# Linux epoll模型详解及源码分析

原创 KiteRunner24 最后发布于2018-06-03 15:50:50 阅读数 12537 ☆ 收藏

# 一、epoll简介

epoll是当前在Linux下开发大规模并发网络程序的热门选择，epoll在Linux2.6内核中正式引入，和select相似，都是**IO多路复用（IO multiplexing）**

按照man手册的说法，**epoll是为处理大批量句柄而做了改进的poll。**

Linux下有以下几个经典的服务器模型：

## 1、PPC模型和TPC模型

PPC（Process Per Connection）模型和TPC（Thread Per Connection）模型的设计思想类似，就是给每一个到来的连接都分配一个独立的进程或者
对于这两种模型，其需要耗费较大的时间和空间资源。当管理连接数较多时，进程或线程的切换开销较大。因此，这类模型能接受的最大连接数都不会
几百个左右。

## 2、select模型

对于select模型，其主要有以下几个特点：

- **最大并发数限制**：由于一个进程所打开的fd（文件描述符）是有限制的，由FD_SETSIZE设置，默认值是1024/2048，因此，select模型的最大并发
了。
- **效率问题**：每次进行select调用都会线性扫描全部的fd集合。这样，效率就会呈现线性下降。
- **内核/用户空间内存拷贝问题**：select在解决将fd消息传递给用户空间时采用了**内存拷贝**的方式。这样，其处理效率不高。

## 3、poll模型

对于poll模型，其虽然解决了select最大并发数的限制，但依然没有解决掉select的效率问题和内存拷贝问题。

## 4、epoll模型

对比于其他模型，epoll做了如下改进：

### 支持一个进程打开较大数目的文件描述符（fd）

select模型对一个进程所打开的文件描述符是有一定限制的，其由FD_SETSIZE设置，默认为1024/2048。这对于那些需要支持上万连接数目的高并发服
然太少了，这个时候，可以选择两种方案：一是可以选择修改FD_SETSIZE宏然后重新编译内核，不过这样做也会带来网络效率的下降；二是可以选择多
方案（传统的Apache方案），不过虽然Linux中创建线程的代价比较小，但仍然是不可忽视的，加上进程间数据同步远不及线程间同步的高效，所以也
美的方案。

但是，epoll则没有对描述符数目的限制，**它所支持的文件描述符上限是整个系统最大可以打开的文件数目**，例如，在1GB内存的机器上，这个限制大概
右。

### IO效率不会随文件描述符（fd）的增加而线性下降

传统的select/poll的一个致命弱点就是当你拥有一个很大的socket集合时，不过任一时间只有部分socket是活跃的，select/poll每次调用都会线性扫描
集合，这将导致IO处理效率呈现线性下降。

但是，epoll不存在这个问题，它只会对活跃的socket进行操作，这是因为在内核实现中，epoll是根据每个fd上面的callback函数实现的。因此，只有
才会主动去调用callback函数，其他idle状态socket则不会。在这一点上，epoll实现了一个伪AIO，其内部推动力在内核。

在一些benchmark中，如果所有的socket基本上都是活跃的，如高速LAN环境，epoll并不比select/poll效率高，相反，过多使用epoll_ctl，其效率反
降。但是，一旦使用idle connections模拟WAN环境，epoll的效率就远在select/poll之上了。

### 使用mmap加速内核与用户空间的消息传递

无论是select，poll还是epoll，它们都需要内核把fd消息通知给用户空间。因此，如何避免不必要的内存拷贝就很重要了。对于该问题，epoll通过内核
mmap同一块内存来实现。

### 内核微调

这一点其实不算epoll的优点了，而是整个Linux平台的优点，Linux赋予开发者微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk_buff结构，可以在运行期间动态调整这个内存池大小（skb_head_pool）来提高性能，该参数可以通过使用`echo xxxx > /proc/sys/net/core/hot_list_length`来完成。可以尝试使用最新的NAPI网卡驱动架构来处理数据包数量巨大但数据包本身很小的特殊场景。

# 二、epoll API

epoll只有`epoll_create`、`epoll_ctl`和`epoll_wait`这三个系统调用。其定义如下：

```
#include <sys/epoll.h>

int epoll_create(int size);

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

## 1、epoll_create

