -= chars;

            /* Was it a packet buffer? Clean up and exit */
            if (buf->flags & PIPE_BUF_FLAG_PACKET) {
                total_len = chars;
                buf->len = 0;
            }

            if (!buf->len) {
                pipe_buf_release(pipe, buf);
                spin_lock_irq(&pipe->rd_wait.lock);
#ifdef CONFIG_WATCH_QUEUE
                if (buf->flags & PIPE_BUF_FLAG_LOSS)
                    pipe->note_loss = true;
#endif
                tail++;
                pipe->tail = tail;
                spin_unlock_irq(&pipe->rd_wait.lock);
            }
            total_len -= chars;
            if (!total_len)
                break;  /* common path: read succeeded */
            if (!pipe_empty(head, tail))    /* More to do? */
                continue;
        }

        if (!pipe->writers)
            break;
        if (ret)
            break;
        if (filp->f_flags & O_NONBLOCK) {
            ret = -EAGAIN;
            break;
        }
        __pipe_unlock(pipe);
```

rendezvous 機制

```
        if (unlikely(was_full))
            wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLLWRNORM);
        kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);

        /*
         * But because we didn't read anything, at this point we can
         * just return directly with -ERESTARTSYS if we're interrupted,
         * since we've done any required wakeups and there's no need
         * to mark anything accessed. And we've dropped the lock.
         */
        if (wait_event_interruptible_exclusive(pipe->rd_wait, pipe_readable(pipe)) < 0)
            return -ERESTARTSYS;

        __pipe_lock(pipe);
        was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
        wake_next_reader = true;
    }
    if (pipe_empty(pipe->head, pipe->tail))
        wake_next_reader = false;
    __pipe_unlock(pipe);

    if (was_full)
        wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLLWRNORM);
    if (wake_next_reader)
        wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
    kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
    if (ret > 0)
        file_accessed(filp);
    return ret;
}
```

Reader rendezvous 過程:

(1) was_full 是「一開始進 pipe_read 時 pipe 有沒有滿」。如果一開始是滿的，那有可能某些 writer 早就 block 在 pipe->wr_wait 上。雖然這一次我們最後什麼也沒讀到（ret==0），但還是把這些 writer 喚醒一下，因為狀態可能有變。

(2) wake_up_interruptible_sync_poll()---Linux epoll (Event Poll)➔通知 Writer 可以寫入:

a. &pipe->wr_wait : 將等待 writer 喚醒。

b. EPOLLOT | EPOLLWRNORM: 告訴 writer 可以寫入資料

```
        if (unlikely(was_full))
            wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLLWRNORM);
        kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
```

(3) rendezvous 核心：wait_event_interruptible_exclusive(***「reader 在 rendezvous 點等待」，也就是目前 pipe 無資料可讀，reader 先去睡了***)

a. 將 reader 掛到 rd_wait(wait queue)，等資料並進入睡眠

b. pipe_readable(): 睡前、喚醒後檢查是否有資料。

c. __exclusive(): 避免多個 reader 同時讀取 pipe，導致沒有讀取到的 reader 白白被叫起來重睡，導致浪費 CPU 資源。

(進入 wait queue 後，就等待 writer 通知，以達到同步 rendezvous 概念)

```
        if (wait_event_interruptible_exclusive(pipe->rd_wait, pipe_readable(pipe)) < 0)
            return -ERESTARTSYS;
```

d.醒之後，更新狀態

- __pipe_lock(): 重新上鎖，準備下一輪讀取
- wake_next_reader = true: 表示此 reader 已經讀取過，如果讀取完還有資料，應換其他 reader

```
        __pipe_lock(pipe);
        was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
        wake_next_reader = true;
    }
```

e. if (was_full) wake_up_interruptible_sync_poll(&pipe->wr_wait, ...)：---喚醒 writer

如果「在這一輪讀之前」 pipe 是滿的（was_full == true），而我們剛剛讀走了一部分，那現在就有空位可以寫 → 喚醒 wr_wait 上睡覺的 writer。

f. if (wake_next_reader) wake_up_interruptible_sync_poll(&pipe->rd_wait, ...)：---喚醒 reader

當我們之前在 rd_wait 睡過一次（wake_next_reader = true）且 pipe 目前仍然不空，代表除了我之外還可能有其他 reader 等著資料 → 把「下一個 reader」也叫醒

