



深入揭秘 epoll 是如何实现 IO 多路复用的



腾讯技术工程
编程话题下的优秀答主

关注

548 人赞同了该文章

作者：yanfeizhang，腾讯 PCG 后开开发工程师

提起 epoll，大家都不陌生，知道它性能不错。但是它内部是如何工作的，如何达到高性能的效果呢，鲜有文章能把原理介绍清楚，所以我就撸起袖子搞了一篇文章，献给大家。

进程在 Linux 上是一个开销不小的家伙，先不说创建，光是上下文切换一次就得几个微秒。所以为了高效地对海量用户提供服务，必须要让一个进程能同时处理很多个 tcp 连接才行。现在假设一个进程保持了 10000 条连接，那么如何发现哪条连接上有数据可读了、哪条连接可写了？

我们当然可以采用循环遍历的方式来发现 IO 事件，但这种方式太低级了。我们希望有一种更高效的机制，在很多连接中的某条上有 IO 事件发生的时候直接快速把它找出来。其实这个事情 Linux 操作系统已经替我们都做好了，它就是我们所熟知的 **IO 多路复用机制**。这里的复用指的就是对进程的复用。

在 Linux 上多路复用方案有 select、poll、epoll。它们三个中 epoll 的性能表现是最优秀的，能支持的并发量也最大。所以我们今天把 epoll 作为要拆解的对象，深入揭秘内核是如何实现多路的 IO 管理的。

为了方便讨论，我们举一个使用了 epoll 的简单示例（只是个例子，实践中不这么写）：

```
int main(){
    listen(lfd, ...);

    cfd1 = accept(...);
    cfd2 = accept(...);
    efd = epoll_create(...);

    epoll_ctl(efd, EPOLL_CTL_ADD, cfd1, ...);
    epoll_ctl(efd, EPOLL_CTL_ADD, cfd2, ...);
    epoll_wait(efd, ...)
}
```

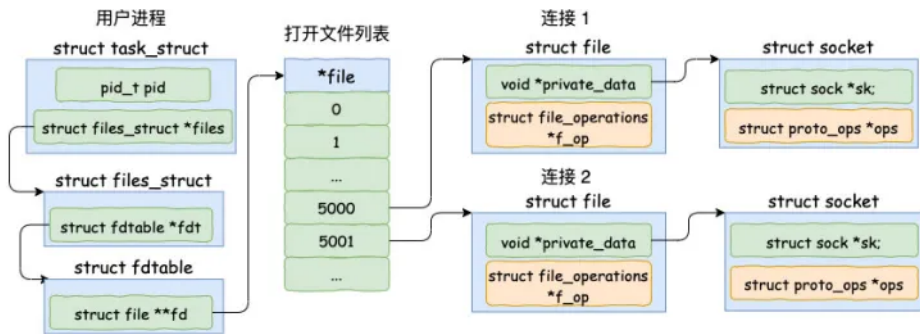
其中和 epoll 相关的函数是如下三个：

- `epoll_create`：创建一个 epoll 对象
- `epoll_ctl`：向 epoll 对象中添加要管理的连接
- `epoll_wait`：等待被管理的连接上的 IO 事件

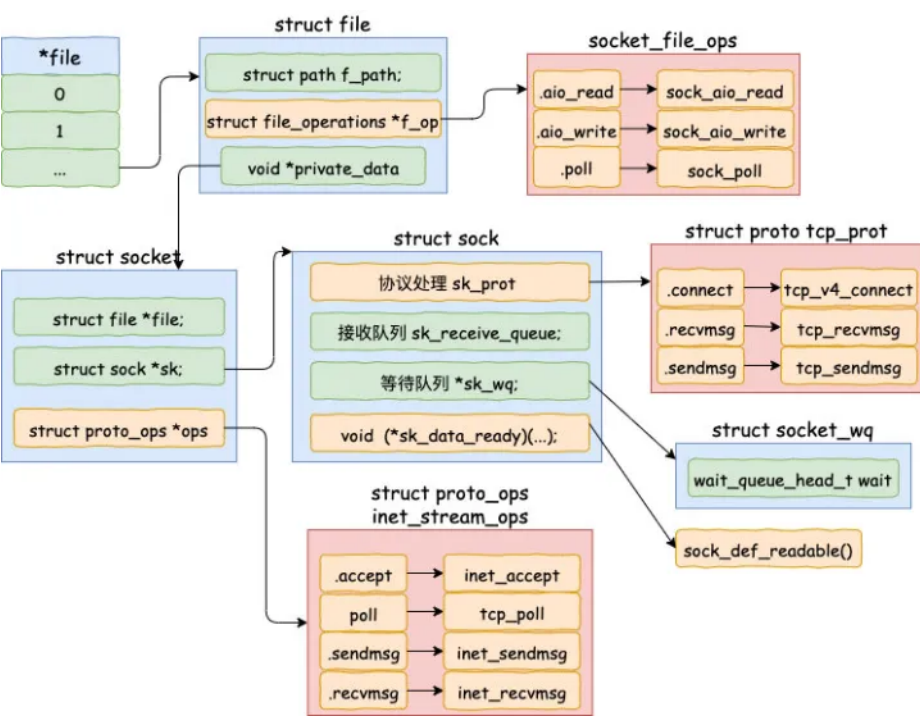
友情提示，万字长文，慎入！！

一、accept 创建新 socket

我们直接从服务器端的 accept 讲起。当 accept 之后，进程会创建一个新的 socket 出来，专门用于和对应的客户端通信，然后把它放到当前进程的打开文件列表中。



其中一条连接的 socket 内核对象更为具体一点的结构图如下。



接下来我们来看一下接收连接时 socket 内核对象的创建源码。accept 的系统调用代码位于源文件 net/socket.c 下。

```
//file: net/socket.c
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr,
                int __user *, upeer_addrlen, int, flags)
{
    struct socket *sock, *newsock;

    //根据 fd 查找到监听的 socket
    sock = sockfd_lookup_light(fd, &err, &fput_needed);

    //1.1 申请并初始化新的 socket
    newsock = sock_alloc();
    newsock->type = sock->type;
    newsock->ops = sock->ops;

    reator->name);
```



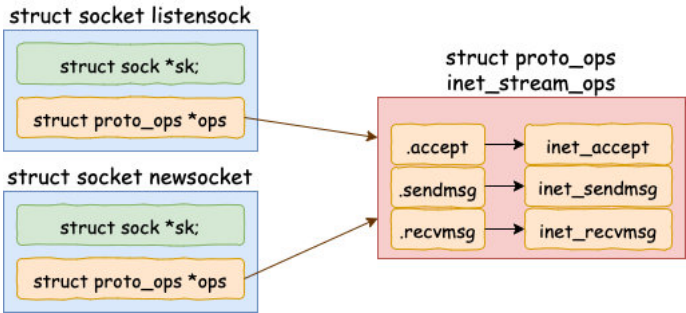
```
.....

//1.3 接收连接
err = sock->ops->accept(sock, newsock, sock->file->f_flags);

//1.4 添加新文件到当前进程的打开文件列表
fd_install(newfd, newfile);
```

