作者：董昊 （要转载的同学帮忙把名字和博客链接http://donghao.org/uii/带上，多谢了！）

poll和epoll的使用应该不用再多说了。当fd很多时，使用epoll比poll效率更高。我们通过内核源码分析来看看到底是为什么。

poll剖析

poll系统调用：
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

内核2.6.9对应的实现代码为：
[fs/select.c -->sys_poll]

```
asmlinkage long sys_poll(struct pollfd __user * ufds, unsigned int nfds, long timeout)
{
    struct poll_wqueues table;
    int fdcount, err;
    unsigned int i;
    struct poll_list *head;
    struct poll_list *walk;

    /* Do a sanity check on nfds ... */ /* 用户给的nfds数不可以超过一个struct file结构支持的最大fd数（默认是256）*/
    if (nfds > current->files->max_fdset && nfds > OPEN_MAX)
        return -EINVAL;

    if (timeout) {
        /* Careful about overflow in the intermediate values */
        if ((unsigned long) timeout < MAX_SCHEDULE_TIMEOUT / HZ)
            timeout = (unsigned long)(timeout*HZ+999)/1000+1;
        else /* Negative or overflow */
            timeout = MAX_SCHEDULE_TIMEOUT;
    }

    poll_initwait(&table);
```

其中poll_initwait较为关键，从字面上看，应该是初始化变量table，注意此处table在整个执行poll的过程中是很关键的变量。
而struct poll_table其实就只包含了一个函数指针：
[fs/poll.h]

```
/*
 * structures and helpers for f_op->poll implementations
 */
typedef void (*poll_queue_proc)(struct file *, wait_queue_head_t *, struct poll_table_struct *);

typedef struct poll_table_struct {
    poll_queue_proc qproc;
} poll_table;
```

现在我们来看看poll_initwait到底在做些什么
[fs/select.c]

```
void __pollwait(struct file *filp, wait_queue_head_t *wait_address, poll_table *p);
```

```
void poll_initwait(struct poll_wqueues *pwq)        注册函数
{
    &(pwq->pt)->qproc = __pollwait; /*此行已经被我"翻译"了，方便观看*/
    pwq->error = 0;
    pwq->table = NULL;
}
```

很明显，poll_initwait的主要动作就是把table变量的成员poll_table对应的回调函数置为__pollwait。这个__pollwait不仅是poll系统调用需要，select系统调用也一样是用这个__pollwait，说白了，这是个操作系统的异步操作的"御用"回调函数。当然了，epoll没有用这个，它另外新增了一个回调函数，以达到其高效运转的目的，这是后话，暂且不表。

我们先不讨论__pollwait的具体实现，还是继续看sys_poll:

[fs/select.c -->sys_poll]

```
    head = NULL;
    walk = NULL;                                    描述符个数
    i = nfds;
    err = -ENOMEM;                                  建立链表
    while(i!=0) {
        struct poll_list *pp;
        pp = kmalloc(sizeof(struct poll_list)+      一个poll_list和i个
                sizeof(struct pollfd)*              pollfd结构
                (i>POLLFD_PER_PAGE?POLLFD_PER_PAGE:i),
                    GFP_KERNEL);
        if(pp==NULL)
            goto out_fds;
        pp->next=NULL;
        pp->len = (i>POLLFD_PER_PAGE?POLLFD_PER_PAGE:i);
        if (head == NULL)
            head = pp;
        else                                        将描述符考入内核
            walk->next = pp;

        walk = pp;
        if (copy_from_user(pp->entries, ufds + nfds-i,
                sizeof(struct pollfd)*pp->len)) {
            err = -EFAULT;
            goto out_fds;
        }
        i -= pp->len;
    }
    fdcount = do_poll(nfds, head, &table, timeout);
```

这一大堆代码就是建立一个链表，每个链表的节点是一个page大小（通常是4k），这链表节点由一个指向struct poll_list的指针掌控，而众多的struct pollfd就通过struct_list的entries成员访问。上面的循环就是把用户态的struct pollfd拷进这些entries里。通常用户程序的poll调用就监控几个fd，所以上面这个链表通常也就只需要一个节点，即操作系统的一页。但是，当用户传入的fd很多时，由于poll系统调用每次都要把所有struct pollfd拷进内核，所以参数传递和页分配此时就成了poll系统调用的性能瓶颈。

