<https://www.cnblogs.com/zengyiwen/p/7213f3303eca4bb08fd270f7d5772100.html>

/\*

 \*  fs/eventpoll.c (Efficient event retrieval implementation)

 \*  Copyright (C) 2001,...,2009  Davide Libenzi

 \*

 \*  This program is free software; you can redistribute it and/or modify

 \*  it under the terms of the GNU General Public License as published by

 \*  the Free Software Foundation; either version 2 of the License, or

 \*  (at your option) any later version.

 \*

 \*  Davide Libenzi <davidel@xmailserver.org>

 \*

 \*/

/\*

 \* 在深入了解epoll的实现之前, 先来了解内核的3个方面.

 \* 1. 等待队列 waitqueue

 \* 我们简单解释一下等待队列:

 \* 队列头(wait\_queue\_head\_t)往往是资源生产者,

 \* 队列成员(wait\_queue\_t)往往是资源消费者,

 \* 当头的资源ready后, 会逐个执行每个成员指定的回调函数,

 \* 来通知它们资源已经ready了, 等待队列大致就这个意思.

 \* 2. 内核的poll机制

 \* 被Poll的fd, 必须在实现上支持内核的Poll技术,

 \* 比如fd是某个字符设备,或者是个socket, 它必须实现

 \* file\_operations中的poll操作, 给自己分配有一个等待队列头.

 \* 主动poll fd的某个进程必须分配一个等待队列成员, 添加到

 \* fd的对待队列里面去, 并指定资源ready时的回调函数.

 \* 用socket做例子, 它必须有实现一个poll操作, 这个Poll是

 \* 发起轮询的代码必须主动调用的, 该函数中必须调用poll\_wait(),

 \* poll\_wait会将发起者作为等待队列成员加入到socket的等待队列中去.

 \* 这样socket发生状态变化时可以通过队列头逐个通知所有关心它的进程.

 \* 这一点必须很清楚的理解, 否则会想不明白epoll是如何

 \* 得知fd的状态发生变化的.

 \* 3. epollfd本身也是个fd, 所以它本身也可以被epoll,

 \* 可以猜测一下它是不是可以无限嵌套epoll下去...

 \*

 \* epoll基本上就是使用了上面的1,2点来完成.

 \* 可见epoll本身并没有给内核引入什么特别复杂或者高深的技术,

 \* 只不过是已有功能的重新组合, 达到了超过select的效果.

 \*/

/\*

 \* 相关的其它内核知识:

 \* 1. fd我们知道是文件描述符, 在内核态, 与之对应的是struct file结构,

 \* 可以看作是内核态的文件描述符.

 \* 2. spinlock, 自旋锁, 必须要非常小心使用的锁,

 \* 尤其是调用spin\_lock\_irqsave()的时候, 中断关闭, 不会发生进程调度,

 \* 被保护的资源其它CPU也无法访问. 这个锁是很强力的, 所以只能锁一些

 \* 非常轻量级的操作.

 \* 3. 引用计数在内核中是非常重要的概念,

 \* 内核代码里面经常有些release, free释放资源的函数几乎不加任何锁,

 \* 这是因为这些函数往往是在对象的引用计数变成0时被调用,

 \* 既然没有进程在使用在这些对象, 自然也不需要加锁.

 \* struct file 是持有引用计数的.

 \*/

/\* --- epoll相关的数据结构 --- \*/

/\*

 \* This structure is stored inside the "private\_data" member of the file

 \* structure and rapresent the main data sructure for the eventpoll

 \* interface.

 \*/

/\* 每创建一个epollfd, 内核就会分配一个eventpoll与之对应, 可以说是

 \* 内核态的epollfd. \*/

struct eventpoll {

    /\* Protect the this structure access \*/

    spinlock\_t lock;

    /\*

     \* This mutex is used to ensure that files are not removed

     \* while epoll is using them. This is held during the event

     \* collection loop, the file cleanup path, the epoll file exit

     \* code and the ctl operations.

     \*/

    /\* 添加, 修改或者删除监听fd的时候, 以及epoll\_wait返回, 向用户空间

     \* 传递数据时都会持有这个互斥锁, 所以在用户空间可以放心的在多个线程

     \* 中同时执行epoll相关的操作, 内核级已经做了保护. \*/

    struct mutex mtx;

    /\* 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 wq;

    /\* Wait queue used by file->poll() \*/

    /\* 这个用于epollfd本事被poll的时候... \*/

    wait\_queue\_head\_t poll\_wait;

    /\* List of ready file descriptors \*/

    /\* 所有已经ready的epitem都在这个链表里面 \*/

    struct list\_head rdllist;

    /\* RB tree root used to store monitored fd structs \*/

    /\* 所有要监听的epitem都在这里 \*/

    struct rb\_root rbr;

    /\*

        这是一个单链表链接着所有的struct epitem当event转移到用户空间时

     \*/

     \* This is a single linked list that chains all the "struct epitem" that

     \* happened while transfering ready events to userspace w/out

     \* holding ->lock.

