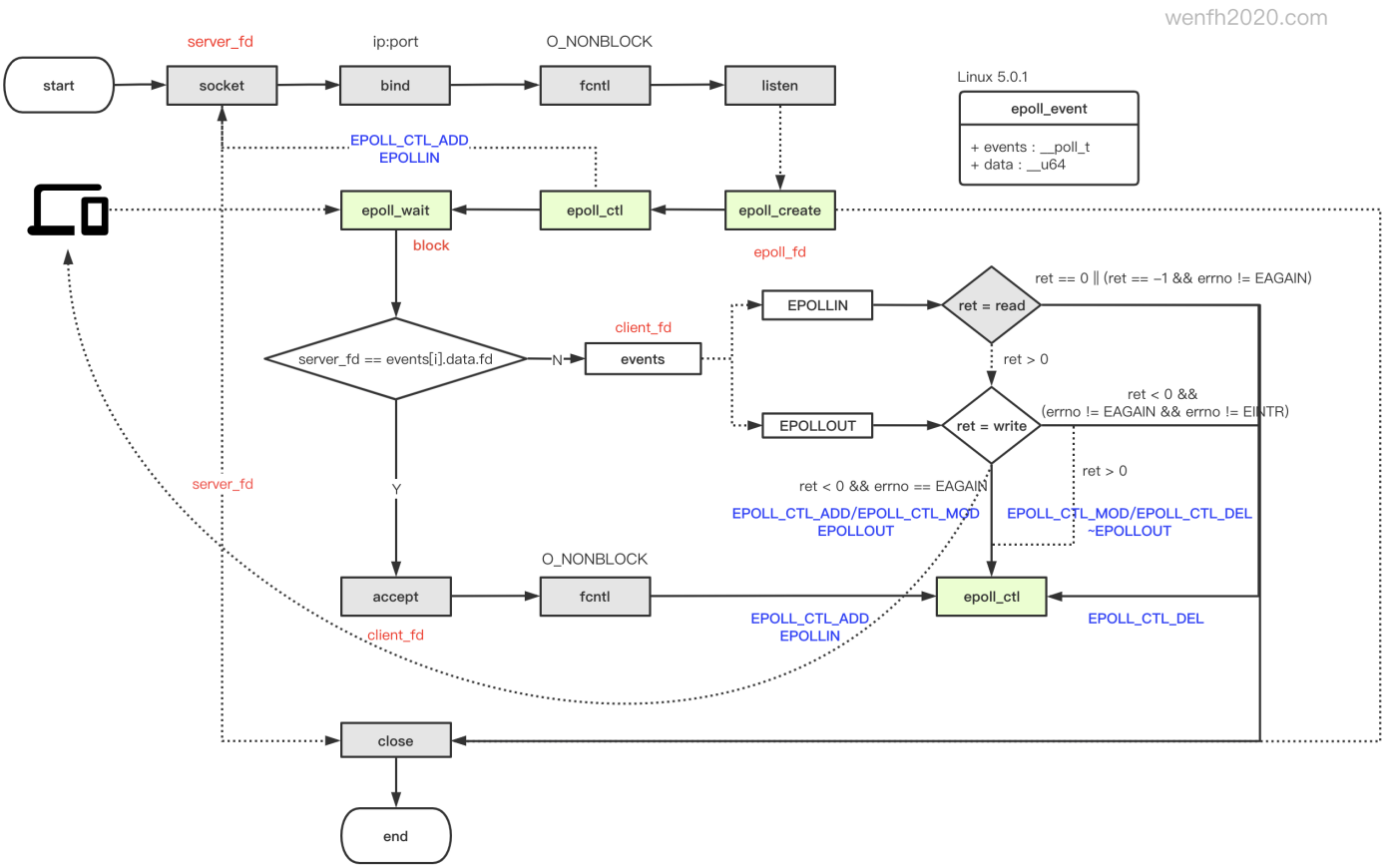
epoll小组汇报

Epoll

epoll如何使用

epoll的使用和之前select和poll的使用流程是有一些不同的。前面的select和poll都是要遍历所有的文件描述符来判断是否需要有就绪的数据。



ET和LT区别

这两种模式的区别在于：

- 对于水平触发模式，一个事件只要有，就会一直触发；
- 对于边缘触发模式，只有一个事件从无到有才会触发。

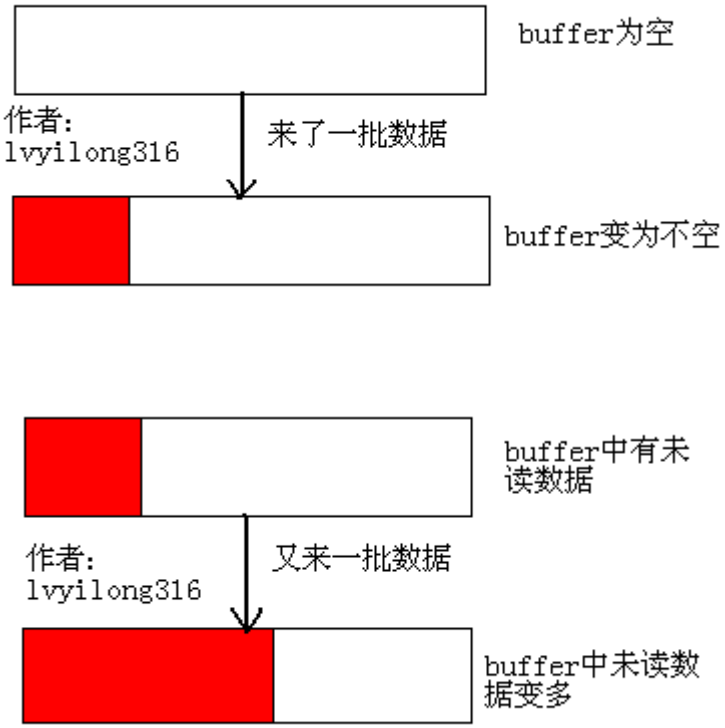
ET模式下被唤醒（返回就绪）的条件为：

1. 对于读取操作：
    a. **当buffer由不可读状态变为可读的时候，即由空变为不空的时候**
    b. **当有新数据到达时，即buffer中的待读内容变多的时候。**

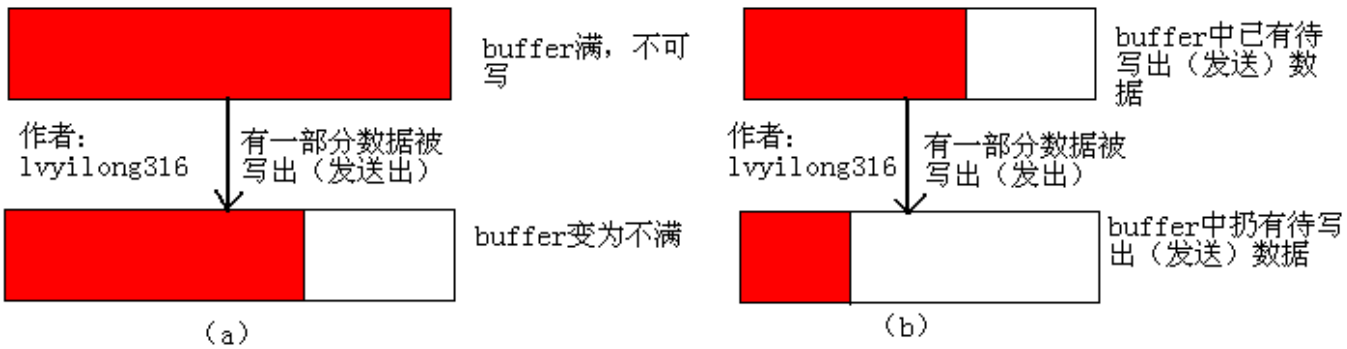
2. 对于写操作：

   a. **当buffer由不可写变为可写的时候，即由满状态变为不满状态的时候。**

   b. **当有旧数据被发送走时，即buffer中待写的内容变少得时候。**



ET读触发的两种情况

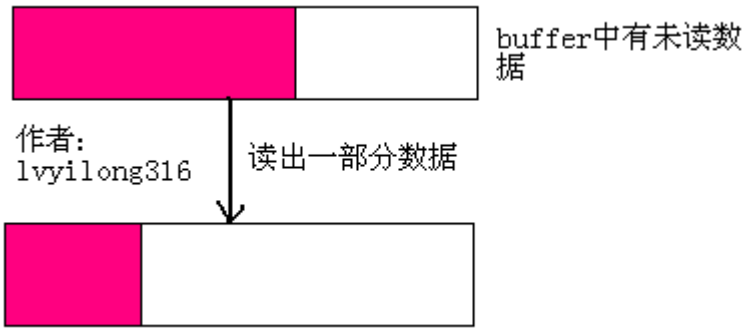


ET写触发的两种情况

LT模式下进程被唤醒（描述符就绪）的条件就简单多了，它包含ET模式的所有条件，也就是上述列出的四种读写被唤醒的条件都是用于LT模式。此外，还有更普通的情况LT可以被唤醒，而ET则不理会，这也是我们需要注意的情况。