最后一句do_poll，我们跟进去：

[fs/select.c-->sys_poll()-->do_poll()]

```
static void do_pollfd(unsigned int num, struct pollfd * fdpage,
    poll_table ** pwait, int *count)
{
    int i;
```

```
    for (i = 0; i < num; i++) {
        int fd;
        unsigned int mask;
        struct pollfd *fdp;

        mask = 0;
        fdp = fdpage+i;
        fd = fdp->fd;
        if (fd >= 0) {
            struct file * file = fget(fd);        描述符不是一个打
            mask = POLLNVAL;                      开的文件
            if (file != NULL) {
                mask = DEFAULT_POLLMASK;
                if (file->f_op && file->f_op->poll)
                    mask = file->f_op->poll(file, *pwait);
                mask &= fdp->events | POLLERR | POLLHUP;
                fput(file);
            }
            if (mask) {
                *pwait = NULL;
                (*count)++;
            }                                     返回值
        }
        fdp->revents = mask;
    }
}

static int do_poll(unsigned int nfds,  struct poll_list *list,
            struct poll_wqueues *wait, long timeout)
{
    int count = 0;
    poll_table* pt = &wait->pt;

    if (!timeout)
        pt = NULL;

    for (;;) {
        struct poll_list *walk;
        set_current_state(TASK_INTERRUPTIBLE);
        walk = list;
        while(walk != NULL) {
            do_pollfd( walk->len, walk->entries, &pt, &count);
            walk = walk->next;
        }
        pt = NULL;
        if (count || !timeout || signal_pending(current))
            break;
        count = wait->error;                      count大于0跳出
        if (count)
            break;
        timeout = schedule_timeout(timeout); /* 让current挂起，别的进程跑，timeout到了
以后再回来运行current*/
```

```
451     }
452     __set_current_state(TASK_RUNNING);
453     return count;
454 }
```

注意438行的set_current_state和445行的signal_pending，它们两句保障了当用户程序在调用poll后挂起时，发信号可以让程序迅速推出poll调用，而通常的系统调用是不会被信号打断的。
纵览do_poll函数，主要是在循环内等待，直到count大于0才跳出循环，而count主要是靠do_pollfd函数处理。
注意标红的440-443行，当用户传入的fd很多时（比如1000个），对do_pollfd就会调用很多次，poll效率瓶颈的另一原因就在这里。
do_pollfd就是针对每个传进来的fd，调用它们各自对应的poll函数，简化一下调用过程，如下：

```
struct file* file = fget(fd);
file->f_op->poll ( file, &(table->pt));
```

如果fd对应的是某个socket，do_pollfd调用的就是网络设备驱动实现的poll；如果fd对应的是某个ext3文件系统上的一个打开文件，那do_pollfd调用的就是ext3文件系统驱动实现的poll。一句话，这个file->f_op->poll是设备驱动程序实现的，那设备驱动程序的poll实现通常又是什么样子呢？其实，设备驱动程序的标准实现是：调用poll_wait，即以设备自己的等待队列为参数（通常设备都有自己的等待队列，不然一个不支持异步操作的设备会让人很郁闷）调用struct poll_table的回调函数。
作为驱动程序的代表，我们看看socket在使用tcp时的代码：
[net/ipv4/tcp.c-->tcp_poll]

```
unsigned int tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
{
    unsigned int mask;
    struct sock *sk = sock->sk;
    struct tcp_opt *tp = tcp_sk(sk);

    poll_wait(file, sk->sk_sleep, wait);
```

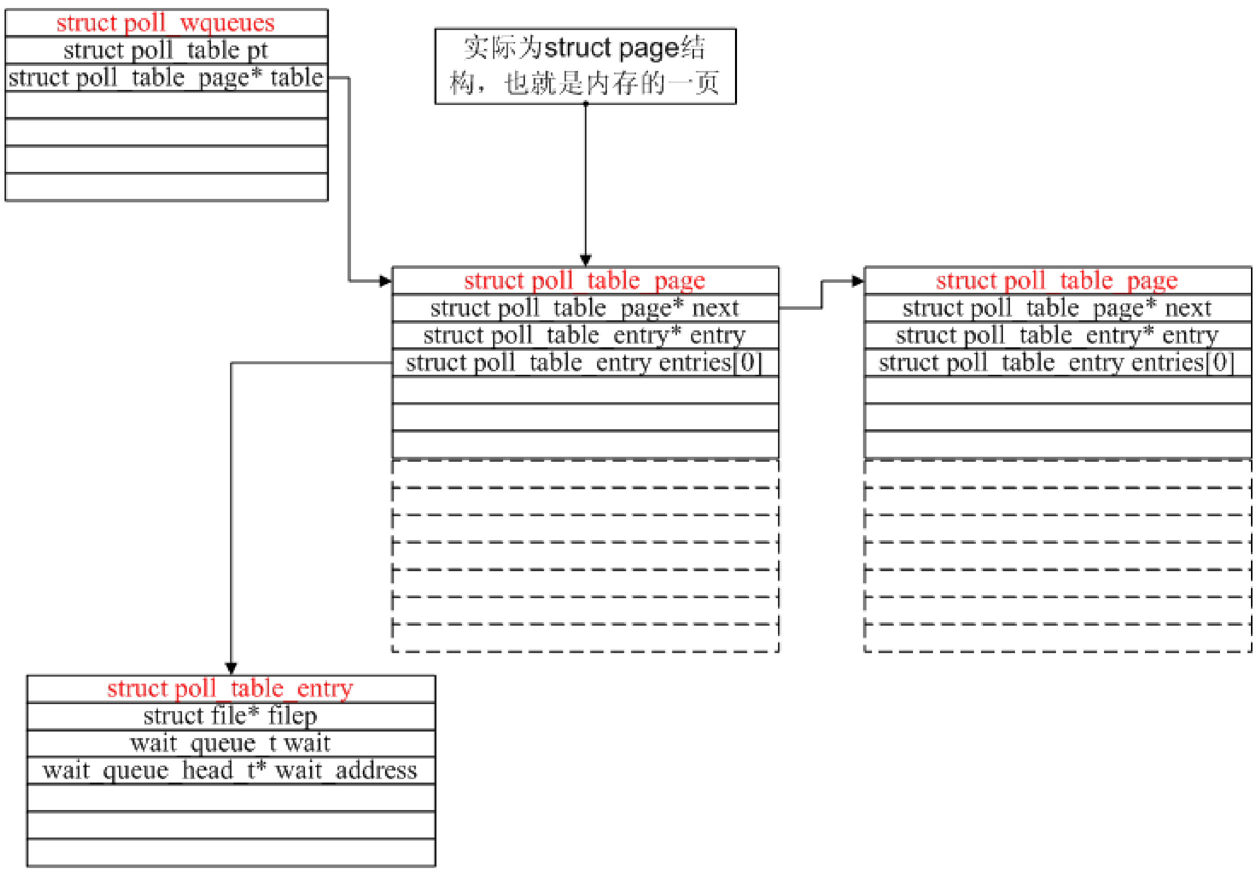
代码就看这些，剩下的无非就是判断状态、返回状态值，tcp_poll的核心实现就是poll_wait，而poll_wait就是调用struct poll_table对应的回调函数，那poll系统调用对应的回调函数就是__poll_wait，所以这里几乎就可以把tcp_poll理解为一个语句：
__poll_wait(file, sk->sk_sleep, wait);
由此也可以看出，每个socket自己都带有一个等待队列sk_sleep，所以上面我们所说的"设备的等待队列"其实不止一个。
这时候我们再看看__poll_wait的实现：