```
    if (was_full)
        wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLLWRNORM);
    if (wake_next_reader)
        wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
    kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
    if (ret > 0)
        file_accessed(filp);
    return ret;
}
```

流程:

T0：R 呼叫 read()

檢查 pipe：空；有 writer 存在；ret=0；阻塞模式。

解鎖 mutex，呼叫 **wait_event_interruptible_exclusive(rd_wait, pipe_readable) → 睡到 rd_wait 上**。

T1：W 呼叫 write()

拿 mutex，往 pipe 裡寫資料，更新 head、bufs。

寫完後 wake_up_interruptible_sync_poll(rd_wait, EPOLLIN|...) → 把 R 從 rd_wait 喚醒。

釋放 mutex，write() return。

T2：R 被喚醒

wait_event 返回 0。

R 重新拿 mutex，回到 for 迴圈前端：

這次 pipe_empty(head, tail) 為 false → 走拷貝資料那條路，

把資料從 pipe_buffer 拿出來，更新 tail。

若 pipe 因為這次讀變「不滿」，就 wake_up wr_wait → 告訴 writer「有空間了」。最後 read() ，R 得到資料。

關鍵觀念：

R 在 T0 就「站在 rendezvous 點等」：rd_wait。

W 在 T1 來到 rendezvous，準備好資料並呼叫 wake_up。

兩人「在這個點碰頭」，資料 transfer 才真正發生

二、pip_write()

```
static ssize_t
pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
    struct file *filp = iocb->ki_filp;
    struct pipe_inode_info *pipe = filp->private_data;
    unsigned int head;
    ssize_t ret = 0;
    size_t total_len = iov_iter_count(from);
    ssize_t chars;
    bool was_empty = false;
    bool wake_next_writer = false;

    /*
     * Reject writing to watch queue pipes before the point where we lock
     * the pipe.
     * Otherwise, lockdep would be unhappy if the caller already has another
     * pipe locked.
     * If we had to support locking a normal pipe and a notification pipe at
     * the same time, we could set up lockdep annotations for that, but
     * since we don't actually need that, it's simpler to just bail here.
     */
    if (pipe_has_watch_queue(pipe))
        return -EXDEV;

    /* Null write succeeds. */
    if (unlikely(total_len == 0))
        return 0;

    __pipe_lock(pipe);

    if (!pipe->readers) {
        send_sig(SIGPIPE, current, 0);
        ret = -EPIPE;
        goto out;
    }

    /*
     * If it wasn't empty we try to merge new data into
     * the last buffer.
     *
     * That naturally merges small writes, but it also
     * page-aligns the rest of the writes for large writes
     * spanning multiple pages.
     */
    head = pipe->head;
    was_empty = pipe_empty(head, pipe->tail);
    chars = total_len & (PAGE_SIZE-1);
    if (chars && !was_empty) {
        unsigned int mask = pipe->ring_size - 1;
        struct pipe_buffer *buf = &pipe->bufs[(head - 1) & mask];
        int offset = buf->offset + buf->len;

        if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) &&
            offset + chars <= PAGE_SIZE) {
            ret = pipe_buf_confirm(pipe, buf);
            if (ret)
                goto out;

            ret = copy_page_from_iter(buf->page, offset, chars, from);
            if (unlikely(ret < chars)) {
                ret = -EFAULT;
                goto out;
            }

            buf->len += ret;
            if (!iov_iter_count(from))
                goto out;
        }
    }
```

```
for (;;) {
    if (!pipe->readers) {
        send_sig(SIGPIPE, current, 0);
        if (!ret)
            ret = -EPIPE;
        break;
    }

    head = pipe->head;
    if (!pipe_full(head, pipe->tail, pipe->max_usage)) {
        unsigned int mask = pipe->ring_size - 1;
        struct pipe_buffer *buf = &pipe->bufs[head & mask];
        struct page *page = pipe->tmp_page;
        int copied;

        if (!page) {
            page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
            if (unlikely(!page)) {
                ret = ret ? : -ENOMEM;
                break;
            }
            pipe->tmp_page = page;
        }