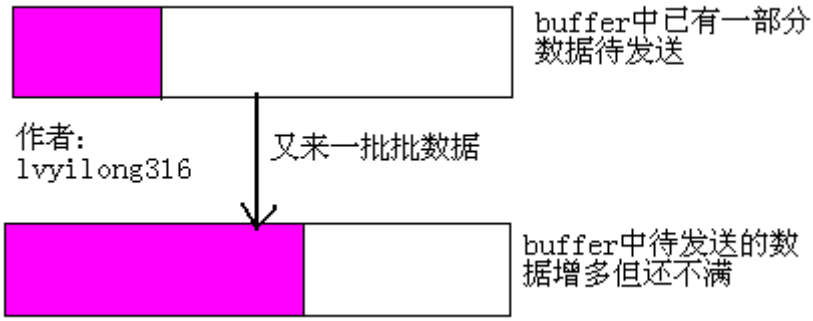
1. 对于读操作

当buffer中有数据，且数据被读出一部分后buffer还不空的时候，即buffer中的内容减少的时候，LT模式返回读就绪。如下图所示。



2. 对于写操作

当buffer不满，又写了一部分数据后仍然不满的的时候，即由于写操作的速度大于发送速度造成buffer中的内容增多的时候，LT模式会返回就绪。如下图所示。




```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <pthread.h>
#include <stdbool.h>

#define MAX_EVENT_NUMBER 1024
#define BUFFER_SIZE 10

// 将文件描述符改为非阻塞的
int set_nonblocking(int fd)
{
```

```
22      int old_option = fcntl(fd, F_GETFL);
23      int new_option = old_option | O_NONBLOCK;
24      fcntl(fd, F_SETFL, new_option);
25      return old_option;
26  }
27
28  // 将文件描述符fd上的EPOLLIN注册到epollfd指向的epoll内核事件表，参数enable_et表示是否
    启动ET模式
29  void addfd(int epollfd, int fd, bool enable_et)
30  {
31      // 封装epoll_event结构体，主要是这个文件描述符有是读还是写或者异常
32      struct epoll_event event;
33      event.data.fd = fd;
34      event.events = EPOLLIN;
35      if (enable_et)
36      {
37          event.events |= EPOLLET;
38          printf("已经改变\n");
39      }
40      // 调用这个注册函数，将文件描述符添加进去
41      epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &event);
42      // set_nonblocking(fd);
43  }
44
45  // LT模式的工作流程
46  void lt(struct epoll_event *events, int number, int epollfd, int listenfd)
47  {
48      char buf[BUFFER_SIZE];
49      for (size_t i = 0; i < number; i++)
50      {
51          int sockfd = events[i].data.fd;
52          if (sockfd == listenfd) // 说明是有新的链接到达
53          {
54              struct sockaddr_in client_addr;
55              socklen_t client_addr_len = sizeof(client_addr);
56              int connfd = accept(listenfd, (struct sockaddr *)&client_addr,
    &client_addr_len);
57              addfd(epollfd, connfd, false); // 将这个用来链接的fd也放到epoll的监管中
58          }
59          else if (events[i].events & EPOLLIN) // 有新的数据到达
60          {
61              // 这是lt模式，只要socket中还有未读出的数据，这段代码就一直触发
62              printf("event triggerr once \n");
63              memset(buf, '\0', BUFFER_SIZE);
64              int ret = recv(sockfd, buf, BUFFER_SIZE - 1, 0);
65              if (ret <= 0)
66              {
```

```
67                  close(sockfd);
68                  continue;
69              }
70              printf("get %d byte of content: %s \n", ret, buf);
71          }
72          else
73          {
74          }
75      }
76  }
77
78  // et模式
79  void et(struct epoll_event *events, int number, int epollfd, int listenfd)
80  {
81      char buf[BUFFER_SIZE];
82      for (size_t i = 0; i < number; i++)
83      {
84          int sockfd = events[i].data.fd;
85          // 如果当前是listenfd，说明有新的连接到来
86          if (sockfd == listenfd)
87          {
88              struct sockaddr_in client_addr;
89              socklen_t client_addr_len = sizeof(client_addr);
90              // 调用accept进行连接，
91              int connfd = accept(listenfd, (struct sockaddr *)&client_addr,
    &client_addr_len);
92              // 将这个fd交由epoll进行管理
93              addfd(epollfd, connfd, true);
94          }
95          else if (events[i].events & EPOLLIN) // 读事件
96          {
97              // 因为是et模式，所以下面这段代码之触发一次，所以我们应该循环读取，以确保数
    据可以完全读取
98              // 同时应该注意在临界情况下，就是没有数据的时候，如果我们使用的是阻塞IO，那
    么在recv的时候就会一直卡在这里，
99              // 导致别的描述符无法读取，只能对这一个连接进行读取
100             printf("event trigger once\n");
101             while (1)
102             {
103
104                 memset(buf, '\0', BUFFER_SIZE);
105
106                 int ret = recv(sockfd, buf, BUFFER_SIZE - 1, 0);
107                 printf("%d\n", ret);
108                 if (ret < 0) // 出错，一般是没有数据读了
109                 {
```

```
110                     // 对于非阻塞来说，下面的条件成立表示数据已经全部读取完毕。如果之后
    这个描述符再次被触发，就说明有新的数据到达
111                     if ((errno == EAGAIN) || (errno == EWOULDBLOCK))
112                     {
113                         printf("这次发送的数据读取完成\n");
114                         break; // 说明这次到达的数据已经全部读到，直接推出这次循环
115                     }
116                     close(sockfd); // 位置错误，直接关闭该描述符
117                     break;
118                 }
119                 else if (ret == 0) // 链接关闭
120                 {
121                     printf("128\n");
122                     close(sockfd);
123                     break;
124                 }
125                 else
126                 {
127                     printf("get %d bytes of content : %s\n", ret, buf);
128                 }
129             }
130         }
131         else
132         {
133         }
134     }
135 }
136 
137 int main(int argc, char *argv[])
138 {
139     if (argc <= 2)
140     {
141         printf("err150");
142         return 1;
143     }
144     // 这些是一个基本的处理
145     const char *ip = argv[1];
146     int port = atoi(argv[2]);
147     int ret = 0;
148     struct sockaddr_in addr;
149     bzero(&addr, sizeof(addr));
150     addr.sin_family = AF_INET;
151     addr.sin_port = htons(port);
152     inet_pton(AF_INET, ip, &addr.sin_addr);
153 
154     int listenfd = socket(PF_INET, SOCK_STREAM, 0);
155     assert(listenfd != -1);
```

```
156     ret = bind(listenfd, (struct sockaddr *)&addr, sizeof(addr));
157     assert(ret != -1);
158     ret = listen(listenfd, 5);
159     assert(ret != -1);
160     // 这里我们就创建出来了listenfd，用来监听的文件描述符
161     // 在用户态空间分配一个数组，这个数组是epoll_wait的参数，传址调用，返回后就是我们到达的事件
162     struct epoll_event events[MAX_EVENT_NUMBER];
163     // 我们首先创建一个epoll描述符，返回值也是一个文件描述符，它使用的是匿名inode，这里就不细说了。
164     int epollfd = epoll_create(5); // 参数没有意义了，只要>=0就可以
165     assert(epollfd != -1);
166     // 接下来我们需要注册监听，就是把这个listenfd的文件描述符注册到红黑树中
167     addfd(epollfd, listenfd, true);
168     // 接下来就是处理一个循环
169     while (1)
170     {
171         // 就是在数据准备阶段生效，如果有数据就会通过events这个数组返回
172         // 四个参数分别是epoll_fd，上述的用户态数组，数组的大小，和一个定时器
173         // 返回值就是有几个io事件
174         int ret = epoll_wait(epollfd, events, MAX_EVENT_NUMBER, -1); // 为什么要加一个超时参数，不仅可以io事件，还可以处理定时器事件
175         if (ret < 0)
176         {
177             printf("err 180\n");
178             break;
179         }
180         // 进行处理
181         //  lt(events,ret,epollfd,listenfd);
182 
183         et(events, ret, epollfd, listenfd);
184     }
185     close(listenfd);
186     return 0;
187 }
```

上面就是分别使用ET和LT来从fd中读取文件描述符中的数据。

其中我们要注意的就是使用ET触发的时候需要非阻塞io。因为ET模式下的读写需要一直读或写直到出错（对于读，当读到的实际字节数小于请求字节数时就可以停止），而如果你的文件描述符如果不是非阻塞的，那这个一直读或一直写势必会在最后一次阻塞。这样就不能在阻塞在epoll_wait上了，造成其他文件描述符的任务饿死。

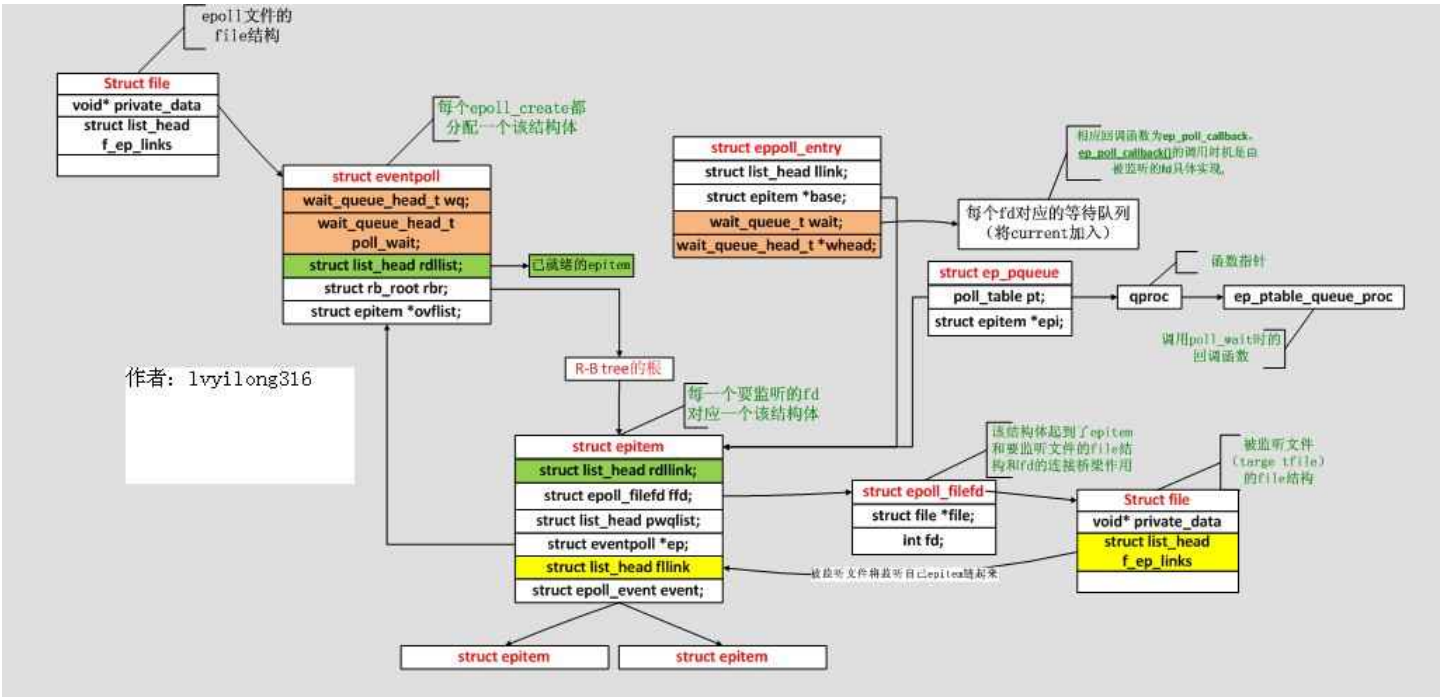
> 这个道理有点像排队打饭，一个队列上，有些同学要打包两份饭，如果每次只能打包一份，lt 模式就是，这些同学打包了一份之后，马上重新回去排队，再打一份。et 模式是，直接要求两份饭。效率比较高。但是如果你业务逻辑写的有问题，只打了一份饭，那么就是有问题的。