     \*/

    struct epitem \*ovflist;

    /\* The user that created the eventpoll descriptor \*/

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

    struct user\_struct \*user;

};

/\*

 \* Each file descriptor added to the eventpoll interface will

 \* have an entry of this type linked to the "rbr" RB tree.

 \*/

/\* epitem 表示一个被监听的fd \*/

struct epitem {

    /\* RB tree node used to link this structure to the eventpoll RB tree \*/

    /\* rb\_node, 当使用epoll\_ctl()将一批fds加入到某个epollfd时, 内核会分配

     \* 一批的epitem与fds们对应, 而且它们以rb\_tree的形式组织起来, tree的root

     \* 保存在epollfd, 也就是struct eventpoll中.

     \* 在这里使用rb\_tree的原因我认为是提高查找,插入以及删除的速度.

     \* rb\_tree对以上3个操作都具有O(lgN)的时间复杂度 \*/

    struct rb\_node rbn;

    /\* List header used to link this structure to the eventpoll ready list \*/

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

    struct list\_head rdllink;

    /\*

     \* Works together "struct eventpoll"->ovflist in keeping the

     \* single linked chain of items.

     \*/

    /\* 这个在代码中再解释... \*/

    struct epitem \*next;

    /\* The file descriptor information this item refers to \*/

    /\* epitem对应的fd和struct file \*/

    struct epoll\_filefd ffd;

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

    int nwait;

    /\* List containing poll wait queues \*/

    struct list\_head pwqlist;

    /\* The "container" of this item \*/

    /\* 当前epitem属于哪个eventpoll \*/

    struct eventpoll \*ep;

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

    struct list\_head fllink;

    /\* The structure that describe the interested events and the source fd \*/

    /\* 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 \*/

    struct epoll\_event event;

};

struct epoll\_filefd {

    struct file \*file;

    int fd;

};

/\* poll所用到的钩子Wait structure used by the poll hooks \*/

struct eppoll\_entry {

    /\* List header used to link this structure to the "struct epitem" \*/

    struct list\_head llink;

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

    struct epitem \*base;

    /\*

     \* Wait queue item that will be linked to the target file wait

     \* queue head.

     \*/

    wait\_queue\_t wait;

    /\* The wait queue head that linked the "wait" wait queue item \*/

    wait\_queue\_head\_t \*whead;

};

/\* Wrapper struct used by poll queueing \*/

struct ep\_pqueue {

    poll\_table pt;

    struct epitem \*epi;

};

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

struct ep\_send\_events\_data {

    int maxevents;

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

};

/\* --- 代码注释 --- \*/

/\* 你没看错, 这就是epoll\_create()的真身, 基本啥也不干直接调用epoll\_create1了,

 \* 另外你也可以发现, size这个参数其实是没有任何用处的... \*/

SYSCALL\_DEFINE1(epoll\_create, int, size)

{

        if (size <= 0)

                return -EINVAL;

        return sys\_epoll\_create1(0);

}

/\* 这才是真正的epoll\_create啊~~ \*/

SYSCALL\_DEFINE1(epoll\_create1, int, flags)

{

    int error;

    struct eventpoll \*ep = NULL;//主描述符

    /\* Check the EPOLL\_\* constant for consistency.  \*/

    /\* 这句没啥用处... \*/

    BUILD\_BUG\_ON(EPOLL\_CLOEXEC != O\_CLOEXEC);

    /\* 对于epoll来讲, 目前唯一有效的FLAG就是CLOEXEC \*/

    if (flags & ~EPOLL\_CLOEXEC)

        return -EINVAL;

    /\*

     \* Create the internal data structure ("struct eventpoll").

     \*/

    /\* 分配一个struct eventpoll, 分配和初始化细节我们随后深聊~ \*/

    error = ep\_alloc(&ep);

    if (error < 0)

        return error;

    /\*

     \* Creates all the items needed to setup an eventpoll file. That is,

     \* a file structure and a free file descriptor.

     \*/

    /\* 这里是创建一个匿名fd, 说起来就话长了...长话短说:

     \* epollfd本身并不存在一个真正的文件与之对应, 所以内核需要创建一个

     \* "虚拟"的文件, 并为之分配真正的struct file结构, 而且有真正的fd.

     \* 这里2个参数比较关键:

     \* eventpoll\_fops, fops就是file operations, 就是当你对这个文件(这里是虚拟的)进行操作(比如读)时,

     \* fops里面的函数指针指向真正的操作实现, 类似C++里面虚函数和子类的概念.

     \* epoll只实现了poll和release(就是close)操作, 其它文件系统操作都有VFS全权处理了.

     \* ep, ep就是struct epollevent, 它会作为一个私有数据保存在struct file的private指针里面.

     \* 其实说白了, 就是为了能通过fd找到struct file, 通过struct file能找到eventpoll结构.

     \* 如果懂一点Linux下字符设备驱动开发, 这里应该是很好理解的,

     \* 推荐阅读 <Linux device driver 3rd>

     \*/

    error = anon\_inode\_getfd("[eventpoll]", &eventpoll\_fops, ep,

                 O\_RDWR | (flags & O\_CLOEXEC));

    if (error < 0)

        ep\_free(ep);

    return error;

}

/\*

\* 创建好epollfd后, 接下来我们要往里面添加fd咯

\* 来看epoll\_ctl

\* epfd 就是epollfd

\* op ADD,MOD,DEL

\* fd 需要监听的描述符

\* event 我们关心的events

\*/

SYSCALL\_DEFINE4(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;

    error = -EFAULT;

    /\*

     \* 错误处理以及从用户空间将epoll\_event结构copy到内核空间.

