<https://blog.csdn.net/weiyuefei/article/details/52242936>

**Epoll实现分析——作者：lvyilong316**

通过上一章分析，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，如果要监听1000个fd，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。  
    另外，epoll机制实现了自己特有的文件系统eventpoll filesystem

**1. 内核数据结构**

**(1) struct** eventpoll {

    spinlock\_t lock;

**struct** mutex mtx;

      wait\_queue\_head\_t wq;  /\* Wait queue used by sys\_epoll\_wait() ,调用epoll\_wait()时, 我们就是"睡"在了这个等待队列上\*/

 wait\_queue\_head\_t poll\_wait;  /\* Wait queue used by file->poll() , 这个用于epollfd本事被poll的时候\*/

**struct**list\_head rdllist; /\* List of ready file descriptors, 所有已经ready的epitem都在这个链表里面\*/

**struct**rb\_root rbr; /\* RB tree root used to store monitored fd structs, 所有要监听的epitem都在这里\*/

epitem \*ovflist;  /\*存放的epitem都是我们在传递数据给用户空间时监听到了事件\*/.

**struct** user\_struct \*user; /\*这里保存了一些用户变量,比如fd监听数量的最大值等\*/

};

通过epoll\_ctl接口加入该epoll描述符监听的套接字则属于socket filesystem，这点一定要注意。每个添加的待监听（这里监听和listen调用不同）都对应于一个epitem结构体，该结构体已红黑树的结构组织，eventpoll结构中保存了树的根节点（rbr成员）。同时有监听事件到来的套接字的该结构以双向链表组织起来，链表头保存在eventpoll中（rdllist成员）。

/\*

 \* Each file descriptor added to the eventpoll interface will  have an entry of this type linked to the "rbr" RB tree.

 \*/

**(2) struct** epitem {

**struct** rb\_node rbn;      /\* RB tree node used to link this structure to the eventpoll RB tree \*/

**struct**list\_head rdllink;  /\* 链表节点, 所有已经ready的epitem都会被链到eventpoll的rdllist中 \*/

**struct** epitem \*next;

**struct** epoll\_filefd ffd;    /\* The file descriptor information this item refers to \*/

**int** nwait;   /\* Number of active wait queue attached to poll operations \*/

**struct** list\_head pwqlist;  /\* List containing poll wait queues \*/

**struct** eventpoll \*ep;  /\* The "container" of this item \*/

**struct** list\_head fllink; /\* List header used to link this item to the "struct file" items list \*/

**struct** epoll\_event event;   /\*当前的epitem关系哪些events, 这个数据是调用epoll\_ctl时从用户态传递过来 \*/

};

**(3)**struct epoll\_filefd {

struct file \*file;

int fd;};

**(4)** struct eppoll\_entry { /\* Wait structure used by the poll hooks \*/

struct list\_head llink; /\* List header used to link this structure to the "struct epitem" \*/

struct epitem \*base; /\* The "base" pointer is set to the container "struct epitem" \*/

wait\_queue\_t wait; / Wait queue item that will be linked to the target file wait queue head. /

wait\_queue\_head\_t \*whead;/The wait queue head that linked the "wait" wait queue item \*/

};//注：后两项相当于等待队列

**(5)** struct ep\_pqueue {/\* Wrapper struct used by poll queueing \*/

poll\_table pt;   // struct poll\_table是一个函数指针的包裹

struct epitem \*epi;

};

**(6)**struct ep\_send\_events\_data {

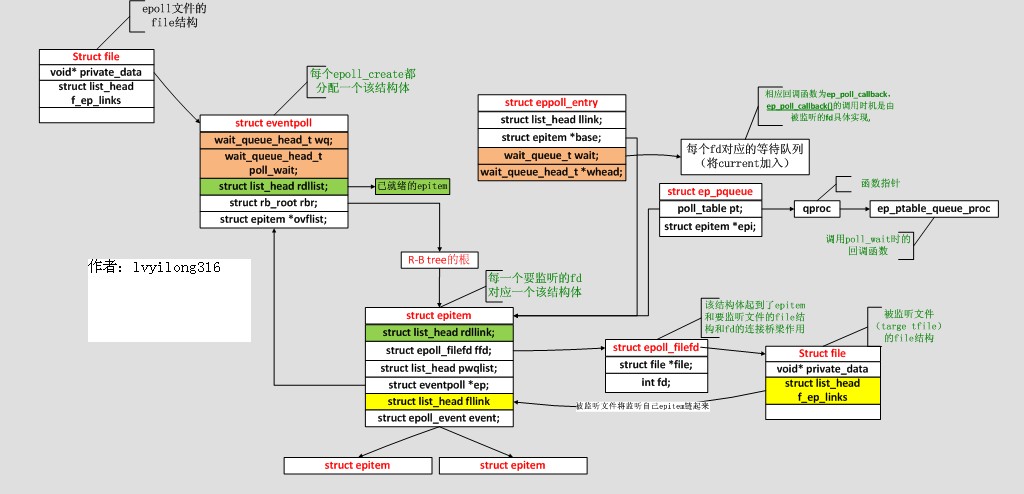
     /\* Used by the ep\_send\_events() function as callback private data \*/