epoll实现细节

通过上面的使用，我们可以看到整个epoll只有三个系统调用。

接口	描述
epoll_create	创建 epoll。
epoll_ctl	fd 事件注册函数，用户通过这个函数关注 fd 读写事件。
epoll_wait	阻塞等待 fd 事件发生。

所以我们具体的实现主要是这三个系统调用。在此之前，我们先简单的看一下数据结构。



数据结构

```
1  struct eventpoll
2  {
3
4      spinlock_t lock;
5
6      struct mutex mtx;
7
8      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*/
9
10     wait_queue_head_t poll_wait; /* Wait queue used by file->poll() , 这个用于
   epollfd本事被poll的时候*/
11
```

```
12      struct list_head rdllist; /* List of ready file descriptors, 所有已经ready的
    epitem都在这个链表里面*/
13
14      structrb_root rbr; /* RB tree root used to store monitored fd structs, 所有
    要监听的epitem都在这里*/
15
16      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*/
17
18      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*/
19  };
```

关键函数

函数	描述
eventpoll_init	初始化 epoll 模块。eventpoll 作为 Linux 内核的一部分，模块化管理。
do_epoll_create	为 eventpoll 结构分配资源。
do_epoll_ctl	epoll 管理 fd 事件接口。
do_epoll_wait	有条件阻塞等待 fd 事件发生，返回对fd 和对应事件数据。
ep_item_poll	获取 fd 就绪事件，并关联 fd 和事件触发回调函数 ep_poll_callback。
ep_poll_callback	fd 事件回调函数。当底层收到数据，中断调用 fd 关联的 ep_poll_callback 回调函数，如 列，然后唤醒阻塞等待的 epoll_wait 处理。
ep_send_events	遍历就绪列表，拷贝内核空间就绪数据到用户空间。结合 ep_scan_ready_list 和 ep_se
ep_scan_ready_list	遍历就绪列表。当 fd 收到数据，回调 ep_poll_callback，如果事件是用户关注的，那么 历这个就绪列表，将数据从内核空间拷贝到用户空间，或者其它操作。
ep_send_events_proc	内核将就绪列表数据，发送到用户空间。结合 ep_scan_ready_list 使用。LT/ET 模式在
ep_ptable_queue_proc	添加 fd 的等待事件到等待队列，关联 fd 与回调函数 ep_poll_callback。

核心源码

初始化

这个就是在内核加载的时候，添加 epoll 模块到内核，slab 算法为 epoll 分配资源。在初始化的过程中，eventpollfs create两个slub分别是：epitem和eppoll_entry。

```
static int __init eventpoll_init(void)
{
    mutex_init(&epmutex);

    /* 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);

    /* 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);

    return 0;
}
fs_initcall(eventpoll_init);
```

epoll_create

eventpoll是通过epoll_create生成，epoll_create传入一个size参数，size参数只要>0即可，没有任何意义。epoll_create调用函数sys_epoll_create1实现eventpoll的初始化。sys_epoll_create1通过ep_alloc生成一个eventpoll对象，并初始化eventpoll的三个等待队列，wait，poll_wait以及rdlist（ready的fd list）。同时还会初始化被监视fs的rbtree 根节点。

同时，他会申请一个没有使用的文件描述符和一个匿名文件，然后将两者和刚申请的ep绑定在一起。

file->private_data指定为指向前面生成的eventpoll，这样就将eventpoll和文件fd关联。通过fd得到最后返回文件描述符fd。

```c
static int do_epoll_create(int flags) {
    int error, fd;
    struct eventpoll *ep = NULL;
    struct file *file;
    ...
    // 为 eventpoll 结构分配内存，并初始化 eventpoll 成员数据。
    error = ep_alloc(&ep);
    if (error < 0)
        return error;

    // 分配一个空闲的文件描述符。
    fd = get_unused_fd_flags(O_RDWR | (flags & O_CLOEXEC));
    if (fd < 0) {
        error = fd;
        goto out_free_ep;
    }

    // slab 分配一个新的文件结构对象（struct 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;
    }
    ep->file = file;

    // fd 与 file* 结构进行绑定。
    fd_install(fd, file);
    return fd;
    ...
}
```

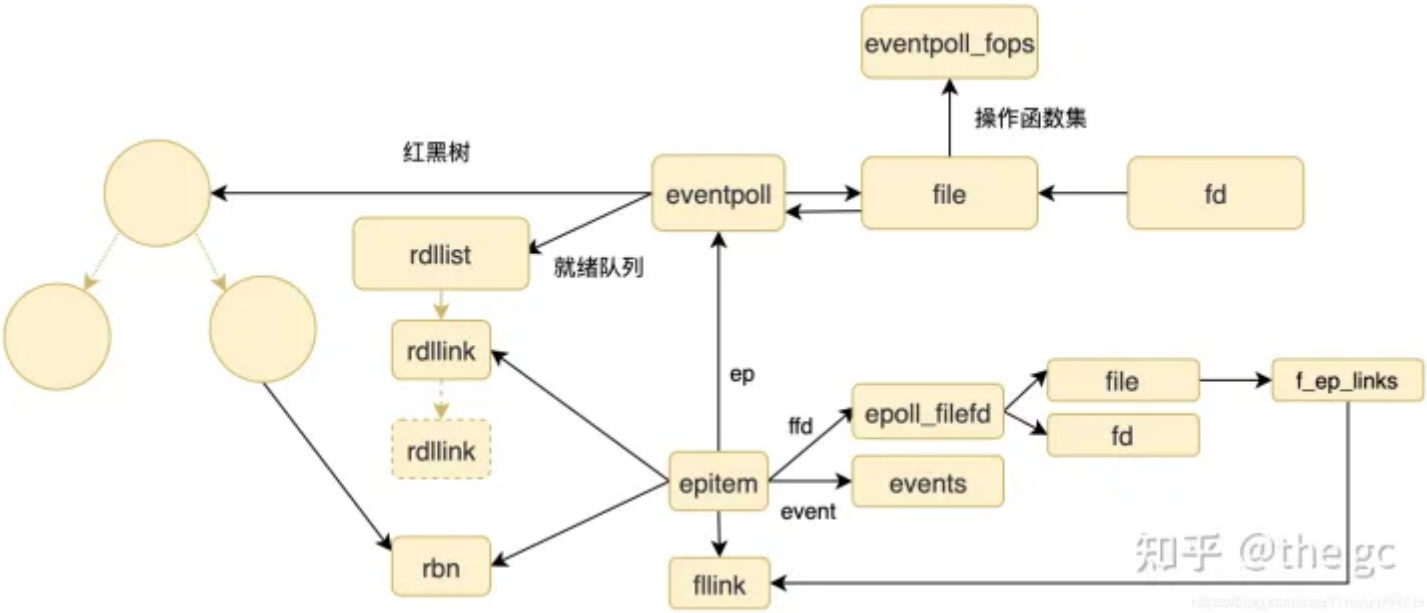
epoll_ctl

这是三个系统调用中最麻烦的一种，涉及的范围很多。

通过epoll_create生成一个eventpoll后，可以通过epoll_ctl提供的相关操作对eventpoll进行ADD，MOD，DEL操作。epoll_ctl有四个参数，分别是：int epfd（需要操作的eventpoll）, int op（操作类型）, int fd（需要被监视的文件）, struct epoll_event *event（被监视文件的相关event）。epoll_ctl首先通过epfd的private_data域获取需要操作的eventpoll，然后通过ep_find确认需要操作的fd是否已经在被监视的红黑树中（eventpoll->rbr）。然后根据op的类型分别作ADD（ep_insert），MOD（ep_modify），DEL（ep_remove）操作。

首先分析ep_insert，ep_insert有四个参数分别为： struct eventpoll *ep（需要操作的eventpoll）, struct epoll_event *event（epoll_create传入的event参数，当然得从user空间拷贝过来）, struct file *tfile（被监视的文件描述符）, int fd（被监视的文件id）。ep_insert首先从slub中分

配一个epitem的对象epi。并初始化epitem的三个list头指针，rdllink（指向eventpoll的rdlist），fllist指向（struct file的f_ep_links），pwqlist（指向包含此epitem的所有poll wait queue）。并将epitem的ep指针，指向传入的eventpoll，并通过传入参数event对ep内部变量event赋值。然后通过ep_set_ffd将目标文件和epitem关联。这样epitem本身就完成了和eventpoll以及被监视文件的关联。



