

31 | 性能篇答疑--epoll源码深度剖析

2019-10-18 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 14:27 大小 13.25M



你好，我是盛延敏，今天是网络编程实战性能篇的答疑模块，欢迎回来。


在性能篇中，我主要围绕 C10K 问题进行了深入剖析，最后引出了事件分发机制和多线程。可以说，基于 epoll 的事件分发能力，是 Linux 下高性能网络编程的不二之选。如果你觉得还不过瘾，期望有更深刻的认识和理解，那么在性能篇的答疑中，我就带你一起梳理一下 epoll 的源代码，从中我们一定可以有更多的发现和领悟。

今天的代码有些多，建议你配合文稿收听音频。

基本数据结构

在开始研究源代码之前，我们先看一下 epoll 中使用的数据结构，分别是 eventpoll、epitem 和 epoll_entry。

我们先看一下 eventpoll 这个数据结构，这个数据结构是我们在调用 epoll_create 之后内核侧创建的一个句柄，表示了一个 epoll 实例。后续如果我们在调用 epoll_ctl 和 epoll_wait 等，都是对这个 eventpoll 数据进行操作，这部分数据会被保存在 epoll_create 创建的匿名文件 file 的 private_data 字段中。

 复制代码

```
1  /*
2   * This structure is stored inside the "private_data" member of the file
3   * structure and represents the main data structure for the eventpoll
4   * interface.
5   */
6  struct eventpoll {
7      /* Protect the access to this structure */
8      spinlock_t lock;
9
10     /*
11      * This mutex is used to ensure that files are not removed
12      * while epoll is using them. This is held during the event
13      * collection loop, the file cleanup path, the epoll file exit
14      * code and the ctl operations.
15      */
16     struct mutex mtx;
17
18     /* Wait queue used by sys_epoll_wait() */
19     // 这个队列里存放的是执行 epoll_wait 从而等待的进程队列
20     wait_queue_head_t wq;
21
22     /* Wait queue used by file->poll() */
23     // 这个队列里存放的是该 eventloop 作为 poll 对象的一个实例，加入到等待的队列
24     // 这是因为 eventpoll 本身也是一个 file，所以也会有 poll 操作
25     wait_queue_head_t poll_wait;
26
27     /* List of ready file descriptors */
28     // 这里存放的是事件就绪的 fd 列表，链表的每个元素是下面的 epitem
29     struct list_head rdllist;
30
31     /* RB tree root used to store monitored fd structs */
32     // 这是用来快速查找 fd 的红黑树
33     struct rb_root_cached rbr;
34
35     /*
36      * This is a single linked list that chains all the "struct epitem" that
37      * happened while transferring ready events to userspace w/out
38      * holding ->lock.
```


```

39     */
40     struct epitem *ovflist;
41
42     /* wakeup_source used when ep_scan_ready_list is running */
43     struct wakeup_source *ws;
44
45     /* The user that created the eventpoll descriptor */
46     struct user_struct *user;
47
48     // 这是 eventloop 对应的匿名文件，充分体现了 Linux 下一切皆文件的思想
49     struct file *file;
50
51     /* used to optimize loop detection check */
52     int visited;
53     struct list_head visited_list_link;
54
55 #ifdef CONFIG_NET_RX_BUSY_POLL
56     /* used to track busy poll napi_id */
57     unsigned int napi_id;
58 #endif
59 };

```

你能看到在代码中我提到了 epitem，这个 epitem 结构是干什么用的呢？

每当我们调用 `epoll_ctl` 增加一个 fd 时，内核就会为我们创建出一个 epitem 实例，并且把这个实例作为红黑树的一个子节点，增加到 eventpoll 结构体中的红黑树中，对应的字段是 `rbr`。这之后，查找每一个 fd 上是否有事件发生都是通过红黑树上的 epitem 来操作。

 复制代码

```

1  /*
2   * Each file descriptor added to the eventpoll interface will
3   * have an entry of this type linked to the "rbr" RB tree.
4   * Avoid increasing the size of this struct, there can be many thousands
5   * of these on a server and we do not want this to take another cache line.
6   */
7  struct epitem {
8      union {
9          /* RB tree node links this structure to the eventpoll RB tree */
10         struct rb_node rbn;
11         /* Used to free the struct epitem */
12         struct rcu_head rcu;
13     };
14
15     /* List header used to link this structure to the eventpoll ready list */