int maxevents;

struct epoll\_event \_\_user \*events;

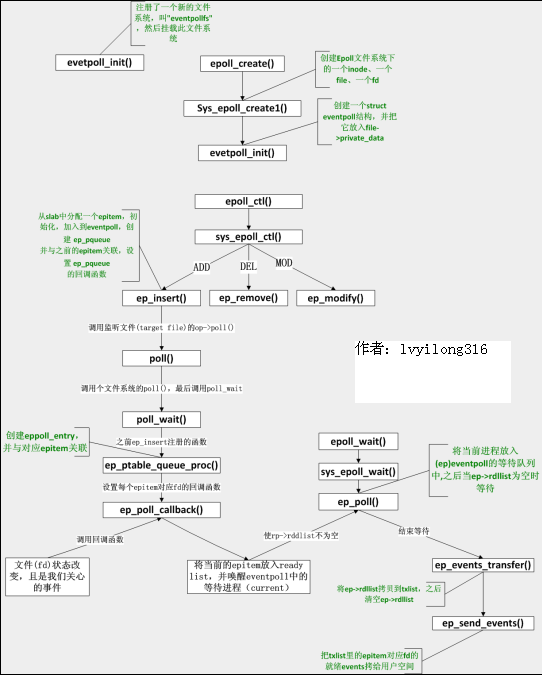
};

各个数据结构的关系如下图：



**2. 函数调用分析**

epoll函数调用关系全局图：



**3. 函数实现分析**

**3.1 eventpoll\_init**

epoll是个module，所以先看看module的入口eventpoll\_init  
[fs/eventpoll.c-->evetpoll\_init()]（简化后）  
 static int \_\_init eventpoll\_init(void)  
 {  
 epi\_cache = kmem\_cache\_create("eventpoll\_epi", sizeof(struct epitem),  
 0, SLAB\_HWCACHE\_ALIGN|EPI\_SLAB\_DEBUG|SLAB\_PANIC, NULL, NULL);

 pwq\_cache = kmem\_cache\_create("eventpoll\_pwq",  
 sizeof(struct eppoll\_entry), 0, EPI\_SLAB\_DEBUG|SLAB\_PANIC, NULL, NULL);  
 //注册了一个新的文件系统，叫"eventpollfs"

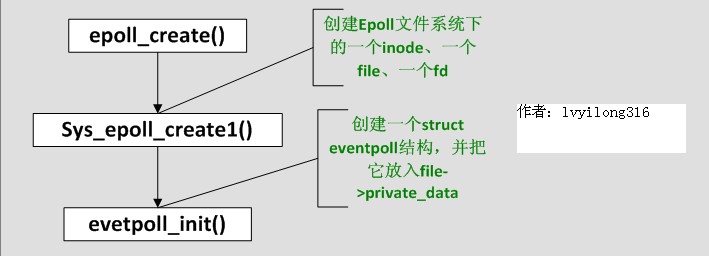
 error = register\_filesystem(&eventpoll\_fs\_type);  
eventpoll\_mnt = kern\_mount(&eventpoll\_fs\_type);;  
 }  
很有趣，这个module在初始化时注册了一个新的文件系统，叫"eventpollfs"（在eventpoll\_fs\_type结构里），然后挂载此文件系统。另外创建两个内核cache（在内核编程中，如果需要频繁分配小块内存，应该创建kmem\_cahe来做“内存池”）,分别用于存放struct epitem和eppoll\_entry。

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：

**3.2 sys\_epoll\_create**

[fs/eventpoll.c-->sys\_epoll\_create()]  
 asmlinkage long sys\_epoll\_create(int size)  
 {  
 int error, fd;  
 struct inode \*inode;  
 struct file \*file;  
 error = ep\_getfd(&fd, &inode, &file);  
 /\* Setup the file internal data structure ( "struct eventpoll" ) \*/  
 error = ep\_file\_init(file);