1.1 初始化 struct socket 对象

在上述的源码中，首先是调用 sock_alloc 申请一个 struct socket 对象出来。然后接着把 listen 状态的 socket 对象上的协议操作函数集合 ops 赋值给新的 socket。（对于所有的 AF_INET 协议族下的 socket 来说，它们的 ops 方法都是一样的，所以这里可以直接复制过来）

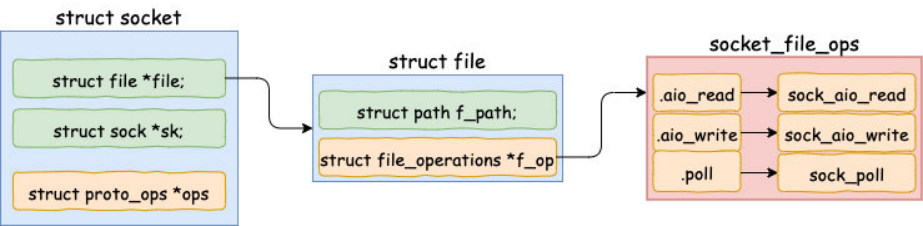


其中 inet_stream_ops 的定义如下

```
//file: net/ipv4/af_inet.c
const struct proto_ops inet_stream_ops = {
    ...
    .accept      = inet_accept,
    .listen      = inet_listen,
    .sendmsg     = inet_sendmsg,
    .recvmsg     = inet_recvmsg,
    ...
}
```

1.2 为新 socket 对象申请 file

struct socket 对象中有一个重要的成员 -- file 内核对象指针。这个指针初始化的时候是空的。在 accept 方法里会调用 sock_alloc_file 来申请内存并初始化。然后将新 file 对象设置到 sock->file 上。



来看 sock_alloc_file 的实现过程：

```
struct file *sock_alloc_file(struct socket *sock, int flags,
                             const char *dname)
{
    struct file *file;
    file = alloc_file(&path, FMODE_READ | FMODE_WRITE,
                     &socket_file_ops);
    .....
    sock->file = file;
}
```



sock_alloc_file 又会接着调用到 alloc_file。注意在 alloc_file 方法中，把 socket_file_ops 函数集合并赋到了新 file->f_op 里了。

```
//file: fs/file_table.c
struct file *alloc_file(struct path *path, fmode_t mode,
                        const struct file_operations *fop)
{
    struct file *file;
    file->f_op = fop;
    .....
}
```

socket_file_ops 的具体定义如下：

```
//file: net/socket.c
static const struct file_operations socket_file_ops = {
    ...
    .aio_read  = sock_aio_read,
    .aio_write = sock_aio_write,
    .poll      = sock_poll,
    .release   = sock_close,
    ...
};
```

这里看到，在 accept 里创建的新 socket 里的 file->f_op->poll 函数指向的是 sock_poll。接下来我们会调用到它，后面我们再说。

其实 file 对象内部也有一个 socket 指针，指向 socket 对象。

1.3 接收连接

在 socket 内核对象中除了 file 对象指针以外，有一个核心成员 sock。

```
//file: include/linux/net.h
struct socket {
    struct file    *file;
    struct sock    *sk;
}
```

这个 struct sock 数据结构非常大，是 socket 的核心内核对象。发送队列、接收队列、等待队列等核心数据结构都位于此。其定义位置文件 include/net/sock.h，由于太长就不展示了。

在 accept 的源码中：

```
//file: net/socket.c
SYSCALL_DEFINE4(accept4, ...)
{
    ...
    //1.3 接收连接
    err = sock->ops->accept(sock, newsock, sock->file->f_flags);
}
```

sock->ops->accept 对应的方法是 inet_accept。它执行的时候会从握手队列里直接获取创建好的 sock。sock 对象的完整创建过程涉及到三次握手，比较复杂，不展开了说了。咱们只看 struct sock 初始化过程中用到的一个函数：

```
void sock_init_data(struct socket *sock, struct sock *sk)
{
    sk->sk_wq = NULL;
    sk->sk_data_ready = sock_def_readable;
```

1.4 添加新文件到当前进程的打开文件列表中

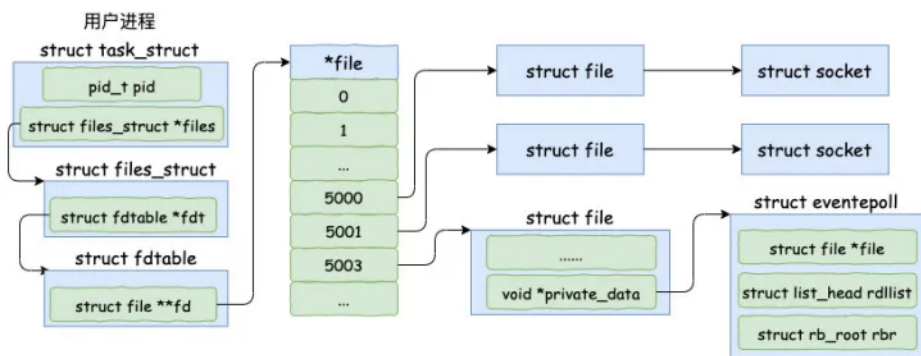
当 file、socket、sock 等关键内核对象创建完毕以后，剩下要做的一件事情就是把它挂到当前进程的打开文件列表中就行了。

```
//file: fs/file.c
void fd_install(unsigned int fd, struct file *file)
{
    __fd_install(current->files, fd, file);
}

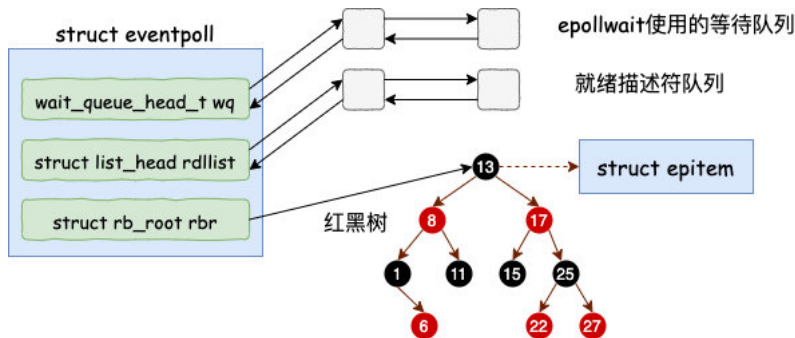
void __fd_install(struct files_struct *files, unsigned int fd,
                 struct file *file)
{
    ...
    fdt = files_fdttable(files);
    BUG_ON(fdt->fd[fd] != NULL);
    rcu_assign_pointer(fdt->fd[fd], file);
}
```