```

```

16 // 将这个 epitem 连接到 eventpoll 里面的 rdllist 的 list 指针
17 struct list_head rdllink;
18
19 /*
20  * Works together "struct eventpoll"->ovflist in keeping the
21  * single linked chain of items.
22  */
23 struct epitem *next;
24
25 /* The file descriptor information this item refers to */
26 //epoll 监听的 fd
27 struct epoll_filefd ffd;
28
29 /* Number of active wait queue attached to poll operations */
30 // 一个文件可以被多个 epoll 实例所监听，这里就记录了当前文件被监听的次数
31 int nwait;
32
33 /* List containing poll wait queues */
34 struct list_head pwqlist;
35
36 /* The "container" of this item */
37 // 当前 epollitem 所属的 eventpoll
38 struct eventpoll *ep;
39
40 /* List header used to link this item to the "struct file" items list */
41 struct list_head flink;
42
43 /* wakeup_source used when EPOLLWAKEUP is set */
44 struct wakeup_source __rcu *ws;
45
46 /* The structure that describe the interested events and the source fd */
47 struct epoll_event event;
48 };

```

每次当一个 fd 关联到一个 epoll 实例，就会有一个 eppoll_entry 产生。eppoll_entry 的结构如下：

 复制代码

```

1 /* Wait structure used by the poll hooks */
2 struct eppoll_entry {
3     /* List header used to link this structure to the "struct epitem" */
4     struct list_head llink;
5
6     /* The "base" pointer is set to the container "struct epitem" */
7     struct epitem *base;
8
9     /*

```

```

10     * Wait queue item that will be linked to the target file wait
11     * queue head.
12     */
13     wait_queue_entry_t wait;
14
15     /* The wait queue head that linked the "wait" wait queue item */
16     wait_queue_head_t *whead;
17 };

```

epoll_create


我们在使用 epoll 的时候，首先会调用 epoll_create 来创建一个 epoll 实例。这个函数是如何工作的呢？

首先，epoll_create 会对传入的 flags 参数做简单的验证。

```

1  /* Check the EPOLL_* constant for consistency. */
2  BUILD_BUG_ON(EPOLL_CLOEXEC != O_CLOEXEC);
3
4  if (flags & ~EPOLL_CLOEXEC)
5      return -EINVAL;
6  /*

```

 复制代码

接下来，内核申请分配 eventpoll 需要的内存空间。

```

1  /* Create the internal data structure ("struct eventpoll").
2  */
3  error = ep_alloc(&ep);
4  if (error < 0)
5      return error;

```


 复制代码

在接下来，epoll_create 为 epoll 实例分配了匿名文件和文件描述字，其中 fd 是文件描述字，file 是一个匿名文件。这里充分体现了 UNIX 下一切都是文件的思想。注意，

eventpoll 的实例会保存一份匿名文件的引用，通过调用 fd_install 函数将匿名文件和文件描述字完成了绑定。

这里还有一个特别需要注意的地方，在调用 anon_inode_get_file 的时候，epoll_create 将 eventpoll 作为匿名文件 file 的 private_data 保存了起来，这样，在之后通过 epoll 实例的文件描述字来查找时，就可以快速地定位到 eventpoll 对象了。

最后，这个文件描述字作为 epoll 的文件句柄，被返回给 epoll_create 的调用者。

 复制代码

```
1 /*
2  * Creates all the items needed to setup an eventpoll file. That is,
3  * a file structure and a free file descriptor.
4  */
5 fd = get_unused_fd_flags(O_RDWR | (flags & O_CLOEXEC));
6 if (fd < 0) {
7     error = fd;
8     goto out_free_ep;
9 }
10 file = anon_inode_getfile("[eventpoll]", &eventpoll_fops, ep,
11     O_RDWR | (flags & O_CLOEXEC));
12 if (IS_ERR(file)) {
13     error = PTR_ERR(file);
14     goto out_free_fd;
15 }
16 ep->file = file;
17 fd_install(fd, file);
18 return fd;
```

epoll_ctl

接下来，我们看一下一个套接字是如何被添加到 epoll 实例中的。这就要解析一下 epoll_ctl 函数实现了。


查找 epoll 实例

首先，epoll_ctl 函数通过 epoll 实例句柄来获得对应的匿名文件，这一点很好理解，UNIX 下一切都是文件，epoll 的实例也是一个匿名文件。

 复制代码


```
1 // 获得 epoll 实例对应的匿名文件
2 f = fdget(epfd);
3 if (!f.file)
4     goto error_return;
```

