## 再好好理解一下同步异步阻塞非阻塞

- https://zhuanlan.zhihu.com/p/393635611
  - 一般STORAGE/NIC与OS之间的数据copy都是DMA，不走CPU
  - kernel buffer与用户态buffer之间的copy一般都依赖CPU
- https://zhuanlan.zhihu.com/p/127170201
  - 专门介绍了同步异步阻塞非阻塞

### 从中断回调的调度再看一下 SSD 为彻底理解 IO 总结一下前置知识（storage的IO）

- **数据指针** 从 **内核到设备** 坐的 **几辆车**

```
// 一个 bh 代表一个块，有一个IO完成的回调，一般会阻塞在这个地方，submit(bh)/wait_on_bit()
struct buffer_head {
    ...
        bh_end_io_t *b_end_io;          // I/O completion
    ...
};
// bh 的回调函数多种多样
bh->b_end_io = end_buffer_async_read_io;
bh->b_end_io = end_buffer_write_sync;
...

// bio 代表设备上的连续空间
struct bio {
    ...
        bio_end_io_t *bi_end_io;  // IO完成的回调
    ...
};
// 赋值 bio 的回调函数也是多种多样的
bio->bi_end_io = end_bio_bh_io_sync;
...

// request 继续装车，准备发给设备了，bio会转变成rq，交给设备的IO队列处理，一个request包含多个bio
struct request {
    ...
        // completion callback.
        rq_end_io_fn *end_io;  // 这个IO函数不一定有，如果没有就简单的将 request free 即可
    ...
};
// rq 是在函数 blk_mq_submit_bio 中经由 blk_mq_bio_to_request(rq, bio, nr_segs) 中创建并初始化的

// 同步IO的语义，核心就是 wait_on_bit + while 循环来搞定的
// submit_bio这个函数无论怎么写，都不可能保证数据持久化之后才返回，一般都会在数据还没有持久化下去的时候返
submit_bio(bh);
// task 一般会在 wait on bit 中 for+poll，所以不会继续向下执行