下面还需要做两个动作：将epitem插入目标文件的polllist并注册回调函数；将epitem插入eventpoll的rbtree。

将epi插入红黑树是比较容易理解的，我们主要关注的点是注册回调函数。

- 添加 fd 事件管理流程：fd 关联回调 ep_poll_callback。

> fd -> socket -> poll -> ep_ptable_queue_proc -> wait_queue -> ep_poll_callback

- 触发了 fd 关注的事件回调处理。

> driver -> ep_poll_callback -> waitup -> epoll_wait(wake up)

do_epoll_ctl

```
int do_epoll_ctl(int epfd, int op, int fd, struct epoll_event *epds, bool nonblo
{
    int error;
    int full_check = 0;
    struct fd f, tf;
    struct eventpoll *ep;
    struct epitem *epi;
    struct eventpoll *tep = NULL;
    ...
        // 检查参数合法性。
```

```
11          ...
12          // 在 do_epoll_create 实现里 anon_inode_getfile 将 private_data 与 eventp
13          ep = f.file->private_data;
14      ...
15          // 红黑树检查 fd 是否已经被添加。
16          epi = ep_find(ep, tf.file, fd);
17
18      error = -EINVAL;
19      switch (op)
20      {
21      case EPOLL_CTL_ADD:
22          if (!epi)
23          {
24              /* epoll 如果没有添加过该 fd，就添加到红黑树进行管理。
25               * 事件默认关注异常处理(EPOLLERR | EPOLLHUP)。*/
26              epds->events |= EPOLLERR | EPOLLHUP;
27              error = ep_insert(ep, epds, tf.file, fd, full_check);
28          }
29          else
30              error = -EEXIST;
31          if (full_check)
32              clear_tfile_check_list();
33          break;
34      case EPOLL_CTL_DEL:
35          if (epi)
36              error = ep_remove(ep, epi);
37          else
38              error = -ENOENT;
39          break;
40      case EPOLL_CTL_MOD:
41          if (epi)
42          {
43              if (!(epi->event.events & EPOLLEXCLUSIVE))
44              {
45                  epds->events |= EPOLLERR | EPOLLHUP;
46                  error = ep_modify(ep, epi, epds);
47              }
48          }
49          else
50              error = -ENOENT;
51          break;
52      }
53      ... return error;
54  }
55
```

ep_insert

```
1  static int ep_insert(struct eventpoll *ep, const struct epoll_event *event,
2                       struct file *tfile, int fd, int full_check)
3  {
4      // epoll 管理 fd 和对应事件节点 epitem 数据结构。
5      struct epitem *epi;
6      struct ep_pqueue epq;
7      ... epq.epi = epi;
8
9      // 初始化就绪事件处理函数调用。poll() 接口调用 ep_ptable_queue_proc。
10     init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
11
12     // 添加等待队列，如果 fd 有用户关注的事件发生，返回对应 fd 关注的事件 revents。
13     revents = ep_item_poll(epi, &epq.pt, 1);
14     ...
15         // 将当前节点，添加到 epoll 文件钩子，将 epoll 文件与 fd 对应文件串联起来。
16         list_add_tail_rcu(&epi->fllink, &tfile->f_ep_links);
17
18     // 将节点添加进二叉树
19     ep_rbtree_insert(ep, epi);
20
21     // 如果有关注的事件发生，将节点关联到就绪事件列表。
22     if (revents && !ep_is_linked(epi))
23     {
24         list_add_tail(&epi->rdllink, &ep->rdllist);
25         ep_pm_stay_awake(epi);
26
27         /* 如果进程正在睡眠等待，唤醒它去处理就绪事件。睡眠事件 ep->wq 在 epoll_wait 中
28         if (waitqueue_active(&ep->wq))
29             // 唤醒进程
30             wake_up(&ep->wq);
31
32         // 如果监控的是另外一个 epoll_create 的 fd，有就绪事件，也唤醒进程。
33         if (waitqueue_active(&ep->poll_wait))
34             pwake++;
35     }
36     ... if (pwake)
37         ep_poll_safewake(&ep->poll_wait);
38
39     return 0;
40 }
```

ep_item_poll

fd 节点就绪事件处理。

```
static __poll_t ep_item_poll(const struct epitem *epi, poll_table *pt, int depth
{
    struct eventpoll *ep;
    bool locked;

    pt->_key = epi->event.events;
    if (!is_file_epoll(epi->ffd.file))
    {
        // 非 epoll fd，tcp_poll 检查 socket 就绪事件，fd 关联回调函数 ep_poll_calll
        return vfs_poll(epi->ffd.file, pt) & epi->event.events;
    }
    else
    {
        // epoll 嵌套。epoll_ctl 添加关注了另外一个 epoll 的 fd(epfd)。
        ep = epi->ffd.file->private_data;
        poll_wait(epi->ffd.file, &ep->poll_wait, pt);
        locked = pt && (pt->_qproc == ep_ptable_queue_proc);

        return ep_scan_ready_list(epi->ffd.file->private_data,
                                  ep_read_events_proc, &depth, depth, locked) &
               epi->event.events;
    }
}

// vfs - Virtual Filesystem Switch（Linux 虚拟文件系统）
// poll.h 就绪事件处理函数。
static inline __poll_t vfs_poll(struct file *file, struct poll_table_struct *pt)
{
    if (unlikely(!file->f_op->poll))
        return DEFAULT_POLLMASK;
    // 这里的 poll 函数指针指向 tcp_poll 函数。
    return file->f_op->poll(file, pt);
}

// tcp.c
// tcp 就绪事件获取函数。
__poll_t tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
{
    __poll_t mask;
    struct sock *sk = sock->sk;
    const struct tcp_sock *tp = tcp_sk(sk);
    int state;

    /* 添加等待队列和关联事件回调函数 ep_poll_callback
```

```
     * (只有 epoll_ctl EPOLL_CTL_ADD 的情况下，才会添加等待事件，否则 wait == NULL) *
    sock_poll_wait(file, sock, wait);

    // 检查 fd 是否有事件发生。
    state = inet_sk_state_load(sk);
    if (state == TCP_LISTEN)
        return inet_csk_listen_poll(sk);
    ...
}

// socket.h
static inline void sock_poll_wait(struct file *filp, struct socket *sock, poll_t
{
    // ep_insert 调用 ep_item_poll 才会插入等待事件。
    if (!poll_does_not_wait(p))
    {
        poll_wait(filp, &sock->wq.wait, p);
        ...
    }
}

// poll.h
static inline void poll_wait(struct file *filp, wait_queue_head_t *wait_address,
{
    if (p && p->_qproc && wait_address)
        // _qproc ---> ep_ptable_queue_proc
        p->_qproc(filp, wait_address, p);
}
```

ep_ptable_queue_proc

socket 的等待队列关联回调函数 ep_poll_callback

```
static void ep_ptable_queue_proc(struct file *file, wait_queue_head_t *whead, po
{
    struct epitem *epi = ep_item_from_epqueue(pt);
    struct eppoll_entry *pwq;

    if (epi->nwait >= 0 && (pwq = kmem_cache_alloc(pwq_cache, GFP_KERNEL)))
    {
        // 关联等待队列和ep_poll_callback。
        init_waitqueue_func_entry(&pwq->wait, ep_poll_callback);

        // whead ---> socket->wq.wait
        pwq->whead = whead;
        pwq->base = epi;
```

```
14
15          /* 等待事件，添加到等待队列。EPOLLEXCLUSIVE 为了解决 epoll_wait 惊群问题。
16           * 如果多线程同时调用 epoll_wait，那么 fd 应该设置 EPOLLEXCLUSIVE 事件。 */
17          if (epi->event.events & EPOLLEXCLUSIVE)
18          {
19              add_wait_queue_exclusive(whead, &pwq->wait);
20          }
21          else
22          {
23              add_wait_queue(whead, &pwq->wait);
24          }
25
26          /* 等待事件，关联 epitem。epitem 为什么要有一个等待队列呢，
27           * 因为有可能一个进程里存在多个 epoll 实例同时 epoll_ctl 关注一个 fd。*/
28          list_add_tail(&pwq->llink, &epi->pwqlist);
29          epi->nwait++;
30      }
31      else
32      {
33          /* We have to signal that an error occurred */
34          epi->nwait = -1;
35      }
36  }
```