接下来，获得添加的套接字对应的文件，这里 tf 表示的是 target file，即待处理的目标文件。

 复制代码


```
1 /* Get the "struct file *" for the target file */
2 // 获得真正的文件，如监听套接字、读写套接字
3 tf = fdget(fd);
4 if (!tf.file)
5     goto error_fput;
```

再接下来，进行了一系列的数据验证，以保证用户传入的参数是合法的，比如 epfd 真的是一个 epoll 实例句柄，而不是一个普通文件描述符。

 复制代码

```
1 /* The target file descriptor must support poll */
2 // 如果不支持 poll，那么该文件描述字是无效的
3 error = -EPERM;
4 if (!tf.file->f_op->poll)
5     goto error_tgt_fput;
6 ...
```


如果获得了一个真正的 epoll 实例句柄，就可以通过 private_data 获取之前创建的 eventpoll 实例了。

 复制代码

```
1 /*
2  * At this point it is safe to assume that the "private_data" contains
3  * our own data structure.
4  */
5 ep = f.file->private_data;
```



红黑树查找

接下来 `epoll_ctl` 通过目标文件和对应描述字，在红黑树中查找是否存在该套接字，这也是 `epoll` 为什么高效的地方。红黑树（RB-tree）是一种常见的数据结构，这里 `eventpoll` 通过红黑树跟踪了当前监听的所有文件描述字，而这棵树的根就保存在 `eventpoll` 数据结构中。

 复制代码

```
1 /* RB tree root used to store monitored fd structs */
2 struct rb_root_cached rbr;
```

对于每个被监听的文件描述字，都有一个对应的 `epitem` 与之对应，`epitem` 作为红黑树中的节点就保存在红黑树中。

 复制代码

```
1 /*
2  * Try to lookup the file inside our RB tree, Since we grabbed "mtx"
3  * above, we can be sure to be able to use the item looked up by
4  * ep_find() till we release the mutex.
5  */
6 epi = ep_find(ep, tf.file, fd);
```

红黑树是一棵二叉树，作为二叉树上的节点，`epitem` 必须提供比较能力，以便可以按大小顺序构建出一棵有序的二叉树。其排序能力是依靠 `epoll_filefd` 结构体来完成的，`epoll_filefd` 可以简单理解为需要监听的文件描述字，它对应到二叉树上的节点。

可以看到这个还是比较好理解的，按照文件的地址大小排序。如果两个相同，就按照文件文件描述字来排序。

 复制代码

```
1 struct epoll_filefd {
2     struct file *file; // pointer to the target file struct corresponding to the fd
3     int fd; // target file descriptor number
4 } __packed;
5
6 /* Compare RB tree keys */
7 static inline int ep_cmp_ffd(struct epoll_filefd *p1,
```



```

8             struct epoll_filefd *p2)
9 {
10     return (p1->file > p2->file ? +1:
11             (p1->file < p2->file ? -1 : p1->fd - p2->fd));
12 }

```

在进行完红黑树查找之后，如果发现是一个 ADD 操作，并且在树中没有找到对应的二叉树节点，就会调用 `ep_insert` 进行二叉树节点的增加。

 复制代码

```

1 case EPOLL_CTL_ADD:
2     if (!epi) {
3         epds.events |= POLLERR | POLLHUP;
4         error = ep_insert(ep, &epds, tf.file, fd, full_check);
5     } else
6         error = -EEXIST;
7     if (full_check)
8         clear_tfile_check_list();
9     break;

```

ep_insert

`ep_insert` 首先判断当前监控的文件值是否超过了 `/proc/sys/fs/epoll/max_user_watches` 的预设最大值，如果超过了则直接返回错误。


 复制代码

```

1 user_watches = atomic_long_read(&ep->user->epoll_watches);
2 if (unlikely(user_watches >= max_user_watches))
3     return -ENOSPC;

```

接下来是分配资源和初始化动作。

 复制代码

```

1 if (!(epi = kmem_cache_alloc(epi_cache, GFP_KERNEL)))
2     return -ENOMEM;
3
4     /* Item initialization follow here ... */


```

```

5 INIT_LIST_HEAD(&epi->rdllink);
6 INIT_LIST_HEAD(&epi->flink);
7 INIT_LIST_HEAD(&epi->pwqlist);
8 epi->ep = ep;
9 ep_set_ffd(&epi->ffd, tfile, fd);
10 epi->event = *event;
11 epi->nwait = 0;
12 epi->next = EP_UNACTIVE_PTR;