        /* Allocate a slot in the ring in advance and attach an
         * empty buffer.  If we fault or otherwise fail to use
         * it, either the reader will consume it or it'll still
         * be there for the next write.
         */
        spin_lock_irq(&pipe->rd_wait.lock);

        head = pipe->head;
        if (pipe_full(head, pipe->tail, pipe->max_usage)) {
            spin_unlock_irq(&pipe->rd_wait.lock);
            continue;
        }

        pipe->head = head + 1;
        spin_unlock_irq(&pipe->rd_wait.lock);

        /* Insert it into the buffer array */
        buf = &pipe->bufs[head & mask];
        buf->page = page;
        buf->ops = &anon_pipe_buf_ops;
        buf->offset = 0;
        buf->len = 0;
        if (is_packetized(filp))
            buf->flags = PIPE_BUF_FLAG_PACKET;
        else
            buf->flags = PIPE_BUF_FLAG_CAN_MERGE;
        pipe->tmp_page = NULL;

        copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);
        if (unlikely(copied < PAGE_SIZE && iov_iter_count(from))) {
            if (!ret)
                ret = -EFAULT;
            break;
        }
        ret += copied;
        buf->offset = 0;
        buf->len = copied;

        if (!iov_iter_count(from))
            break;
    }
```

```
        }
        if (signal_pending(current)) {
            if (!ret)
                ret = -ERESTARTSYS;
            break;
        }

        /*
         * We're going to release the pipe lock and wait for more
         * space. We wake up any readers if necessary, and then
         * after waiting we need to re-check whether the pipe
         * become empty while we dropped the lock.
         */
        __pipe_unlock(pipe);
        if (was_empty)
            wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
        kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
        wait_event_interruptible_exclusive(pipe->wr_wait, pipe_writable(pipe));
        __pipe_lock(pipe);
        was_empty = pipe_empty(pipe->head, pipe->tail);
        wake_next_writer = true;
    }
out:
    if (pipe_full(pipe->head, pipe->tail, pipe->max_usage))
        wake_next_writer = false;
    __pipe_unlock(pipe);

    /*
     * If we do do a wakeup event, we do a 'sync' wakeup, because we
     * want the reader to start processing things asap, rather than
     * leave the data pending.
     *
     * This is particularly important for small writes, because of
     * how (for example) the GNU make jobserver uses small writes to
     * wake up pending jobs
     *
     * Epoll nonsensically wants a wakeup whether the pipe
     * was already empty or not.
     */
    if (was_empty || pipe->poll_usage)
        wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
    kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
    if (wake_next_writer)
        wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT | EPOLLWRNORM);
    if (ret > 0 && sb_start_write_trylock(file_inode(filp)->i_sb)) {
        int err = file_update_time(filp);
        if (err)
            ret = err;
        sb_end_write(file_inode(filp)->i_sb);
    }
    return ret;
}
```

備註:因某些部分與 reader 相同，因此只列出不同處。

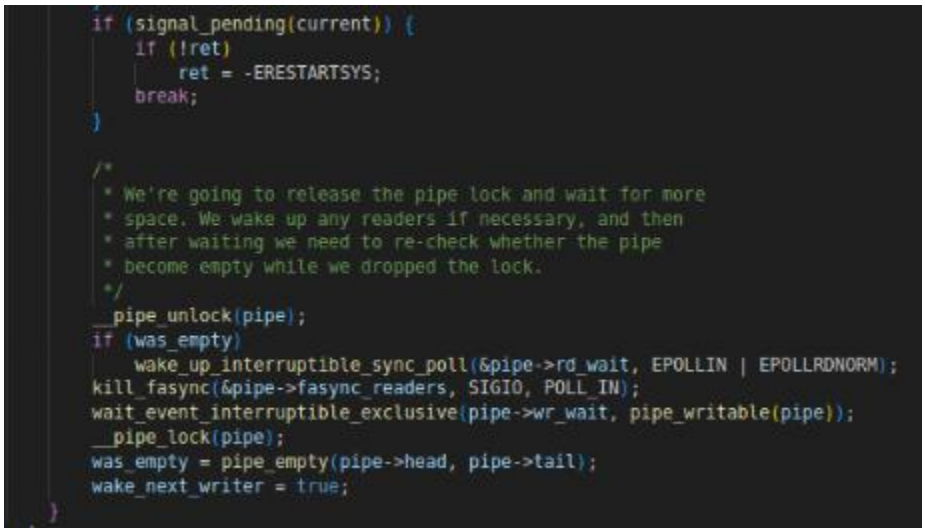
1. 檢查有沒有 reader：

