从linux源码看socket的阻塞和非阻塞

笔者一直觉得如果能知道从应用到框架再到操作系统的每一处代码，是一件Exciting的事情。
大部分高性能网络框架采用的是非阻塞模式。笔者这次就从linux源码的角度来阐述socket阻塞(block)和非阻塞(non_block)的区别。 本文源码均来自采用Linux-2.6.24内核版本。

一个TCP非阻塞client端简单的例子

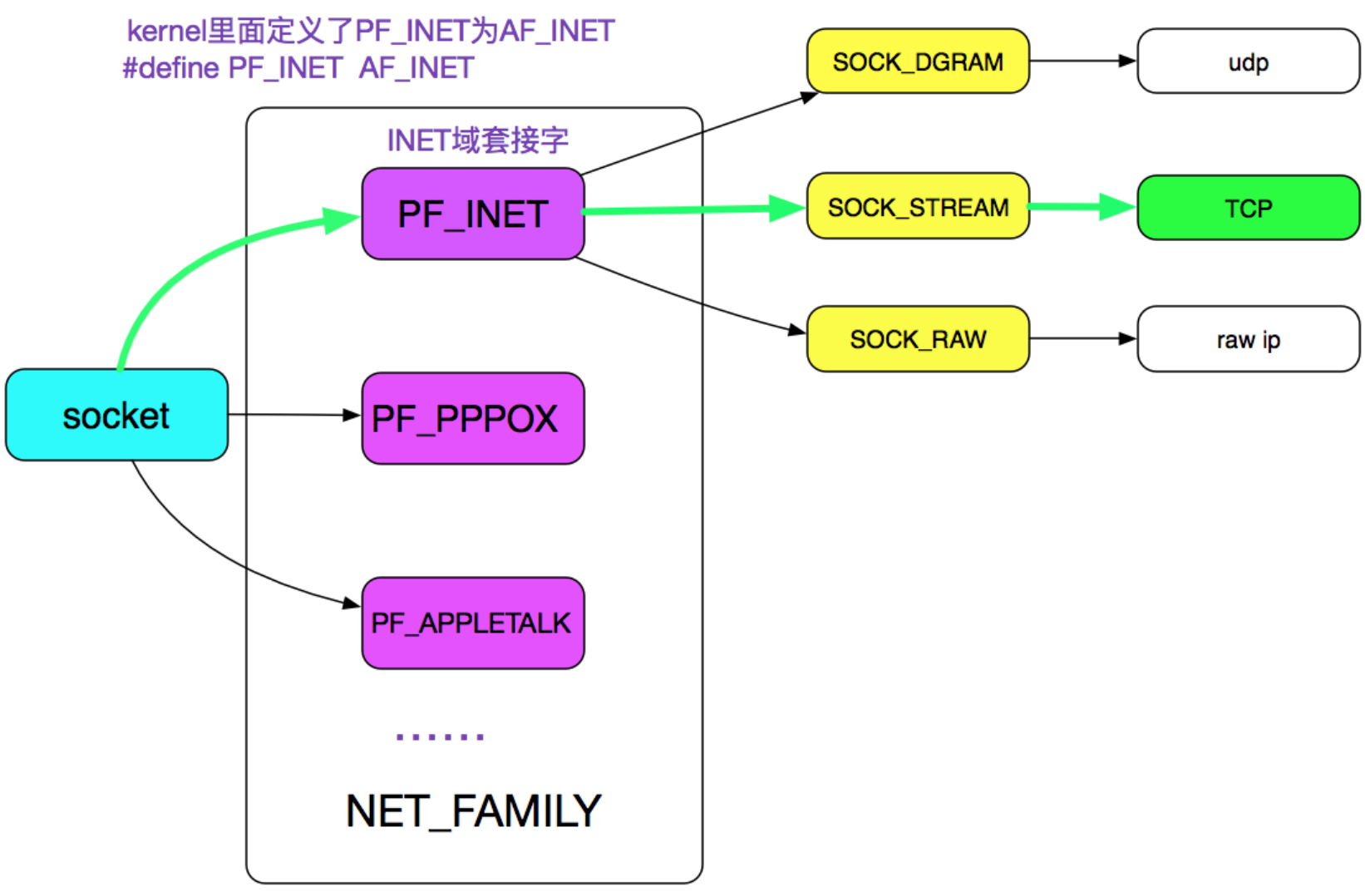
如果我们要产生一个非阻塞的socket,在C语言中如下代码所示:

```
// 创建socket
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
...
// 更改socket为nonblock
fcntl(sock_fd, F_SETFL, fdflags | O_NONBLOCK);
// connect
....
while(1)  {
    int recvlen = recv(sock_fd, recvbuf, RECV_BUF_SIZE) ;
    ......
}
...
```

由于网络协议非常复杂，内核里面用到了大量的面向对象的技巧，所以我们从创建连接开始，一步一步追述到最后代码的调用点。

socket的创建

很明显，内核的第一步应该是通过AF_INET、SOCK_STREAM以及最后一个参数0定位到需要创建一个TCP的socket,如下图绿线所示:



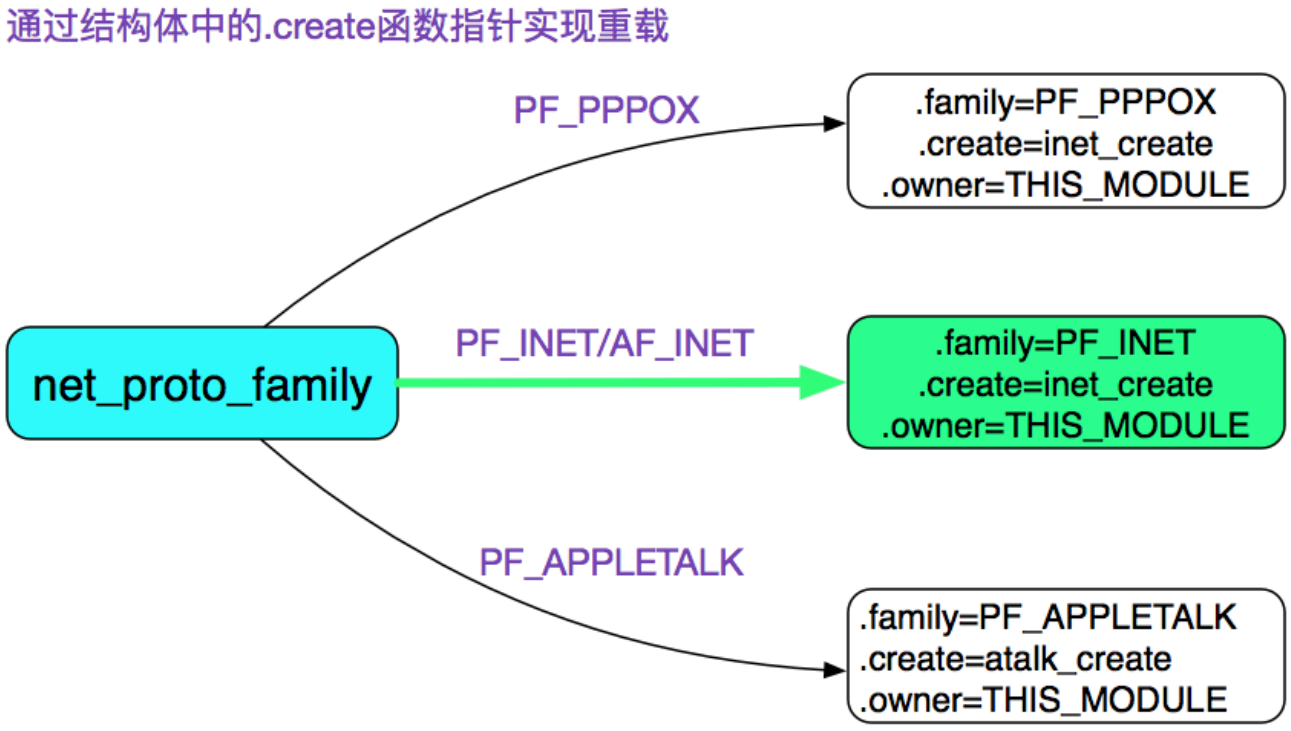
我们跟踪源码调用

```
socket(AF_INET, SOCK_STREAM, 0)
        |->sys_socket 进入系统调用
                |->sock_create
                        |->__sock_create
```

进一步分析__sock_create的代码判断:

```
const struct net_proto_family *pf;
// RCU(Read-Copy Update)是linux的一种内核同步方法，在此不阐述
// family=INET
pf = rcu_dereference(net_families[family]);
err = pf->create(net, sock, protocol);
```

由于family是AF_INET协议，注意在操作系统里面定义了PF_INET等于AF_INET, 内核通过函数指针实现了对pf(net_proto_family)的重载。如下图所示:



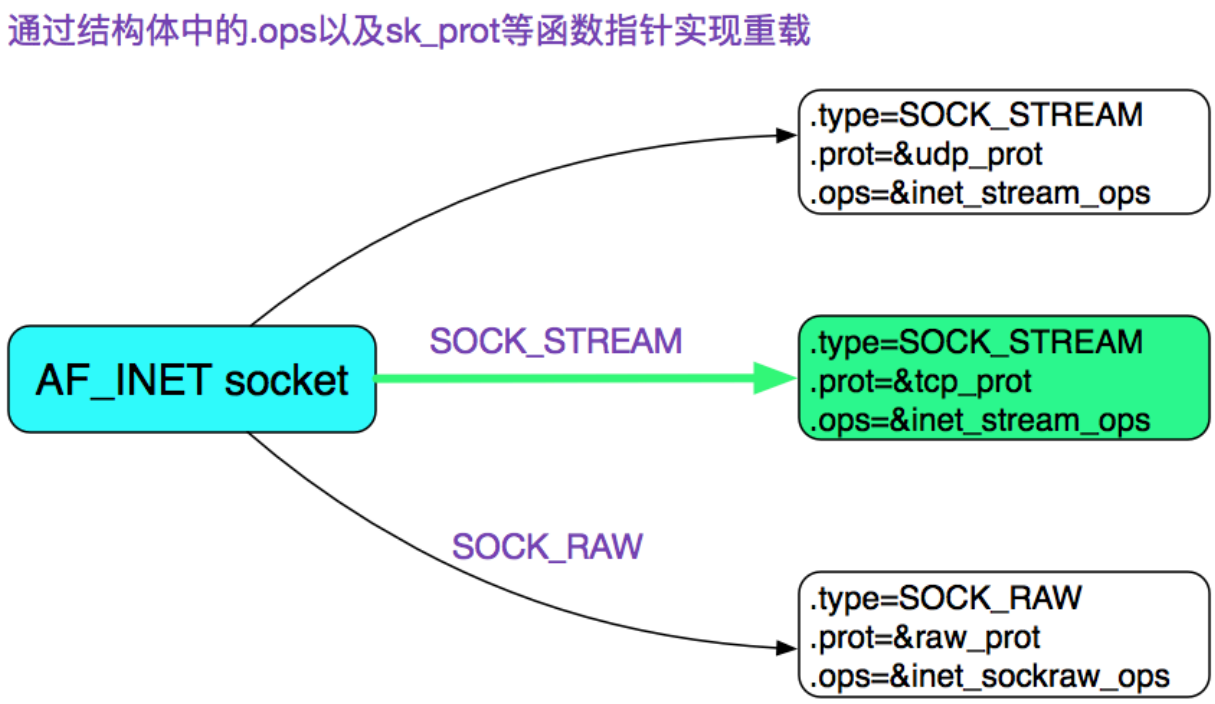
则通过源码可知，由于是AF_INET(PF_INET),所以net_families[PF_INET].create=inet_create(以后我们都用PF_INET表示)，即pf->create = inet_create; 进一步追溯调用:

```
inet_create(struct net *net, struct socket *sock, int protocol){
        Sock* sock;
        ......
        // 此处是寻找对应协议处理器的过程
lookup_protocol:
        // 迭代寻找protocol==answer->protocol的情况
        list_for_each_rcu(p, &inetsw[sock->type]) answer = list_entry(p, st

                /* Check the non-wild match. */
                if (protocol == answer->protocol) {
                        if (protocol != IPPROTO_IP)
                                break;
                }
        ......
        // 这边answer指的是SOCK_STREAM
        sock->ops = answer->ops;
        answer_no_check = answer->no_check;
        // 这边sk->prot就是answer_prot=>tcp_prot
        sk = sk_alloc(net, PF_INET, GFP_KERNEL, answer_prot);
        sock_init_data(sock, sk);
        ......
}
```

上面的代码就是在INET中寻找SOCK_STREAM的过程了 我们再看一下inetsw[SOCK_STREAM]的具体配置:

```
static struct inet_protosw inetsw_array[] =
{
        {
                .type =       SOCK_STREAM,
                .protocol =   IPPROTO_TCP,
                .prot =       &tcp_prot,
                .ops =        &inet_stream_ops,
                .capability = -1,
                .no_check =   0,
                .flags =      INET_PROTOSW_PERMANENT |
                              INET_PROTOSW_ICSK,
        },
        ......
}
```

这边也用了重载，AF_INET有TCP、UDP以及Raw三种:



从上述代码，我们可以清楚的发现sock->ops=&inet_stream_ops;

```
const struct proto_ops inet_stream_ops = {
        .family             = PF_INET,
        .owner              = THIS_MODULE,
        ......
        .sendmsg            = tcp_sendmsg,
        .recvmsg            = sock_common_recvmsg,
        ......
}
```

即sock->ops->recvmsg = sock_common_recvmsg;

