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)
{
```

```
    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;
    }
```

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 */
```

```c
    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()]

```c
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
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：