<https://www.cnblogs.com/sduzh/p/6714281.html>

一、说明

针对的内核版本为4.4.10。

本文只是我自己看源码的简单笔记，如果想了解epoll的实现，强烈推荐下面的文章:

[The Implementation of epoll(1)](https://idndx.com/2014/09/01/the-implementation-of-epoll-1/)

[The Implementation of epoll(2)](https://idndx.com/2014/09/01/the-implementation-of-epoll-2/)

[The Implementation of epoll(3)](https://idndx.com/2014/09/01/the-implementation-of-epoll-3/)

[The Implementation of epoll(4)](https://idndx.com/2014/09/01/the-implementation-of-epoll-4/)

二、epoll\_create()

系统调用epoll\_create()会创建一个epoll实例并返回该实例对应的文件描述符fd。在内核中，每个epoll实例会和一个struct eventpoll类型的对象一一对应，该对象是epoll的核心，其[声明](http://lxr.free-electrons.com/source/fs/eventpoll.c" \l "L185" \t "_blank)在fs/eventpoll.c文件中.

epoll\_create的接口定义在[这里](http://lxr.free-electrons.com/source/fs/eventpoll.c" \l "L1836" \t "_blank)，主要源码分析如下：

首先创建一个struct eventpoll对象：

|  |  |
| --- | --- |
| 1  2  3  4  5 | struct eventpoll \*ep = NULL;  ...  error = ep\_alloc(&ep);  if (error < 0)      return error; |

然后分配一个未使用的文件描述符：

|  |  |
| --- | --- |
| 1  2  3  4  5 | fd = get\_unused\_fd\_flags(O\_RDWR | (flags & O\_CLOEXEC));  if (fd < 0) {      error = fd;      goto out\_free\_ep;  } |

然后创建一个struct file对象，将file中的struct file\_operations \*f\_op设置为全局变量eventpoll\_fops，将void \*private指向刚创建的eventpoll对象ep：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | struct file \*file;  ...  file = anon\_inode\_getfile("[eventpoll]", &eventpoll\_fops, ep, O\_RDWR | (flags & O\_CLOEXEC));  if (IS\_ERR(file)) {      error = PTR\_ERR(file);      goto out\_free\_fd;  } |