```

再接下来的事情非常重要，`ep_insert` 会为加入的每个文件描述字设置回调函数。这个回调函数是通过函数 `ep_ptable_queue_proc` 来进行设置的。这个回调函数是干什么的呢？其实，对应的文件描述字上如果有事件发生，就会调用这个函数，比如套接字缓冲区有数据了，就会回调这个函数。这个函数就是 `ep_poll_callback`。这里你会发现，原来内核设计也是充满了事件回调的原理。

 复制代码

```


1 /*
2  * This is the callback that is used to add our wait queue to the
3  * target file wakeup lists.
4  */
5 static void ep_ptable_queue_proc(struct file *file, wait_queue_head_t *whead, poll_table
6 {
7     struct epitem *epi = ep_item_from_epqueue(pt);
8     struct eppoll_entry *pwq;
9
10    if (epi->nwait >= 0 && (pwq = kmem_cache_alloc(pwq_cache, GFP_KERNEL))) {
11        init_waitqueue_func_entry(&pwq->wait, ep_poll_callback);
12        pwq->whead = whead;
13        pwq->base = epi;
14        if (epi->event.events & EPOLLEXCLUSIVE)
15            add_wait_queue_exclusive(whead, &pwq->wait);
16        else
17            add_wait_queue(whead, &pwq->wait);
18        list_add_tail(&pwq->llink, &epi->pwqlist);
19        epi->nwait++;
20    } else {
21        /* We have to signal that an error occurred */
22        epi->nwait = -1;
23    }
24 }

```

ep_poll_callback


ep_poll_callback 函数的作用非常重要，它将内核事件真正地 and epoll 对象联系了起来。它又是怎么实现的呢？

首先，通过这个文件的 wait_queue_entry_t 对象找到对应的 epitem 对象，因为 epoll_entry 对象里保存了 wait_queue_entry_t，根据 wait_queue_entry_t 这个对象的地址就可以简单计算出 epoll_entry 对象的地址，从而可以获得 epitem 对象的地址。这部分工作在 ep_item_from_wait 函数中完成。一旦获得 epitem 对象，就可以寻迹找到 eventpoll 实例。

 复制代码

```
1 /*
2  * This is the callback that is passed to the wait queue wakeup
3  * mechanism. It is called by the stored file descriptors when they
4  * have events to report.
5  */
6 static int ep_poll_callback(wait_queue_entry_t *wait, unsigned mode, int sync, void *key)
7 {
8     int pwake = 0;
9     unsigned long flags;
10     struct epitem *epi = ep_item_from_wait(wait);
11     struct eventpoll *ep = epi->ep;
```

接下来，进行一个加锁操作。

 复制代码

```
1 spin_lock_irqsave(&ep->lock, flags);
```

下面对发生的事件进行过滤，为什么需要过滤呢？为了性能考虑，ep_insert 向对应监控文件注册的是所有的事件，而实际用户侧订阅的事件未必和内核事件对应。比如，用户向内核订阅了一个套接字的可读事件，在某个时刻套接字的可写事件发生时，并不需要向用户空间传递这个事件。

 复制代码


```
1 /*
2  * Check the events coming with the callback. At this stage, not
3  * every device reports the events in the "key" parameter of the
4  * callback. We need to be able to handle both cases here, hence the
```

```

5  * test for "key" != NULL before the event match test.
6  */
7  if (key && !((unsigned long) key & epi->event.events))
8      goto out_unlock;

```

接下来，判断是否需要把该事件传递给用户空间。

 复制代码

```

1  if (unlikely(ep->ovflist != EP_UNACTIVE_PTR)) {
2      if (epi->next == EP_UNACTIVE_PTR) {
3          epi->next = ep->ovflist;
4          ep->ovflist = epi;
5          if (epi->ws) {
6              /*
7               * Activate epi->ws since epi->ws may get
8               * deactivated at any time.
9               */
10             __pm_stay_awake(ep->ws);
11         }
12     }
13     goto out_unlock;
14 }

```

如果需要，而且该事件对应的 `event_item` 不在 `eventpoll` 对应的已完成队列中，就把它放入该队列，以便将该事件传递给用户空间。

 复制代码

```

1  /* If this file is already in the ready list we exit soon */
2  if (!ep_is_linked(&epi->rdllink)) {
3      list_add_tail(&epi->rdllink, &ep->rdllist);
4      ep_pm_stay_awake_rcu(epi);
5  }