```
__pipe_lock(pipe);

if (!pipe->readers) {
    send_sig(SIGPIPE, current, 0);
    ret = -EPIPE;
    goto out;
}
```

2. merge 寫入到上一格 buffer: 把多個小 write 合併到同一個 page 裡，減少 pipe buffer 的數量，並讓資料 page-align，整個過程不用新增 ring buffer entry，只有在「塞不進去」時才走下面「重新拿一個 page + 新增 slot」的路

```
/*
 * If it wasn't empty we try to merge new data into
 * the last buffer.
 *
 * That naturally merges small writes, but it also
 * page-aligns the rest of the writes for large writes
 * spanning multiple pages.
 */
head = pipe->head;
was_empty = pipe_empty(head, pipe->tail);
chars = total_len & (PAGE_SIZE-1);
if (chars && !was_empty) {
    unsigned int mask = pipe->ring_size - 1;
    struct pipe_buffer *buf = &pipe->bufs[(head - 1) & mask];
    int offset = buf->offset + buf->len;

    if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) &&
        offset + chars <= PAGE_SIZE) {
        ret = pipe_buf_confirm(pipe, buf);
        if (ret)
            goto out;

        ret = copy_page_from_iter(buf->page, offset, chars, from);
        if (unlikely(ret < chars)) {
            ret = -EFAULT;
            goto out;
        }

        buf->len += ret;
        if (!iov_iter_count(from))
            goto out;
    }
}
```

3. writer 端的 rendezvous：等待「有空間可以寫」的 handshake，pipe 滿了 (pipe_full) 且這次 write 還沒寫完 時才走到。也就是「writer ready，但是 buffer 已經塞滿」的情境



```
        if (signal_pending(current)) {
            if (!ret)
                ret = -ERESTARTSYS;
            break;
        }

        /*
         * We're going to release the pipe lock and wait for more
         * space. We wake up any readers if necessary, and then
         * after waiting we need to re-check whether the pipe
         * become empty while we dropped the lock.
         */
        __pipe_unlock(pipe);
        if (was_empty)
            wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
        kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
        wait_event_interruptible_exclusive(pipe->wr_wait, pipe_writable(pipe));
        __pipe_lock(pipe);
        was_empty = pipe_empty(pipe->head, pipe->tail);
        wake_next_writer = true;
    }
```

4. writer 的 rendezvous 流程

(1) 解鎖讓 reader 有機會進來

```
__pipe_unlock(pipe);
```

(2) 如果原本 pipe 是空的 → 叫醒可能在等的 reader。若 was_empty == true，表示 這一輪 write 之前 pipe 是空的：很可能有某些 reader 已經睡在 rd_wait 等資料

```
if (was_empty)
    wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
```

rendezvous 機制

(3) writer 睡在 wr_wait 上，等「可以寫」。pipe_writable(pipe)➔ pipe 不滿：有空間可以寫。

沒有 reader 了：代表再寫也沒意義；wait_event 會醒來，迴圈會走到 !pipe->readers ，傳 SIGPIPE。

wait_event_interruptible_exclusive 會：

a.把這個 writer 掛到 pipe->wr_wait wait queue。

b.若 pipe_writable(pipe) 一開始就 true → 不睡。否則 schedule() 讓出 CPU，直到：有 reader 把資料讀走，讓 pipe 不再 full → reader 會 wake_up wr_wait。或沒有 reader 了（pipe_writable 也會變 true），或收到 signal → 返回負值，後面會變成 -ERESTARTSYS。。

```
wait_event_interruptible_exclusive(pipe->wr_wait, pipe_writable(pipe));
```

三、統整 rendezvous synchronization

read：was_full → 喚醒 writer。

writer：was_empty → 喚醒 reader。

1. **pipe_read() rendezvous**：
   (1)若 `pipe_empty` 且仍有 writer，blocking read 會釋放 mutex，
   (2)呼叫 `wait_event_interruptible_exclusive(pipe->rd_wait, pipe_readable(pipe))`，
   (3)把自己睡在 `rd_wait` 上，等待「有資料或 EOF」。
   `pipe_write()` 在寫入資料後，若 pipe 原本是空，會 `wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN|...)` 喚醒 reader。

2. **pipe_write() rendezvous**：
   若 `pipe_full` 且仍有 reader，blocking write 會釋放 mutex，
   呼叫 `wait_event_interruptible_exclusive(pipe->wr_wait,pipe_writable(pipe))`，把自己睡在 `wr_wait` 上，等待「有空間或沒 reader」。`pipe_read()` 在讀走資料後，若 pipe 原本是滿，會 `wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT|...)` 喚醒 writer。

3. **因此，reader 與 writer 透過兩個 wait queue 互相等待與喚醒：**
   沒資料 → reader 等 writer；
   沒空間 → writer 等 reader；
   只有當「reader ready + pipe 有資料」或「writer ready + pipe 有空間」同時成立時，真正的資料搬移才會發生

Note:

1._pipe_lock()、_pipe_unlock()、__pipe_double_lock()