     \*/

    if (ep\_op\_has\_event(op) &&

        copy\_from\_user(&epds, event, sizeof(struct epoll\_event)))

        goto error\_return;

    /\* Get the "struct file \*" for the eventpoll file \*/

    /\* 取得struct file结构, epfd既然是真正的fd, 那么内核空间

     \* 就会有与之对于的一个struct file结构

     \* 这个结构在epoll\_create1()中, 由函数anon\_inode\_getfd()分配 \*/

    error = -EBADF;

    file = fget(epfd);

    if (!file)

        goto error\_return;

    /\* Get the "struct file \*" for the target file \*/

    /\* 我们需要监听的fd, 它当然也有个struct file结构, 上下2个不要搞混了哦 \*/

    tfile = fget(fd);

    if (!tfile)

        goto error\_fput;

    /\* The target file descriptor must support poll \*/

    error = -EPERM;

    /\* 如果监听的文件不支持poll, 那就没辙了.

     \* 你知道什么情况下, 文件会不支持poll吗?

     \*/

    if (!tfile->f\_op || !tfile->f\_op->poll)

        goto error\_tgt\_fput;

    /\*

     \* We have to check that the file structure underneath the file descriptor

     \* the user passed to us \_is\_ an eventpoll file. And also we do not permit

     \* adding an epoll file descriptor inside itself.

     \*/

    error = -EINVAL;

    /\* epoll不能自己监听自己... \*/

    if (file == tfile || !is\_file\_epoll(file))

        goto error\_tgt\_fput;

    /\*

     \* At this point it is safe to assume that the "private\_data" contains

     \* our own data structure.

     \*/

    /\* 取到我们的eventpoll结构, 来自与epoll\_create1()中的分配 \*/

    ep = file->private\_data;

    /\* 接下来的操作有可能修改数据结构内容, 锁之~ \*/

    mutex\_lock(&ep->mtx);

    /\*

     \* Try to lookup the file inside our RB tree, Since we grabbed "mtx"

     \* above, we can be sure to be able to use the item looked up by

     \* ep\_find() till we release the mutex.

     \*/

    /\* 对于每一个监听的fd, 内核都有分配一个epitem结构,

     \* 而且我们也知道, epoll是不允许重复添加fd的,

     \* 所以我们首先查找该fd是不是已经存在了.

     \* ep\_find()其实就是RBTREE查找, 跟C++STL的map差不多一回事, O(lgn)的时间复杂度.

     \*/

    epi = ep\_find(ep, tfile, fd);

    error = -EINVAL;

    switch (op) {

        /\* 首先我们关心添加 \*/

    case EPOLL\_CTL\_ADD:

        if (!epi) {

            /\* 之前的find没有找到有效的epitem, 证明是第一次插入, 接受!

             \* 这里我们可以知道, POLLERR和POLLHUP事件内核总是会关心的

             \* \*/

            epds.events |= POLLERR | POLLHUP;

            /\* rbtree插入, 详情见ep\_insert()的分析

             \* 其实我觉得这里有insert的话, 之前的find应该

             \* 是可以省掉的... \*/

            error = ep\_insert(ep, &epds, tfile, fd);

        } else

            /\* 找到了!? 重复添加! \*/

            error = -EEXIST;

        break;

        /\* 删除和修改操作都比较简单 \*/

    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;

    }

    mutex\_unlock(&ep->mtx);

error\_tgt\_fput:

    fput(tfile);

error\_fput:

    fput(file);

error\_return:

    return error;

}

/\* 分配一个eventpoll结构 \*/

static int ep\_alloc(struct eventpoll \*\*pep)

{

    int error;

    struct user\_struct \*user;

    struct eventpoll \*ep;

    /\* 获取当前用户的一些信息, 比如是不是root啦, 最大监听fd数目啦 \*/

    user = get\_current\_user();

    error = -ENOMEM;

    ep = kzalloc(sizeof(\*ep), GFP\_KERNEL);

    if (unlikely(!ep))

        goto free\_uid;

    /\* 这些都是初始化啦 \*/

    spin\_lock\_init(&ep->lock);

    mutex\_init(&ep->mtx);

    init\_waitqueue\_head(&ep->wq);//初始化自己睡在的等待队列

    init\_waitqueue\_head(&ep->poll\_wait);//初始化

    INIT\_LIST\_HEAD(&ep->rdllist);//初始化就绪链表

    ep->rbr = RB\_ROOT;

    ep->ovflist = EP\_UNACTIVE\_PTR;

    ep->user = user;

    \*pep = ep;

    return 0;

free\_uid:

    free\_uid(user);

    return error;

}

/\*

 \* Must be called with "mtx" held.