二、epoll_create 实现

在用户进程调用 epoll_create 时，内核会创建一个 struct eventpoll 的内核对象。并同样把它关联到当前进程的已打开文件列表中。



对于 struct eventpoll 对象，更详细的结构如下（同样只列出和今天主题相关的成员）。



epoll_create 的源代码相对比较简单。在 fs/eventpoll.c 下

```
// file: fs/eventpoll.c
SYSCALL_DEFINE1(epoll_create1, int, flags)
{
    struct eventpoll *ep = NULL;

    //创建一个 eventpoll 对象
    error = en_alloc(&en):
```



struct eventpoll 的定义也在这个源文件中。

```
// file: fs/eventpoll.c
struct eventpoll {

    //sys_epoll_wait用到的等待队列
    wait_queue_head_t wq;

    //接收就绪的描述符都会放到这里
    struct list_head rdllist;

    //每个epoll对象中都有一颗红黑树
    struct rb_root rbr;

    .....
}
```

eventpoll 这个结构体中的几个成员的含义如下：

- **wq**：等待队列链表。软中断数据就绪的时候会通过 wq 来找到阻塞在 epoll 对象上的用户进程。
- **rbr**：一棵红黑树。为了支持对海量连接的高效查找、插入和删除，eventpoll 内部使用了一棵红黑树。通过这棵树来管理用户进程下添加进来的所有 socket 连接。
- **rdllist**：就绪的描述符的链表。当有的连接就绪的时候，内核会把就绪的连接放到 rdllist 链表里。这样应用进程只需要判断链表就能找出就绪进程，而不用去遍历整棵树。

当然这个结构被申请完之后，需要做一点点的初始化工作，这都在 ep_alloc 中完成。

```
//file: fs/eventpoll.c
static int ep_alloc(struct eventpoll **pep)
{
    struct eventpoll *ep;

    //申请 epollevnt 内存
    ep = kzalloc(sizeof(*ep), GFP_KERNEL);

    //初始化等待队列头
    init_waitqueue_head(&ep->wq);

    //初始化就绪列表
    INIT_LIST_HEAD(&ep->rdllist);

    //初始化红黑树指针
    ep->rbr = RB_ROOT;

    .....
}
```

说到这儿，这些成员其实只是刚被定义或初始化了，还都没有被使用。它们会在下面被用到。

三、epoll_ctl 添加 socket

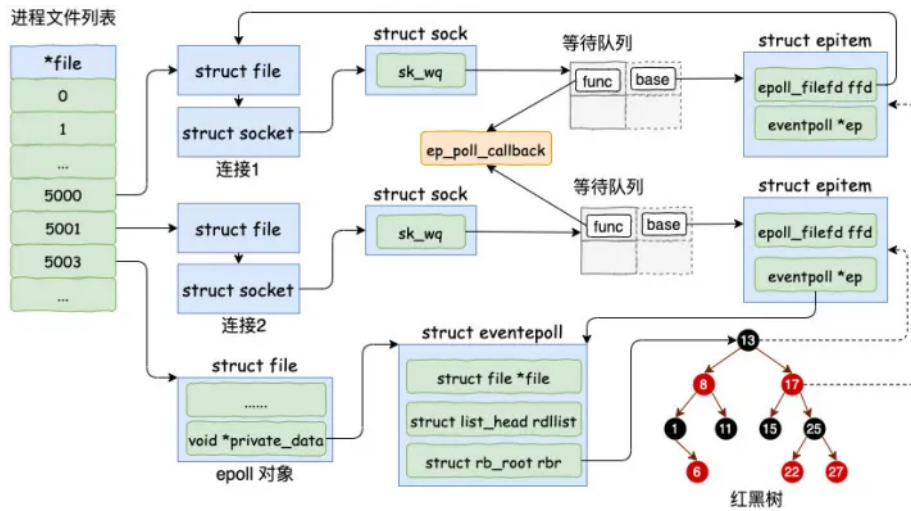
理解这一步是理解整个 epoll 的关键。

为了简单，我们只考虑使用 EPOLL_CTL_ADD 添加 socket，先忽略删除和更新。

假设我们现在和客户端们的多个连接的 socket 都创建好了，也创建好了 epoll 内核对象。在使用 epoll_ctl 注册每一个 socket 的时候，内核会做如下三件事情

1. 分配一个红黑树节点对象 epitem,
2. 添加等待事件到 socket 的等待队列中，其回调函数是 ep_poll_callback
3. 将 epitem 插入到 epoll 对象的红黑树里

关系图大致如下：



我们来详细看看 socket 是如何添加到 epoll 对象里的，找到 epoll_ctl 的源码。

```
// file: fs/eventpoll.c
SYSCALL_DEFINE4(epoll_ctl, int, epfd, int, op, int, fd,
                struct epoll_event __user *, event)
{
    struct eventpoll *ep;
    struct file *file, *tfile;

    //根据 epfd 找到 eventpoll 内核对象
    file = fget(epfd);
    ep = file->private_data;

    //根据 socket 句柄号, 找到其 file 内核对象
    tfile = fget(fd);

    switch (op) {
    case EPOLL_CTL_ADD:
        if (!epi) {
            epds.events |= POLLERR | POLLHUP;
            error = ep_insert(ep, &epds, tfile, fd);
        } else
            error = -EEXIST;
        clear_tfile_check_list();
        break;
    }
}
```

在 epoll_ctl 中首先根据传入 fd 找到 eventpoll、socket 相关的内核对象。对于 EPOLL_CTL_ADD 操作来说，会然后执行到 ep_insert 函数。所有的注册都是在这个函数中完成的。

```
//file: fs/eventpoll.c
static int ep_insert(struct eventpoll *ep,
                    struct epoll_event *event,
                    struct file *tfile, int fd)
{
    //3.1 分配并初始化 epitem
    //分配一个epi对象
    struct epitem *epi;
    if (!(epi = kmem_cache_alloc(epi_cache, GFP_KERNEL)))
        return -ENOMEM;

    //对分配的epi进行初始化
    //epi->ffd中存了句柄号和struct file对象地址
    INIT_LIST_HEAD(&epi->pwqlist);
    epi->ep = ep;
    ep_set_ffd(&epi->ffd, tfile, fd);
}
```



```
//定义并初始化 ep_queue 对象
struct ep_queue epq;
epq.epi = epi;
init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);

//调用 ep_ptable_queue_proc 注册回调函数
//实际注入的函数为 ep_poll_callback
revents = ep_item_poll(epi, &epq.pt);

.....
//3.3 将epi插入到 eventpoll 对象中的红黑树中
ep_rbtrees_insert(ep, epi);
.....
}
```