wait_on_bit();
```

- 首先明确一点，读写设备不可能是同步的，都是通过发命令的方式异步操作的，想要同步的效果，一般调用方式如上
  - `wait_on_bit + while`
  - 而 task 则等待在 bit 上，bit 被置位之后即表明数据到位（**已经由DMA将数据搬运到了内存中**），然后等待在 bit 上的task就开始干活了
    - 是一个 cond 的语义
- 但是这个 bit 是谁来 set 呢
  - 先说结论，**中断处理函数来 set**
  - 设备会给CPU发中断，然后中断分上下部，不着急由下半部搞定

```
#define BUFFER_FNS(bit, name)                                           \
static __always_inline void set_buffer_##name(struct buffer_head *bh)   \
{                                                                       \
        if (!test_bit(BH_##bit, &(bh)->b_state))                        \
                set_bit(BH_##bit, &(bh)->b_state);                      \
}
```

- __do_softirq in kernel/softirq.c
  - h->action(h);
    - blk_done_softirq(struct softirq_action *h) in block/blk-mq.c
      - rq->q->mq_ops->complete(rq);
        - scsi_io_completion(struct scsi_cmnd *cmd, unsigned int good_bytes) in scsi_lib.c
          - scsi_end_request(req, blk_stat, good_bytes)
            - blk_update_request(req, error, bytes)
              - req_bio_endio(req, bio, bio_bytes, error);
- req_bio_endio(req, bio, bio_bytes, error);
  - bio_endio(bio);