 \*/

/\*

 \* ep\_insert()在epoll\_ctl()中被调用, 完成往epollfd里面添加一个监听fd的工作

 \* tfile是fd在内核态的struct file结构

 \*/

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;

    /\* 查看是否达到当前用户的最大监听数 \*/

    if (unlikely(atomic\_read(&ep->user->epoll\_watches) >=

             max\_user\_watches))

        return -ENOSPC;

    /\* 从著名的slab中分配一个epitem \*/

    if (!(epi = kmem\_cache\_alloc(epi\_cache, GFP\_KERNEL)))

        return -ENOMEM;

    /\* Item initialization follow here ... \*/

    /\* 这些都是相关成员的初始化... \*/

    INIT\_LIST\_HEAD(&epi->rdllink);

    INIT\_LIST\_HEAD(&epi->fllink);

    INIT\_LIST\_HEAD(&epi->pwqlist);

    epi->ep = ep;

    /\* 这里保存了我们需要监听的文件fd和它的file结构 \*/

    ep\_set\_ffd(&epi->ffd, tfile, fd);

    epi->event = \*event;

    epi->nwait = 0;

    /\* 这个指针的初值不是NULL哦... \*/

    epi->next = EP\_UNACTIVE\_PTR;

    /\* Initialize the poll table using the queue callback \*/

    /\* 好, 我们终于要进入到poll的正题了 \*/

    epq.epi = epi;

    /\* 初始化一个poll\_table

     \* 其实就是指定调用poll\_wait(注意不是epoll\_wait!!!)时的回调函数,和我们关心哪些events,

     \* ep\_ptable\_queue\_proc()就是我们的回调啦, 初值是所有event都关心 \*/

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

    /\*

     \* Attach the item to the poll hooks and get current event bits.

     \* We can safely use the file\* here because its usage count has

     \* been increased by the caller of this function. Note that after

     \* this operation completes, the poll callback can start hitting

     \* the new item.

     \*/

    /\* 这一部很关键, 也比较难懂, 完全是内核的poll机制导致的...

     \* 首先, 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(), 最后调用到我们上面指定的

     \* ep\_ptable\_queue\_proc()这个回调函数...(好深的调用路径...).

     \* 完成这一步, 我们的epitem就跟这个socket关联起来了, 当它有状态变化时,

     \* 会通过ep\_poll\_callback()来通知.

     \* 最后, 这个函数还会查询当前的fd是不是已经有啥event已经ready了, 有的话

     \* 会将event返回. \*/

    revents = tfile->f\_op->poll(tfile, &epq.pt);

    /\*

     \* We have to check if something went wrong during the poll wait queue

     \* install process. Namely an allocation for a wait queue failed due

     \* high memory pressure.

     \*/

    error = -ENOMEM;

    if (epi->nwait < 0)

        goto error\_unregister;

    /\* Add the current item to the list of active epoll hook for this file \*/

    /\* 这个就是每个文件会将所有监听自己的epitem链起来 \*/

    spin\_lock(&tfile->f\_lock);

    list\_add\_tail(&epi->fllink, &tfile->f\_ep\_links);

    spin\_unlock(&tfile->f\_lock);

    /\*

     \* Add the current item to the RB tree. All RB tree operations are

     \* protected by "mtx", and ep\_insert() is called with "mtx" held.

     \*/

    /\* 都搞定后, 将epitem插入到对应的eventpoll中去 \*/

    ep\_rbtree\_insert(ep, epi);

    /\* We have to drop the new item inside our item list to keep track of it \*/

    spin\_lock\_irqsave(&ep->lock, flags);

    /\* If the file is already "ready" we drop it inside the ready list \*/

    /\* 到达这里后, 如果我们监听的fd已经有事件发生, 那就要处理一下 \*/

    if ((revents & event->events) && !ep\_is\_linked(&epi->rdllink)) {

        /\* 将当前的epitem加入到ready list中去 \*/

        list\_add\_tail(&epi->rdllink, &ep->rdllist);

        /\* Notify waiting tasks that events are available \*/

        /\* 谁在epoll\_wait, 就唤醒它... \*/

        if (waitqueue\_active(&ep->wq))

            wake\_up\_locked(&ep->wq);

        /\* 谁在epoll当前的epollfd, 也唤醒它... \*/

        if (waitqueue\_active(&ep->poll\_wait))

            pwake++;

    }

    spin\_unlock\_irqrestore(&ep->lock, flags);

    atomic\_inc(&ep->user->epoll\_watches);

    /\* We have to call this outside the lock \*/

    if (pwake)

        ep\_poll\_safewake(&ep->poll\_wait);

    return 0;

error\_unregister:

    ep\_unregister\_pollwait(ep, epi);

    /\*

     \* We need to do this because an event could have been arrived on some

     \* allocated wait queue. Note that we don't care about the ep->ovflist

     \* list, since that is used/cleaned only inside a section bound by "mtx".

     \* And ep\_insert() is called with "mtx" held.

     \*/

    spin\_lock\_irqsave(&ep->lock, flags);

    if (ep\_is\_linked(&epi->rdllink))

        list\_del\_init(&epi->rdllink);

    spin\_unlock\_irqrestore(&ep->lock, flags);

    kmem\_cache\_free(epi\_cache, epi);

    return error;

}

/\*

 \* This is the callback that is used to add our wait queue to the

 \* target file wakeup lists.