```c
static inline void __pipe_lock(struct pipe_inode_info *pipe)
{
    mutex_lock_nested(&pipe->mutex, I_MUTEX_PARENT);
}
```

2. Linux epoll (Event Poll):

(1) 允許 process 監視多個 file descriptors ，並在 I/O 可以執行時得到提醒 (edge-triggered 和 level-triggered)

3. pipe_readable():「有資料」或「所有 writer 都沒了 → EOF」➔readable

```c
static inline bool pipe_readable(const struct pipe_inode_info *pipe)
{
    unsigned int head = READ_ONCE(pipe->head);
    unsigned int tail = READ_ONCE(pipe->tail);
    unsigned int writers = READ_ONCE(pipe->writers);

    return !pipe_empty(head, tail) || !writers;
}
```

```c
static inline void __pipe_unlock(struct pipe_inode_info *pipe)
{
    mutex_unlock(&pipe->mutex);
}

void pipe_double_lock(struct pipe_inode_info *pipe1,
                      struct pipe_inode_info *pipe2)
{
    BUG_ON(pipe1 == pipe2);

    if (pipe1 < pipe2) {
        pipe_lock_nested(pipe1, I_MUTEX_PARENT);
        pipe_lock_nested(pipe2, I_MUTEX_CHILD);
    } else {
        pipe_lock_nested(pipe2, I_MUTEX_PARENT);
        pipe_lock_nested(pipe1, I_MUTEX_CHILD);
    }
}
```

```c
static void pipe_lock_nested(struct pipe_inode_info *pipe, int subclass)
{
    if (pipe->files)
        mutex_lock_nested(&pipe->mutex, subclass);
}
```

2. unlikey(): 編譯器會假設 condition 為 假。它會把 else (或 if 之後) 的程式碼緊接著放在跳轉指令之後，讓 CPU 順順地讀下去 (Sequential prefetching)。而 if 裡面的程式碼 (處理 total_len == 0 的部分) 則會被丟到記憶體中比較遠的地方（冷門區段）。

3. smp_load_acquire(&pipe->head) 的意思是：「安全地讀取 head 的值，並保證我接下來要去讀 Buffer 資料的動作，絕對不會偷跑到讀取 head 之前發生。」常用於多核心

Pipe 資料結構:

(1)pipe_buffer: 主要由 page 為單位組成的環形陣列 (Ring Buffer)

```
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

* @page: the page containing the data for the pipe buffer
* @offset: offset of data inside the @page
* @len: length of data inside the @page
* @ops: operations associated with this buffer. See @pipe_buf_operations.
* @flags: pipe buffer flags. See above.

```
#define PIPE_BUF_FLAG_LRU       0x01    /* page is on the LRU */
#define PIPE_BUF_FLAG_ATOMIC    0x02    /* was atomically mapped */
#define PIPE_BUF_FLAG_GIFT      0x04    /* page is a gift */
#define PIPE_BUF_FLAG_PACKET    0x08    /* read() as a packet */
#define PIPE_BUF_FLAG_CAN_MERGE 0x10    /* can merge buffers */
#define PIPE_BUF_FLAG_WHOLE     0x20    /* read() must return entire buffer or error */
#ifdef CONFIG_WATCH_QUEUE
#define PIPE_BUF_FLAG_LOSS      0x40    /* Message loss happened after this buffer */
#endif
```

* @private: private data owned by the ops.---取決於 pipe_buffer_operation()，ops 函式所需的額外上下文資料 (Context Data)