回调函数

[fs/select.c-->__poll_wait()]

```
void __pollwait(struct file *filp, wait_queue_head_t *wait_address, poll_table *_p)
{
    struct poll_wqueues *p = container_of(_p, struct poll_wqueues, pt);
    struct poll_table_page *table = p->table;

    if (!table || POLL_TABLE_FULL(table)) {
        struct poll_table_page *new_table;

        new_table = (struct poll_table_page *) __get_free_page(GFP_KERNEL);
        if (!new_table) {
            p->error = -ENOMEM;
             __set_current_state(TASK_RUNNING);
             return;
         }
         new_table->entry = new_table->entries;
         new_table->next = table;
         p->table = new_table;
```

```
        table = new_table;
    }

    /* Add a new entry */        描述符节点
    {
        struct poll_table_entry * entry = table->entry;
        table->entry = entry+1;
        get_file(filp);
        entry->filp = filp;
        entry->wait_address = wait_address;
        init_waitqueue_entry(&entry->wait, current);
        add_wait_queue(wait_address,&entry->wait);
    }
}
```



__poll_wait的作用就是创建了上图所示的数据结构（一次__poll_wait即一次设备poll调用只创建一个poll_table_entry），并通过struct poll_table_entry的wait成员，把current挂在了设备的等待队列上，此处的等待队列是wait_address，对应tcp_poll里的sk->sk_sleep。
现在我们可以回顾一下poll系统调用的原理了：先注册回调函数__poll_wait，再初始化table变量（类型为struct poll_wqueues），接着拷贝用户传入的struct pollfd（其实主要是fd），然后轮流调用所有fd对应的poll（把current挂到各个fd对应的设备等待队列上）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上的进程，这时current便被唤醒了。current醒来后离开sys_poll的操作相对简单，这里就不逐行分析了。

作者：董昊 （要转载的同学帮忙把名字和博客链接http://donghao.org/uii/带上，多谢了！）

epoll原理简介

通过上面的分析，poll运行效率的两个瓶颈已经找出，现在的问题是怎么改进。首先，每次poll都要把1000个fd 拷入内核，太不科学了，内核干嘛不自己保存已经拷入的fd呢？答对了，epoll就是自己保存拷入的fd，它的API就已经说明了这一点——不是 epoll_wait的时候才传入fd，而是通过epoll_ctl把所有fd传入内核再一起"wait"，这就省掉了不必要的重复拷贝。其次，在 epoll_wait时，也不是把current轮流的加入fd对应的设备等待队列，而是在设备等待队列醒来时调用一个回调函数（当然，这就需要"唤醒回调"机制），把产生事件的fd归入一个链表，然后返回这个链表上的fd。

epoll剖析

epoll是个module，所以先看看module的入口eventpoll_init
[fs/eventpoll.c-->evetpoll_init()]

```
static int __init eventpoll_init(void)
{
    int error;

    init_MUTEX(&epsem);

    /* Initialize the structure used to perform safe poll wait head wake ups */
    ep_poll_safewake_init(&psw);

    /* Allocates slab cache used to allocate "struct epitem" items */
    epi_cache = kmem_cache_create("eventpoll_epi", sizeof(struct epitem),
            0, SLAB_HWCACHE_ALIGN|EPI_SLAB_DEBUG|SLAB_PANIC,
            NULL, NULL);

    /* Allocates slab cache used to allocate "struct eppoll_entry" */
    pwq_cache = kmem_cache_create("eventpoll_pwq",
            sizeof(struct eppoll_entry), 0,
            EPI_SLAB_DEBUG|SLAB_PANIC, NULL, NULL);

    /*
     * Register the virtual file system that will be the source of inodes
     * for the eventpoll files
     */
    error = register_filesystem(&eventpoll_fs_type);
    if (error)
        goto epanic;

    /* Mount the above commented virtual file system */
    eventpoll_mnt = kern_mount(&eventpoll_fs_type);
    error = PTR_ERR(eventpoll_mnt);
    if (IS_ERR(eventpoll_mnt))
        goto epanic;

    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: successfully initialized.\n",
            current));
    return 0;