3.1 分配并初始化 epitem

对于每一个 socket，调用 epoll_ctl 的时候，都会为之分配一个 epitem。该结构的主要数据如下：

```
//file: fs/eventpoll.c
struct epitem {

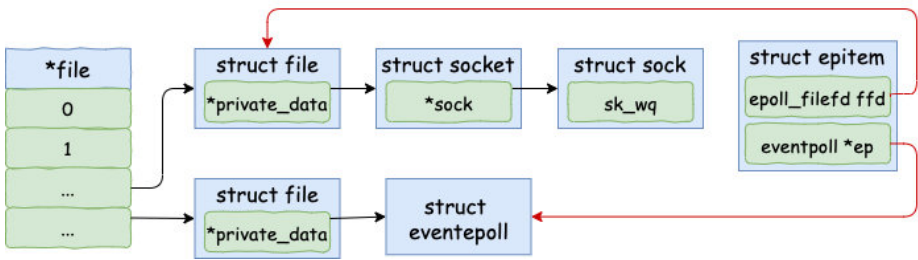
    //红黑树节点
    struct rb_node rbn;

    //socket文件描述符信息
    struct epoll_filefd ffd;

    //所归属的 eventpoll 对象
    struct eventpoll *ep;

    //等待队列
    struct list_head pwqlist;
}
```

对 epitem 进行了一些初始化，首先在 epi->ep = ep 这行代码中将其 ep 指针指向 eventpoll 对象。另外用要添加的 socket 的 file、fd 来填充 epitem->ffd。

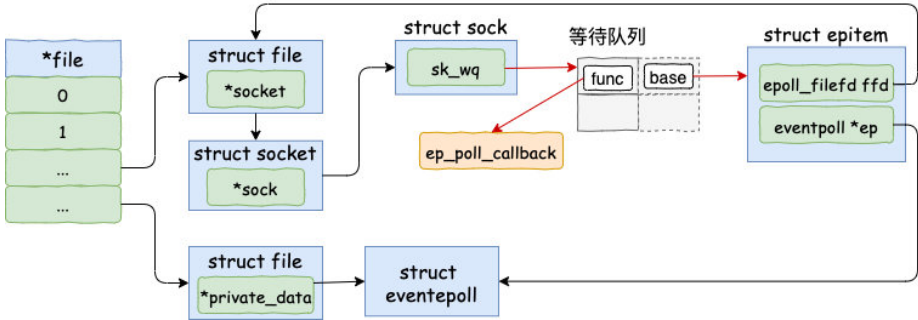


其中使用到的 ep_set_ffd 函数如下。

```
static inline void ep_set_ffd(struct epoll_filefd *ffd,
                             struct file *file, int fd)
{
    ffd->file = file;
    ffd->fd = fd;
}
```

3.2 设置 socket 等待队列

在创建 epitem 并初始化之后，ep_insert 中第二件事情就是设置 socket 对象上的等待任务队列。并把函数 fs/eventpoll.c 文件下的 ep_poll_callback 设置为数据就绪时候的回调函数。



这一块的源代码稍微有点绕，没有耐心的话直接跳到下面的加粗字体来看。首先来看 `ep_item_poll`。

```
static inline unsigned int ep_item_poll(struct epitem *epi, poll_table *pt)
{
    pt->_key = epi->event.events;

    return epi->ffd.file->f_op->poll(epi->ffd.file, pt) & epi->event.events;
}
```

看，这里调用到了 `socket` 下的 `file->f_op->poll`。通过上面第一节的 `socket` 的结构图，我们知道这个函数实际上是 `sock_poll`。

```
/* No kernel lock held - perfect */
static unsigned int sock_poll(struct file *file, poll_table *wait)
{
    ...
    return sock->ops->poll(file, sock, wait);
}
```

同样回看第一节里的 `socket` 的结构图，`sock->ops->poll` 其实指向的是 `tcp_poll`。

```
//file: net/ipv4/tcp.c
unsigned int tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
{
    struct sock *sk = sock->sk;

    sock_poll_wait(file, sk_sleep(sk), wait);
}
```

在 `sock_poll_wait` 的第二个参数传参前，先调用了 `sk_sleep` 函数。**在这个函数里它获取了 `sock` 对象下的等待队列列表头 `wait_queue_head_t`，待会等待队列项就插入这里。**这里稍微注意下，是 `socket` 的等待队列，不是 `epoll` 对象的。来看 `sk_sleep` 源码：

```
//file: include/net/sock.h
static inline wait_queue_head_t *sk_sleep(struct sock *sk)
{
    BUILD_BUG_ON(offsetof(struct socket_wq, wait) != 0);
    return &rcu_dereference_raw(sk->sk_wq)->wait;
}
```

接着真正进入 `sock_poll_wait`。

```
static inline void sock_poll_wait(struct file *filp,
    wait_queue_head_t *wait_address, poll_table *p)
{
    poll_wait(filp, wait_address, p);
}

static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, pol
```



```
p->qproc(filp, wait_address, p);
}
```

这里的 qproc 是个函数指针，它在前面的 init_poll_funcptr 调用时被设置成了 ep_ptable_queue_proc 函数。

```
static int ep_insert(...)
{
    ...
    init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
    ...
}

//file: include/linux/poll.h
static inline void init_poll_funcptr(poll_table *pt,
    poll_queue_proc qproc)
{
    pt->qproc = qproc;
    pt->key = ~0UL; /* all events enabled */
}
```

敲黑板!!! 注意，废了半天的劲，终于到了重点了！在 ep_ptable_queue_proc 函数中，新建了一个等待队列项，并注册其回调函数为 ep_poll_callback 函数。然后再将这个等待项添加到 socket 的等待队列中。

```
//file: fs/eventpoll.c
static void ep_ptable_queue_proc(struct file *file, wait_queue_head_t *whead,
    poll_table *pt)
{
    struct eppoll_entry *pwq;
    f (epi->nwait >= 0 && (pwq = kmem_cache_alloc(pwq_cache, GFP_KERNEL))) {
        //初始化回调方法
        init_waitqueue_func_entry(&pwq->wait, ep_poll_callback);

        //将ep_poll_callback放入socket的等待队列whead（注意不是epoll的等待队列）
        add_wait_queue(whead, &pwq->wait);
    }
}
```

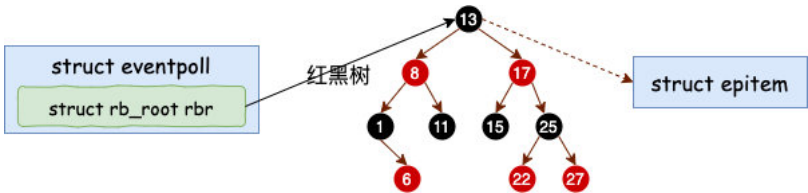
在前文 **深入理解高性能网络开发路上的绊脚石 - 同步阻塞网络 IO** 里阻塞式的系统调用 recvfrom 里，由于需要在数据就绪的时候唤醒用户进程，所以等待对象项的 private (这个变量名起的也是醉了) 会设置成当前用户进程描述符 current。而我们今天的 socket 是交给 epoll 来管理的，不需要在一个 socket 就绪的时候就唤醒进程，所以这里的 q->private 没有啥卵用就设置成了 NULL。

```
//file:include/linux/wait.h
static inline void init_waitqueue_func_entry(
    wait_queue_t *q, wait_queue_func_t func)
{
    q->flags = 0;
    q->private = NULL;

    //ep_poll_callback 注册到 wait_queue_t对象上
    //有数据到达的时候调用 q->func
    q->func = func;
}
```