epoll_wait

```
1  SYSCALL_DEFINE4(epoll_wait, int, epfd, struct epoll_event __user *, events,
2                  int, maxevents, int, timeout)
3  {
4      return do_epoll_wait(epfd, events, maxevents, timeout);
5  }
6
7  static int do_epoll_wait(int epfd, struct epoll_event __user *events,
8                           int maxevents, int timeout)
9  {
10     ...
11         // timeout 阻塞等待处理并返回就绪事件。
12         error = ep_poll(ep, events, maxevents, timeout);
13     ...
14 }
15
16 static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events,
17                    int maxevents, long timeout)
18 {
19     int res = 0, eavail, timed_out = 0;
```

```
20      u64 slack = 0;
21      bool waiter = false;
22      wait_queue_entry_t wait;
23      ktime_t expires, *to = NULL;
24
25      // 计算 timeout 睡眠时间。如果有就绪事件，处理并发送到用户空间。
26      ...
27
28          fetch_events :
29
30          if (!ep_events_available(ep))
31          // napi 中断缓解技术，避免网卡频繁中断 cpu，提高数据获取的效率。这里为了积攒网络
32          ep_busy_loop(ep, timed_out);
33
34      // 检查就绪队列是否有数据。
35      eavail = ep_events_available(ep);
36      if (eavail)
37          // 如果有就绪事件了，就直接不用睡眠等待了，进入发送环节。
38          goto send_events;
39
40      ...
41
42          // 没有就绪事件发生，需要睡眠等待。
43          if (!waiter)
44      {
45          waiter = true;
46          // 等待事件，关联当前进程。
47          init_waitqueue_entry(&wait, current);
48
49          spin_lock_irq(&ep->wq.lock);
50          // 添加等待事件。（为了解决惊群效应，所以等待事件添加了 WQ_FLAG_EXCLUSIVE 标识。
51          __add_wait_queue_exclusive(&ep->wq, &wait);
52          spin_unlock_irq(&ep->wq.lock);
53      }
54
55      for (;;)
56      {
57          /*
58           * We don't want to sleep if the ep_poll_callback() sends us
59           * a wakeup in between. That's why we set the task state
60           * to TASK_INTERRUPTIBLE before doing the checks.
61           */
62
63          // 设置当前进程状态为等待状态，可以被信号解除等待。
64          set_current_state(TASK_INTERRUPTIBLE);
65          /*
66           * Always short-circuit for fatal signals to allow
```

```
67          * threads to make a timely exit without the chance of
68          * finding more events available and fetching
69          * repeatedly.
70          */
71
72         // 信号中断，不要执行睡眠了。
73         if (fatal_signal_pending(current))
74         {
75             res = -EINTR;
76             break;
77         }
78
79         // 检查就绪队列。
80         eavail = ep_events_available(ep);
81         if (eavail)
82             break;
83
84         // 信号中断，不要执行睡眠了。
85         if (signal_pending(current))
86         {
87             res = -EINTR;
88             break;
89         }
90
91         // 进程进入睡眠状态。
92         if (!schedule_hrtimeout_range(to, slack, HRTIMER_MODE_ABS))
93         {
94             timed_out = 1;
95             break;
96         }
97     }
98
99     // 进程等待超时，或者被唤醒，设置进程进入运行状态，等待内核调度运行。
100    __set_current_state(TASK_RUNNING);
101
102 send_events:
103    /*
104     * Try to transfer events to user space. In case we get 0 events and
105     * there's still timeout left over, we go trying again in search of
106     * more luck.
107     */
108
109    // 有就绪事件就发送到用户空间，否则继续获取数据直到超时。
110    if (!res && eavail && !(res = ep_send_events(ep, events, maxevents)) &&
111        !timed_out)
112        goto fetch_events;
113
```

```
114     // 从等待队列中，删除等待事件。
115     if (waiter)
116     {
117         spin_lock_irq(&ep->wq.lock);
118         __remove_wait_queue(&ep->wq, &wait);
119         spin_unlock_irq(&ep->wq.lock);
120     }
121 
122     return res;
123 }
124 
125 /* Used by the ep_send_events() function as callback private data */
126 struct ep_send_events_data
127 {
128     int maxevents;
129     struct epoll_event __user *events;
130     int res;
131 };
132 
133 static int ep_send_events(struct eventpoll *ep,
134                           struct epoll_event __user *events, int maxevents)
135 {
136     struct ep_send_events_data esed;
137 
138     esed.maxevents = maxevents;
139     esed.events = events;
140 
141     // 遍历事件就绪列表，发送就绪事件到用户空间。
142     ep_scan_ready_list(ep, ep_send_events_proc, &esed, 0, false);
143     return esed.res;
144 }
```

ep_scan_ready_list

遍历就绪列表，处理 sproc 函数。这里 sproc 函数指针的使用，是为了减少代码冗余，将 ep_scan_ready_list 做成一个通用的函数。

```
1 //
2 static __poll_t ep_scan_ready_list(struct eventpoll *ep,
3                                     __poll_t (*sproc)(struct eventpoll *,
4                                                       struct list_head *, void *)
5                                     void *priv, int depth, bool ep_locked)
6 {
7     __poll_t res;
8     struct epitem *epi, *nepi;
9     LIST_HEAD(txlist);
```

```
    ...
        // 将就绪队列分片链接到 txlist 链表中。
        list_splice_init(&ep->rdllist, &txlist);
    res = (*sproc)(ep, &txlist, priv);
    ...
        // 在处理 sproc 回调处理过程中，可能产生新的就绪事件被写入 ovflist，将 ovflist
        for (nepi = READ_ONCE(ep->ovflist); (epi = nepi) != NULL;
             nepi = epi->next, epi->next = EP_UNACTIVE_PTR)
    {
        if (!ep_is_linked(epi))
        {
            list_add(&epi->rdllink, &ep->rdllist);
            ep_pm_stay_awake(epi);
        }
    }
    ...
        // txlist 在 epitem 回调中，可能没有完全处理完，那么重新放回到 rdllist，下次处
        list_splice(&txlist, &ep->rdllist);
    ...
}
```

ep_send_events_proc

处理就绪列表，将数据从内核空间拷贝到用户空间。

```
static __poll_t ep_send_events_proc(struct eventpoll *ep, struct list_head *head
{
    struct ep_send_events_data *esed = priv;
    __poll_t revents;
    struct epitem *epi, *tmp;
    struct epoll_event __user *uevent = esed->events;
    struct wakeup_source *ws;
    poll_table pt;
    init_poll_funcptr(&pt, NULL);
    ...

        // 遍历处理 txlist（原 ep->rdllist 数据）就绪队列结点，获取事件拷贝到用户空间。
        list_for_each_entry_safe(epi, tmp, head, rdllink)
    {
        if (esed->res >= esed->maxevents)
            break;
        ...
            // 先从就绪队列中删除 epi，如果是 LT 模式，就绪事件还没处理完，再把它添加回
            list_del_init(&epi->rdllink);

        // 获取 epi 对应 fd 的就绪事件。
```

```
22          revents = ep_item_poll(epi, &pt, 1);
23          if (!revents)
24              // 如果没有就绪事件就返回（这时候，epi 已经从就绪列表中删除了。）
25              continue;
26
27          // 内核空间向用户空间传递数据。__put_user 成功拷贝返回 0。
28          if (__put_user(revents, &uevent->events) ||
29              __put_user(epi->event.data, &uevent->data))
30          {
31              // 如果拷贝失败，继续保存在就绪列表里。
32              list_add(&epi->rdllink, head);
33              ep_pm_stay_awake(epi);
34              if (!esed->res)
35                  esed->res = -EFAULT;
36              return 0;
37          }
38
39          // 成功处理就绪事件的 fd 个数。
40          esed->res++;
41          uevent++;
42          if (epi->event.events & EPOLLONESHOT)
43              // #define EP_PRIVATE_BITS (EPOLLWAKEUP | EPOLLONESHOT | EPOLLET | E
44              epi->event.events &= EP_PRIVATE_BITS;
45          else if (!(epi->event.events & EPOLLET))
46          {
47              /*
48               * If this file has been added with Level
49               * Trigger mode, we need to insert back inside
50               * the ready list, so that the next call to
51               * epoll_wait() will check again the events
52               * availability. At this point, no one can insert
53               * into ep->rdllist besides us. The epoll_ctl()
54               * callers are locked out by
55               * ep_scan_ready_list() holding "mtx" and the
56               * poll callback will queue them in ep->ovflist.
57               */
58              /* lt 模式下，当前事件被处理完后，不会从就绪列表中删除，留待下一次 epoll_wa
59               * 调用，再查看是否还有事件没处理，如果没有事件了就从就绪列表中删除。
60               * 在遍历事件的过程中，不能写 ep->rdllist，因为已经上锁，只能把新的就绪信息
61               * 添加到 ep->ovflist */
62              list_add_tail(&epi->rdllink, &ep->rdllist);
63              ep_pm_stay_awake(epi);
64          }
65      }
66
67      return 0;
68  }
```

ep_poll_callback

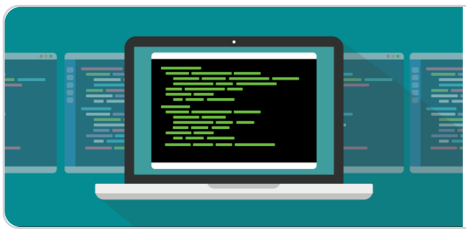
fd 事件回调。当 fd 有网络事件发生，就会通过等待队列，进行回调。参考 __wake_up_common，如果事件是用户关注的事件，回调会唤醒进程进行处理。

```
static int ep_poll_callback(wait_queue_entry_t *wait, unsigned mode, int sync, v
{
    int pwake = 0;
    struct epitem *epi = ep_item_from_wait(wait);
    struct eventpoll *ep = epi->ep;
    __poll_t pollflags = key_to_poll(key);
    unsigned long flags;
    int ewake = 0;

    // 禁止本地中断并获得指定读锁。
    read_lock_irqsave(&ep->lock, flags);

    ep_set_busy_poll_napi_id(epi);

    // #define EP_PRIVATE_BITS (EPOLLWAKEUP | EPOLLONESHOT | EPOLLET | EPOLLEXCL
    // 如果 fd 没有关注除了 EP_PRIVATE_BITS 之外的事件，那么走解锁流程。
    if (!(epi->event.events & ~EP_PRIVATE_BITS))
        goto out_unlock;

    // 如果回调的事件，不是用户关注的 fd 事件，那么走解锁流程。
    if (pollflags && !(pollflags & epi->event.events))
        goto out_unlock;

    /*
     * If we are transferring 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 happen during that period of time are
     * chained in ep->ovflist and requeued later on.
     */
    // 当内核空间向用户空间拷贝数据时，不添加 epi 到 rdllist，将它添加到 ovflist。
    if (READ_ONCE(ep->ovflist) != EP_UNACTIVE_PTR)
    {
        if (epi->next == EP_UNACTIVE_PTR && chain_epi_lockless(epi))
            ep_pm_stay_awake_rcu(epi);
        goto out_unlock;
    }

    // epi 已经加入就绪链表就不需要添加了。
    if (!ep_is_linked(epi) &&
        list_add_tail_lockless(&epi->rdllink, &ep->rdllist))
```

```
41      {
42          ep_pm_stay_awake_rcu(epi);
43      }
44
45      // 当回调事件是用户关注的事件，那么需要唤醒进程处理。
46
47      // ep->wq 在 epoll_wait 时添加，当没有就绪事件，epoll_wait 进行睡眠等待唤醒。
48      if (waitqueue_active(&ep->wq))
49      {
50          if ((epi->event.events & EPOLLEXCLUSIVE) &&
51              !(pollflags & POLLFREE))
52          {
53              // #define EPOLLINOUT_BITS (EPOLLIN | EPOLLOUT)
54              switch (pollflags & EPOLLINOUT_BITS)
55              {
56              case EPOLLIN:
57                  if (epi->event.events & EPOLLIN)
58                      ewake = 1;
59                  break;
60              case EPOLLOUT:
61                  if (epi->event.events & EPOLLOUT)
62                      ewake = 1;
63                  break;
64              case 0:
65                  ewake = 1;
66                  break;
67              }
68          }
69          wake_up(&ep->wq);
70      }
71
72      // ep->poll_wait 是 epoll 监控另外一个 epoll fd 的等待队列。如果触发事件，也需要唤
73      if (waitqueue_active(&ep->poll_wait))
74          pwake++;
75
76  out_unlock:
77      read_unlock_irqrestore(&ep->lock, flags);
78
79      /* We have to call this outside the lock */
80      if (pwake)
81          ep_poll_safewake(&ep->poll_wait);
82
83      if (!(epi->event.events & EPOLLEXCLUSIVE))
84          ewake = 1;
85
86      if (pollflags & POLLFREE)
87      {
```

```
88          /*
89           * If we race with ep_remove_wait_queue() it can miss
90           * ->whead = NULL and do another remove_wait_queue() after
91           * us, so we can't use __remove_wait_queue().
92           */
93          list_del_init(&wait->entry);
94          /*
95           * ->whead != NULL protects us from the race with ep_free()
96           * or ep_remove(), ep_remove_wait_queue() takes whead->lock
97           * held by the caller. Once we nullify it, nothing protects
98           * ep/epi or even wait.
99           */
100         smp_store_release(&ep_pwq_from_wait(wait)->whead, NULL);
101     }
102
103     return ewake;
104 }
```