```

```
epanic:
    panic("eventpoll_init() failed\n");
}
```

很有趣，这个module在初始化时注册了一个新的文件系统，叫"eventpollfs"（在eventpoll_fs_type结构里），然后挂载此文件系统。另外创建两个内核cache（在内核编程中，如果需要频繁分配小块内存，应该创建kmem_cahe来做"内存池"）,分别用于存放struct epitem和eppoll_entry。如果以后要开发新的文件系统，可以参考这段代码。

现在想想epoll_create为什么会返回一个新的fd？因为它就是在这个叫做"eventpollfs"的文件系统里创建了一个新文件！如下：

[fs/eventpoll.c-->sys_epoll_create()]

```
asmlinkage long sys_epoll_create(int size)
{
    int error, fd;
    struct inode *inode;
    struct file *file;

    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: sys_epoll_create(%d)\n",
            current, size));

    /* Sanity check on the size parameter */
    error = -EINVAL;
    if (size <= 0)
        goto eexit_1;

    /*
     * Creates all the items needed to setup an eventpoll file. That is,
     * a file structure, and inode and a free file descriptor.
     */
    error = ep_getfd(&fd, &inode, &file);
    if (error)
        goto eexit_1;

    /* Setup the file internal data structure ( "struct eventpoll" ) */
    error = ep_file_init(file);
    if (error)
        goto eexit_2;
```

函数很简单，其中ep_getfd看上去是"get"，其实在第一次调用epoll_create时，它是要创建新inode、新的file、新的fd。而ep_file_init则要创建一个struct eventpoll结构，并把它放入file->private_data，注意，这个private_data后面还要用到的。

看到这里，也许有人要问了，为什么epoll的开发者不做一个内核的超级大map把用户要创建的epoll句柄存起来，在epoll_create时返回一个指针？那似乎很直观呀。但是，仔细看看，linux的系统调用有多少是返回指针的？你会发现几乎没有！（特此强调，malloc不是系统调用，malloc调用的brk才是）因为linux做为unix的最杰出的继承人，它遵循了unix的一个巨大优点——一切皆文件，输入输出是文件、socket也是文件，一切皆文件意味着使用这个操作系统的程序可以非常简单，因为一切都是文件操作而已！（unix还不是完全做到，plan 9才算）。而且使用文件系统有个好处：epoll_create返回的是一个fd，而不是该死的指针，指针如果指错了，你简直没办法判断，而fd则可以通过current->files->fd_array[]找到其真伪。

epoll_create好了，该epoll_ctl了，我们略去判断性的代码：

[fs/eventpoll.c-->sys_epoll_ctl()]

```
asmlinkage long
sys_epoll_ctl(int epfd, int op, int fd, struct epoll_event __user *event)
{
```

error = DEFAULT;
if(ep_op_has_event(op) &&copy_from_user(&epds,event,sizeof(struct epoll_event)))   goto error_return;   //拷贝用户参数，如果需要
error = EBADF;
file = fget(epdf); if(!file) goto error_return;   //获取epdf对应的file实例
tfile = fget(fd); error = -EPERM;   //获取要操作的文件描述符对应的file实例
if(!tfile->f_op || !tfile->f_op->poll) return error_tgt_fput;
error = -EINVAL;
if(file == tfile ||!is_file_epoll(file)) goto error_tgt_fput；
ep = file->private_data; // 获取eventpoll文件中的私有数据，该数据是在epoll_create中创建的。
mutex_lock(&ep_mtx);

```
    int error;
    struct file *file, *tfile;
    struct eventpoll *ep;
    struct epitem *epi;
    struct epoll_event epds;
....
    epi = ep_find(ep, tfile, fd);

    error = -EINVAL;
    switch (op) {
    case EPOLL_CTL_ADD:
        if (!epi) {
            epds.events |= POLLERR | POLLHUP;

            error = ep_insert(ep, &epds, tfile, fd);
        } else
            error = -EEXIST;
        break;
    case EPOLL_CTL_DEL:
        if (epi)
            error = ep_remove(ep, epi);
        else
            error = -ENOENT;
        break;
    case EPOLL_CTL_MOD:
        if (epi) {
            epds.events |= POLLERR | POLLHUP;
            error = ep_modify(ep, epi, &epds);
        } else
            error = -ENOENT;
        break;
    }
```

在eventpoll中存储文件描述符信息的红黑树中查找指定的fd对应的epitem实例

原来就是在一个大的结构（现在先不管是什么大结构）里先ep_find，如果找到了struct epitem而用户操作是ADD，那么返回-EEXIST；如果是DEL，则ep_remove。如果找不到struct epitem而用户操作是ADD，就ep_insert创建并插入一个。很直白。那这个"大结构"是什么呢？看ep_find的调用方式，ep参数应该是指向这个"大结构"的指针，再看ep = file->private_data，我们才明白，原来这个"大结构"就是那个在epoll_create时创建的struct eventpoll，具体再看看ep_find的实现，发现原来是struct eventpoll的rbr成员（struct rb_root），原来这是一个红黑树的根！而红黑树上挂的都是struct epitem。