如上，等待队列项中仅仅只设置了回调函数 q->func 为 ep_poll_callback。在后面的第 5 节数据来啦我们将看到，软中断将数据收到 socket 的接收队列后，会通过注册的这个 ep_poll_callback 函数来回调，进而通知到 epoll 对象。

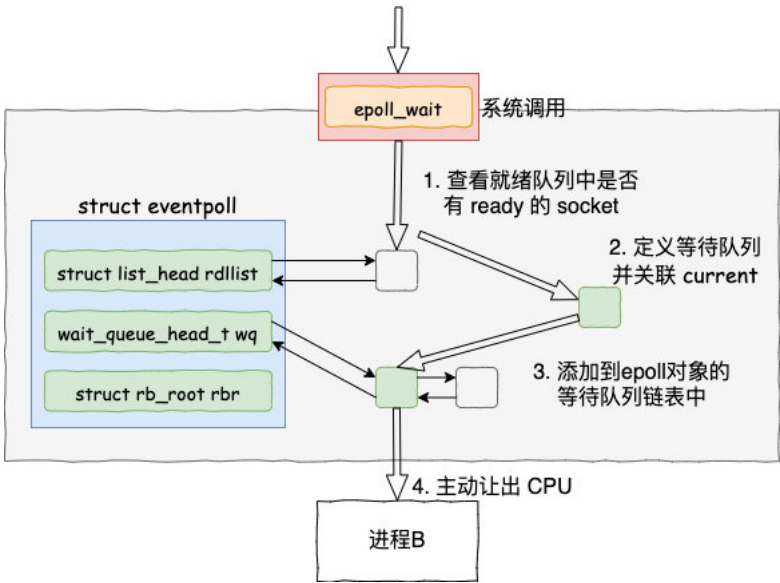
分配完 `epitem` 对象后，紧接着并把它插入到红黑树中。一个插入了一些 `socket` 描述符的 `epoll` 里的红黑树的示意图如下：



这里我们再聊聊为啥要用红黑树，很多人说是因为效率高。其实我觉得这个解释不够全面，要说查找效率树哪能比的上 `HASHTABLE`。我个人认为觉得更为合理的一个解释是为了让 `epoll` 在查找效率、插入效率、内存开销等等多个方面比较均衡，最后发现最适合这个需求的数据结构是红黑树。

四、`epoll_wait` 等待接收

`epoll_wait` 做的事情不复杂，当它被调用时它观察 `eventpoll->rdllist` 链表里有没有数据即可。有数据就返回，没有数据就创建一个等待队列项，将其添加到 `eventpoll` 的等待队列上，然后把自己阻塞掉就完事。



注意：`epoll_ctl` 添加 `socket` 时也创建了等待队列项。不同的是这里的等待队列项是挂在 `epoll` 对象上的，而前者是挂在 `socket` 对象上的。

其源代码如下：

```
//file: fs/eventpoll.c
SYSCALL_DEFINE4(epoll_wait, int, epfd, struct epoll_event __user *, events,
                int, maxevents, int, timeout)
{
    ...
    error = ep_poll(ep, events, maxevents, timeout);
}

static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events,
                  int maxevents, long timeout)
{
    wait_queue_t wait;
    .....

    fetch_events:
        //4.1 判断就绪队列上有没有事件就绪
        if (!ep_events_available(ep)) {
            //4.2 定义等待事件并关联当前进程
```



```
//4.3 把新 waitqueue 添加到 epoll->wq 链表里
__add_wait_queue_exclusive(&ep->wq, &wait);

for (;;) {
    ...
    //4.4 让出CPU 主动进入睡眠状态
    if (!schedule_hrtimeout_range(to, slack, HRTIMER_MODE_ABS))
        timed_out = 1;
    ...
}
```

4.1 判断就绪队列上有没有事件就绪

首先调用 `ep_events_available` 来判断就绪链表中是否有可处理的事件。

```
//file: fs/eventpoll.c
static inline int ep_events_available(struct eventpoll *ep)
{
    return !list_empty(&ep->rdllist) || ep->ovflist != EP_UNACTIVE_PTR;
}
```

4.2 定义等待事件并关联当前进程

假设确实没有就绪的连接，那接着会进入 `init_waitqueue_entry` 中定义等待任务，并把 `current`（当前进程）添加到 `waitqueue` 上。

是的，当没有 IO 事件的时候，`epoll` 也是会阻塞掉当前进程。这个是合理的，因为没有事情可做了占着 CPU 也没啥意义。网上的很多文章有个很不好的习惯，讨论阻塞、非阻塞等概念的时候都不说主语。这会导致你看的云里雾里。拿 `epoll` 来说，`epoll` 本身是阻塞的，但一般会把 `socket` 设置成非阻塞。只有说了主语，这些概念才有意义。

```
//file: include/linux/wait.h
static inline void init_waitqueue_entry(wait_queue_t *q, struct task_struct *p)
{
    q->flags = 0;
    q->private = p;
    q->func = default_wake_function;
}
```

注意这里的回调函数名称是 `default_wake_function`。后续在第 5 节数据来啦时将会调用到该函数。

4.3 添加到等待队列

```
static inline void __add_wait_queue_exclusive(wait_queue_head_t *q,
                                              wait_queue_t *wait)
{
    wait->flags |= WQ_FLAG_EXCLUSIVE;
    __add_wait_queue(q, wait);
}
```