 \*/

/\*

 \* 该函数在调用f\_op->poll()时会被调用.

 \* 也就是epoll主动poll某个fd时, 用来将epitem与指定的fd关联起来的.

 \* 关联的办法就是使用等待队列(waitqueue)

 \*/

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 = kmem\_cache\_alloc(pwq\_cache, GFP\_KERNEL))) {

        /\* 初始化等待队列, 指定ep\_poll\_callback为唤醒时的回调函数,

         \* 当我们监听的fd发生状态改变时, 也就是队列头被唤醒时,

         \* 指定的回调函数将会被调用. \*/

        init\_waitqueue\_func\_entry(&pwq->wait, ep\_poll\_callback);

        pwq->whead = whead;

        pwq->base = epi;

        /\* 将刚分配的等待队列成员加入到头中, 头是由fd持有的 \*/

        add\_wait\_queue(whead, &pwq->wait);

        list\_add\_tail(&pwq->llink, &epi->pwqlist);

        /\* nwait记录了当前epitem加入到了多少个等待队列中,

         \* 我认为这个值最大也只会是1... \*/

        epi->nwait++;

    } else {

        /\* We have to signal that an error occurred \*/

        epi->nwait = -1;

    }

}

/\*

 \* This is the callback that is passed to the wait queue wakeup

 \* machanism. It is called by the stored file descriptors when they

 \* have events to report.

 \*/

/\*

 \* 这个是关键性的回调函数, 当我们监听的fd发生状态改变时, 它会被调用.

 \* 参数key被当作一个unsigned long整数使用, 携带的是events.

 \*/

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);//从等待队列获取epitem.需要知道哪个进程挂载到这个设备

    struct eventpoll \*ep = epi->ep;//获取

    spin\_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 out\_unlock;

    /\*

     \* Check the events coming with the callback. At this stage, not

     \* every device reports the events in the "key" parameter of the

     \* callback. We need to be able to handle both cases here, hence the

     \* test for "key" != NULL before the event match test.

     \*/

    /\* 没有我们关心的event... \*/

    if (key && !((unsigned long) key & epi->event.events))

        goto out\_unlock;

    /\*

     \* If we are trasfering events to userspace, we can hold no locks

     \* (because we're accessing user memory, and because of linux f\_op->poll()

     \* semantics). All the events that happens during that period of time are

     \* chained in ep->ovflist and requeued later on.

     \*/

    /\*

     \* 这里看起来可能有点费解, 其实干的事情比较简单:

     \* 如果该callback被调用的同时, epoll\_wait()已经返回了,

     \* 也就是说, 此刻应用程序有可能已经在循环获取events,

     \* 这种情况下, 内核将此刻发生event的epitem用一个单独的链表

     \* 链起来, 不发给应用程序, 也不丢弃, 而是在下一次epoll\_wait

     \* 时返回给用户.

     \*/

    if (unlikely(ep->ovflist != EP\_UNACTIVE\_PTR)) {

        if (epi->next == EP\_UNACTIVE\_PTR) {

            epi->next = ep->ovflist;

            ep->ovflist = epi;

        }

        goto out\_unlock;

    }

    /\* If this file is already in the ready list we exit soon \*/

    /\* 将当前的epitem放入ready list \*/

    if (!ep\_is\_linked(&epi->rdllink))

        list\_add\_tail(&epi->rdllink, &ep->rdllist);

    /\*

     \* Wake up ( if active ) both the eventpoll wait list and the ->poll()

     \* wait list.

     \*/

    /\* 唤醒epoll\_wait... \*/

    if (waitqueue\_active(&ep->wq))

        wake\_up\_locked(&ep->wq);

    /\* 如果epollfd也在被poll, 那就唤醒队列里面的所有成员. \*/

    if (waitqueue\_active(&ep->poll\_wait))

        pwake++;

out\_unlock:

    spin\_unlock\_irqrestore(&ep->lock, flags);

    /\* We have to call this outside the lock \*/

    if (pwake)

        ep\_poll\_safewake(&ep->poll\_wait);

    return 1;

}

/\*

 \* Implement the event wait interface for the eventpoll file. It is the kernel

 \* part of the user space epoll\_wait(2).

 \*/

SYSCALL\_DEFINE4(epoll\_wait, int, epfd, struct epoll\_event \_\_user \*, events,

        int, maxevents, int, timeout)

{

    int error;

    struct file \*file;

    struct eventpoll \*ep;

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

    if (maxevents <= 0 || maxevents > EP\_MAX\_EVENTS)

        return -EINVAL;

    /\* Verify that the area passed by the user is writeable \*/

    /\* 这个地方有必要说明一下:

     \* 内核对应用程序采取的策略是"绝对不信任",

     \* 所以内核跟应用程序之间的数据交互大都是copy, 不允许(也时候也是不能...)指针引用.

     \* epoll\_wait()需要内核返回数据给用户空间, 内存由用户程序提供,

     \* 所以内核会用一些手段来验证这一段内存空间是不是有效的.