```
#include <sys/epoll.h>

int epoll_create(int size);
```

可以调用epoll_create方法创建一个epoll的句柄。

需要注意的是，当创建好epoll句柄后，它就会占用一个fd值。在使用完epoll后，必须调用close函数进行关闭，否则可能导致fd被耗尽。

## 2、epoll_ctl

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

epoll的事件注册函数，它不同于select是在监听事件时告诉内核要监听什么类型的事件，而是通过epoll_ctl注册要监听的事件类型。

第一个参数epfd：epoll_create函数的返回值。

第二个参数events：表示动作类型。有三个宏来表示：
* EPOLL_CTL_ADD：注册新的fd到epfd中；
* EPOLL_CTL_MOD：修改已经注册的fd的监听事件；
* EPOLL_CTL_DEL：从epfd中删除一个fd。

第三个参数fd：需要监听的fd。

第四个参数event：告诉内核需要监听什么事件。

struct epoll_event结构如下所示：

```
// 保存触发事件的某个文件描述符相关的数据
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;

// 感兴趣的事件和被触发的事件
struct epoll_event {
    __uint32_t events; // Epoll events
    epoll_data_t data; // User data variable
};
```

如上所示，对于Epoll Events，其可以是以下几个宏的集合：

- EPOLLIN：表示对应的文件描述符可读（包括对端Socket）；
- EPOLLOUT：表示对应的文件描述符可写；

- EPOLLPRI：表示对应的文件描述符有紧急数据可读（带外数据）；
- EPOLLERR：表示对应的文件描述符发生错误；
- EPOLLHUP：表示对应的文件描述符被挂断；
- EPOLLET：将EPOLL设为边缘触发（Edge Triggered），这是相对于水平触发（Level Triggered）而言的。
- EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket，需要再次

### 3、epoll_wait

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

收集在epoll监控的事件中已经发生的事件。参数events是分配好的epoll_event结构体数组，epoll将会把发生的事件赋值到events数组中（events不
针，内核只负责把数据赋值到这个event数组中，不会去帮助我们在用户态分配内存）。maxevents告诉内核这个events数组有多大，这个maxevents
创建epoll_create时的size。参数timeout是超时时间（毫秒）。如果函数调用成功，则返回对应IO上已准备好的文件描述符数目，如果返回0则表示已

## 三、epoll工作模式

### 1. LT模式（Level Triggered，水平触发）

该模式是epoll的缺省工作模式，其同时支持阻塞和非阻塞socket。内核会告诉开发者一个文件描述符是否就绪，如果开发者不采取任何操作，内核仍

### 2. ET模式（Edge Triggered，边缘触发）

该模式是一种高速处理模式，当且仅当状态发生变化时才会获得通知。在该模式下，其假定开发者在接收到一次通知后，会完整地处理该事件，因此内
这一事件。注意，缓冲区中还有未处理的数据不能说是状态变化，因此，在ET模式下，开发者如果只读取了一部分数据，其将再也得不到通知了。正确
开发者自己确认读完了所有的字节（一直调用read/write直到出错EAGAGIN为止）。

Nginx默认采用的就是ET（边缘触发）。

## 四、epoll高效性探讨

epoll的高效性主要体现在以下三个方面：

（1）select/poll每次调用都要传递所要监控的所有fd给select/poll系统调用，这意味着每次调用select/poll时都要将fd列表从用户空间拷贝到内核，
时，这会造成性能低效。对于epoll_wait，每次调用epoll_wait时，其不需要将fd列表传递给内核，**epoll_ctl不需要每次都拷贝所有的fd列表，只需要
作**。因此，**在调用epoll_create函数之后，内核已经在内核开始准备数据结构用于存放需要监控的fd了。其后，每次epoll_ctl只是对这个数据结构进行
操作即可。**

（2）**内核使用slab机制，为epoll提供了快速的数据结构**。在内核里，一切都是文件。因此，**epoll向内核注册了一个文件系统，用于存储所有被监控**
epoll_create时，就会在这个虚拟的epoll文件系统中创建一个file节点。epoll在被内核初始化时，同时会分配出epoll自己的内核告诉cache区，用于有
希望监控的fd。**这些fd会以红黑树的形式保存在内核cache里，以支持快速查找、插入和删除**。这个内核高速cache，就是建立连续的物理内存页，然
slab层，简单的说，就是物理上分配好想要的size的内存对象，每次使用时都使用空闲的已分配好的对象。

（3）当调用epoll_ctl往epfd注册百万个fd时，epoll_wait仍然能够快速返回，并有效地将发生的事件fd返回给用户。原因在于，**当我们调用epoll_cre
除了帮我们在epoll文件系统新建file节点，同时在内核cache创建红黑树用于存储以后由epoll_ctl传入的fd外，还会再建立一个list链表，用于存储准
件。当调用epoll_wait时，仅仅观察这个list链表中有无数据即可。**如果list链表中有数据，则返回这个链表中的所有元素；如果list链表中没有数据，则
到timeout超时返回。所以，epoll_wait非常高效，而且，通常情况下，即使我们需要监控百万计的fd，但大多数情况下，一次也只返回少量准备就绪的
此，每次调用epoll_wait，其仅需要从内核态复制少量的fd到用户空间而已。那么，这个准备就绪的list链表是怎么维护的呢？过程如下：当我们执行e
除了把fd放入到epoll文件系统里file对象对应的红黑树之外，还会给内核中断处理程序注册一个回调函数，其告诉内核，如果这个fd的中断到了，就把
绪的list链表中。

如此，一棵红黑树、一张准备就绪的fd链表以及少量的内核cache，就帮我们解决了高并发下fd的处理问题。

总结一下：

- 执行epoll_create时，创建了红黑树和就绪list链表；
- 执行epoll_ctl时，如果增加fd，则检查在红黑树中是否存在，存在则立即返回，不存在则添加到红黑树中，然后向内核注册回调函数，用于当中断
  准备就绪的list链表中插入数据。
- 执行epoll_wait时立即返回准备就绪链表里的数据即可。

## 五、epoll源码分析

eventpoll_init过程：