(2) pipe_inode_info

```
struct pipe_inode_info {
        struct mutex mutex;
        wait_queue_head_t rd_wait, wr_wait;
        unsigned int head;
        unsigned int tail;
        unsigned int max_usage;
        unsigned int ring_size;
#ifdef CONFIG_WATCH_QUEUE
        bool note_loss;
#endif
        unsigned int nr_accounted;
        unsigned int readers;
        unsigned int writers;
        unsigned int files;
        unsigned int r_counter;
        unsigned int w_counter;
        bool poll_usage;
        struct page *tmp_page;
        struct fasync_struct *fasync_readers;
        struct fasync_struct *fasync_writers;
        struct pipe_buffer *bufs;
        struct user_struct *user;
#ifdef CONFIG_WATCH_QUEUE
        struct watch_queue *watch_queue;
#endif
};
```

* @mutex: mutex protecting the whole thing
* @rd_wait: reader wait point in case of empty pipe--pipe 空的時候 reader 會儲存在這
* @wr_wait: writer wait point in case of full pipe--pipe 滿的時候 writer 會儲存在這
* @head: The point of buffer production--下一個要寫入的 slot
* @tail: The point of buffer consumption--下一個要讀出的 slot
* @note_loss: The next read() should insert a data-lost message
* @max_usage: The maximum number of slots that may be used in the ring-- ring buffer 上最多可以用多少 slot，用來判斷 full
* @ring_size: total number of buffers (should be a power of 2)
* @nr_accounted: The amount this pipe accounts for in user->pipe_bufs
* @tmp_page: cached released page
* @readers: number of current readers of this pipe--此值變為 0，寫入者會收到 SIGPIPE 信號
* @writers: number of current writers of this pipe--此值變為 0，讀取者會讀到 EOF
* @files: number of struct file referring this pipe (protected by ->i_lock)
* @r_counter: reader counter
* @w_counter: writer counter
* @poll_usage: is this pipe used for epoll, which has crazy wakeups?
* @fasync_readers: reader side fasync

* @fasync_writers: writer side fasync

* @bufs: the circular array of pipe buffers--每個元素代表一個「page + offset + len」的小段資料

* @user: the user who created this pipe

* @watch_queue: If this pipe is a watch_queue, this is the stuff for that

(2) pipe_buf_operation

```
struct pipe_buf_operations {
        /*
         * ->confirm() verifies that the data in the pipe buffer is there
         * and that the contents are good. If the pages in the pipe belong
         * to a file system, we may need to wait for IO completion in this
         * hook. Returns 0 for good, or a negative error value in case of
         * error.  If not present all pages are considered good.
         */
        int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);

        /*
         * When the contents of this pipe buffer has been completely
         * consumed by a reader, ->release() is called.
         */
        void (*release)(struct pipe_inode_info *, struct pipe_buffer *);

        /*
         * Attempt to take ownership of the pipe buffer and its contents.
         * ->try_steal() returns %true for success, in which case the contents
         * of the pipe (the buf->page) is locked and now completely owned by the
         * caller. The page may then be transferred to a different mapping, the
         * most often used case is insertion into different file address space
         * cache.
         */
        bool (*try_steal)(struct pipe_inode_info *, struct pipe_buffer *);

        /*
         * Get a reference to the pipe buffer.
         */
        bool (*get)(struct pipe_inode_info *, struct pipe_buffer *);
};
```

a. comfirm():確認資料有效性，資料已經準備好了，它回傳 0。如果資料還在傳輸中（例如硬碟還在轉），這個函式會睡眠等待 (Sleep/Wait)
b. release():Buffer 裡的資料被讀取完畢，或者 Pipe 被關閉、清除時，Kernel 會呼叫此函式
c. get():增加引用計數 (Reference Increment)

其他相關函式

(1)pipe buffer operation 操作

```
static const struct pipe_buf_operations anon_pipe_buf_ops = {
    .release    = anon_pipe_buf_release,
    .try_steal  = anon_pipe_buf_try_steal,
    .get        = generic_pipe_buf_get,
};
```

(2)pipe_empty(head, tail)：head == tail → pipe 目前沒有資料。

1.

```
 * pipe_empty - Return true if the pipe is empty
 * @head: The pipe ring head pointer
 * @tail: The pipe ring tail pointer
 */
static inline bool pipe_empty(unsigned int head, unsigned int tail)
{
        return head == tail;
}
```

2.