在这里，把上一小节定义的等待事件添加到了 `epoll` 对象的等待队列中。

4.4 让出 CPU 主动进入睡眠状态

通过 `set_current_state` 把当前进程设置为可打断。调用 `schedule_hrtimeout_range` 让出 CPU，主动进入睡眠状态

```
//file: kernel/hrtimer.c
```



```
{
    return schedule_hrtimeout_range_clock(
        expires, delta, mode, CLOCK_MONOTONIC);
}

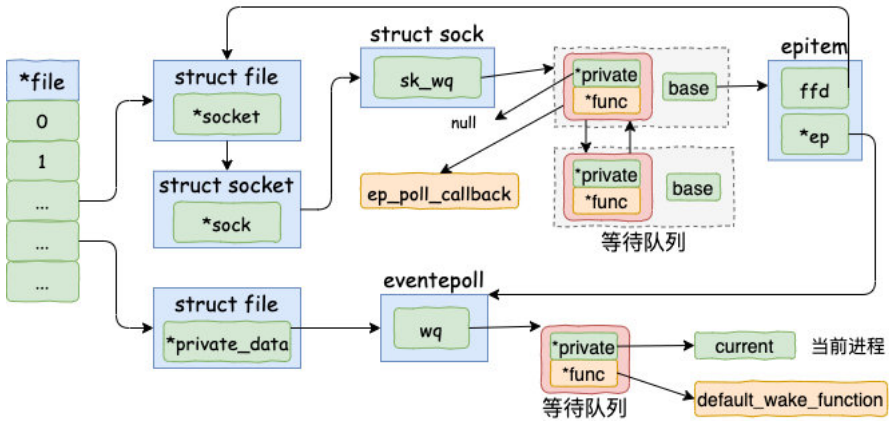
int __sched schedule_hrtimeout_range_clock(...)
{
    schedule();
    ...
}
```

在 schedule 中选择下一个进程调度

```
//file: kernel/sched/core.c
static void __sched __schedule(void)
{
    next = pick_next_task(rq);
    ...
    context_switch(rq, prev, next);
}
```

五、数据来啦

在前面 epoll_ctl 执行的时候，内核为每一个 socket 上都添加了一个等待队列项。在 epoll_wait 运行完的时候，又在 event poll 对象上添加了等待队列元素。在讨论数据开始接收之前，我们把这些队列项的内容再稍微总结一下。



- socket->sock->sk_data_ready 设置的就绪处理函数是 sock_def_readable
- 在 socket 的等待队列项中，其回调函数是 ep_poll_callback。另外其 private 没有用了，指向的是空指针 null。
- 在 eventpoll 的等待队列项中，回调函数是 default_wake_function。其 private 指向的是等待该事件的用户进程。

在这一小节里，我们将看到软中断是怎么样在数据处理完之后依次进入各个回调函数，最后通知到用户进程的。

5.1 接收数据到任务队列

关于软中断是怎么处理网络帧，为了避免篇幅过于臃肿，这里不再介绍。感兴趣的可以看文章《图解 Linux 网络包接收过程》。我们今天直接从 tcp 协议栈的处理入口函数 tcp_v4_rcv 开始说起。

```
// file: net/ipv4/tcp_ipv4.c
int tcp_v4_rcv(struct sk_buff *skb)
{
    .....
    th = tcp_hdr(skb); //获取tcp header
```



```
//根据数据包 header 中的 ip、端口信息查找到对应的socket
sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
.....

//socket 未被用户锁定
if (!sock_owned_by_user(sk)) {
    {
        if (!tcp_prequeue(sk, skb))
            ret = tcp_v4_do_rcv(sk, skb);
    }
}
}
```

在 tcp_v4_rcv 中首先根据收到的网络包的 header 里的 source 和 dest 信息来在本机上查询对应的 socket。找到以后，我们直接进入接收的主体函数 tcp_v4_do_rcv 来看。

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    if (sk->sk_state == TCP_ESTABLISHED) {

        //执行连接状态下的数据处理
        if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {
            rsk = sk;
            goto reset;
        }
        return 0;
    }

    //其它非 ESTABLISH 状态的数据包处理
    .....
}
```

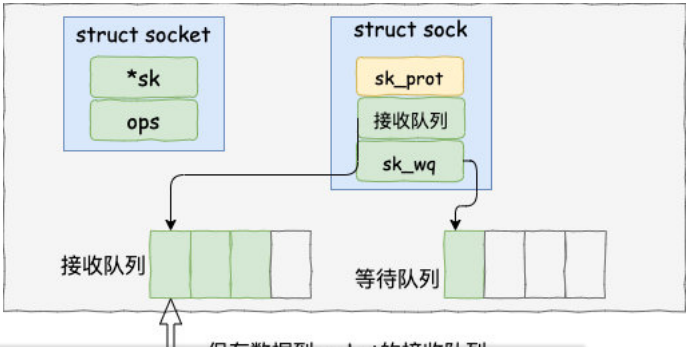
我们假设处理的是 ESTABLISH 状态下的包，这样就又进入 tcp_rcv_established 函数中进行处理。

```
//file: net/ipv4/tcp_input.c
int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
                        const struct tcphdr *th, unsigned int len)
{
    .....

    //接收数据到队列中
    eaten = tcp_queue_rcv(sk, skb, tcp_header_len,
                          &fragstolen);

    //数据 ready，唤醒 socket 上阻塞掉的进程
    sk->sk_data_ready(sk, 0);
}
```

在 tcp_rcv_established 中通过调用 tcp_queue_rcv 函数中完成了将接收数据放到 socket 的接收队列上。



如下源码所示：

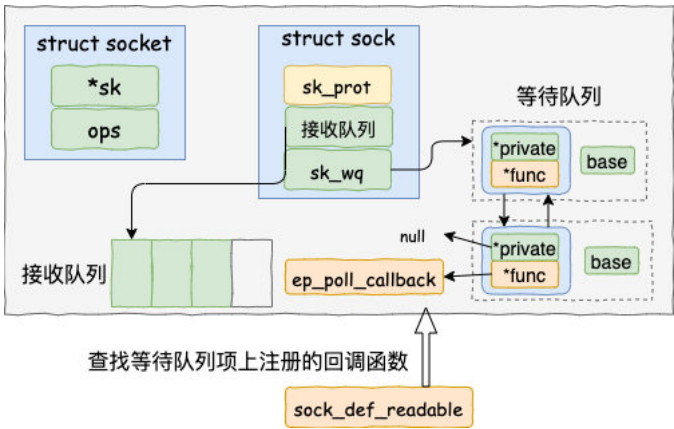


```
//file: net/ipv4/tcp_input.c
static int __must_check tcp_queue_rcv(struct sock *sk, struct sk_buff *skb, int hdrlen,
    bool *fragstolen)
{
    //把接收到的数据放到 socket 的接收队列的尾部
    if (!eaten) {
        __skb_queue_tail(&sk->sk_receive_queue, skb);
        skb_set_owner_r(skb, sk);
    }
    return eaten;
}
```

5.2 查找就绪回调函数

调用 tcp_queue_rcv 接收完成之后，接着再调用 sk_data_ready 来唤醒在 socket 上等待的用户进程。这又是一个函数指针。回想上面第一节我们在 accept 函数创建 socket 流程里提到的 sock_init_data 函数，在这个函数里已经把 sk_data_ready 设置成 sock_def_readable 函数了。它是默认的数据就绪处理函数。

当 socket 上数据就绪时候，内核将以 sock_def_readable 这个函数为入口，找到 epoll_ctl 添加 socket 时在其上设置的回调函数 ep_poll_callback。



我们来详细看下细节：