```
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");
}
```

其中，epoll用slab分配器kmem_cache_create分配内存用于存放`struct epitem`和`struct eppoll_entry`。

当向系统中添加一个fd时，就会创建一个epitem结构体，这是内核管理epoll的基本数据结构：

```
/*
 * Each file descriptor added to the eventpoll interface will
 * have an entry of this type linked to the hash.
 */
struct epitem {
    /* RB-Tree node used to link this structure to the eventpoll rb-tree */
    struct rb_node rbn;    // 用于主结构管理的红黑树

    /* List header used to link this structure to the eventpoll ready list */
    struct list_head rdllink;  // 事件就绪队列

    /* The file descriptor information this item refers to */
    struct epoll_filefd ffd; // 用于主结构中的链表

    /* Number of active wait queue attached to poll operations */
    int nwait; // 事件个数

    /* List containing poll wait queues */
    struct list_head pwqlist; // 双向链表，保存着被监控文件的等待队列

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

```
    struct eventpoll *ep;  //该项属于哪个主结构体

    /* The structure that describe the interested events and the source fd */
    struct epoll_event event; //注册的感兴趣的时间

    /*
    * Used to keep track of the usage count of the structure. This avoids
    * that the structure will desappear from underneath our processing.
    */
    atomic_t usecnt;

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

    /* List header used to link the item to the transfer list */
    struct list_head txlink;

    /*
    * This is used during the collection/transfer of events to userspace
    * to pin items empty events set.
    */
    unsigned int revents;
};
```

对于每个epfd，其对应的数据结构为：

```
/*
* This structure is stored inside the "private_data" member of the file
* structure and rapresent the main data sructure for the eventpoll
* interface.
*/
struct eventpoll {
    /* Protect the this structure access */
    rwlock_t lock;

    /*
    * This semaphore is used to ensure that files are not removed
    * while epoll is using them. This is read-held during the event
    * collection loop and it is write-held during the file cleanup
    * path, the epoll file exit code and the ctl operations.
    */
    struct rw_semaphore sem;

    /* Wait queue used by sys_epoll_wait() */
    wait_queue_head_t wq;

    /* Wait queue used by file->poll() */
    wait_queue_head_t poll_wait;

    /* List of ready file descriptors */
    struct list_head rdllist;  //准备就绪的事件链表

    /* RB-Tree root used to store monitored fd structs */
    struct rb_root rbr;    //用于管理所有fd的红黑树（根节点）
};
```

eventpoll在epoll_create时创建：

```
/*
* It opens an eventpoll file descriptor by suggesting a storage of "size"
* file descriptors. The size parameter is just an hint about how to size
* data structures. It won't prevent the user to store more than "size"
* file descriptors inside the epoll interface. It is the kernel part of
* the userspace epoll_create(2).
*/
asmlinkage long sys_epoll_create(int size)
{