    - bio->bi_end_io(bio);
      - end_bio_bh_io_sync
        - bh->b_end_io(bh,!bio->bi_status);
          - end_buffer_read_sync
            - __end_buffer_read_notouch(bh, uptodate);
              - set_buffer_uptodate(bh);
- BUFFER_FNS(Uptodate, uptodate)
  - 很显然这个是软中断处理的内核thread来做的这个工作，可能会唤醒某些task，也或许某些task本来就在poll这个bit
- **总结一下**
  - 最大的收获是之前有一个误区，关于内核中数据写到设备的实现，一直没理解清楚，核心就是wait_on_bit
    - **这个bit由设备完成IO后的中断来set**

**read操作的返回**

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
        return ksys_read(fd, buf, count);
}
```

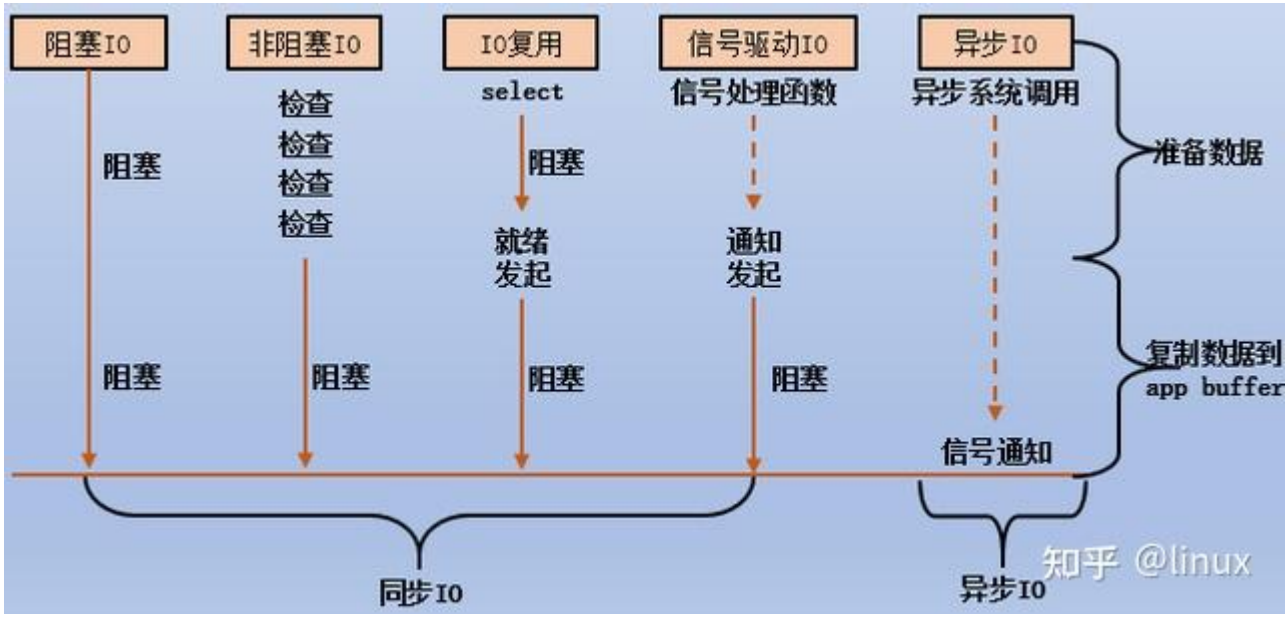
- 假设数据需要由 `storage->kernel buffer->app buffer`
  - 所以看一下读系统调用
- `ksys_read(fd, buf, count);`
  - `ret = vfs_read(f.file, buf, count, ppos);`
    - `ext4_file_read_iter(struct kiocb *iocb, struct iov_iter *to)`
      - `generic_file_read_iter(iocb, to);`
        - `retval = generic_file_buffered_read(iocb, iter, retval);`
          - `wait_on_bit`
          - `copied = copy_page_to_iter(pages[i], offset, bytes, iter);`
- 发起read系统调用的task先sleep等待数据由storage到kernel buffer中，随后该task被唤醒
  - 该task完成数据copy工作
    - 由kernel buffer copy 到 app buffer

# 同步异步阻塞非阻塞 的关键的理解

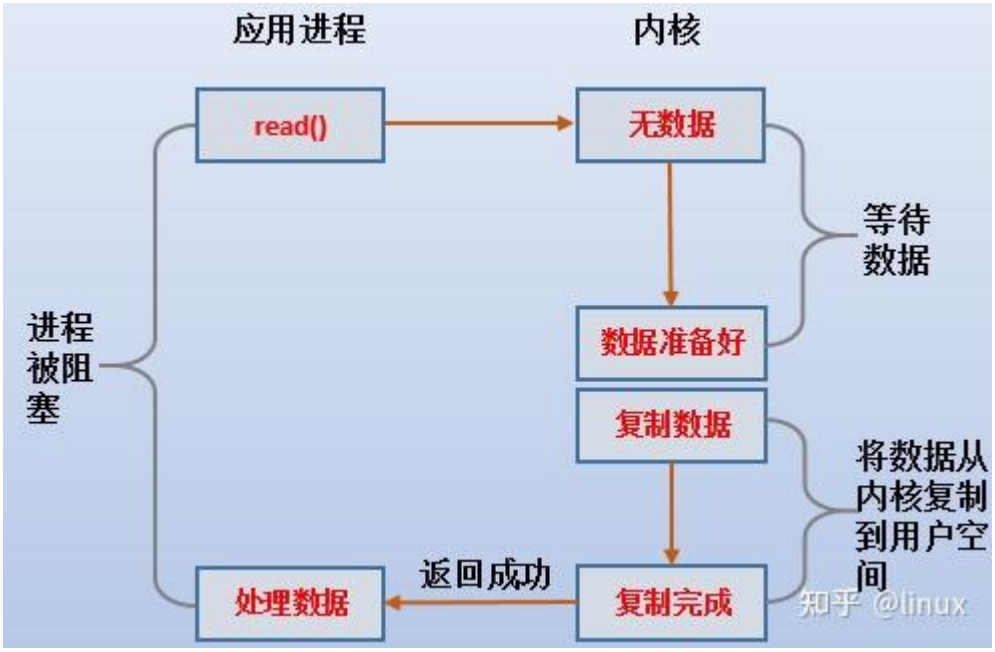