```

我们知道，当我们调用 `epoll_wait` 的时候，调用进程被挂起，在内核看来调用进程陷入休眠。如果该 `epoll` 实例上对应描述字有事件发生，这个休眠进程应该被唤醒，以便及时处理事件。下面的代码就是起这个作用的，`wake_up_locked` 函数唤醒当前 `eventpoll` 上的等待进程。

```

1  /*
2   * Wake up ( if active ) both the eventpoll wait list and the ->poll()
3   * wait list.
4   */
5  if (waitqueue_active(&ep->wq)) {
6      if ((epi->event.events & EPOLLEXCLUSIVE) &&
7          !((unsigned long)key & POLLFREE)) {
8          switch ((unsigned long)key & EPOLLINOUT_BITS) {
9              case POLLIN:
10                 if (epi->event.events & POLLIN)
11                     ewake = 1;
12                 break;
13             case POLLOUT:
14                 if (epi->event.events & POLLOUT)
15                     ewake = 1;
16                 break;
17             case 0:
18                 ewake = 1;
19                 break;
20             }
21         }
22         wake_up_locked(&ep->wq);
23     }

```

查找 epoll 实例

epoll_wait 函数首先进行一系列的检查，例如传入的 maxevents 应该大于 0。

```

1  /* The maximum number of event must be greater than zero */
2  if (maxevents <= 0 || maxevents > EP_MAX_EVENTS)
3      return -EINVAL;
4
5  /* Verify that the area passed by the user is writeable */
6  if (!access_ok(VERIFY_WRITE, events, maxevents * sizeof(struct epoll_event)))
7      return -EFAULT;

```

和前面介绍的 epoll_ctl 一样，通过 epoll 实例找到对应的匿名文件和描述字，并且进行检查和验证。

```

1 /* Get the "struct file *" for the eventpoll file */
2 f = fdget(epfd);
3 if (!f.file)
4     return -EBADF;
5
6 /*
7  * We have to check that the file structure underneath the fd
8  * the user passed to us _is_ an eventpoll file.
9  */
10 error = -EINVAL;
11 if (!is_file_epoll(f.file))
12     goto error_fput;

```

还是通过读取 epoll 实例对应匿名文件的 private_data 得到 eventpoll 实例。


 复制代码

```

1 /*
2  * At this point it is safe to assume that the "private_data" contains
3  * our own data structure.
4  */
5 ep = f.file->private_data;

```

接下来调用 ep_poll 来完成对应的事件收集并传递到用户空间。

 复制代码


```

1 /* Time to fish for events ... */
2 error = ep_poll(ep, events, maxevents, timeout);

```

ep_poll

还记得[第 23 讲](#)里介绍 epoll 函数的时候，对应的 timeout 值可以是大于 0，等于 0 和小于 0 么？这里 ep_poll 就分别对 timeout 不同值的场景进行了处理。如果大于 0 则产生了一个超时时间，如果等于 0 则立即检查是否有事件发生。

 复制代码

```

1 */
2 static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events, int maxevent:


```

```

3 {
4 int res = 0, eavail, timed_out = 0;
5 unsigned long flags;
6 u64 slack = 0;
7 wait_queue_entry_t wait;
8 ktime_t expires, *to = NULL;
9
10 if (timeout > 0) {
11     struct timespec64 end_time = ep_set_mstimeout(timeout);
12     slack = select_estimate_accuracy(&end_time);
13     to = &expires;
14     *to = timespec64_to_ktime(end_time);
15 } else if (timeout == 0) {
16     /*
17      * Avoid the unnecessary trip to the wait queue loop, if the
18      * caller specified a non blocking operation.
19      */
20     timed_out = 1;
21     spin_lock_irqsave(&ep->lock, flags);
22     goto check_events;
23 }

```

接下来尝试获得 eventpoll 上的锁:


 复制代码

```

1 spin_lock_irqsave(&ep->lock, flags);

```

获得这把锁之后，检查当前是否有事件发生，如果没有，就把当前进程加入到 eventpoll 的等待队列 wq 中，这样做的目的是当事件发生时，ep_poll_callback 函数可以把该等待进程唤醒。

 复制代码

```

1 if (!ep_events_available(ep)) {
2     /*
3      * Busy poll timed out. Drop NAPI ID for now, we can add
4      * it back in when we have moved a socket with a valid NAPI
5      * ID onto the ready list.
6      */
7     ep_reset_busy_poll_napi_id(ep);
8
9     /*
10     * We don't have any available event to return to the caller.