```
//file: net/core/sock.c
static void sock_def_readable(struct sock *sk, int len)
{
    struct socket_wq *wq;

    rcu_read_lock();
    wq = rcu_dereference(sk->sk_wq);

    //这个名字起的不好，并不是有阻塞的进程，
    //而是判断等待队列不为空
    if (wq_has_sleeper(wq))
        //执行等待队列项上的回调函数
        wake_up_interruptible_sync_poll(&wq->wait, POLLIN | POLLPRI |
            POLLRDNORM | POLLRDBAND);
    sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_IN);
    rcu_read_unlock();
}
```

这里的函数名其实都有迷惑人的地方。

- wq_has_sleeper，对于简单的 recvfrom 系统调用来说，确实是判断是否有进程阻塞。但是对



一定有唤醒进程的操作。

那接下来就是我们重点看 `wake_up_interruptible_sync_poll`。

我们看一下内核是怎么找到等待队列项里注册的回调函数的。

```
//file: include/linux/wait.h
#define wake_up_interruptible_sync_poll(x, m) \
    __wake_up_sync_key((x), TASK_INTERRUPTIBLE, 1, (void *) (m))

//file: kernel/sched/core.c
void __wake_up_sync_key(wait_queue_head_t *q, unsigned int mode,
    int nr_exclusive, void *key)
{
    ...
    __wake_up_common(q, mode, nr_exclusive, wake_flags, key);
}
```

接着进入 `__wake_up_common`

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
    int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

在 `__wake_up_common` 中，选出等待队列里注册某个元素 `curr`，回调其 `curr->func`。回忆我们 `ep_insert` 调用的时候，把这个 `func` 设置成 `ep_poll_callback` 了。

5.3 执行 socket 就绪回调函数

在上一小节找到了 socket 等待队列项里注册的函数 `ep_poll_callback`，软中断接着就会调用它。

```
//file: fs/eventpoll.c
static int ep_poll_callback(wait_queue_t *wait, unsigned mode, int sync, void *key)
{
    //获取 wait 对应的 epitem
    struct epitem *epi = ep_item_from_wait(wait);

    //获取 epitem 对应的 eventpoll 结构体
    struct eventpoll *ep = epi->ep;

    //1. 将当前epitem 添加到 eventpoll 的就绪队列中
    list_add_tail(&epi->rdllink, &ep->rdllist);

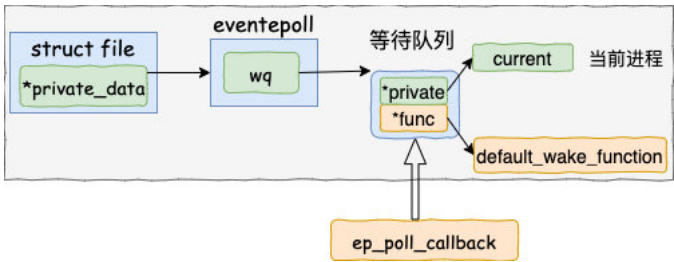
    //2. 查看 eventpoll 的等待队列上是否有在等待
    if (waitqueue_active(&ep->wq))
        wake_up_locked(&ep->wq);
}
```

在 `ep_poll_callback` 根据等待任务队列项上的额外的 `base` 指针可以找到 `epitem`，进而也可以找到 `eventpoll` 对象。

接着它又会查看 eventpoll 对象上的等待队列里是否有等待项（epoll_wait 执行的时候会设置）。



如果没执行软中断的事情就做完了。如果有等待项，那就查找到等待项里设置的回调函数。



调用 wake_up_locked() => __wake_up_locked() => __wake_up_common.

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                             int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

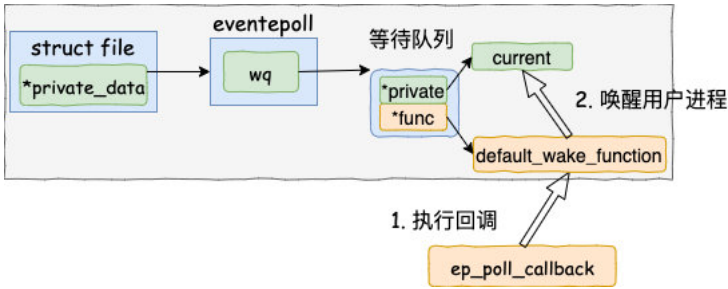
    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

在 __wake_up_common 里，调用 curr->func。这里的 func 是在 epoll_wait 是传入的 default_wake_function 函数。

5.4 执行 epoll 就绪通知

在 default_wake_function 中找到等待队列项里的进程描述符，然后唤醒之。



源代码如下：

```
//file:kernel/sched/core.c
int default_wake_function(wait_queue_t *curr, unsigned mode, int wake_flags,
                          void *key)
{
    return try_to_wake_up(curr->private, mode, wake_flags);
}
```

等待队列项 curr->private 指针是在 epoll 对象上等待而被阻塞掉的进程。

将 epoll_wait 进程推入可运行队列，等待内核重新调度进程。然后 epoll_wait 对应的这个进程重新运行后，就从 schedule 恢复

当进程醒来后，继续从 epoll_wait 时暂停的代码继续执行。把 rdlist 中就绪的事件返回给用户进



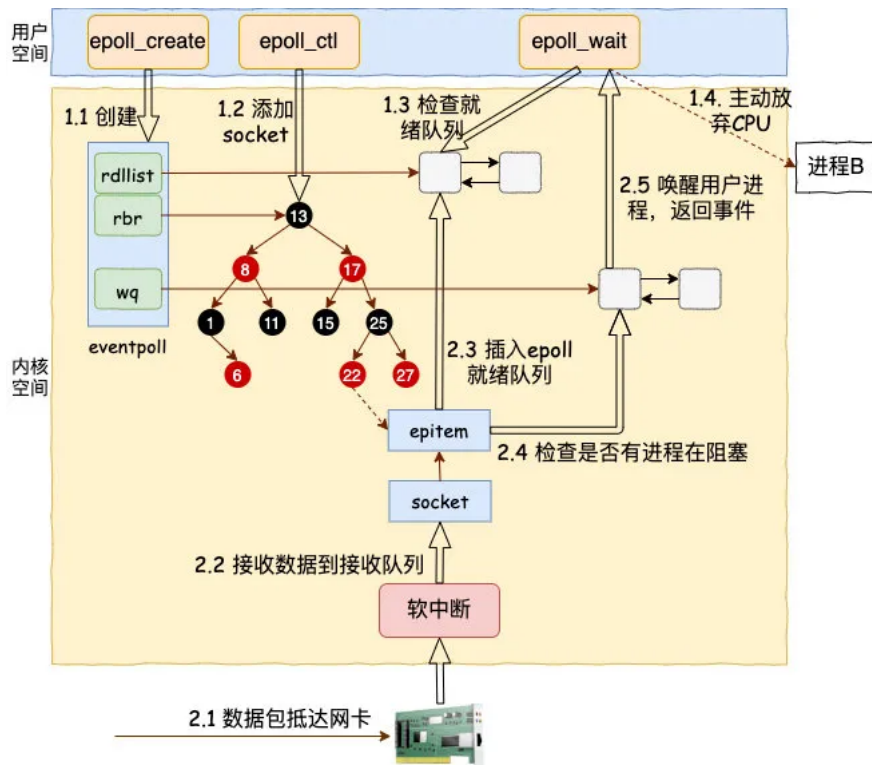
```
//file: fs/eventpoll.c
static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events,
                  int maxevents, long timeout)
{
    .....
    __remove_wait_queue(&ep->wq, &wait);