}  
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。

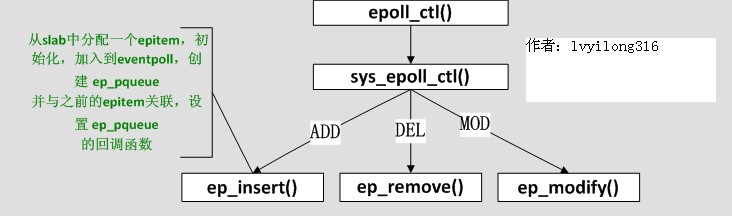


**3.3 epoll\_ctl**

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)  
 {  
 struct file \*file, \*tfile;  
 struct eventpoll \*ep;  
 struct epitem \*epi;  
 struct epoll\_event epds;  
....  
 epi = ep\_find(ep, tfile, fd);//tfile存放要监听的fd对应在rb-tree中的epitem  
 switch (op) {//省略了判空处理  
 case EPOLL\_CTL\_ADD: epds.events |= POLLERR | POLLHUP;

 error = ep\_insert(ep, &epds, tfile, fd); break;  
 case EPOLL\_CTL\_DEL: error = ep\_remove(ep, epi); break;  
 case EPOLL\_CTL\_MOD: epds.events |= POLLERR | POLLHUP;

 error = ep\_modify(ep, epi, &epds); break;  
 }  
    原来就是在一个“大的结构”（struct eventpoll）里先ep\_find，如果找到了struct epitem,而根据用户操作是ADD、DEL、MOD调用相应的函数，这些函数在epitem组成红黑树中增加、删除、修改相应节点（每一个监听fd对应一个节点）。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！



**3.4 sys\_epoll\_wait**

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)  
 {  
 struct file \*file;  
 struct eventpoll \*ep;  
 /\* Get the "struct file \*" for the eventpoll file \*/  
 file = fget(epfd);  
/\*  
 \* We have to check that the file structure underneath the fd  
 \* the user passed to us \_is\_ an eventpoll file.(所以如果这里是普通的文件fd会出错)  
 \*/  
 if (!IS\_FILE\_EPOLL(file))  
 goto eexit\_2;

 ep = file->private\_data;  
 error = ep\_poll(ep, events, maxevents, timeout);

……

 }

故伎重演，从file->private\_data中拿到struct eventpoll，再调用ep\_poll

**3.5 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;  
 wait\_queue\_t wait;//等待队列项  
 if (list\_empty(&ep->rdllist)) {  
 //ep->rdllist存放的是已就绪(read)的fd，为空时说明当前没有就绪的fd，所以需要将当前  
 init\_waitqueue\_entry(&wait, current);//创建一个等待队列项，并使用当前进程（current）初始化  
 add\_wait\_queue(&ep->wq, &wait);//将刚创建的等待队列项加入到ep中的等待队列（即将当前进程添加到等待队列）  
 for (;;) {  
 /\*将进程状态设置为TASK\_INTERRUPTIBLE，因为我们不希望这期间ep\_poll\_callback()发信号唤醒进程的时候，进程还在sleep \*/  
 set\_current\_state(TASK\_INTERRUPTIBLE);  
 if (!list\_empty(&ep->rdllist) || !jtimeout)//如果ep->rdllist非空(即有就绪的fd)或时间到则跳                           出循环

 break;  
 if (signal\_pending(current)) {  
 res = -EINTR;  
 break;  
 }  
 }  
 remove\_wait\_queue(&ep->wq, &wait);//将等待队列项移出等待队列(将当前进程移出)  
 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.



**3.6 ep\_insert()**

[fs/eventpoll.c-->sys\_epoll\_ctl()-->ep\_insert()]  
static int ep\_insert(struct eventpoll \*ep, struct epoll\_event \*event, struct file \*tfile, int fd)  
{

struct epitem \*epi;  
 struct ep\_pqueue epq;// 创建ep\_pqueue对象  
epi = EPI\_MEM\_ALLOC();//分配一个epitem  
/\* 初始化这个epitem ... \*/  
 epi->ep = ep;//将创建的epitem添加到传进来的struct eventpoll

/\*后几行是设置epitem的相应字段\*/  
 EP\_SET\_FFD(&epi->ffd, tfile, fd);//将要监听的fd加入到刚创建的epitem  
 epi->event = \*event;  
 epi->nwait = 0;

/\* Initialize the poll table using the queue callback \*/  
epq.epi = epi;  //将一个epq和新插入的epitem(epi)关联