```



```


11     * We need to sleep here, and we will be wake up by
12     * ep_poll_callback() when events will become available.
13     */
14     init_waitqueue_entry(&wait, current);
15     __add_wait_queue_exclusive(&ep->wq, &wait);

```

紧接着是一个无限循环, 这个循环中通过调用 `schedule_hrtimeout_range`, 将当前进程陷入休眠, CPU 时间被调度器调度给其他进程使用, 当然, 当前进程可能会被唤醒, 唤醒的条件包括有以下四种:

1. 当前进程超时;
2. 当前进程收到一个 signal 信号;
3. 某个描述字上有事件发生;
4. 当前进程被 CPU 重新调度, 进入 for 循环重新判断, 如果没有满足前三个条件, 就又重新进入休眠。

对应的 1、2、3 都会通过 `break` 跳出循环, 直接返回。

 复制代码

```

1 // 这个循环里, 当前进程可能会被唤醒, 唤醒的途径包括
2 //1. 当前进程超时
3 //2. 当前进程收到一个 signal 信号
4 //3. 某个描述字上有事件发生
5 // 对应的 1.2.3 都会通过 break 跳出循环
6 // 第 4 个可能是当前进程被 CPU 重新调度, 进入 for 循环的判断, 如果没有满足 1.2.3 的条件, 就又
7 for (;;) {
8     /*
9      * We don't want to sleep if the ep_poll_callback() sends us
10     * a wakeup in between. That's why we set the task state
11     * to TASK_INTERRUPTIBLE before doing the checks.
12     */
13     set_current_state(TASK_INTERRUPTIBLE);
14     /*
15     * Always short-circuit for fatal signals to allow
16     * threads to make a timely exit without the chance of
17     * finding more events available and fetching
18     * repeatedly.
19     */
20     if (fatal_signal_pending(current)) {
21         res = -EINTR;
22         break;
23     }
24     if (ep_events_available(ep) || timed_out)


```

```

25         break;
26     if (signal_pending(current)) {
27         res = -EINTR;
28         break;
29     }
30
31     spin_unlock_irqrestore(&ep->lock, flags);
32
33     // 通过调用 schedule_hrtimeout_range, 当前进程进入休眠, CPU 时间被调度器调度给其他进程使
34     if (!schedule_hrtimeout_range(to, slack, HRTIMER_MODE_ABS))
35         timed_out = 1;
36
37     spin_lock_irqsave(&ep->lock, flags);
38 }

```

如果进程从休眠中返回, 则将当前进程从 `eventpoll` 的等待队列中删除, 并且设置当前进程为 `TASK_RUNNING` 状态。


 复制代码

```

1 // 从休眠中结束, 将当前进程从 wait 队列中删除, 设置状态为 TASK_RUNNING, 接下来进入 check_event
2 __remove_wait_queue(&ep->wq, &wait);
3 __set_current_state(TASK_RUNNING);

```

最后, 调用 `ep_send_events` 将事件拷贝到用户空间。

 复制代码

```

1 //ep_send_events 将事件拷贝到用户空间
2 /*
3  * Try to transfer events to user space. In case we get 0 events and
4  * there's still timeout left over, we go trying again in search of
5  * more luck.
6  */
7 if (!res && eavail &&
8     !(res = ep_send_events(ep, events, maxevents)) && !timed_out)
9     goto fetch_events;
10
11
12 return res;

```

ep_send_events

ep_send_events 这个函数会将 ep_send_events_proc 作为回调函数并调用 ep_scan_ready_list 函数，ep_scan_ready_list 函数调用 ep_send_events_proc 对每个已经就绪的事件循环处理。

ep_send_events_proc 循环处理就绪事件时，会再次调用每个文件描述符的 poll 方法，以便确定确实有事件发生。为什么这样做呢？这是为了确定注册的事件在这个时刻还是有效的。

可以看到，尽管 ep_send_events_proc 已经尽可能的考虑周全，使得用户空间获得的事件通知都是真实有效的，但还是有一定的概率，当 ep_send_events_proc 再次调用文件上的 poll 函数之后，用户空间获得的事件通知已经不再有效，这可能是用户空间已经处理掉了，或者其他什么情形。还记得[第 22 讲](#)吗，在这种情况下，如果套接字不是非阻塞的，整个进程将会被阻塞，这也是为什么将非阻塞套接字配合 epoll 使用作为最佳实践的原因。