epoll惊群问题



知乎 https://zhuanlan.zhihu.com/p/385410196

深入浅出 Linux 惊群：现象、原因和解决方案

作者：morganhuang，腾讯 IEG 后台开发工程师 1. Accept"惊群"现象我们知道，在网络分组通信中，网络数据包的接收是异步进行的，因为你不知道什么时…

https://mp.weixin.qq.com/s/xxjCrFH1361iG-srfNL9_Q

在我们直接介绍惊群问题之前，我们先来介绍两种编程模型。

1. 多进程共用一个epfd来监听同一listen_fd
2. 多进程拥有各自的epfd来监听listen_fd。

我们使用epoll_ctl的EPOLL_CTL_ADD时,将会在对应socket的等待队列添加一个任务,任务的回调函数将是ep_poll_callback。此函数的作用也比较简单,将就绪事件添加到任务所对应进程的就绪链表中,同时唤醒在另外一个等待队列睡眠的进程。

另外一个等待队列?其实就是epoll_fd的等待队列。当我们调用epoll_wait时,实际上我们的进程是休眠在epfd的等待队列中。而此等待队列最终的回调函数都default_wake_function。

我们先了解下Linux内核都是通过socket的睡眠队列来组织所有等待此socket事件变化的任务,一旦事件来了将遍历此socket的等待队列所有任务并调用每个任务的callback,如果遇到此任务被设置为WQ_FLAG_EXCLUSIVE则内核调用此任务callback就直接退出,不再继续遍历。

我们先来说第一种情况，共用一个epfd，那么在socket上的睡眠队列就只会有一个任务在这里阻塞，有数据来的时候，指挥唤醒一个epoll。不同的进程会封装成一个数据结构挂在这个ep上。在这种情况下，只会在LT工作模式下可能会有类似惊群的情况。待会我们在分析。

第二种呢就是有多个epfd，他们都会阻塞在socket上的队列，如果有数据来，那么他们都会被唤醒，这才是一般意义上的唤醒。解决这种问题就是添加一个WQ_FLAG_EXCLUSIVE标志，那么到时候内核就直接唤醒一个就可以了。

现在我们来看第一种情况，这种情况是最难搞的，涉及到源码。

上面我们提到的是类似惊群的问题，因为他不是想一般惊群问题那样一下子全部唤醒，而是一种链式的唤醒方式，并且这个不断唤醒的过程是可以终止的。

我们再来看一下epoll_wait的源码。

```
/* epoll_wait 执行逻辑。 */
static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events,
                   int maxevents, long timeout)
{
    ...
        /* epoll_wait 处理就绪事件前，先添加等待唤醒事件。 */
        if (!waiter)
    {
        waiter = true;
        /* current 是当前进程。 */
        init_waitqueue_entry(&wait, current);

        spin_lock_irq(&ep->wq.lock);
        __add_wait_queue_exclusive(&ep->wq, &wait);
        spin_unlock_irq(&ep->wq.lock);
    }
    ...
        /* 如果就绪队列没有就绪事件了，那么进程进入睡眠状态，等待唤醒。 */
        for (;;)
    {
        /* 进程设置为可中断睡眠状态。 */
        set_current_state(TASK_INTERRUPTIBLE);
        ... eavail = ep_events_available(ep);
        if (eavail)
            break;
        ...
            /* 没有就绪事件了，超时阻塞睡眠，等待唤醒。 */
            if (!schedule_hrtimeout_range(to, slack, HRTIMER_MODE_ABS))
```

```
29          {
30              timed_out = 1;
31              break;
32          }
33      }
34
35      /* 进程设置为唤醒状态。 */
36      __set_current_state(TASK_RUNNING);
37      ...
38          /* 就绪队列有事件，处理就绪事件逻辑。 */
39          if (!res && eavail &&
40              !(res = ep_send_events(ep, events, maxevents)) && !timed_out) goto f
41      ...
42          /* 处理完逻辑，从等待唤醒事件队列，删除自己的等待事件。 */
43          if (waiter)
44      {
45          spin_lock_irq(&ep->wq.lock);
46          __remove_wait_queue(&ep->wq, &wait);
47          spin_unlock_irq(&ep->wq.lock);
48      }
49      ...
50  }
51
52  static int ep_send_events(struct eventpoll *ep,
53                            struct epoll_event __user *events, int maxevents)
54  {
55      struct ep_send_events_data esed;
56
57      esed.maxevents = maxevents;
58      esed.events = events;
59
60      /* 遍历事件就绪队列，发送就绪事件到用户空间。 */
61      ep_scan_ready_list(ep, ep_send_events_proc, &esed, 0, false);
62      return esed.res;
63  }
64
65  static __poll_t ep_scan_ready_list(struct eventpoll *ep,
66                                     __poll_t (*sproc)(struct eventpoll *,
67                                                       struct list_head *, void *)
68                                     void *priv, int depth, bool ep_locked)
69  {
70      ...
71          /* 将就绪队列分片链接到 txlist 链表中。 */
72          list_splice_init(&ep->rdllist, &txlist);
73      /* 执行 ep_send_events_proc，唤醒进程 A。 */
74      res = (*sproc)(ep, &txlist, priv);
75      ... if (!list_empty(&ep->rdllist))
```

```
76      {
77          if (waitqueue_active(&ep->wq))
78              /* A 进程已被唤醒，但是就绪队列（ep->rdllist）还有数据，
79               * 进程 B，也在等待队列中，那么唤醒进程 B。 */
80              wake_up_locked(&ep->wq);
81          ...
82      }
83      ...
84  }
85
86  static __poll_t ep_send_events_proc(struct eventpoll *ep, struct list_head *head
87  {
88      ...
89          // 遍历处理 txlist（原 ep->rdllist 数据）就绪队列结点，获取事件拷贝到用户空间。
90          list_for_each_entry_safe(epi, tmp, head, rdllink)
91      {
92          if (esed->res >= esed->maxevents)
93              break;
94          ...
95              /* 先从就绪队列中删除 epi，如果是 lt 模式，就绪事件还没处理完，再把它添加回z
96              list_del_init(&epi->rdllink);
97
98          /* 获取 epi 对应 fd 的就绪事件。 */
99          revents = ep_item_poll(epi, &pt, 1);
100         if (!revents)
101             /* 如果没有就绪事件就返回（这时候，epi 已经从就绪队列中删除了。） */
102             continue;
103
104         /* 内核空间通过 __put_user 向用户空间拷贝传递数据。 */
105         if (__put_user(revents, &uevent->events) ||
106             __put_user(epi->event.data, &uevent->data))
107         {
108             /* 如果拷贝失败，将 epi 重新保存回就绪队列，以便下一次处理。 */
109             list_add(&epi->rdllink, head);
110             ep_pm_stay_awake(epi);
111             if (!esed->res)
112                 esed->res = -EFAULT;
113             return 0;
114         }
115
116         /* 增加成功处理就绪事件的个数。 */
117         esed->res++;
118         uevent++;
119         if (epi->event.events & EPOLLONESHOT)
120             /* #define EP_PRIVATE_BITS (EPOLLWAKEUP | EPOLLONESHOT | EPOLLET | E
121             epi->event.events &= EP_PRIVATE_BITS;
122         else if (!(epi->event.events & EPOLLET))
```

```
123         {
124             /* lt 模式，重新将前面从就绪队列删除的 epi 添加回去。
125              * 等待下一次 epoll_wait 调用，重新走上面的逻辑。
126              * et 模式，前面从就绪队列里删除的 epi 将不会被重新添加，
127              * 直到用户关注的事件再次发生。*/
128             list_add_tail(&epi->rdllink, &ep->rdllist);
129             ep_pm_stay_awake(epi);
130         }
131     }
132 
133     return 0;
134 }
```

LT的描述“如果事件来了，不管来了几个，只要仍然有未处理的事件，epoll都会通知你。”，显然，epoll_wait刚刚取到事件的时候的时候，不可能马上就调用accept去处理，事实上，逻辑在epoll_wait函数调用的ep_poll中还没返回的，这个时候，显然符合“仍然有未处理的事件”这个条件，显然这个时候为了实现这个语义，需要做的就是通知别的同样阻塞在同一个epoll句柄睡眠队列上的进程！在实现上，这个语义由两点来保证：