(3)pipe_full(head, tail, max_usage)：佔用的 slot 數 ≥ max_usage → pipe 已滿。

```
/**
 * pipe_full - Return true if the pipe is full
 * @head: The pipe ring head pointer
 * @tail: The pipe ring tail pointer
 * @limit: The maximum amount of slots available.
 */
static inline bool pipe_full(unsigned int head, unsigned int tail,
                             unsigned int limit)
{
        return pipe_occupancy(head, tail) >= limit;
}
```

(4)pipe_buf(pipe, slot): 取得 ring 上某個 slot 的 struct pipe_buffer *

```
/**
 * pipe_buf_get - get a reference to a pipe_buffer
 * @pipe:       the pipe that the buffer belongs to
 * @buf:        the buffer to get a reference to
 *
 * Return: %true if the reference was successfully obtained.
 */
static inline __must_check bool pipe_buf_get(struct pipe_inode_info *pipe,
                                             struct pipe_buffer *buf)
{
        return buf->ops->get(pipe, buf);
}
```

Q7.

呈上，參考 pipe.c，用 character device 實作一個 mini pipe，

a. 當 writer ready、reader not ready 時，writer waits

b. 當 reader ready、writer not ready 時，reader waits

c. 當雙方都 ready 時，喚醒對方並同步傳輸資料。

d. 你可以制定自己的 writing policy。

提示： writer 呼叫 write()想寫資料，即為 writer ready。

提示：所謂 wait 可以用 wait_event_interruptible() 與在某個 queue 上等待條

件成立。

提示：所謂喚醒，是指當狀態或條件改變時，使用 wake_up_interruptible()

喚醒對方。

提示：應該需要寫 read()、write()、poll()等 function。

提示：可以使用固定大小的 buffer。繳交：mini_pipe.c, Makefile, test code（cat, echo 或自己的測試程

式）, test logs, README, diff（說明你的 mini_pipe 跟原來的 pipe.c

有什麼不同、你的 writing policy。)

```
jonathan@jonathan-ASUS-TUF-Gaming-F15-FX507VV4-FX507VV4:~/Documents/Jonathan/EOS/WS5$ ./test_pipe
=== Starting Mini Pipe Test ===
[Writer] Ready to write 200 bytes (Buffer size is 128)...
[Writer] Wrote 128 bytes (Total: 128/200). Buffer might be full now...
[Reader] Starting to read (should unblock Writer)...
[Reader] Read 64 bytes
[Writer] Wrote 64 bytes (Total: 192/200). Buffer might be full now...
[Reader] Read 64 bytes
[Writer] Wrote 8 bytes (Total: 200/200). Buffer might be full now...
[Writer] Done. All data sent.
```

Writer 先執行： 嘗試寫入 200 bytes 的資料。

- Writer 卡住 (Wait)： 因為 Buffer 只有 128 bytes，Writer 寫滿後會停住不動。這代表它正在 wait_queue 裡面等待空間。
- Reader 啟動 (2 秒後)： Reader 開始讀取資料。
- Writer 解鎖 (Wake up)： 一旦 Reader 讀走了資料（騰出空間），Writer 被喚醒，完成剩下的寫入並循環直到結束。

根據作業 Q7 要求 ，以下實作一個名為 mini_pipe 的 Character Device。此模組模擬了 Pipe 的 Blocking I/O 行為（Rendezvous Synchronization）。

Writing Policy 設計 :

- Buffer: 使用一個固定大小的循環陣列（Circular Buffer）。
- Writer Wait: 當 Buffer 滿時（視為 Reader not ready），Writer 進入 Wait Queue 等待。
- Reader Wait: 當 Buffer 空時（視為 Writer not ready），Reader 進入 Wait Queue 等待。
- Synchronization: 雙方透過 mutex 保護 Buffer，並在操作完成後互相 wake_up。

mini_pipe.c 與 pipe.c 的差異:

	mini_pipe.c	pipe.c
buffer	char[128]	dynamic page buffers
synchronization	mutex + wait queue	spinlock + fasync + poll
data transfer	copy	reference
merging policy	X	Y
wait	wait_event_interruptible()	wait_event_interruptible_exclusive()
wake up	wake_up_interruptible()	wake_up_interruptible_sync_poll()

檔案列表:

1. mini_pipe.c: 驅動程式源碼
2. Makefile: 編譯腳本
3. test_pipe.c: 測試程式 (Reader/Writer)
4. README.txt: 說明文件