在进行简单的事件掩码校验之后，ep_send_events_proc 将事件结构体拷贝到用户空间需要的数据结构中。这是通过 __put_user 方法完成的。

 复制代码


```
1 // 这里对一个 fd 再次进行 poll 操作，以确认事件
2 revents = ep_item_poll(epi, &pt);
3
4 /*
5  * If the event mask intersect the caller-requested one,
6  * deliver the event to userspace. Again, ep_scan_ready_list()
7  * is holding "mtx", so no operations coming from userspace
8  * can change the item.
9  */
10 if (revents) {
11     if (__put_user(revents, &uevent->events) ||
12         __put_user(epi->event.data, &uevent->data)) {
13         list_add(&epi->rdllink, head);
14         ep_pm_stay_awake(epi);
15         return eventcnt ? eventcnt : -EFAULT;
16     }
17     eventcnt++;
18     uevent++;
```

Level-triggered VS Edge-triggered

在[前面的文章](#)里，我们一直都在强调 level-triggered 和 edge-triggered 之间的区别。

从实现角度来看其实非常简单，在 ep_send_events_proc 函数的最后，针对 level-triggered 情况，当前的 epoll_item 对象被重新加到 eventpoll 的就绪列表中，这样在下次 epoll_wait 调用时，这些 epoll_item 对象就会被重新处理。

在前面我们提到，在最终拷贝到用户空间有效事件列表中之前，会调用对应文件的 poll 方法，以确定这个事件是不是依然有效。所以，如果用户空间程序已经处理掉该事件，就不会被再次通知；如果没有处理，意味着该事件依然有效，就会被再次通知。

 复制代码


```
1 // 这里是 Level-triggered 的处理，可以看到，在 Level-triggered 的情况下，这个事件被重新加回到
2 // 这样，下一轮 epoll_wait 的时候，这个事件会被重新 check
3 else if (!(epi->event.events & EPOLLET)) {
4     /*
5      * If this file has been added with Level
6      * Trigger mode, we need to insert back inside
7      * the ready list, so that the next call to
8      * epoll_wait() will check again the events
9      * availability. At this point, no one can insert
10     * into ep->rdllist besides us. The epoll_ctl()
11     * callers are locked out by
12     * ep_scan_ready_list() holding "mtx" and the
13     * poll callback will queue them in ep->ovflist.
14     */
15     list_add_tail(&epi->rdllink, &ep->rdllist);
16     ep_pm_stay_awake(epi);
17 }
```

epoll VS poll/select

最后，我们从实现角度来说明一下为什么 epoll 的效率要远远高于 poll/select。

首先，poll/select 先将要监听的 fd 从用户空间拷贝到内核空间，然后在内核空间里面进行处理之后，再拷贝给用户空间。这里就涉及到内核空间申请内存，释放内存等等过程，这在大量 fd 情况下，是非常耗时的。而 epoll 维护了一个红黑树，通过对这棵黑红树进行操作，可以避免大量的内存申请和释放的操作，而且查找速度非常快。

下面的代码就是 poll/select 在内核空间申请内存的展示。可以看到 select 是先尝试申请栈上资源, 如果需要监听的 fd 比较多, 就会去申请堆空间的资源。

 复制代码

```
1 int core_sys_select(int n, fd_set __user *inp, fd_set __user *outp,
2                     fd_set __user *exp, struct timespec64 *end_time)
3 {
4     fd_set_bits fds;
5     void *bits;
6     int ret, max_fds;
7     size_t size, alloc_size;
8     struct fdtable *fdt;
9     /* Allocate small arguments on the stack to save memory and be faster */
10    long stack_fds[SELECT_STACK_ALLOC/sizeof(long)];
11
12    ret = -EINVAL;
13    if (n < 0)
14        goto out_nofds;
15
16    /* max_fds can increase, so grab it once to avoid race */
17    rcu_read_lock();
18    fdt = files_fdtable(current->files);
19    max_fds = fdt->max_fds;
20    rcu_read_unlock();
21    if (n > max_fds)
22        n = max_fds;
23
24    /*
25     * We need 6 bitmaps (in/out/ex for both incoming and outgoing),
26     * since we used fdset we need to allocate memory in units of
27     * long-words.
28     */
29    size = FDS_BYTES(n);
30    bits = stack_fds;
31    if (size > sizeof(stack_fds) / 6) {
32        /* Not enough space in on-stack array; must use kmalloc */
33        ret = -ENOMEM;
34        if (size > (SIZE_MAX / 6))
35            goto out_nofds;
36
37        alloc_size = 6 * size;
38        bits = kvmalloc(alloc_size, GFP_KERNEL);
39        if (!bits)
40            goto out_nofds;
41    }
42    fds.in      = bits;
43    fds.out     = bits +  size;
44    fds.ex      = bits + 2*size;
45    fds.res_in  = bits + 3*size;
```

```
47     fds.res_out = bits + 4*size;
48     fds.res_ex  = bits + 5*size;
49     ...
```