现在清楚了，一个新创建的epoll文件带有一个struct eventpoll结构，这个结构上再挂一个红黑树，而这个红黑树就是每次epoll_ctl时fd存放的地方！

现在数据结构都已经清楚了，我们来看最核心的：

[fs/eventpoll.c-->sys_epoll_wait()]

```
asmlinkage long sys_epoll_wait(int epfd, struct epoll_event __user *events,
                               int maxevents, int timeout)
{
    int error;
    struct file *file;
    struct eventpoll *ep;

    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: sys_epoll_wait(%d, %p, %d, %d)\n",
            current, epfd, events, maxevents, timeout));

    /* The maximum number of event must be greater than zero */
```

```
    if (maxevents <= 0)
        return -EINVAL;

    /* Verify that the area passed by the user is writeable */
    if ((error = verify_area(VERIFY_WRITE, events, maxevents * sizeof(struct epoll_event))))
        goto eexit_1;

    /* Get the "struct file *" for the eventpoll file */
    error = -EBADF;
    file = fget(epfd);
    if (!file)
        goto eexit_1;

    /*
     * We have to check that the file structure underneath the fd
     * the user passed to us _is_ an eventpoll file.
     */
    error = -EINVAL;
    if (!IS_FILE_EPOLL(file))
        goto eexit_2;

    /*
     * At this point it is safe to assume that the "private_data" contains
     * our own data structure.
     */
    ep = file->private_data;

    /* Time to fish for events ... */
    error = ep_poll(ep, events, maxevents, timeout);

eexit_2:
    fput(file);
eexit_1:
    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: sys_epoll_wait(%d, %p, %d, %d) = %d\n",
             current, epfd, events, maxevents, timeout, error));

    return error;
}
```

故伎重演，从file->private_data中拿到struct eventpoll，再调用ep_poll

[fs/eventpoll.c-->sys_epoll_wait()->ep_poll()]

```
static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events,
           int maxevents, long timeout)
{
    int res, eavail;
    unsigned long flags;
    long jtimeout;
    wait_queue_t wait;

    /*
     * Calculate the timeout by checking for the "infinite" value ( -1 )
     * and the overflow condition. The passed timeout is in milliseconds,
```

```
     * that why (t * HZ) / 1000.
     */
    jtimeout = timeout == -1 || timeout > (MAX_SCHEDULE_TIMEOUT - 1000) / HZ ?
        MAX_SCHEDULE_TIMEOUT: (timeout * HZ + 999) / 1000;

retry:
    write_lock_irqsave(&ep->lock, flags);

    res = 0;
    if (list_empty(&ep->rdllist)) {
        /*
         * We don't have any available event to return to the caller.
         * We need to sleep here, and we will be wake up by
         * ep_poll_callback() when events will become available.
         */
        init_waitqueue_entry(&wait, current);
        add_wait_queue(&ep->wq, &wait);

        for (;;) {
            /*
             * We don't want to sleep if the ep_poll_callback() sends us
             * a wakeup in between. That's why we set the task state
             * to TASK_INTERRUPTIBLE before doing the checks.
             */
            set_current_state(TASK_INTERRUPTIBLE);
            if (!list_empty(&ep->rdllist) || !jtimeout)
                break;
            if (signal_pending(current)) {
                res = -EINTR;
                break;
            }

            write_unlock_irqrestore(&ep->lock, flags);
            jtimeout = schedule_timeout(jtimeout);
            write_lock_irqsave(&ep->lock, flags);
        }
        remove_wait_queue(&ep->wq, &wait);

        set_current_state(TASK_RUNNING);
    }
....
```

又是一个大循环，不过这个大循环比poll的那个好，因为仔细一看——它居然除了睡觉和判断ep->rdllist是否为空以外，啥也没做！

什么也没做当然效率高了，但到底是谁来让ep->rdllist不为空呢？

答案是ep_insert时设下的回调函数：

[fs/eventpoll.c-->sys_epoll_ctl()-->ep_insert()]

```
static int ep_insert(struct eventpoll *ep, struct epoll_event *event,
             struct file *tfile, int fd)
{
    int error, revents, pwake = 0;
    unsigned long flags;
    struct epitem *epi;
    struct ep_pqueue epq;
```

```
930
931     error = -ENOMEM;
932     if (!(epi = EPI_MEM_ALLOC()))
933         goto eexit_1;
934
```

申请一个epi空间
下面初始化

```
935     /* Item initialization follow here ... */
936     EP_RB_INITNODE(&epi->rbn);
937     INIT_LIST_HEAD(&epi->rdllink);
938     INIT_LIST_HEAD(&epi->fllink);
939     INIT_LIST_HEAD(&epi->txlink);
940     INIT_LIST_HEAD(&epi->pwqlist);
941     epi->ep = ep;
942     EP_SET_FFD(&epi->ffd, tfile, fd);
943     epi->event = *event;
944     atomic_set(&epi->usecnt, 1);
945     epi->nwait = 0;
946
947     /* Initialize the poll table using the queue callback */
948     epq.epi = epi;
949     init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
950
951     /*
952      * Attach the item to the poll hooks and get current event bits.
953      * We can safely use the file* here because its usage count has
954      * been increased by the caller of this function.
955      */
956     revents = tfile->f_op->poll(tfile, &epq.pt);
```