保证1：在LT模式下，“就绪链表”上取出的epi上报完事件后会重新加回“就绪链表”；

保证2：如果“就绪链表”不为空，且此时有进程阻塞在同一个epoll句柄的睡眠队列上，则唤醒它。

```
1  ep_scan_ready_list()
2  {
3      // 遍历“就绪链表”
4      ready_list_for_each() {
5          list_del_init(&epi->rdllink);
6          revents = ep_item_poll(epi, &pt);
7          // 保证1
8          if (revents) {
9              __put_user(revents, &uevent->events);
10             if (!(epi->event.events & EPOLLET)) {
11                 list_add_tail(&epi->rdllink, &ep->rdllist);
12             }
13         }
14     }
15     // 保证2
16     if (!list_empty(&ep->rdllist)) {
17         if (waitqueue_active(&ep->wq))
18             wake_up_locked(&ep->wq);
19     }
20 }
21 
```

我们来看一个例子

> 1. 假设进程a的epoll_wait首先被ep_poll_callback唤醒，那么满足1和2，则唤醒了进程B；
> 2. 进程B在处理ep_scan_ready_list的时候，发现依然满足1和2，于是唤醒了进程C....
> 3. 上面1)和2)的过程一直到之前某个进程将client取出，此时下一个被唤醒的进程在ep_scan_ready_list中的ep_item_poll调用中将得不到任何事件，此时便不会再将该epi加回“就绪链表”了，LT水平触发结束，结束了这场悲伤的梦！

所以说这种情形并不算是标准的惊群现象，好像没有什么办法解决。至于网上的什么nginx的解决方法都是说的第二种情况，在Linux内核没有出现WQ_FLAG_EXCLUSIVE标志之前，可以用加锁的方式来保证只有一个进程被唤醒。

所以在实践中我们应该多个线程多个epfd来使用，或者使用et来。

非阻塞IO

套接字的默认状态是阻塞的。这表示发出一个不能立即完成的套接字调用时，其进程会被投入睡眠，等待相应操作完成。可能阻塞的套接字调用可分为下面 4 种：

1. 输入操作，包括 read，readv，recv，recvfrom 和 recvmsg 共 5 个函数。如果某个进程对一个阻塞的 TCP 套接字调用这些输入函数之一，并且缓冲区中没有数据可以读取，进程会进入休眠，知道数据到达。TCP 是字节流协议，该进程的唤醒只要一些数据，如果要设置固定的量，可以使用 readn 或者设置 MSG_WAITALL 标志。对于非阻塞的套接字，如果输入操作不被满足，调用会立即返回一个 EWOULDBLOCK 错误
2. 输出操作，包括 write，writev，send，sendto 和 sendmsg 共 5 个函数，和输入类似。对于一个非阻塞的 TCP 套接字，如果发送缓冲区没有空间，输出函数调用将立即返回一个 EWOULDBLOCK 错误。
3. 接受外来连接，即 accept 函数。如果对一个阻塞的套接字调用该 accept 函数，并且没有新的连接到达，调用进程将被投入睡眠。如果一个非阻塞的套接字调用 accept 函数，并且尚无新的连接到达，accept 调用将立即返回一个 EWOLULDBLOCK 错误。
4. 发起外出连接，即用于 TCP 的 connect 函数。TCP 连接的建立涉及一个三路握手过程，而且 connect 函数一直要等到客户收到对于自己的 SYN 的 ACK 才会返回。这意味着 TCP 的每个 connect 总会阻塞其调用进程至少一个 RTT 时间。如果对一个非阻塞的 TCP 套接字调用 connect，并且连接不能立即建立，那么连接的建立能照样发起，不过会返回一个 EINPROGRESS 错误，注意这个错误和上面的错误并不相同。还需要注意同一主机上的连接会立即建立完成，通常发生在同一主机的情况下。因此对于非阻塞的 connect，我们也要预备 connect 成功返回的情况发生。

对于非阻塞IO这边我就想把unp上的几个例子拿来说一说。分别是一个使用非阻塞io的回射函数和非阻塞connect在浏览器上的应用。

非阻塞读和写：str_cli 函数（修订版）

```
1
2  #include "unp.h"
3  #include <time.h>
4
5
6  void str_cli(FILE *fp, int sockfd){
7          int maxfdp1, val, stdineof;
8          ssize_t n, nwritten;
9          fd_set rset, wset;
10         char to[MAXLINE], fr[MAXLINE];  //to是从标准输入到服务器的缓冲区，fr是服务器
11         char *toiptr, *tooptr, *friptr, *froptr; //opt是输出，ipt是输入。
12
13         val = Fcntl(sockfd, F_GETFL, 0);  //暂存套接字描述符的原值
14         Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);  //用|的形式去添加非阻塞位
15
16         val = Fcntl(STDIN_FILENO, F_GETFL, 0);  //暂存标准输入字描述符的原值
17         Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);  //设置为非阻塞
18
19         val = Fcntl(STDOUT_FILENO, F_GETFL, 0);  //暂存标准输出的原值
20         Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);  //设置为非阻塞
21
22         toiptr = tooptr = to;  //缓冲区指针初始化，一开始都是相同的,在数组开头
23         friptr = froptr = fr;
24         stdineof = 0; //这个标志用来判断标准输入是否结束
25         maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
26         for( ; ; ){
27                 FD_ZERO(&rset);   //读写集合的初始化清零
28                 FD_ZERO(&wset);
29                 //如果可读，且标准输入缓冲区还够，则重置该描述符到reset，接下来select能
30                 if (stdineof == 0 && toiptr < &to[MAXLINE])
31                         FD_SET(STDIN_FILENO, &rset);
32                 if (friptr < &fr[MAXLINE]) //从服务器接收数据缓冲区还够，则绑定
33                         FD_SET(sockfd, &rset);
34                 if (tooptr != toiptr) //输入数据中还有一些数据没有发往服务器，则绑定
35                         FD_SET(sockfd, &wset);
36                 if(froptr != friptr) //接收的数据中，还有一些没发往屏幕，则绑定
37                         FD_SET(STDOUT_FILENO, &wset);
38                 Select(maxfdp1, &rset, &wset, NULL, NULL); //等待哪个描述符有可读或
39                 /* ……等待……，好! Select返回了，我们检查一下是哪些描述符有反应了 */
40                 if (FD_ISSET(STDIN_FILENO, &rset)){ //如果是输入可读，即有标准输入的
41                         if( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toipt
```

```
42                                  if (errno != EWOULDBLOCK)  //我们忽略阻塞错误
43                                          err_sys("read error on stdin");
44                          }else if (n == 0){  //n=0说明是输入了EOF
45                                  fprintf(stderr, "%s:EOF on stdin\n", gf_time());
46                                  stdineof = 1;
47                                  if (tooptr == toiptr)  //当发往服务器的缓冲区用完，
48                                          Shutdown(sockfd, SHUT_WR);  //也可能没用完
49                          }else{
50                                  fprintf(stderr, "%s: read %d bytes from stdin\n"
51                                  toiptr += n; //又输入了一些！指针右移
52                                  FD_SET(sockfd, &wset);  //既然有输入，那么又可以给服
53                          }
54                  }
55                  if (FD_ISSET(sockfd, &rset)){ //收到服务器发来的数据啦
56                          if( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) <
57                                  if (errno != EWOULDBLOCK)  //我们忽略阻塞错误
58                                          err_sys("read error on stdin");
59                          }else if (n == 0){  //n=0说明是服务器发给我的数据发完了
60                                  fprintf(stderr, "%s:EOF on socket\n", gf_time())
61                                  if (stdineof)  //刚好我的输入也结束了的，说明，整个回
62                                          return ;
63                                  else  //whatfuck!  输入还没结束，你就告诉我回射完了?
64                                          err_quit("str_cli: server terminated pre
65                          }else{
66                                  fprintf(stderr, "%s: read %d bytes from socket\n
67                                  friptr += n; //收到了一些数据！指针右移
68                                  FD_SET(STDOUT_FILENO, &wset);  //那么接下来可以让标
69                          }
70                  }
71
72                  //想输出到屏幕？？先看看给你存的那些服务器发来的数据够不够
73                  //服务器的数据都给你发完了？说明那边堵住了，你别急着写，先歇着！
74                  if (FD_ISSET(STDOUT_FILENO, &wset) && ((n = friptr - froptr) > 0
75                          if( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0){
76                                  if (errno != EWOULDBLOCK)  //我们忽略阻塞错误
77                                          err_sys("write error on stdin");
78                          }else{
79                                  fprintf(stderr, "%s: wrote %d bytes to stdout\n"
80                                  froptr += nwritten; //往屏幕发了一下数据，缓冲区里的
81                                  if(froptr == friptr)  //哇发完了？那我们重头开始（免
82                                          froptr = friptr = fr;
83                          }
84                  }
85
86                  //我要往服务器发数据！先看看内容够不够。。。
87                  if (FD_ISSET(sockfd, &wset) && ((n = toiptr - tooptr) > 0)){
88                          if( (nwritten = write(sockfd, tooptr, n)) < 0){ //可能一次
```

```
89                                  if (errno != EWOULDBLOCK)  //我们忽略阻塞错误
90                                          err_sys("write error on stdin");
91                          }else{
92                                  fprintf(stderr, "%s: wrote %d bytes to socket\n"
93                                  tooptr += nwritten; //往屏幕发了一下数据，缓冲区里的
94                                  if(tooptr == toiptr){  //哇发完了？那我们重头开始（
95                                          tooptr = toiptr = to;
96                                          if (stdineof)
97                                                  Shutdown(sockfd, SHUT_WR);
98                                  }
99                          }
100                 }
101         }
102
103 }
104
105 int main(int argc, char **argv){
106         int sockfd,i;
107         struct sockaddr_in servaddr;
108         if (argc != 2){
109                 err_quit("usage: tcpli<IPaddress>");
110         }
111         sockfd = Socket(AF_INET, SOCK_STREAM, 0);
112         bzero(&servaddr, sizeof(servaddr));
113         servaddr.sin_family = AF_INET;
114         servaddr.sin_port = htons(SERV_PORT);
115         Inet_pton(AF_INET, argv[1],  &servaddr.sin_addr);
116         Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
117         str_cli(stdin, sockfd);
118         exit(0);
119 }
120
```