     \*/

    if (!access\_ok(VERIFY\_WRITE, events, maxevents \* sizeof(struct epoll\_event))) {

        error = -EFAULT;

        goto error\_return;

    }

    /\* Get the "struct file \*" for the eventpoll file \*/

    error = -EBADF;

    /\* 获取epollfd的struct file, epollfd也是文件嘛 \*/

    file = fget(epfd);

    if (!file)

        goto error\_return;

    /\*

     \* We have to check that the file structure underneath the fd

     \* the user passed to us \_is\_ an eventpoll file.

     \*/

    error = -EINVAL;

    /\* 检查一下它是不是一个真正的epollfd... \*/

    if (!is\_file\_epoll(file))

        goto error\_fput;

    /\*

     \* At this point it is safe to assume that the "private\_data" contains

     \* our own data structure.

     \*/

    /\* 获取eventpoll结构 \*/

    ep = file->private\_data;

    /\* Time to fish for events ... \*/

    /\* OK, 睡觉, 等待事件到来~~ \*/

    error = ep\_poll(ep, events, maxevents, timeout);

error\_fput:

    fput(file);

error\_return:

    return error;

}

/\* 这个函数真正将执行epoll\_wait的进程带入睡眠状态... \*/

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;//等待队列

    /\*

     \* 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.

     \*/

    /\* 计算睡觉时间, 毫秒要转换为HZ \*/

    jtimeout = (timeout < 0 || timeout >= EP\_MAX\_MSTIMEO) ?

        MAX\_SCHEDULE\_TIMEOUT : (timeout \* HZ + 999) / 1000;

retry:

    spin\_lock\_irqsave(&ep->lock, flags);

    res = 0;

    /\* 如果ready list不为空, 就不睡了, 直接干活... \*/

    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.

         \*/

        /\* OK, 初始化一个等待队列, 准备直接把自己挂起,

         \* 注意current是一个宏, 代表当前进程 \*/

        init\_waitqueue\_entry(&wait, current);//初始化等待队列,wait表示当前进程

        \_\_add\_wait\_queue\_exclusive(&ep->wq, &wait);//挂载到ep结构的等待队列

        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.

             \*/

            /\* 将当前进程设置位睡眠, 但是可以被信号唤醒的状态,

             \* 注意这个设置是"将来时", 我们此刻还没睡! \*/

            set\_current\_state(TASK\_INTERRUPTIBLE);

            /\* 如果这个时候, ready list里面有成员了,

             \* 或者睡眠时间已经过了, 就直接不睡了... \*/

            if (!list\_empty(&ep->rdllist) || !jtimeout)

                break;

            /\* 如果有信号产生, 也起床... \*/

            if (signal\_pending(current)) {

                res = -EINTR;

                break;

            }

            /\* 啥事都没有,解锁, 睡觉... \*/

            spin\_unlock\_irqrestore(&ep->lock, flags);

            /\* jtimeout这个时间后, 会被唤醒,

             \* ep\_poll\_callback()如果此时被调用,

             \* 那么我们就会直接被唤醒, 不用等时间了...

             \* 再次强调一下ep\_poll\_callback()的调用时机是由被监听的fd

             \* 的具体实现, 比如socket或者某个设备驱动来决定的,

             \* 因为等待队列头是他们持有的, epoll和当前进程

             \* 只是单纯的等待...

             \*\*/

            jtimeout = schedule\_timeout(jtimeout);//睡觉

            spin\_lock\_irqsave(&ep->lock, flags);

        }

        \_\_remove\_wait\_queue(&ep->wq, &wait);

        /\* OK 我们醒来了... \*/

        set\_current\_state(TASK\_RUNNING);

    }

    /\* Is it worth to try to dig for events ? \*/

    eavail = !list\_empty(&ep->rdllist) || ep->ovflist != EP\_UNACTIVE\_PTR;

    spin\_unlock\_irqrestore(&ep->lock, flags);

    /\*

     \* 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.

     \*/

    /\* 如果一切正常, 有event发生, 就开始准备数据copy给用户空间了... \*/

    if (!res && eavail &&

        !(res = ep\_send\_events(ep, events, maxevents)) && jtimeout)

        goto retry;

    return res;

}

/\* 这个简单, 我们直奔下一个... \*/

static int ep\_send\_events(struct eventpoll \*ep,

              struct epoll\_event \_\_user \*events, int maxevents)

{

    struct ep\_send\_events\_data esed;

    esed.maxevents = maxevents;

    esed.events = events;

    return ep\_scan\_ready\_list(ep, ep\_send\_events\_proc, &esed);

}

/\*\*

 \* ep\_scan\_ready\_list - Scans the ready list in a way that makes possible for

 \*                      the scan code, to call f\_op->poll(). Also allows for

 \*                      O(NumReady) performance.

 \*

 \* @ep: Pointer to the epoll private data structure.

 \* @sproc: Pointer to the scan callback.

 \* @priv: Private opaque data passed to the @sproc callback.

 \*

 \* Returns: The same integer error code returned by the @sproc callback.