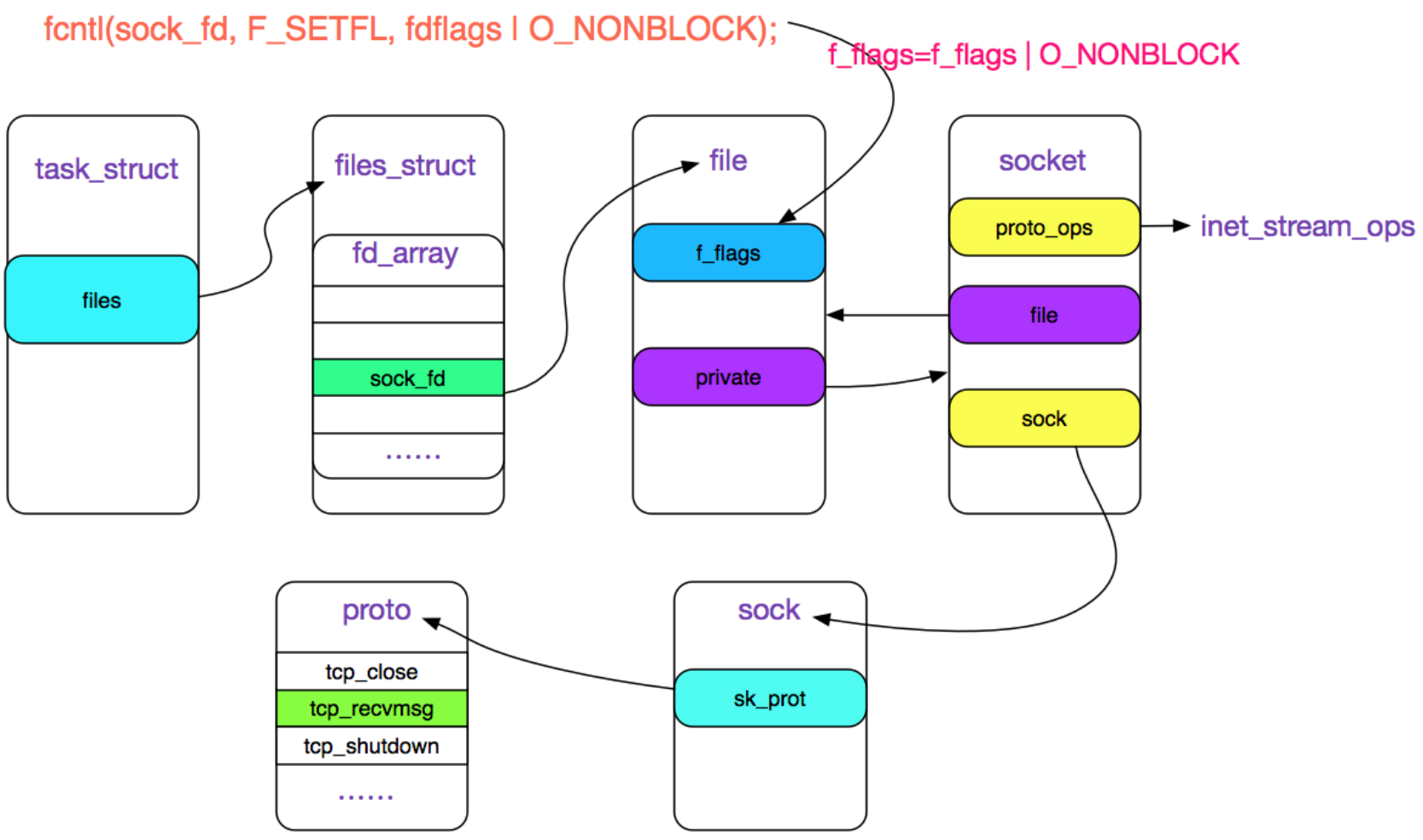
同时sock->sk->sk_prot = tcp_prot;

我们再看下tcp_prot中的各个函数重载的定义:

```
struct proto tcp_prot = {
        .name                   = "TCP",
        .close                  = tcp_close,
        .connect                = tcp_v4_connect,
        .disconnect             = tcp_disconnect,
        .accept                 = inet_csk_accept,
        ......
        // 我们重点考察tcp的读
        .recvmsg                = tcp_recvmsg,
        ......
}
```

fcntl控制socket的阻塞\非阻塞状态

我们用fcntl修改socket的阻塞\非阻塞状态。 事实上: fcntl的作用就是将O_NONBLOCK标志位存储在sock_fd对应的filp结构的f_lags里,如下图所示。




```
fcntl(sock_fd, F_SETFL, fdflags | O_NONBLOCK);
        |->setfl
```

追踪setfl代码:

```
static int setfl(int fd, struct file * filp, unsigned long arg) {
        ......
        filp->f_flags = (arg & SETFL_MASK) | (filp->f_flags & ~SETFL_MASK);
        ......
}
```

上图中，由sock_fd在task_struct(进程结构体)->files_struct->fd_array中找到对应的socket的file描述符，再修改file->flags

在调用socket.recv的时候

我们跟踪源码调用:

```
socket.recv
        |->sys_recv
                |->sys_recvfrom
                        |->sock_recvmsg
                                |->__sock_recvmsg
                                        |->sock->ops->recvmsg
```

由上文可知: sock->ops->recvmsg = sock_common_recvmsg;

sock

值得注意的是,在sock_recmsg中,有对标识O_NONBLOCK的处理

```
        if (sock->file->f_flags & O_NONBLOCK)
                flags |= MSG_DONTWAIT;
```

上述代码中sock关联的file中获取其f_flags,如果flags有O_NONBLOCK标识，那么就设置msg_flags为MSG_DONTWAIT(不等待)。
fcntl与socket就是通过其共同操作File结构关联起来的。