```

```
    int error, fd;
    struct eventpoll *ep;
    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, and create the internal data
    * structure ( "struct eventpoll" ).
    */
    error = -EINVAL;
    if (size <= 0 || (error = ep_alloc(&ep)) != 0)  // ep_alloc为eventpoll分配内存并初始化
        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, ep); // 创建于eventpoll相关的数据结构，包括file、inode和fd等信息
    if (error)
        goto eexit_2;

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

    return fd;

eexit_2:
    ep_free(ep);
    kfree(ep);
eexit_1:
    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: sys_epoll_create(%d) = %d\n",
             current, size, error));
    return error;
}
```

如上，内核中维护了一棵红黑树，大致结构如下：

下面是epoll_ctl函数过程：

```
/*
* The following function implements the controller interface for
* the eventpoll file that enables the insertion/removal/change of
 * file descriptors inside the interest set.  It represents
* the kernel part of the user space epoll_ctl(2).
*/
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;

    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: sys_epoll_ctl(%d, %d, %d, %p)\n",
             current, epfd, op, fd, event));

    error = -EFAULT;
    if (ep_op_hash_event(op) &&
        copy_from_user(&epds, event, sizeof(struct epoll_event)))
        goto eexit_1;

```

```
    /* Get the "struct file *" for the eventpoll file */
    error = -EBADF;
    file = fget(epfd);  //获取epfd对应的文件
    if (!file)
        goto eexit_1;

    /* Get the "struct file *" for the target file */
    tfile = fget(fd);  //获取fd对应的文件
    if (!tfile)
        goto eexit_2;

    /* The target file descriptor must support poll */
    error = -EPERM;
    if (!tfile->f_op || !tfile->f_op->poll)
        goto eexit_3;

    /*
    * 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;
    if (file == tfile || !is_file_epoll(file))
        goto eexit_3;

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

    down_write(&ep->sem);

    /* Try to lookup the file inside our hash table */
    epi = ep_find(ep, tfile, fd);  //在哈希表中查询，防止重复添加

    error = -EINVAL;
    switch (op) {
    case EPOLL_CTL_ADD:  //添加节点，调用ep_insert函数
        if (!epi) {
            epds.events |= POLLERR | POLLHUP;

            error = ep_insert(ep, &epds, tfile, fd);
        } else
            error = -EEXIST;
        break;
    case EPOLL_CTL_DEL:  //删除节点，调用ep_remove函数
        if (epi)
            error = ep_remove(ep, epi);
        else
            error = -ENOENT;
        break;
    case EPOLL_CTL_MOD:  //修改节点，调用ep_modify函数
        if (epi) {
            epds.events |= POLLERR | POLLHUP;
            error = ep_modify(ep, epi, &epds);
        } else
            error = -ENOENT;
        break;
    }

    /*
    * The function ep_find() increments the usage count of the structure
    * so, if this is not NULL, we need to release it.
    */
    if (epi)
        ep_release_epitem(epi);

    up_write(&ep->sem);
```

```
96
97  eexit_3:
98      fput(tfile);
99  eexit_2:
100     fput(file);
101 eexit_1:
102     DNPRINTK(3, (KERN_INFO "[%p] eventpoll: sys_epoll_ctl(%d, %d, %d, %p) = %d\n",
103              current, epfd, op, fd, event, error));

        return error;
    }
```