 \*/

static int ep\_scan\_ready\_list(struct eventpoll \*ep,

                  int (\*sproc)(struct eventpoll \*,

                       struct list\_head \*, void \*),

                  void \*priv)

{

    int error, pwake = 0;

    unsigned long flags;

    struct epitem \*epi, \*nepi;

    LIST\_HEAD(txlist);

    /\*

     \* We need to lock this because we could be hit by

     \* eventpoll\_release\_file() and epoll\_ctl().

     \*/

    mutex\_lock(&ep->mtx);

    /\*

     \* Steal the ready list, and re-init the original one to the

     \* empty list. Also, set ep->ovflist to NULL so that events

     \* happening while looping w/out locks, are not lost. We cannot

     \* have the poll callback to queue directly on ep->rdllist,

     \* because we want the "sproc" callback to be able to do it

     \* in a lockless way.

     \*/

    spin\_lock\_irqsave(&ep->lock, flags);

    /\* 这一步要注意, 首先, 所有监听到events的epitem都链到rdllist上了,

     \* 但是这一步之后, 所有的epitem都转移到了txlist上, 而rdllist被清空了,

     \* 要注意哦, rdllist已经被清空了! \*/

    list\_splice\_init(&ep->rdllist, &txlist);

    /\* ovflist, 在ep\_poll\_callback()里面我解释过, 此时此刻我们不希望

     \* 有新的event加入到ready list中了, 保存后下次再处理... \*/

    ep->ovflist = NULL;

    spin\_unlock\_irqrestore(&ep->lock, flags);

    /\*

     \* Now call the callback function.

     \*/

    /\* 在这个回调函数里面处理每个epitem

     \* sproc 就是 ep\_send\_events\_proc, 下面会注释到. \*/

    error = (\*sproc)(ep, &txlist, priv);

    spin\_lock\_irqsave(&ep->lock, flags);

    /\*

     \* During the time we spent inside the "sproc" callback, some

     \* other events might have been queued by the poll callback.

     \* We re-insert them inside the main ready-list here.

     \*/

    /\* 现在我们来处理ovflist, 这些epitem都是我们在传递数据给用户空间时

     \* 监听到了事件. \*/

    for (nepi = ep->ovflist; (epi = nepi) != NULL;

         nepi = epi->next, epi->next = EP\_UNACTIVE\_PTR) {

        /\*

         \* We need to check if the item is already in the list.

         \* During the "sproc" callback execution time, items are

         \* queued into ->ovflist but the "txlist" might already

         \* contain them, and the list\_splice() below takes care of them.

         \*/

        /\* 将这些直接放入readylist \*/

        if (!ep\_is\_linked(&epi->rdllink))

            list\_add\_tail(&epi->rdllink, &ep->rdllist);

    }

    /\*

     \* We need to set back ep->ovflist to EP\_UNACTIVE\_PTR, so that after

     \* releasing the lock, events will be queued in the normal way inside

     \* ep->rdllist.

     \*/

    ep->ovflist = EP\_UNACTIVE\_PTR;

    /\*

     \* Quickly re-inject items left on "txlist".

     \*/

    /\* 上一次没有处理完的epitem, 重新插入到ready list \*/

    list\_splice(&txlist, &ep->rdllist);

    /\* ready list不为空, 直接唤醒... \*/

    if (!list\_empty(&ep->rdllist)) {

        /\*

         \* Wake up (if active) both the eventpoll wait list and

         \* the ->poll() wait list (delayed after we release the lock).

         \*/

        if (waitqueue\_active(&ep->wq))

            wake\_up\_locked(&ep->wq);

        if (waitqueue\_active(&ep->poll\_wait))

            pwake++;

    }

    spin\_unlock\_irqrestore(&ep->lock, flags);

    mutex\_unlock(&ep->mtx);

    /\* We have to call this outside the lock \*/

    if (pwake)

        ep\_poll\_safewake(&ep->poll\_wait);

    return error;

}

/\* 该函数作为callbakc在ep\_scan\_ready\_list()中被调用

 \* head是一个链表, 包含了已经ready的epitem,

 \* 这个不是eventpoll里面的ready list, 而是上面函数中的txlist.

 \*/

static int ep\_send\_events\_proc(struct eventpoll \*ep, struct list\_head \*head,

                   void \*priv)

{

    struct ep\_send\_events\_data \*esed = priv;

    int eventcnt;

    unsigned int revents;

    struct epitem \*epi;

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

    /\*

     \* We can loop without lock because we are passed a task private list.

     \* Items cannot vanish during the loop because ep\_scan\_ready\_list() is

     \* holding "mtx" during this call.

     \*/

    /\* 扫描整个链表... \*/

    for (eventcnt = 0, uevent = esed->events;

         !list\_empty(head) && eventcnt < esed->maxevents;) {

        /\* 取出第一个成员 \*/

        epi = list\_first\_entry(head, struct epitem, rdllink);

        /\* 然后从链表里面移除 \*/

        list\_del\_init(&epi->rdllink);

        /\* 读取events,

         \* 注意events我们ep\_poll\_callback()里面已经取过一次了, 为啥还要再取?