//下面一句等价于&(epq.pt)->qproc = ep\_ptable\_queue\_proc;

init\_poll\_funcptr(&epq.pt, ep\_ptable\_queue\_proc);

revents = tfile->f\_op->poll(tfile, &epq.pt);  //tfile代表target file，即被监听的文件,poll()返回就绪事件的掩码，赋给revents.

list\_add\_tail(&epi->fllink, &tfile->f\_ep\_links);// 每个文件会将所有监听自己的epitem链起来

ep\_rbtree\_insert(ep, epi);// 都搞定后, 将epitem插入到对应的eventpoll中去

……

}

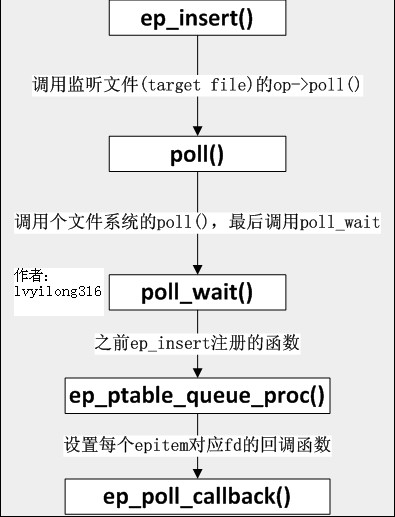
紧接着 tfile->f\_op->poll(tfile, &epq.pt)其实就是调用被监控文件（epoll里叫“target file”)的poll方法，而这个poll其实就是调用poll\_wait（还记得poll\_wait吗？每个支持poll的设备驱动程序都要调用的），最后就是调用ep\_ptable\_queue\_proc。**（注：f\_op->poll()一般来说只是个wrapper, 它会调用真正的poll实现, 拿UDP的socket来举例, 这里就是这样的调用流程: f\_op->poll(), sock\_poll(), udp\_poll(), datagram\_poll(), sock\_poll\_wait()。）**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。

**3.7 ep\_ptable\_queue\_proc**

[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进程时，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(ep->rbr)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！

**3.8 ep\_poll\_callback**

[fs/eventpoll.c-->ep\_poll\_callback()]  
static int ep\_poll\_callback(wait\_queue\_t \*wait, unsigned mode, int sync, void \*key)  
{  
 int pwake = 0;  
 struct epitem \*epi = EP\_ITEM\_FROM\_WAIT(wait);  
 struct eventpoll \*ep = epi->ep;  
 /\* 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++;  
 }



**4. 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，回调函数就把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：

**4.1 ep\_poll()    (接3.5)**

[fs/eventpoll.c->ep\_poll()]

 /\* Try to transfer events to user space. \*/  
 ep\_events\_transfer(ep, events, maxevents)  
 ......  
把rdllist里的fd拷到用户空间，这个任务是ep\_events\_transfer做的.

**4.2 ep\_events\_transfer**

[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);  
/\* 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：

**4.3 ep\_send\_events()**

[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;  
list\_for\_each(lnk, txlist) {  
epi = list\_entry(lnk, struct epitem, txlink);  
revents = epi->ffd.file->f\_op->poll(epi->ffd.file, NULL);//调用每个监听文件的poll方法获取就绪事件（掩码），并赋值给revents

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))//将event从内核空间发送到用户空间  
     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，但是请注意红色的一行，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;  
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 \*/  
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？