继续跟踪调用

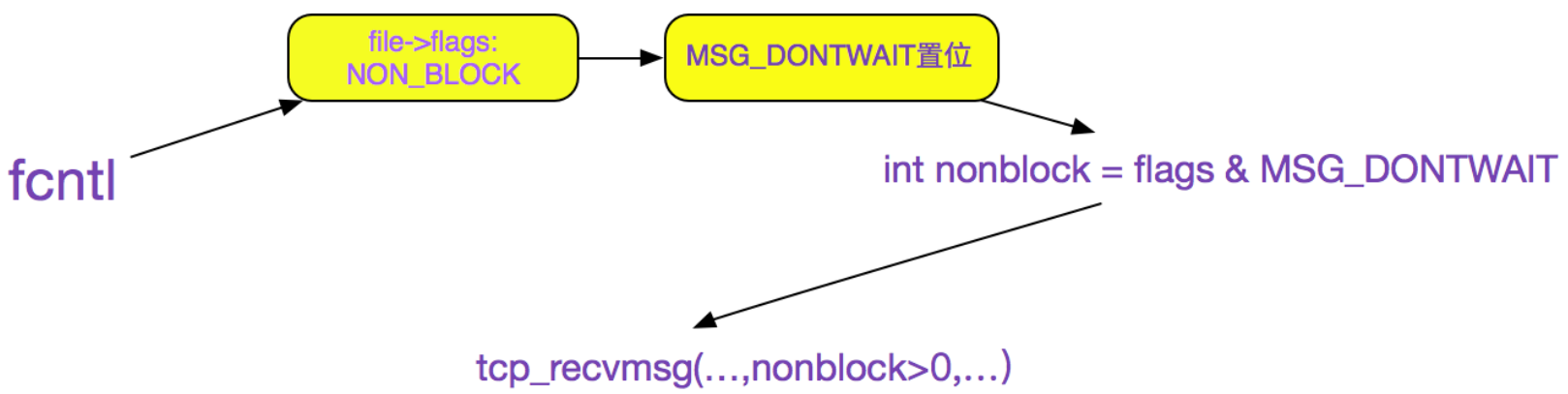
sock_common_recvmsg

```
int sock_common_recvmsg(struct kiocb *iocb, struct socket *sock,
                        struct msghdr *msg, size_t size, int flags) {
```

```
        ......
        // 如果flags的MSG_DONTWAIT标识置位，则传给recvmsg的第5个参数为正,否则为0
        err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags & MSG_DONTWAI
                                   flags & ~MSG_DONTWAIT, &addr_len);
        .....
}
```

由上文可知: sk->sk_prot->recvmsg 其中sk_prot=tcp_prot,即最终调用的是tcp_prot->tcp_recvmsg,

上面的代码可以看出，如果fcntl(O_NONBLOCK)=>MSG_DONTWAIT置位=>(flags & MSG_DONTWAIT)>0, 再结合tcp_recvmsg的函数签名,即如果设置了O_NONBLOCK的话，设置给tcp_recvmsg的nonblock参数>0,关系如下图所示:



最终的调用逻辑tcp_recvmsg

首先我们看下tcp_recvmsg的函数签名:

```
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len, int nonblock, int flags, int *addr_len)
```

显然我们关注焦点在(int nonblock这个参数上):

```
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len, int nonblock, int flags, int *addr_len){
        ......
        // copied是指向用户空间拷贝了多少字节，即读了多少
        int copied;
        // target指的是期望多少字节
        int target;
        // 等效为timo = noblock ? 0 : sk->sk_rcvtimeo;
        timeo = sock_rcvtimeo(sk, nonblock);
        ......
```

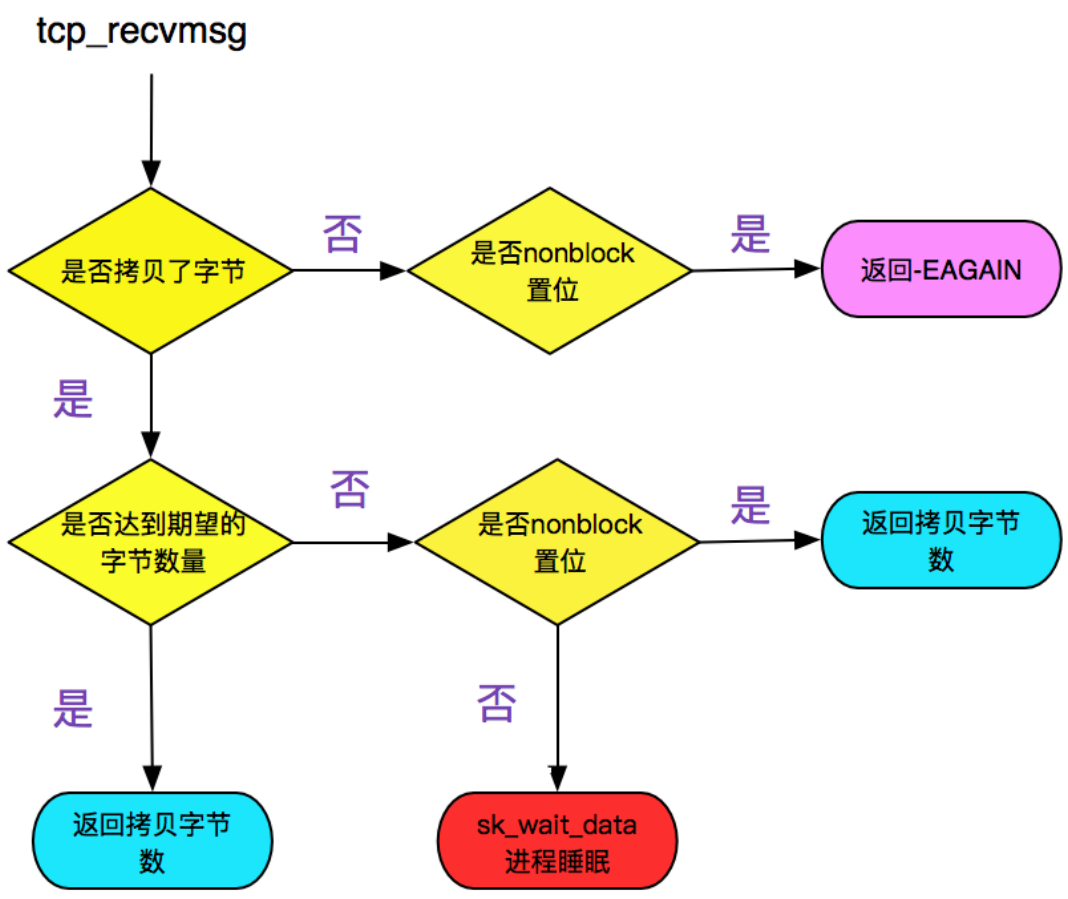
```
// 如果设置了MSG_WAITALL标识target=需要读的长度
// 如果未设置，则为最低低水位值
target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
......

do{
        // 表明读到数据
        if (copied) {
                // 注意，这边只要!timeo，即nonblock设置了就会跳出循环
                if (sk->sk_err ||
                    sk->sk_state == TCP_CLOSE ||
                    (sk->sk_shutdown & RCV_SHUTDOWN) ||
                    !timeo ||
                    signal_pending(current) ||
                    (flags & MSG_PEEK))
                break;
        }else{
                // 到这里，表明没有读到任何数据
                // 且nonblock设置了导致timeo=0，则返回-EAGAIN,符合我们的
                if (!timeo) {
                        copied = -EAGAIN;
                        break;
        }
        // 这边如果堵到了期望的数据，继续，否则当前进程阻塞在sk_wait_data上
                if (copied >= target) {
                        /* Do not sleep, just process backlog. */
                        release_sock(sk);
                        lock_sock(sk);
                } else
                        sk_wait_data(sk, &timeo);
        } while (len > 0);
        ......
        return copied
}
```