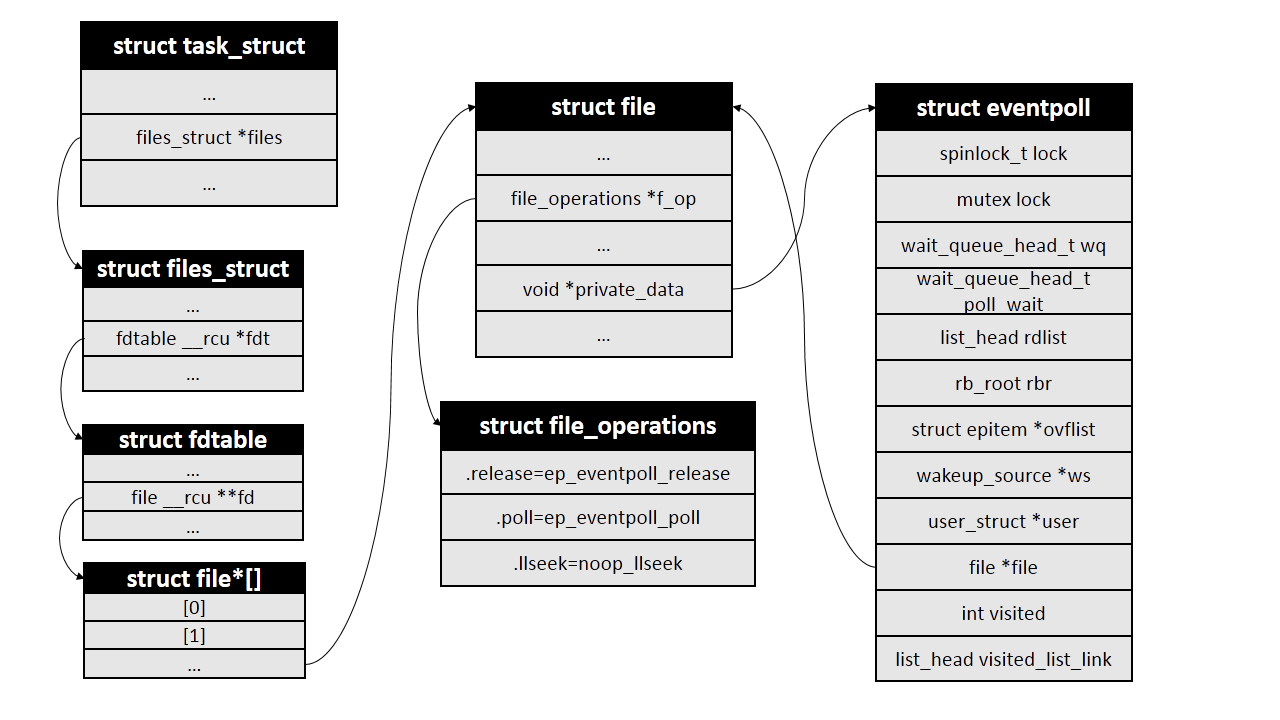
然后设置eventpoll中的file指针：

|  |  |
| --- | --- |
| 1 | ep->file = file; |

最后将文件描述符添加到当前进程的文件描述符表中，并返回给用户

|  |  |
| --- | --- |
| 1 | fd\_install(fd, file);<br>return fd; |

操作结束后主要结构关系如下图：



三、epoll\_ctl()

系统调用epoll\_ctl()在内核中的定义如下，各个参数的含义可参见epoll\_ctl的man手册

|  |  |
| --- | --- |
| 1 | SYSCALL\_DEFINE4(epoll\_ctl, int, epfd, int, op, int, fd, struct epoll\_event \_\_user \*, event) |

epoll\_ctl()首先判断op是不是删除操作，如果不是则将event参数从用户空间拷贝到内核中：

|  |  |
| --- | --- |
| 1  2  3  4  5 | struct epoll\_event epds;  ...  if (ep\_op\_has\_event(op) &&       copy\_from\_user(&epds, event, sizeof(struct epoll\_event)))           goto error\_return; |

ep\_op\_has\_event()实际就是判断op是不是删除操作：

|  |  |
| --- | --- |
| 1  2  3  4 | static inline int ep\_op\_has\_event(int op)  {      return op != EPOLL\_CTL\_DEL;  } |

接下来判断用户是否设置了EPOLLEXCLUSIVE标志，这个标志是4.5版本内核才有的，主要是为了解决同一个文件描述符同时被添加到多个epoll实例中造成的“惊群”问题，详细描述可以看[这里](http://man7.org/linux/man-pages/man2/epoll_ctl.2.html" \t "_blank)。 这个标志的设置有一些限制条件，比如只能是在EPOLL\_CTL\_ADD操作中设置，而且对应的文件描述符本身不能是一个epoll实例，下面代码就是对这些限制的检查：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | /\*   \*epoll adds to the wakeup queue at EPOLL\_CTL\_ADD time only,   \* so EPOLLEXCLUSIVE is not allowed for a EPOLL\_CTL\_MOD operation.   \* Also, we do not currently supported nested exclusive wakeups.   \*/   if (epds.events & EPOLLEXCLUSIVE) {       if (op == EPOLL\_CTL\_MOD)           goto error\_tgt\_fput;       if (op == EPOLL\_CTL\_ADD && (is\_file\_epoll(tf.file) ||              (epds.events & ~EPOLLEXCLUSIVE\_OK\_BITS)))           goto error\_tgt\_fput;  } |

接下来从传入的文件描述符开始，一步步获得struct file对象，再从struct file中的private\_data字段获得struct eventpoll对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | struct fd f, tf;  struct eventpoll \*ep;  ...  f = fdget(epfd);  ...  tf = fdget(fd);  ...  ep = f.file->private\_data; |