非阻塞connect

当在一个非阻塞 tcp 套接字上调用 connect 时，connect 将立即返回 EINPROGRESS 错误 (注意一种特殊情况: 在本机上建立 tcp 连接 connect 可能会返回成功)，不过已经发起的 tcp 三次握手仍在进行，接着可以使用 select 来检测这个连接或成功或失败。非阻塞 connect 有三个用途：

1. 可以把三次握手叠加到其他处理上。完成一个 connect 至少一个 RTT 时间，从局域网可能是几毫秒到几百毫秒甚至广域网上的几秒。这一段时间也许有我们想要执行的其他处理工作可执行。
2. 可以利用此同时建立多个连接。
3. 利用 select 的超时时间来缩短 connect 的超时。即在指定时间内检测 socket 是否可用。如果超时发生，需要主动关闭套接字，防止已经启动的三次握手继续进行。

接下来我们将使用两个例子来看一下非阻塞的connect如何使用。

非阻塞connect示例

1. 创建socket，并将 socket 设置成非阻塞模式；
2. 调用 connect 函数，此时无论 connect 函数是否连接成功会立即返回；如果返回-1并不表示连接出错，如果此时错误码是EINPROGRESS
3. 接着调用 select 函数，在指定的时间内判断该 socket 是否可写，如果可写说明连接成功，反之则认为连接失败。

```
int connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
        int                                     flags, n, error;
        socklen_t               len;
        fd_set                          rset, wset;
        struct timeval          tval;
    // 先获取原套接字描述符
        flags = Fcntl(sockfd, F_GETFL, 0);
    // 调用 fcntl 设置为非阻塞
        Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
    // 发起非阻塞的 connect，期望的错误是 EINPROGRESS，表示连接建立已经启动但是尚未完成。connect 返回的任何其他错误返回给本函数的调用者。
        error = 0;
        if ( (n = connect(sockfd, saptr, salen)) < 0)
                if (errno != EINPROGRESS)
                        return(-1);

        /* Do whatever we want while the connect is taking place. */
    // 此时可以在 RTT 时间中做我们想做的事
    // h == 0，连接已经建立，处于同一主机，立即建立连接，直接跳转到 done
        if (n == 0)
                goto done;      /* connect completed immediately */
    // 调用 select 等带连接的建立完成
        FD_ZERO(&rset);
        FD_SET(sockfd, &rset);
        wset = rset;
        tval.tv_sec = nsec;
        tval.tv_usec = 0;
    // select 返回 0，超时情况发生，返回 ETIMEOUT 错误返回给调用者。还需要关闭套接字，防止三路握手继续下去
        if ( (n = Select(sockfd+1, &rset, &wset, NULL,
                                        nsec ? &tval : NULL)) == 0) {
                close(sockfd);                  /* timeout */
```

```
32                  errno = ETIMEDOUT;
33                  return(-1);
34          }
35      // 变成可读或者可读可写
36          if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
37                  len = sizeof(error);
38          // 获取待处理错误，建立成功返回 0.建立出错，返回对应的 error
39                  if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
40                          return(-1);                         /* Solaris pending
    error */
41          } else
42                  err_quit("select error: sockfd not set");
43
44  done:
45      // 恢复套接字原来的文件状态标志
46          Fcntl(sockfd, F_SETFL, flags);          /* restore file status flags */
47      // 有 error 标志，关闭描述符并返回 -1
48          if (error) {
49                  close(sockfd);                  /* just in case */
50                  errno = error;
51                  return(-1);
52          }
53          return(0);
54  }
```

非阻塞 connect 听起来虽然简单，但是仍然有一些细节问题要处理：

1. 即使套接字是非阻塞的，如果连接的服务器在同一台主机上，那么在调用 connect 建立连接时，连接通常会立即建立成功。我们必须处理这种情况；
2. 源自 Berkeley 的实现有两条与 select 和非阻塞 I/O 相关的规则：
   a. 当连接建立成功时，套接口描述符变成 可写 （连接建立时，写缓冲区空闲，所以可写）；
   b. 当连接建立出错时，套接口描述符变成 既可读又可写 （由于有未决的错误，从而可读又可写）；

      注意：当一个套接口出错时，它会被 select 调用标记为既可读又可写。
      非阻塞 connect 有这么多好处，但是处理非阻塞 connect 时会遇到很多【可移植性问题】。

非阻塞connect实现一个web客户端

```
1  int nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
2  fd_set rset, wset;
3
```

```
int main(int argc, char **argv)
{
    int i, fd, n, maxnconn, flags, error;
    char buf[MAXLINE];
    fd_set rs, ws;

    if (argc < 5)
        err_quit("usage: web <#conns> <hostname> <homepage> <file1> ...");

    maxnconn = atoi(argv[1]); /*最大连接数*/

    nfiles = min(argc - 4, MAXFILES);
    for (i = 0; i < nfiles; i++) {
        file[i].f_name = argv[i + 4];
        file[i].f_host = argv[2];
        file[i].f_flags = 0;
    }
    printf("nfiles = %d\n", nfiles);
    /*访问服务器主页*/
    home_page(argv[2], argv[3]);
    FD_ZERO(&rset);
    FD_ZERO(&wset);
    maxfd = -1;
    nlefttoread = nlefttoconn = nfiles;
    nconn = 0;

    while (nlefttoread > 0) { /*还有文件未读取*/
        while (nconn < maxnconn && nlefttoconn > 0) { /*还有空闲的连接*/
            for (i = 0; i < nfiles; i++) /*找到一个待读取的文件*/
                if (file[i].f_flags == 0)
                    break;
            if (i == nfiles)
                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
            start_connect(&file[i]); /*发起非阻塞连接*/
            nconn++;
            nlefttoconn--;
        }
        rs = rset;
        ws = wset;
        n = Select(maxfd + 1, &rs, &ws, NULL, NULL); /*监听所有的套接字描述符*/
        for (i = 0; i < nfiles; i++) {
            flags = file[i].f_flags;
            if (flags == 0 || flags & F_DONE) /*忽略已经读取的文件*/
                continue;
            fd = file[i].f_fd;
            if (flags & F_CONNECTING &&
                (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) { /*连接正在进行*/
```

```
51                  n = sizeof(error);
52                  if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &n) < 0 ||
53                      error != 0) { /*连接失败*/
54                      err_ret("nonblocking connect failed for %s",
55                          file[i].f_name);
56                  }
57                  /*连接成功*/
58                  printf("connection established for %s\n", file[i].f_name);
59                  FD_CLR(fd, &wset); /*不需要再测试描述符是否可写*/
60                  write_get_cmd(&file[i]); /*使用GET命令获取文件*/
61
62              } else if(flags & F_READING && FD_ISSET(fd, &rs)) { /*正在读取文件*/
63                  if ((n = Read(fd, buf, sizeof(buf))) == 0) {
64                      printf("end-of-file on %s\n", file[i].f_name);
65                      Close(fd); /*文件读取完毕，断开连接*/
66                      file[i].f_flags = F_DONE;
67                      FD_CLR(fd, &rset); /*不需要再测试描述符是否可读*/
68                      nconn--;
69                      nlefttoread--;
70                  } else {
71                      printf("read %d bytes from %s\n", n, file[i].f_name);
72                  }
73              }
74          }
75      }
76
77      exit(0);
78  }
```

我们只把这个主函数拿出来，来看一下基本的思路，里面的一些细节我们就不说了。

首先调用 `home_page` 来将主页面加载进来，然后循环调用非阻塞connect连接，随后调用select来检测连接是否建立成功。随后再次请求。整个处理的逻辑是比较简单的。

非阻塞accept

当有一个已完成的连接准备好被 accept 时，select 将作为可读描述符返回该连接的监听套接字。因此，如果使用 select 来监听某个套接字上等待的一个外来连接，那就没有必要把监听套接字设置为非阻塞。

但这里存在一个潜在问题：利用了 IO 多路复用的 accept 正常情况下不会被阻塞，即使是阻塞 tcp 套接字。但考虑一下情况：

a. 客户端申请建立连接，并建立后发送 RST 终止连接

b. select 向服务器返回监听套接字可读，但过一段时间再 accept

c. 服务器从 select 返回到调用 accept 期间，服务器收到客户的 RST

d. 该连接被弹出已完成连接队列，假设队列中没有其他连接，那么服务器将阻塞在 accept，无法处理其他事务，除非新来一个连接

可以发现，以上情况并不满足正常情况的 IO 多路复用。所以解决方法是将监听套接字设置为非阻塞。

http://blog.chinaunix.net/uid-28541347-id-4238524.html

poll&&epoll实现分析(二)——epoll实现-lvyilong316-ChinaUnix博客

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

彻底学会使用epoll(一)——ET模式实现分析_lvyilong316-ChinaUnix博客

http://blog.chinaunix.net/uid-26339466-id-3292595.html

EPOLL Linux内核源代码实现原理分析-huangjiangwei-ChinaUnix博客

EPOLL内核源代码实现原理分析 黄江伟 will.huang@aliyun-inc.com epoll的实现主要依赖于一个迷你文件系统:eventpollfs。此文件系统通过初始化。在初始化的过程中，eventpollfs cr

https://wenfh2020.com/2020/04/23/epoll-code/

[内核源码] epoll 实现原理

文章主要对 tcp 通信进行 epoll 源码走读。Linux 源码:Linux 5.7 版本。epoll 核心源码:eventpoll.h / eventpoll.c。 搭建 epoll 内核调试环境视频:vscode + gdb 远程调试 linux (EPOLL) 内核源码