**4.4 poll\_wait**

[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。

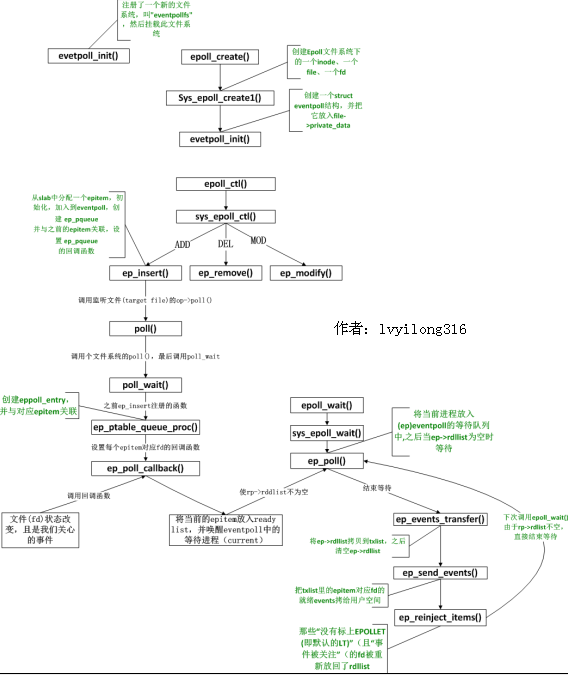
**4.5 p\_reinject\_items**

[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;  
     while (!list\_empty(txlist)) {//遍历txlist（此时txlist存放的是已就绪的epitem）  
     epi = list\_entry(txlist->next, struct epitem, txlink);  
     EP\_LIST\_DEL(&epi->txlink);//将当前的epitem从txlist中删除  
     if (EP\_RB\_LINKED(&epi->rbn) && !(epi->event.events & EPOLLET) &&  
     (epi->revents & epi->event.events) && !EP\_IS\_LINKED(&epi->rdllink)) {

     list\_add\_tail(&epi->rdllink, &ep->rdllist);//将当前epitem重新加入ep->rdllist  
     ricnt++;// ep->rdllist中epitem的个数（即从新加入就绪的epitem的个数）  
      }  
    }  
 if (ricnt) {//如果ep->rdllist不空，重新唤醒等、等待队列的进程（current）  
    if (waitqueue\_active(&ep->wq))  
    wake\_up(&ep->wq);  
    if (waitqueue\_active(&ep->poll\_wait))  
    pwake++;  
    }  
   ……

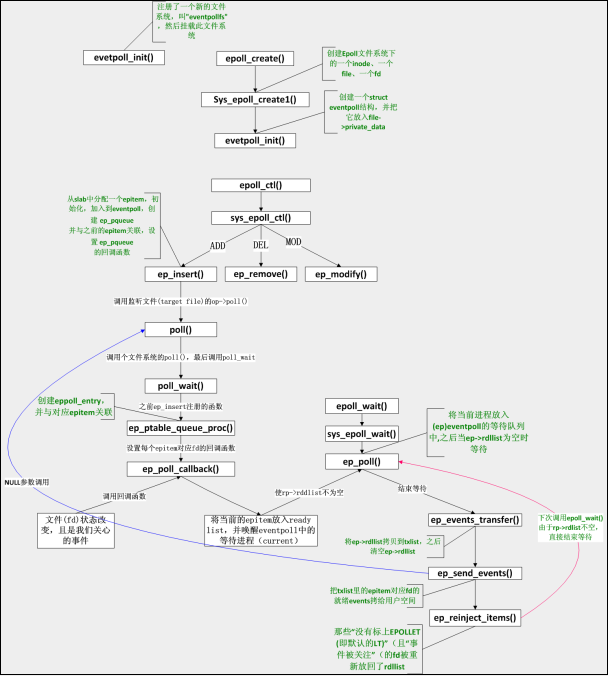
}  
ep\_reinject\_items把txlist里的一部分fd又放回rdllist，那么，是把哪一部分fd放回去呢？看上面那个判断——是**那些“没有标上EPOLLET(即默认的LT)”（标红代码）且“事件被关注”（标蓝代码）的fd被重新放回了rdllist。那么下次epoll\_wait当然会又把rdllist里的fd拿来拷给用户了。**举个例子。假设一个socket，只是connect，还没有收发数据，那么它的poll事件掩码总是有POLLOUT的（参见上面的驱动示例），每次调用epoll\_wait总是返回POLLOUT事件（比较烦），因为它的fd就总是被放回rdllist；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，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。

**总结：**

epoll函数调用关系全局图：  


**注：**上述函数关系图中有个问题，当ep\_reinject\_items()将LT的上次就绪的eptiem重新放回就绪链表，下次ep\_poll()直接返回，这不就造成了一个循环了吗？什么时候这些LT的epitem才不再加入就绪链表呢？这个问题的解决在4.3——ep\_send\_events()中，注意这个函数中标红的那个poll调用，我们分析过当传入NULL时，poll仅仅是拿到事件掩码，所以如果之前用户对事件的处理导致的文件的revents（状态）改变，那么这里就会得到更新。例如：用户以可读监听，当读完数据后文件的会变为不可读，这时ep\_send\_events()中获取的revents中将不再有可读事件，也就不满足ep\_reinject\_items()中的蓝色判断，所以epitem不再被加入就绪链表（ep->rdllist）。但是如果只读部分数据，并不会引起文件状态改变（文件仍可读），所以仍会加入就绪链表通知用户空间，这也就是如果是TL，就会一直通知用户读事件，直到某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。

将上述调用添加到函数调用关系图后，如下（添加的为蓝线）：



epoll实现数据结构全局关系图：