- 同步异步，阻塞非阻塞啥的是针对用户态的应用来说的，别与kernel弄混了
- 数据通路一般是这样的
  - **storage->kernel buffer->app buffer**
    - **storage->kernel buffer** 一般由 DMA 搞定
    - **kernel buffer->app buffer** 需要 CPU 参与
      - 如果用户态程序执行copy，**则用户态程序在这个过程中是被阻塞的**
      - 如果这个copy过程是由后台线程完成的，则用户态app执行该过程是非阻塞的
- **阻塞非阻塞**说的是程序能够继续执行
  - **是否能够继续持有CPU，rip是否能够继续++**
  - 例如read操作执行系统调用之后，在内核中如果数据不在，然后发起IO，把自己挂在这个等待完成的IO上就直接sleep了
    - 所以用户态的这个read操作是无法继续执行任何一条指令的，直到数据由 **storage->kernel buffer**
      - 所以是阻塞的
    - 然后数据到位后，该程序被唤醒，主动执行 `copy_from_kernel_to_user`
      - 在数据 copy 的过程中，task也是被阻塞的

        - **没法继续执行指令**
      - **除非这个copy过程由后台来干**
- **同步异步**
  - **同步指的是 kernel_buffer 与 app_buffer 之间的同步**
  - 关键说的是在 **kernel buffer->app buffer** 我们的目标进程在干啥
    - 同步：**阻塞，等待copy的完成**
      - 同步IO模型中，调用read()的进程会切换到内核，由内核占用CPU来执行数据拷贝，**所以原进程在此阶段一直被阻塞**
    - 异步：**非阻塞，copy完成之后再通知**
      - 异步IO模型中，由内核在后台默默的执行数据拷贝，**所以原进程在此阶段不被阻塞**
- levelDB中关于**同/异步写**的一个描述
  - By default, each write to leveldb is asynchronous
    - it returns after **pushing** the write **from the process into the operating system**
      - write推给OS就返回就算是异步的