第二，select/poll 从休眠中被唤醒时，如果监听多个 fd，只要其中有一个 fd 有事件发生，内核就会遍历内部的 list 去检查到底是哪一个事件到达，并没有像 epoll 一样，通过 fd 直接关联 eventpoll 对象，快速地把 fd 直接加入到 eventpoll 的就绪列表中。

 复制代码

```
1 static int do_select(int n, fd_set_bits *fds, struct timespec64 *end_time)
2 {
3     ...
4     retval = 0;
5     for (;;) {
6         unsigned long *rinp, *routp, *rexp, *inp, *outp, *exp;
7         bool can_busy_loop = false;
8
9         inp = fds->in; outp = fds->out; exp = fds->ex;
10        rinp = fds->res_in; routp = fds->res_out; rexp = fds->res_ex;
11
12        for (i = 0; i < n; ++rinp, ++routp, ++rexp) {
13            unsigned long in, out, ex, all_bits, bit = 1, mask, j;
14            unsigned long res_in = 0, res_out = 0, res_ex = 0;
15
16            in = *inp++; out = *outp++; ex = *exp++;
17            all_bits = in | out | ex;
18            if (all_bits == 0) {
19                i += BITS_PER_LONG;
20                continue;
21            }
22
23            if (!poll_schedule_timeout(&table, TASK_INTERRUPTIBLE,
24                                     to, slack))
25                timed_out = 1;
26        ...
```

总结

在这次答疑中，我希望通过深度分析 epoll 的源码实现，帮你理解 epoll 的实现原理。

epoll 维护了一棵红黑树来跟踪所有待检测的文件描述字，黑红树的使用减少了内核和用户空间大量的数据拷贝和内存分配，大大提高了性能。

同时，epoll 维护了一个链表来记录就绪事件，内核在每个文件有事件发生时将自己登记到这个就绪事件列表中，通过内核自身的文件 file-eventpoll 之间的回调和唤醒机制，减少了对内核描述字的遍历，大大加速了事件通知和检测的效率，这也为 level-triggered 和 edge-triggered 的实现带来了便利。

通过对比 poll/select 的实现，我们发现 epoll 确实克服了 poll/select 的种种弊端，不愧是 Linux 下高性能网络编程的皇冠。我们应该感谢 Linux 社区的大神们设计了这么强大的事件分发机制，让我们在 Linux 下可以享受高性能网络服务器带来的种种技术红利。




网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 真正的大杀器：异步I/O探索

下一篇 32 | 自己动手写高性能HTTP服务器（一）：设计和思路

 **tamzr**
2019-10-20

老师，能系统讲一下边缘触发饥饿问题和解决方案？面试常遇到，期待ing ~



1



鱼向北游
2019-10-18

select这种是把等待队列和就绪队列混在一起，epoll根据这两种队列的特性用两种数据结构把这两个队列分开，果然在程序世界没有解决不了的事情，如果有，就加一个中间层

作者回复: 你这个理解倒是比较有趣，程序是伟大的。



1



影帝
2019-10-20

我发现看留言学到的更多。🤖

展开 ▾



沉淀的梦想
2019-10-19

缺乏C语言和linux内核基础的人读起这些源码来相当吃力，虽然老师讲得很好



TM
2019-10-18

hi 老师您好，有个问题想咨询下。把 redis 的 backlog 设置为 1，然后在 redis 里 debug sleep 50，然后发起两个请求，一个成功连接，另一个会出『operation timeout』，而不是 connect timeout，然后大概是 26 s，反复试了几次、都是26s左右的时间。很奇怪这个报错是内核爆出来的吗？为什么是26s这个时间呢？扩展是 phpredis，php 底层 socket 超时是 60s。

展开 ▾

作者回复: 你好像问过一次了吧，这个我认为不是内核报出来的，我建议你debug一下。