对于ep_insert函数，基本代码如下：

```
1   static int ep_insert(struct eventpoll *ep, struct epoll_event *event,
2                struct file *tfile, int fd)
3   {
4       int error, revents, pwake = 0;
5       unsigned long flags;
6       struct epitem *epi;
7       struct ep_pqueue epq;
8
9       error = -ENOMEM;
10      // 分配一个epitem结构体来保存每个加入的fd
11      if (!(epi = kmem_cache_alloc(epi_cache, SLAB_KERNEL)))
12          goto eexit_1;
13
14      /* Item initialization follow here ... */
15      // 初始化结构体
16      ep_rb_initnode(&epi->rbn);
17      INIT_LIST_HEAD(&epi->rdllink);
18      INIT_LIST_HEAD(&epi->fllink);
19      INIT_LIST_HEAD(&epi->txlink);
20      INIT_LIST_HEAD(&epi->pwqlist);
21      epi->ep = ep;
22      ep_set_ffd(&epi->ffd, tfile, fd);
23      epi->event = *event;
24      atomic_set(&epi->usecnt, 1);
25      epi->nwait = 0;
26
27      /* Initialize the poll table using the queue callback */
28      epq.epi = epi;
29      // 安装poll回调函数
30      init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
31
32      /*
33     * Attach the item to the poll hooks and get current event bits.
34     * We can safely use the file* here because its usage count has
35     * been increased by the caller of this function.
36     */
37      // 将当前item添加至poll hook中，然后获取当前event位
38      revents = tfile->f_op->poll(tfile, &epq.pt);
39
40      /*
41     * We have to check if something went wrong during the poll wait queue
42     * install process. Namely an allocation for a wait queue failed due
43     * high memory pressure.
44     */
45      if (epi->nwait < 0)
46          goto eexit_2;
47
48      /* Add the current item to the list of active epoll hook for this file */
49      spin_lock(&tfile->f_ep_lock);
50      list_add_tail(&epi->fllink, &tfile->f_ep_links);
51      spin_unlock(&tfile->f_ep_lock);
52
53      /* We have to drop the new item inside our item list to keep track of it */
54      write_lock_irqsave(&ep->lock, flags);
55
56
```

```
    /* Add the current item to the rb-tree */
    ep_rbtree_insert(ep, epi);

    /* If the file is already "ready" we drop it inside the ready list */
    if ((revents & event->events) && !ep_is_linked(&epi->rdllink)) {
        list_add_tail(&epi->rdllink, &ep->rdllist);

        /* Notify waiting tasks that events are available */
        if (waitqueue_active(&ep->wq))
            wake_up(&ep->wq);
        if (waitqueue_active(&ep->poll_wait))
            pwake++;
    }

    write_unlock_irqrestore(&ep->lock, flags);

    /* We have to call this outside the lock */
    if (pwake)
        ep_poll_safewake(&psw, &ep->poll_wait);

    DNPRINTK(3, (KERN_INFO "[%p] eventpoll: ep_insert(%p, %p, %d)\n",
             current, ep, tfile, fd));

    return 0;

eexit_2:
    ep_unregister_pollwait(ep, epi);

    /*
   * We need to do this because an event could have been arrived on some
   * allocated wait queue.
   */
    write_lock_irqsave(&ep->lock, flags);
    if (ep_is_linked(&epi->rdllink))
        ep_list_del(&epi->rdllink);
    write_unlock_irqrestore(&ep->lock, flags);

    kmem_cache_free(epi_cache, epi);
eexit_1:
    return error;
}
```

其中，`init_poll_funcptr`和`tfile->f_op->poll`将ep_ptable_queue_proc注册到epq.pt中的qproc中。

ep_ptable_queue_proc函数设置了等待队列的ep_poll_callback回调函数。在设备硬件数据到来时，硬件中断函数唤醒该等待队列上等待的进程时，会
数ep_poll_callback。

ep_poll_callback函数主要的功能是将被监视文件的等待事件就绪时，将文件对应的epitem实例添加到就绪队列中，当用户调用epoll_wait时，内核会
的事件报告给用户。

epoll_wait的实现如下：

```
/*
 * Implement the event wait interface for the eventpoll file. It is the kernel
 * part of the user space epoll_wait(2).
 */
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 || maxevents > MAX_EVENTS) //检查maxevents参数
        return -EINVAL;
```

```
    /* Verify that the area passed by the user is writeable */
    // 检查用户空间传入的events指向的内存是否可写
    if (!access_ok(VERIFY_WRITE, events, maxevents * sizeof(struct epoll_event))) {
        error = -EFAULT;
        goto eexit_1;
    }

    /* Get the "struct file *" for the eventpoll file */
    error = -EBADF;
    file = fget(epfd); // 获取epfd对应的eventpoll文件的file实例，file结构是在epoll_create中创建的
    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 ... */
    // 核心处理函数
    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;
}
```

其中，调用ep_poll函数，具体流程如下：

```
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 < 0 || timeout >= EP_MAX_MSTIMEO) ?
        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);
    }

    /* Is it worth to try to dig for events ? */
    eavail = !list_empty(&ep->rdllist);

    write_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.
    */
    if (!res && eavail &&
        !(res = ep_events_transfer(ep, events, maxevents)) && jtimeout)
        goto retry;

    return res;
}
```

ep_send_events函数用于向用户空间发送就绪事件。ep_send_events函数将用户传入的内存简单封装到ep_send_events_data结构中，然后调用ep_scan_ready_list将就绪队列中的事件传入用户空间的内存。

# 六、参考

Epoll详解及源码分析——CSDN博客




KiteRunner24
发布了32 篇原创文章 · 获赞 48 · 访问量 10万+
私信