我们注意949行，其实就是

&(epq.pt)->qproc = ep_ptable_queue_proc;

紧接着 tfile->f_op->poll(tfile, &epq.pt)其实就是调用被监控文件（epoll里叫"target file")的poll方法，而这个poll其实就是调用poll_wait（还记得poll_wait吗？每个支持poll的设备驱动程序都要调用的），最后就是调用ep_ptable_queue_proc。这是比较难解的一个调用关系，因为不是语言级的直接调用。

ep_insert还把struct epitem放到struct file里的f_ep_links连表里，以方便查找，struct epitem里的fllink就是担负这个使命的。

[fs/eventpoll.c-->ep_ptable_queue_proc()]

```
static void ep_ptable_queue_proc(struct file *file, wait_queue_head_t *whead,
                poll_table *pt)
{
    struct epitem *epi = EP_ITEM_FROM_EPQUEUE(pt);
    struct eppoll_entry *pwq;

    if (epi->nwait >= 0 && (pwq = PWQ_MEM_ALLOC())) {
        init_waitqueue_func_entry(&pwq->wait, ep_poll_callback);
        pwq->whead = whead;
        pwq->base = epi;
        add_wait_queue(whead, &pwq->wait);
        list_add_tail(&pwq->llink, &epi->pwqlist);
        epi->nwait++;
    } else {
        /* We have to signal that an error occurred */
        epi->nwait = -1;
    }
```

```
}
```

上面的代码就是ep_insert中要做的最重要的事：创建struct eppoll_entry，设置其唤醒回调函数为ep_poll_callback，然后加入设备等待队列（注意这里的whead就是上一章所说的每个设备驱动都要带的等待队列）。只有这样，当设备就绪，唤醒等待队列上的等待着时，ep_poll_callback就会被调用。每次调用poll系统调用，操作系统都要把current（当前进程）挂到fd对应的所有设备的等待队列上，可以想象，fd多到上千的时候，这样“挂”法很费事；而每次调用epoll_wait则没有这么罗嗦，epoll只在epoll_ctl时把current挂一遍（这第一遍是免不了的）并给每个fd一个命令“好了就调回调函数”，如果设备有事件了，通过回调函数，会把fd放入rdllist，而每次调用epoll_wait就只是收集rdllist里的fd就可以了——epoll巧妙的利用回调函数，实现了更高效的事件驱动模型。

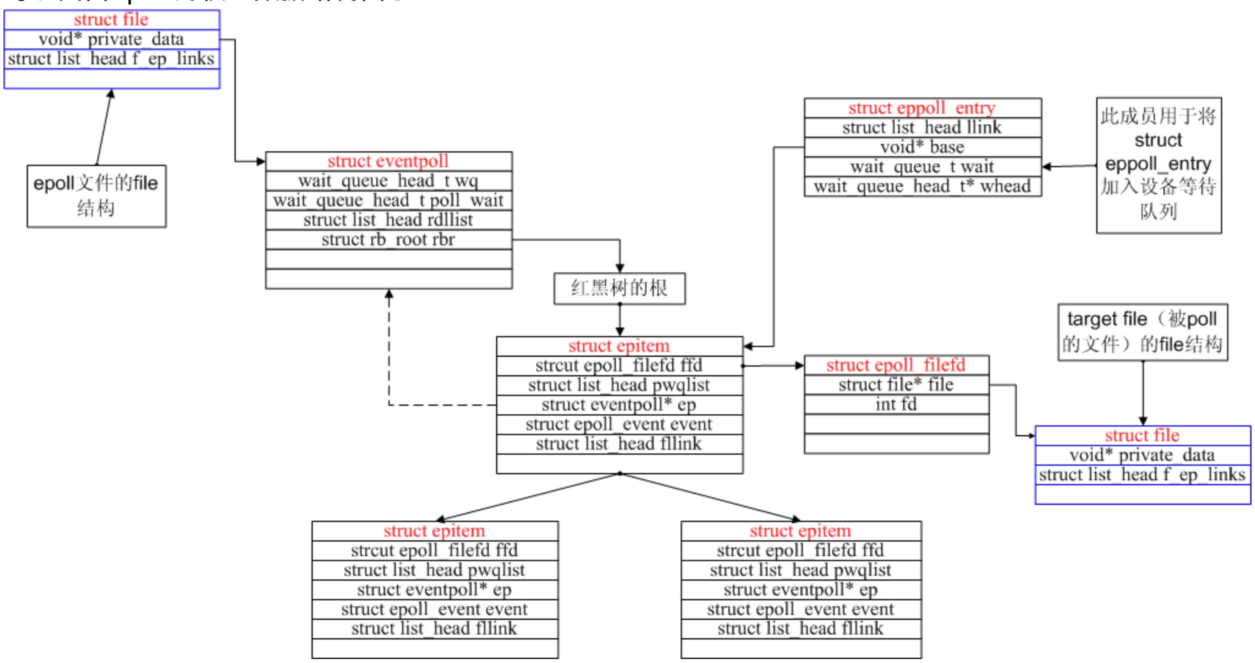
现在我们猜也能猜出来ep_poll_callback会干什么了——肯定是把红黑树上的收到event的epitem（代表每个fd）插入ep->rdllist中，这样，当epoll_wait返回时，rdllist里就都是就绪的fd了！