    set_current_state(TASK_RUNNING);
}
check_events:
    //返回就绪事件给用户进程
    ep_send_events(ep, events, maxevents))
}
```

从用户角度来看，epoll_wait 只是多等了一会儿而已，但执行流程还是顺序的。

总结

我们用一幅图总结一下 epoll 的整个工作路程。



其中软中断回调的时候回调函数也整理一下：

sock_def_readable: sock 对象初始化时设置的

=> ep_poll_callback: epoll_ctl 时添加到 socket 上的

=> default_wake_function: epoll_wait 是设置到 epoll 上的

总结下，epoll 相关的函数里内核运行环境分两部分：

- 用户进程内核态。进行调用 `epoll_wait` 等函数时会将进程陷入内核态来执行。这部分代码负责查看接收队列，以及负责把当前进程阻塞掉，让出 CPU。
- 硬软中断上下文。在这些组件中，将包从网卡接收过来进行处理，然后放到 socket 的接收队列。对于 epoll 来说，再找到 socket 关联的 `epitem`，并把它添加到 epoll 对象的就绪链表中。这个时候再捎带检查一下 epoll 上是否有被阻塞的进程，如果有唤醒之。

为了介绍到每个细节，本文涉及到的流程比较多，把阻塞都介绍进来了。

但**其实在实践中**，只要活儿足够的多，`epoll_wait` 根本都不会让进程阻塞。用户进程会一直干活，直到 `epoll_wait` 用完了没活干可干的时候才主动让出 CPU。这就是 epoll 高效的

包括本文在内，飞哥总共用三篇文章分析了一件事情，一个网络包是如何从网卡达到你的用户进程里的。另外两篇如下：



- [图解 | Linux 网络包接收过程](#)
- [图解 | 深入理解高性能网络开发路上的绊脚石 - 同步阻塞网络 IO](#)

恭喜你没被内核源码劝退，一直能坚持到了现在。赶快给先自己鼓个掌，晚饭去加个鸡腿！

当然网络编程剩下还有一些概念我们没有讲到，比如 Reactor 和 Proactor 等。不过相对内核来讲，这些用户层的技术相对就很简单了。这些只是在讨论当多进程一起配合工作时谁负责查看 IO 事件、谁该负责计算、谁负责发送和接收，仅仅是用户进程的不同分工模式罢了。

参考

- [epoll 原理-csdn](#)
- [关于 AVL 树和红黑树的一点看法](#)
- [源码解读 epoll 内核机制](#)
- [一个基于 epoll 的 demo 程序](#)
- [tcp/ip 协议栈——epoll 的内部实现原理](#)
- [从 linux 源码看 epoll](#)
- [套接字之 recvfrom 系统调用实现](#)
- [如果这篇文章说不清 epoll 的本质，那就过来掐死我吧](#)
- [Linux 内核文件描述符表的演变](#)
- [三次握手过程的源码分析](#)

发布于 2022-03-28 17:30

C / C++ IO多路复用 epoll



评论千万条，友善第一条

18 条评论

默认 最新



BBAB

前段时间找实习的时候看过epoll的实现，可惜面试官没一个问的...

2022-03-29

回复 6



BBAB 猫与少年

我疯狂勾引面试官往这个方向问我，可惜他们浅尝辄止

2022-03-30

回复 3



阿灰想写code BBAB

面试官不懂罢了，一般如果候选人如果能够说出等待队列这个东西，我就认为很不错了

2022-04-16

回复 赞

展开其他 1 条回复



惊呆了

又臭又长

2022-03-30

回复 5



韩立

如果你本身就是搞底层的，可能对你来说这篇文章挺鸡肋；但是对于我们想探究底层的java、Go工程师来说，能找到这么一篇好文实属不易，网上大多数文章都是抄来抄去，我们想找一篇说的清楚明白的文章真的很累。所以，你可以不爱，但请别伤害。

2022-06-25

回复 6



vision

插个眼，后续再啃


2022-10-09

回复 赞


赞同 548 18 条评论 分享 喜欢 收藏 申请转载

图文并茂，非常全面
2022-08-02

● 回复 ● 赞

 **张子诚**
写太好了，这特么要给你点个外卖送鸡腿👍
2022-07-04


● 回复 ● 赞

 **韩立**
赞一个，从头到尾讲的真清楚！
2022-06-25


● 回复 ● 赞

 **拍爱娶拍程序员**
不用多路复用，一个进程怎么接收10000个连接？
2022-04-20


● 回复 ● 赞

 **拍爱娶拍程序员**
不用多路复用的情况下，一个进程怎么连接10000个socket
2022-04-20


● 回复 ● 赞

 **仔子弟**
开10000个线程😏 但很不实际
2022-10-25


● 回复 ● 赞

 **喽哈啊**
总结的很好，我也省得去看内核源码了，直接看你的
2022-03-30

● 回复 ● 赞

 **青年真理报** 🏠
非常棒👍
2022-03-30


● 回复 ● 赞

 **客串一回**
主要讲了使用方法🐱
2022-03-29

● 回复 ● 赞

 **海胖子**
后开开发...
2022-03-28

● 回复 ● 赞

 **客串一回**
后端？
2022-03-29

● 回复 ● 赞



评论千万条，友善第一条



腾讯技术
跟腾讯技术相关的文章都在这里了

推荐阅读



图解 | 深入揭秘 epoll 是如何实现 IO 多路复用的！

5种IO模型 详解 包含select
epoll原理

哪五种：阻塞IO，非阻塞IO，IO多路复用，信号驱动IO，异步IO下面我们来——拔毛，搞得你懂为止 一. 阻塞IO应用程序(用户空间)向内核(内核空间)发起读取数据申请时 (recvfrom) 在内核数



彻底搞懂 IO 底层原理

彻底搞懂 IO 底层原理

武侠小说里有很多“招式”。计算机“法”和“招式”呢为“道”和“术”的计算机理论，隐非常抽象、晦涩难

vivo互...

▲ 赞同 548 ▼ ● 18 条评论 ↗ 分享 ❤ 喜欢 ★ 收藏 📄 申请转载 ...

IP开源中国



]