如果要添加的文件描述符本身也代表一个epoll实例，那么有可能会造成死循环，内核对此情况做了检查，如果存在死循环则返回错误。这部分的代码目前我还没细看，这里不再贴出。

接下来会从epoll实例的红黑树里寻找和被监控文件对应的epollitem对象，如果不存在，也就是之前没有添加过该文件，返回的会是NULL。

|  |  |
| --- | --- |
| 1  2  3 | struct epitem \*epi;  ...  epi = ep\_find(ep, tf.file, fd); |

ep\_find()函数本质是一个红黑树查找过程，红黑树查找和插入使用的比较函数是ep\_cmp\_ffd()，先比较struct file对象的地址大小，相同的话再比较文件描述符大小。struct file对象地址相同的一种情况是通过dup()系统调用将不同的文件描述符指向同一个struct file对象。

|  |  |
| --- | --- |
| 1  2  3  4  5 | static inline int ep\_cmp\_ffd(struct epoll\_filefd \*p1, <br>                             struct epoll\_filefd \*p2)  {          return (p1->file > p2->file ? +1:                  (p1->file < p2->file ? -1 : p1->fd - p2->fd));  } |

接下来会根据操作符op的不同做不同的处理，这里我们只看op等于EPOLL\_CTL\_ADD时的添加操作。首先会判断上一步操作中返回的epollitem对象地址是否为NULL，不是NULL说明该文件已经添加过了，返回错误，否则调用ep\_insert()函数进行真正的添加操作。在添加文件之前内核会自动为该文件增加POLLERR和POLLHUP事件。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | if (!epi) {      epds.events |= POLLERR | POLLHUP;      error = ep\_insert(ep, &epds, tf.file, fd, full\_check);  } else      error = -EEXIST;  if (full\_check)      clear\_tfile\_check\_list(); |

 ep\_insert()返回之后会判断full\_check标志，该标志和上文提到的死循环检测相关，这里也略去。

四、ep\_insert()

ep\_insert()函数中，首先判断epoll实例中监视的文件数量是否已超过限制，没问题则为待添加的文件创建一个epollitem对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int error, revents, pwake = 0;  unsigned long flags;  long user\_watches;  struct epitem \*epi;  struct ep\_pqueue epq;    user\_watches = atomic\_long\_read(&ep->user->epoll\_watches);  if (unlikely(user\_watches >= max\_user\_watches))          return -ENOSPC;  if (!(epi = kmem\_cache\_alloc(epi\_cache, GFP\_KERNEL)))          return -ENOMEM; |

接下来是对epollitem的初始化：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | INIT\_LIST\_HEAD(&epi->rdllink);  INIT\_LIST\_HEAD(&epi->fllink);  INIT\_LIST\_HEAD(&epi->pwqlist);  epi->ep = ep;  ep\_set\_ffd(&epi->ffd, tfile, fd);  epi->event = \*event;  epi->nwait = 0;  epi->next = EP\_UNACTIVE\_PTR;  if (epi->event.events & EPOLLWAKEUP) {          error = ep\_create\_wakeup\_source(epi);          if (error)                  goto error\_create\_wakeup\_source;  } else {          RCU\_INIT\_POINTER(epi->ws, NULL);  } |

接下来是比较重要的操作：将epollitem对象添加到被监视文件的等待队列上去。等待队列实际上就是一个回调函数链表，定义在[/include/linux/wait.h](http://lxr.free-electrons.com/source/include/linux/wait.h" \t "_blank)文件中。因为不同文件系统的实现不同，无法直接通过struct file对象获取等待队列，因此这里通过struct file的poll操作，以回调的方式返回对象的等待队列，这里设置的回调函数是ep\_ptable\_queue\_proc:

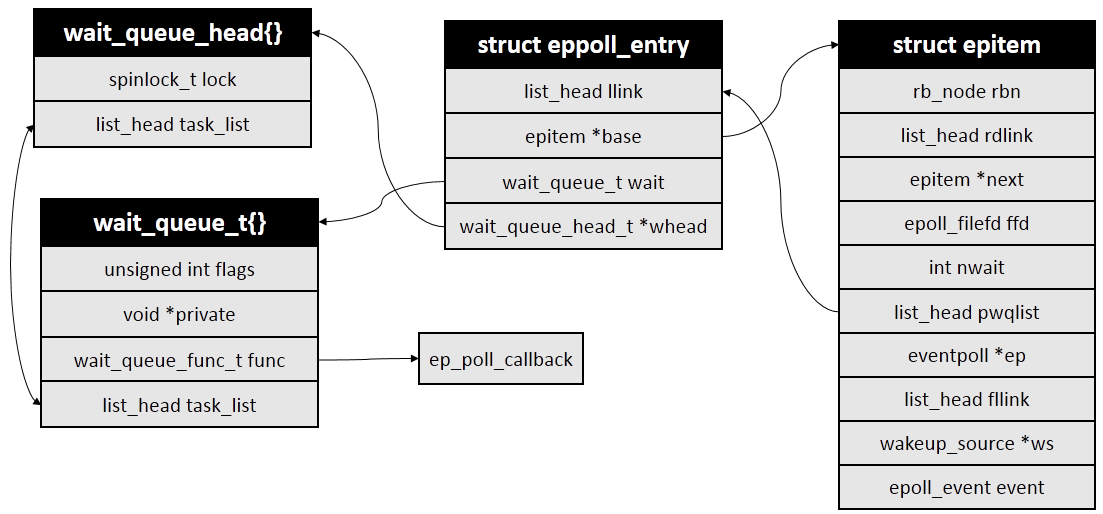
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | struct ep\_pqueue epq;  ...  /\* Initialize the poll table using the queue callback \*/  epq.epi = epi;  init\_poll\_funcptr(&epq.pt, ep\_ptable\_queue\_proc);    /\*   \* Attach the item to the poll hooks and get current event bits.   \* 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.   \*/  revents = ep\_item\_poll(epi, &epq.pt); |

上面代码中结构体ep\_queue的作用是能够在poll的回调函数中取得对应的epollitem对象，这种做法在Linux内核里非常常见。

在回调函数ep\_ptable\_queue\_proc中，内核会创建一个struct eppoll\_entry对象，然后将等待队列中的回调函数设置为ep\_poll\_callback()。也就是说，当被监控文件有事件到来时，比如socker收到数据时，ep\_poll\_callback()会被回调。ep\_ptable\_queue\_proc()代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 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))) {                  init\_waitqueue\_func\_entry(&pwq->wait, ep\_poll\_callback);                  pwq->whead = whead;                  pwq->base = epi;                  if (epi->event.events & EPOLLEXCLUSIVE)                          add\_wait\_queue\_exclusive(whead, &pwq->wait);                  else                          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;          }  } |

eppoll\_entry和epitem等结构关系如下图：



在回到ep\_insert()函数中。ep\_item\_poll()调用完成之后，会将epitem中的fllink字段添加到struct file中的f\_ep\_links链表中，这样就可以通过struct file找到所有对应的struct epollitem对象，进而通过struct epollitem找到所有的epoll实例对应的struct eventpoll。

|  |  |
| --- | --- |
| 1  2  3 | spin\_lock(&tfile->f\_lock);  list\_add\_tail\_rcu(&epi->fllink, &tfile->f\_ep\_links);  spin\_unlock(&tfile->f\_lock); |

然后就是将epollitem插入到红黑树中：

|  |  |
| --- | --- |
| 1 | ep\_rbtree\_insert(ep, epi) |

最后再更新下状态就返回了，插入操作也就完成了。

在返回之前还会判断一次刚才添加的文件是不是当前已经有事件就绪了，如果是就将其加入到epoll的就绪链表中，关于就绪链表放到下一部分中讲，这里略过。

最后是我画的几个结构体之间的结构图。