         \* 1. 我们当然希望能拿到此刻的最新数据, events是会变的~

         \* 2. 不是所有的poll实现, 都通过等待队列传递了events, 有可能某些驱动压根没传

         \* 必须主动去读取. \*/

        revents = epi->ffd.file->f\_op->poll(epi->ffd.file, NULL) &

            epi->event.events;

        /\*

         \* If the event mask intersect the caller-requested one,

         \* deliver the event to userspace. Again, ep\_scan\_ready\_list()

         \* is holding "mtx", so no operations coming from userspace

         \* can change the item.

         \*/

        if (revents) {

            /\* 将当前的事件和用户传入的数据都copy给用户空间,

             \* 就是epoll\_wait()后应用程序能读到的那一堆数据. \*/

            if (\_\_put\_user(revents, &uevent->events) ||

                \_\_put\_user(epi->event.data, &uevent->data)) {

                /\* 如果copy过程中发生错误, 会中断链表的扫描,

                 \* 并把当前发生错误的epitem重新插入到ready list.

                 \* 剩下的没处理的epitem也不会丢弃, 在ep\_scan\_ready\_list()

                 \* 中它们也会被重新插入到ready list \*/

                list\_add(&epi->rdllink, head);

                return eventcnt ? eventcnt : -EFAULT;

            }

            eventcnt++;

            uevent++;

            if (epi->event.events & EPOLLONESHOT)

                epi->event.events &= EP\_PRIVATE\_BITS;

            else if (!(epi->event.events & EPOLLET)) {

                /\*

                 \* If this file has been added with Level

                 \* Trigger mode, we need to insert back inside

                 \* the ready list, so that the next call to

                 \* epoll\_wait() will check again the events

                 \* availability. At this point, noone can insert

                 \* into ep->rdllist besides us. The epoll\_ctl()

                 \* callers are locked out by

                 \* ep\_scan\_ready\_list() holding "mtx" and the

                 \* poll callback will queue them in ep->ovflist.

                 \*/

                /\* 嘿嘿, EPOLLET和非ET的区别就在这一步之差呀~

                 \* 如果是ET, epitem是不会再进入到readly list,

                 \* 除非fd再次发生了状态改变, ep\_poll\_callback被调用.

                 \* 如果是非ET, 不管你还有没有有效的事件或者数据,

                 \* 都会被重新插入到ready list, 再下一次epoll\_wait

                 \* 时, 会立即返回, 并通知给用户空间. 当然如果这个

                 \* 被监听的fds确实没事件也没数据了, epoll\_wait会返回一个0,

                 \* 空转一次.

                 \*/

                list\_add\_tail(&epi->rdllink, &ep->rdllist);

            }

        }

    }

    return eventcnt;

}

/\* ep\_free在epollfd被close时调用,

 \* 释放一些资源而已, 比较简单 \*/

static void ep\_free(struct eventpoll \*ep)

{

    struct rb\_node \*rbp;

    struct epitem \*epi;

    /\* We need to release all tasks waiting for these file \*/

    if (waitqueue\_active(&ep->poll\_wait))

        ep\_poll\_safewake(&ep->poll\_wait);

    /\*

     \* We need to lock this because we could be hit by

     \* eventpoll\_release\_file() while we're freeing the "struct eventpoll".

     \* We do not need to hold "ep->mtx" here because the epoll file

     \* is on the way to be removed and no one has references to it

     \* anymore. The only hit might come from eventpoll\_release\_file() but

     \* holding "epmutex" is sufficent here.

     \*/

    mutex\_lock(&epmutex);

    /\*

     \* Walks through the whole tree by unregistering poll callbacks.

     \*/

    for (rbp = rb\_first(&ep->rbr); rbp; rbp = rb\_next(rbp)) {

        epi = rb\_entry(rbp, struct epitem, rbn);

        ep\_unregister\_pollwait(ep, epi);

    }

    /\*

     \* Walks through the whole tree by freeing each "struct epitem". At this

     \* point we are sure no poll callbacks will be lingering around, and also by

     \* holding "epmutex" we can be sure that no file cleanup code will hit

     \* us during this operation. So we can avoid the lock on "ep->lock".

     \*/

    /\* 之所以在关闭epollfd之前不需要调用epoll\_ctl移除已经添加的fd,

     \* 是因为这里已经做了... \*/

    while ((rbp = rb\_first(&ep->rbr)) != NULL) {

        epi = rb\_entry(rbp, struct epitem, rbn);

        ep\_remove(ep, epi);

    }

    mutex\_unlock(&epmutex);

    mutex\_destroy(&ep->mtx);

    free\_uid(ep->user);

    kfree(ep);

}

/\* File callbacks that implement the eventpoll file behaviour \*/

static const struct file\_operations eventpoll\_fops = {

    .release    = ep\_eventpoll\_release,

    .poll       = ep\_eventpoll\_poll

};

/\* Fast test to see if the file is an evenpoll file \*/

static inline int is\_file\_epoll(struct file \*f)

{

    return f->f\_op == &eventpoll\_fops;

}

/\* OK, eventpoll我认为比较重要的函数都注释完了... \*/