上面的逻辑归结起来就是:

⑴在设置了nonblock的时候，如果copied>0,则返回读了多少字节,如果copied=0，则返回-EAGAIN,提示应用重复调用。

⑵如果没有设置nonblock，如果读取的数据>=期望，则返回读取了多少字节。如果没有则用sk_wait_data将当前进程等待。

如下流程图所示:

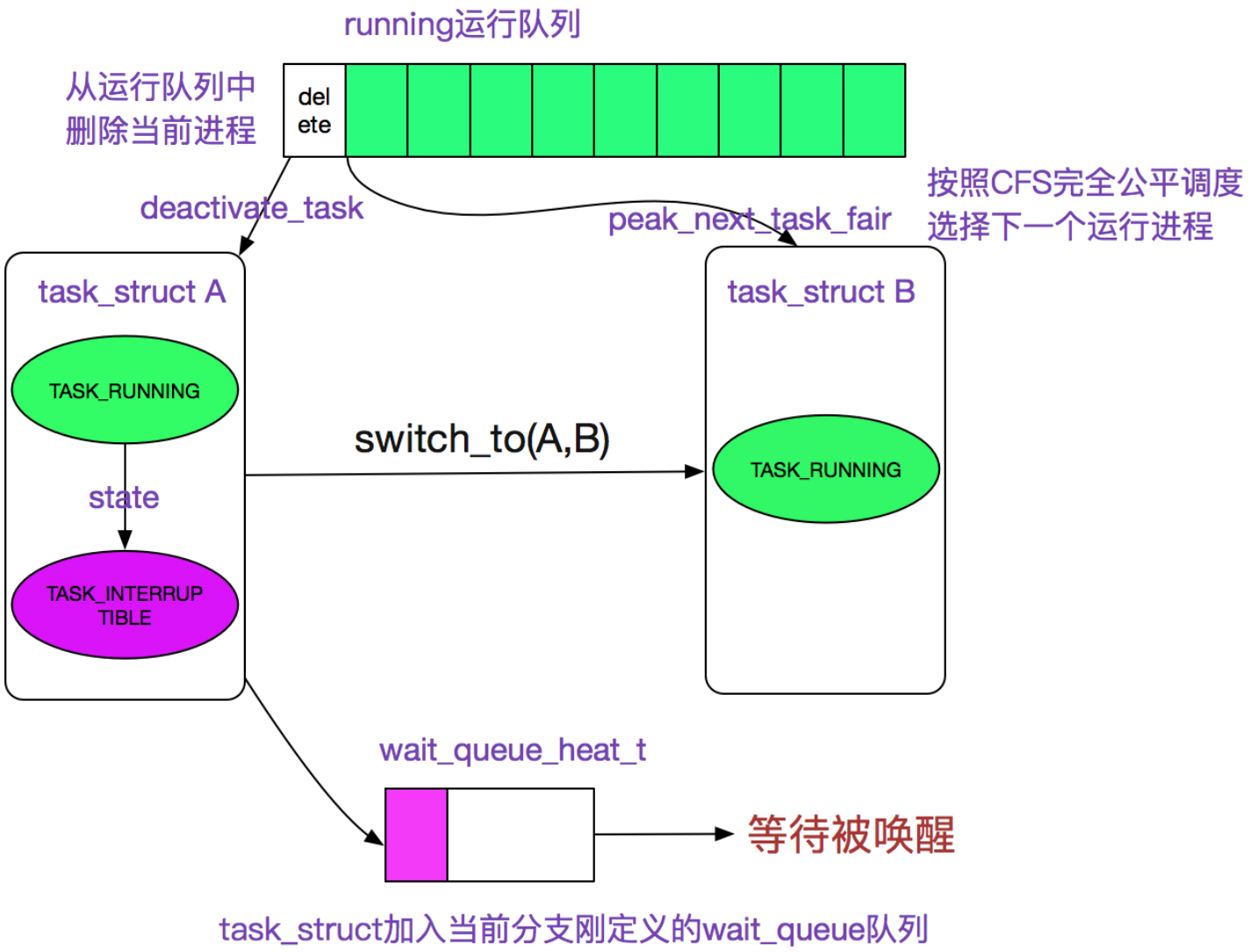


阻塞函数sk_wait_data

sk_wait_data代码-函数为:

```
// 将进程状态设置为可打断INTERRUPTIBLE
prepare_to_wait(sk->sk_sleep, &wait, TASK_INTERRUPTIBLE);
set_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);