[fs/eventpoll.c-->ep_poll_callback()]

```
static int ep_poll_callback(wait_queue_t *wait, unsigned mode, int sync, void *key)
{
    int pwake = 0;
    unsigned long flags;
    struct epitem *epi = EP_ITEM_FROM_WAIT(wait);
    struct eventpoll *ep = epi->ep;

    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: poll_callback(%p) epi=%p ep=%p\n",
            current, epi->file, epi, ep));

    write_lock_irqsave(&ep->lock, flags);

    /*
     * If the event mask does not contain any poll(2) event, we consider the
     * descriptor to be disabled. This condition is likely the effect of the
     * EPOLLONESHOT bit that disables the descriptor when an event is received,
     * until the next EPOLL_CTL_MOD will be issued.
     */
    if (!(epi->event.events & ~EP_PRIVATE_BITS))
        goto is_disabled;

    /* If this file is already in the ready list we exit soon */
    if (EP_IS_LINKED(&epi->rdllink))
        goto is_linked;

    list_add_tail(&epi->rdllink, &ep->rdllist);

is_linked:
    /*
     * Wake up ( if active ) both the eventpoll wait list and the ->poll()
     * wait list.
     */
    if (waitqueue_active(&ep->wq))
        wake_up(&ep->wq);
    if (waitqueue_active(&ep->poll_wait))
        pwake++;

is_disabled:
    write_unlock_irqrestore(&ep->lock, flags);

```

```
1246     /* We have to call this outside the lock */
1247     if (pwake)
1248         ep_poll_safewake(&psw, &ep->poll_wait);
1249
1250     return 1;
1251 }
```

真正重要的只有1231行的只一句，就是把struct epitem放到struct eventpoll的rdllist中去。现在我们可以画出epoll的核心数据结构图了：



作者：董昊 （要转载的同学帮忙把名字和博客链接http://donghao.org/uii/带上，多谢了！）

epoll独有的EPOLLET

EPOLLET是epoll系统调用独有的flag，ET就是Edge Trigger（边缘触发）的意思，具体含义和应用大家可google之。有了EPOLLET，重复的事件就不会总是出来打扰程序的判断，故而常被使用。那EPOLLET的原理是什么呢？

上篇我们讲到epoll把fd都挂上一个回调函数，当fd对应的设备有消息时，就把fd放入rdllist链表，这样epoll_wait只要检查这个rdllist链表就可以知道哪些fd有事件了。我们看看ep_poll的最后几行代码：

[fs/eventpoll.c->ep_poll()]

```

    /*
     * Try to transfer events to user space. In case we get 0 events and
     * there's still timeout left over, we go trying again in search of
     * more luck.
     */
    if (!res && eavail &&
        !(res = ep_events_transfer(ep, events, maxevents)) && jtimeout)
        goto retry;

    return res;
}
```

把rdllist里的fd拷到用户空间，这个任务是ep_events_transfer做的：

[fs/eventpoll.c->ep_events_transfer()]

```
static int ep_events_transfer(struct eventpoll *ep,
                  struct epoll_event __user *events, int maxevents)
{
    int eventcnt = 0;
    struct list_head txlist;

    INIT_LIST_HEAD(&txlist);

    /*
     * We need to lock this because we could be hit by
     * eventpoll_release_file() and epoll_ctl(EPOLL_CTL_DEL).
     */
    down_read(&ep->sem);

    /* Collect/extract ready items */
    if (ep_collect_ready_items(ep, &txlist, maxevents) > 0) {
        /* Build result set in userspace */
        eventcnt = ep_send_events(ep, &txlist, events);

        /* Reinject ready items into the ready list */
        ep_reinject_items(ep, &txlist);
    }

    up_read(&ep->sem);

    return eventcnt;
}
```

代码很少，其中ep_collect_ready_items把rdllist里的fd挪到txlist里（挪完后rdllist就空了），接着ep_send_events把txlist里的fd拷给用户空间，然后ep_reinject_items把一部分fd从txlist里“返还”给rdllist以便下次还能从rdllist里发现它。
其中ep_send_events的实现：

[fs/eventpoll.c->ep_send_events()]

```
static int ep_send_events(struct eventpoll *ep, struct list_head *txlist,
              struct epoll_event __user *events)
{
    int eventcnt = 0;
    unsigned int revents;
    struct list_head *lnk;
    struct epitem *epi;

    /*
     * We can loop without lock because this is a task private list.
     * The test done during the collection loop will guarantee us that
     * another task will not try to collect this file. Also, items
     * cannot vanish during the loop because we are holding "sem".
     */
    list_for_each(lnk, txlist) {
        epi = list_entry(lnk, struct epitem, txlink);

        /*
         * Get the ready file event set. We can safely use the file
         * because we are holding the "sem" in read and this will
         * guarantee that both the file and the item will not vanish.
         */
        revents = epi->ffd.file->f_op->poll(epi->ffd.file, NULL);

        /*
         * Set the return event set for the current file descriptor.
         * Note that only the task task was successfully able to link
         * the item to its "txlist" will write this field.
         */
        epi->revents = revents & epi->event.events;

        if (epi->revents) {
            if (__put_user(epi->revents,
                       &events[eventcnt].events) ||
                __put_user(epi->event.data,
                       &events[eventcnt].data))
                return -EFAULT;
            if (epi->event.events & EPOLLONESHOT)
                epi->event.events &= EP_PRIVATE_BITS;
            eventcnt++;
        }
    }
    return eventcnt;
}
```