        - 不确定数据由 kernel copy 到 levelDB 是不是一个异步线程在做的，不过看样子应该是的

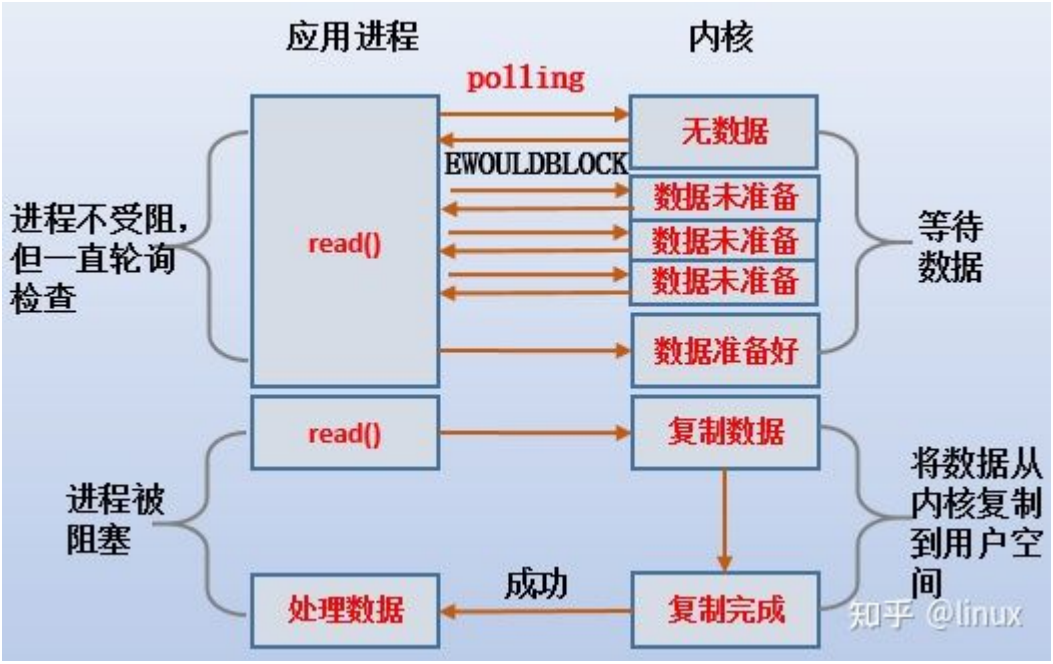
## 来总结一下



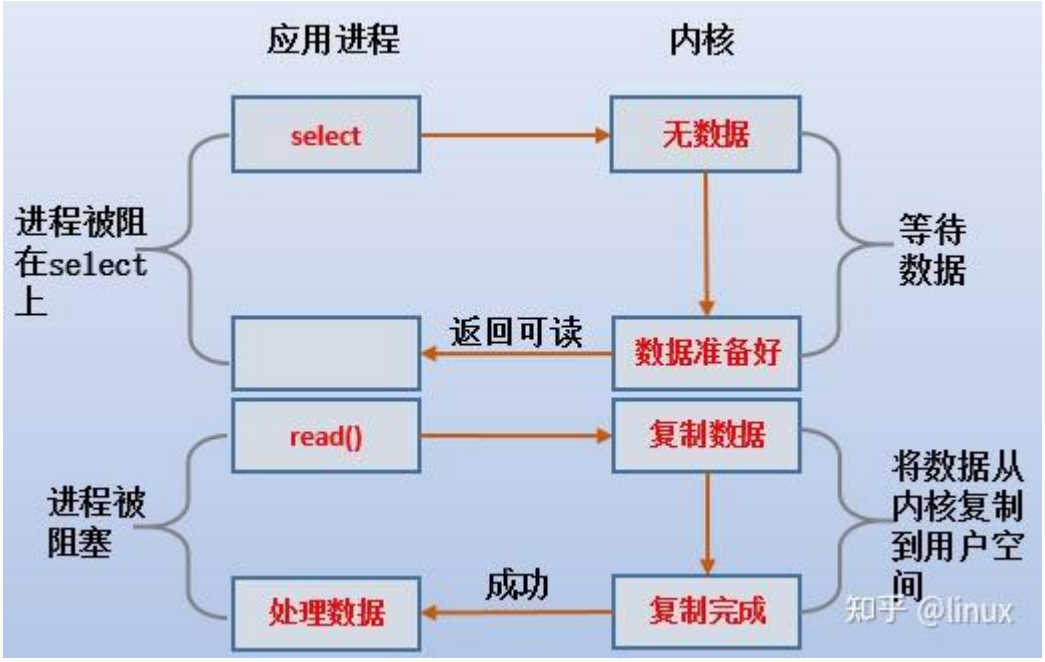

- 绝大多数IO都是同步的（依赖FLAG的），异步有一个专门的异步aio接口
  - 异步IO最核心的地方说的是 **kernel buffer->app buffer** 的过程是由 **原进程完成的呢还是由后台进程完成的呢**
  - **信号驱动与异步的区别也在这里**



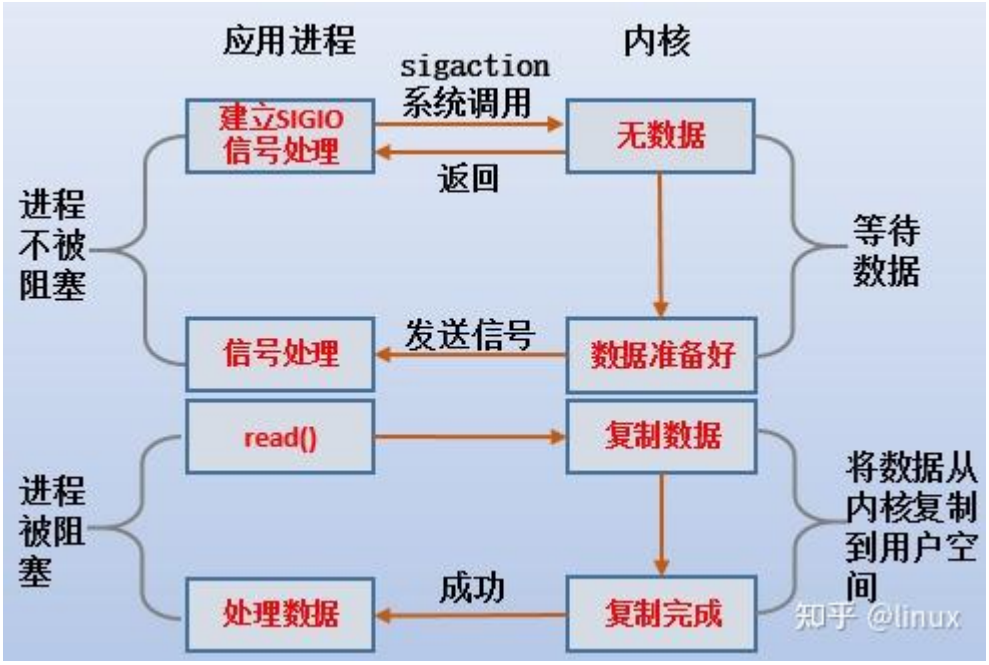

- 这就是最简单的 **同步阻塞**



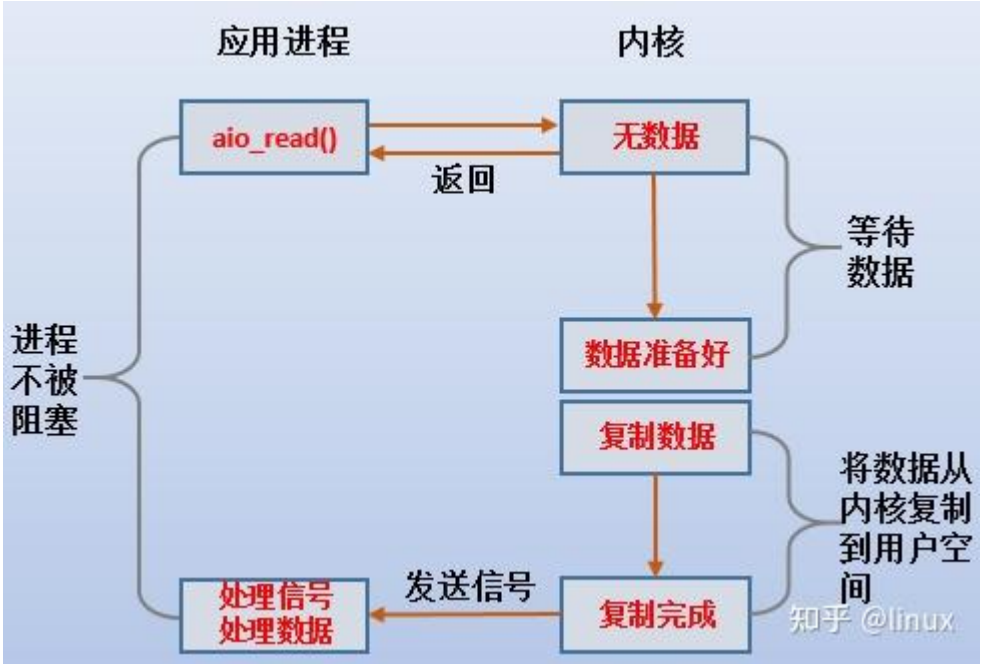

- 是同步的，**但是进程在IO的过程中并没有被阻塞**




- 多路复用，可以处理多个fd(socket)
  - 首先被阻塞在 select 上
  - 有 fd 继续后，数据由 kernel buffer->app buffer **的过程中继续被阻塞**




- 当文件描述符上设置了O_ASYNC标记时，就表示该文件描述符是信号驱动的IO
  - 确实信号驱动在概念上很向异步的
    - 但是同步异步说的是 kernel buffer->app buffer 的过程
      - **所以信号驱动也依然是同步IO**




- 真正的异步IO是这样的，**可以用信号可以用回调**
  - 数据完全不可用到数据完全可用，**原进程一点都不需要操心**
  - 数据由 **kernel buffer->app buffer** 是其它线程的操作，实际上会与原进程抢CPU的
  - **就是原进程本身可以不进入内核态，就进入一次，就再也不需要了**

## 异步 IO

- https://www.fsl.cs.sunysb.edu/~vass/linux-aio.txt
  - 式例代码有一个 inline 的问题

## 网络IO中的Reactor（反应器）与Proactor（前摄器）

- **均为 事件驱动**，仅仅是**同步**与**异步**的**差异**
- **Reactor**
  - **同步IO**，**例如epoll**，通知的是**就绪事件**
    - 数据由kernel到用户态是work的CPU做的
- **Proactor**
  - **异步IO**，是事件驱动的另一种模式。**不是通知就绪事件**，**而是通知完成事件**
    - **比如 c++ boost库 的异步网络IO**