// 通过调用schedule_timeout让出CPU，然后进行睡眠
rc = sk_wait_event(sk, timeo, !skb_queue_empty(&sk->sk_receive_queu
// 到这里的时候，有网络事件或超时事件唤醒了此进程，继续运行
clear_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);
finish_wait(sk->sk_sleep, &wait);
```

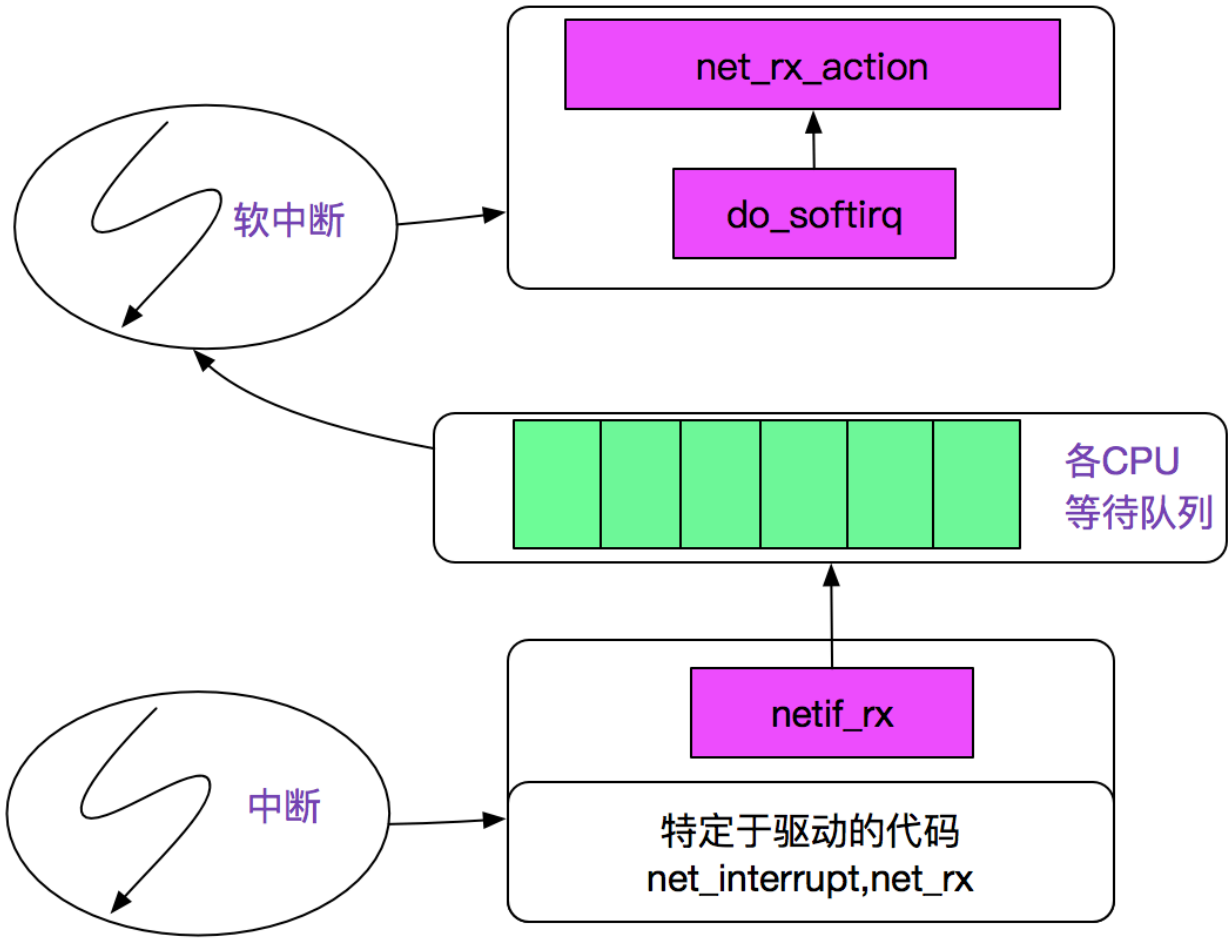
该函数调用schedule_timeout进入睡眠，其进一步调用了schedule函数，首先从运行队列删除，其次加入到等待队列，最后调用和体系结构相关的switch_to宏来完成进程间的切换。
如下图所示:



阻塞后什么时候恢复运行呢

情况1:有对应的网络数据到来

首先我们看下网络分组到来的内核路径，网卡发起中断后调用netif_rx将事件挂入CPU的等待队列，并唤起软中断(soft_irq)，再通过linux的软中断机制调用net_rx_action，如下图所示:



注:上图来自PLKA(<<深入Linux内核架构>>)

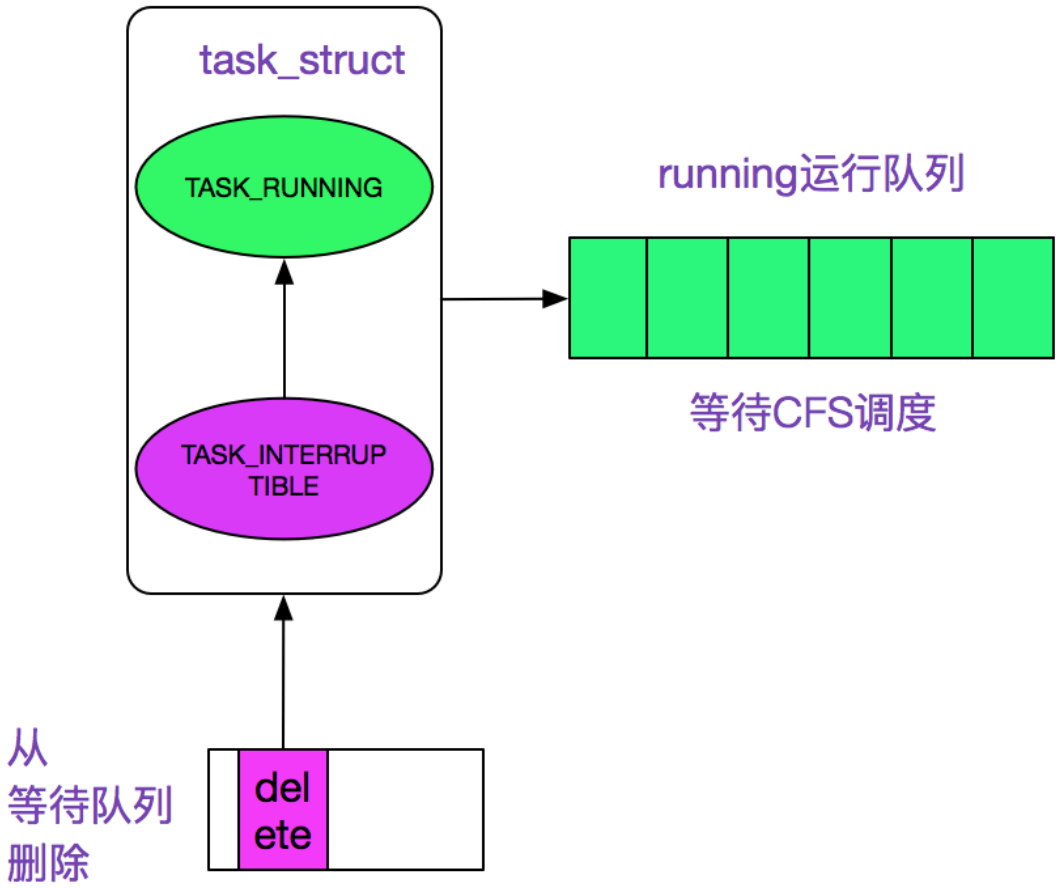
紧接着跟踪next_rx_action

```
next_rx_action
        |-process_backlog
                .......
                        |->packet_type->func 在这里我们考虑ip_rcv
                                        |->ipprot->handler 在这里ipprot重载为
                                                (handler 即为tcp_v4_rcv)
```

紧接着tcp_v4_rcv:

```
tcp_input.c
tcp_v4_rcv
        |-tcp_v4_do_rcv
                |-tcp_rcv_state_process
                        |-tcp_data_queue
                                |-sk->sk_data_ready=sock_def_readable
                                        |-wake_up_interruptible
                                                |-__wake_up
                                                        |-__wake_up_common
```

在这里__wake_up_common将停在当前wait_queue_head_t中的进程唤醒，即状态改为task_running，等待CFS调度以进行下一步的动作,如下图所示。



情况2:设定的超时时间到来

在前面调用sk_wait_event中调用了schedule_timeout

```
fastcall signed long __sched schedule_timeout(signed long timeout) {
        ......
        // 设定超时的回掉函数为process_timeout
        setup_timer(&timer, process_timeout, (unsigned long)current);
        __mod_timer(&timer, expire);
        // 这边让出CPU
        schedule();
        del_singleshot_timer_sync(&timer);
        timeout = expire - jiffies;
 out:
        // 返回经过了多长事件
        return timeout < 0 ? 0 : timeout;
}
```

process_timeout函数即是将此进程重新唤醒

```
static void process_timeout(unsigned long __data)
{
        wake_up_process((struct task_struct *)__data);
}
```

总结

linux内核源代码博大精深，阅读其代码很费周折。希望笔者这篇文章能帮助到阅读linux网络协议栈代码的人。

原文地址

https://my.oschina.net/alchemystar/blog/1791017

额外添加

我的博客即将搬运同步至腾讯云+社区，邀请大家一同入驻：https://cloud.tencent.com/developer/support-plan

标签： tcpip