这个拷贝实现其实没什么可看的，但是请注意1359行，这个poll很狡猾，它把第二个参数置为NULL来调用。我们先看一下设备驱动通常是怎么实现poll的：

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
        struct scull_pipe *dev = filp->private_data;
        unsigned int mask = 0;

        /*
        * The buffer is circular; it is considered full
        * if "wp" is right behind "rp" and empty if the
        * two are equal.
        */
        down(&dev->sem);
        poll_wait(filp, &dev->inq,  wait);
        poll_wait(filp, &dev->outq, wait);
        if (dev->rp != dev->wp)
                mask |= POLLIN | POLLRDNORM;  /* readable */
        if (spacefree(dev))
                mask |= POLLOUT | POLLWRNORM;  /* writable */
        up(&dev->sem);
        return mask;
}
```

上面这段代码摘自《linux设备驱动程序（第三版）》，绝对经典，设备先要把current（当前进程）挂在inq和outq两个队列上（这个“挂”操作是wait回调函数指针做的），然后等设备来唤醒，唤醒后就能通过mask拿到事件掩码了（注意那个mask参数，它就是负责拿事件掩码的）。那如果wait为NULL，poll_wait会做些什么呢？

```
[include/linux/poll.h->poll_wait]
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
    if (p && wait_address)
        p->qproc(filp, wait_address, p);
}
```

喏，看见了，如果poll_table为空，什么也不做。我们倒回ep_send_events，那句标红的poll，实际上就是“我不想休眠，我只想拿到事件掩码”的意思。然后再把拿到的事件掩码拷给用户空间。ep_send_events完成后，就轮到ep_reinject_items了：

```
[fs/eventpoll.c->ep_reinject_items]
static void ep_reinject_items(struct eventpoll *ep, struct list_head *txlist)
{
    int ricnt = 0, pwake = 0;
    unsigned long flags;
    struct epitem *epi;

    write_lock_irqsave(&ep->lock, flags);

    while (!list_empty(txlist)) {
        epi = list_entry(txlist->next, struct epitem, txlink);

```

```
1400         /* Unlink the current item from the transfer list */
1401         EP_LIST_DEL(&epi->txlink);
1402
1403         /*
1404          * If the item is no more linked to the interest set, we don't
1405          * have to push it inside the ready list because the following
1406          * ep_release_epitem() is going to drop it. Also, if the current
1407          * item is set to have an Edge Triggered behaviour, we don't have
1408          * to push it back either.
1409          */
1410         if (EP_RB_LINKED(&epi->rbn) && !(epi->event.events & EPOLLET) &&
1411             (epi->revents & epi->event.events) && !EP_IS_LINKED(&epi->rdllink)) {
1412             list_add_tail(&epi->rdllink, &ep->rdllist);
1413             ricnt++;
1414         }
1415     }
1416
1417     if (ricnt) {
1418         /*
1419          * Wake up ( if active ) both the eventpoll wait list and the ->poll()
1420          * wait list.
1421          */
1422         if (waitqueue_active(&ep->wq))
1423             wake_up(&ep->wq);
1424         if (waitqueue_active(&ep->poll_wait))
1425             pwake++;
1426     }
1427
1428     write_unlock_irqrestore(&ep->lock, flags);
1429
1430     /* We have to call this outside the lock */
1431     if (pwake)
1432         ep_poll_safewake(&psw, &ep->poll_wait);
1433 }
```

ep_reinject_items把txlist里的一部分fd又放回rdllist，那么，是把哪一部分fd放回去呢？看上面1410行的那个判断——是哪些"没有标上EPOLLET"（标红代码）且"事件被关注"（标蓝代码）的fd被重新放回了rdllist。那么下次epoll_wait当然会又把rdllist里的fd拿来拷给用户了。

举个例子。假设一个socket，只是connect，还没有收发数据，那么它的poll事件掩码总是有POLLOUT的（参见上面的驱动示例），每次调用epoll_wait总是返回POLLOUT事件（比较烦），因为它的fd就总是被放回rdllist；假如此时有人往这个socket里写了一大堆数据，造成socket塞住（不可写了），那么1411行里标蓝色的判断就不成立了（没有POLLOUT了），fd不会放回rdllist，epoll_wait将不会再返回用户POLLOUT事件。现在我们给这个socket加上EPOLLET，然后connect，没有收发数据，此时，1410行标红的判断又不成立了，所以epoll_wait只会返回一次POLLOUT通知给用户（因为此fd不会再回到rdllist了），接下来的epoll_wait都不会有任何事件通知了。