作者：董昊 （要转载的同学帮忙把名字和博客链接http://donghao.org/uii/带上，多谢了！）

poll和epoll的使用应该不用再多说了。当fd很多时，使用epoll比poll效率更高。我们通过内核源码分析来看看到底是为什么。

poll剖析

poll系统调用：
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

内核2.6.9对应的实现代码为：
[fs/select.c -->sys_poll]

```
 456 asmlinkage long sys_poll(struct pollfd __user * ufds, unsigned int nfds, long timeout)
 457 {
 458     struct poll_wqueues table;
 459     int fdcount, err;
 460     unsigned int i;
 461     struct poll_list *head;
 462     struct poll_list *walk;
 463
 464     /* Do a sanity check on nfds ... */ /* 用户给的nfds数不可以超过一个struct file结构支持的最大fd数（默认是256）*/
 465     if (nfds > current->files->max_fdset && nfds > OPEN_MAX)
 466         return -EINVAL;
 467
 468     if (timeout) {
 469         /* Careful about overflow in the intermediate values */
 470         if ((unsigned long) timeout < MAX_SCHEDULE_TIMEOUT / HZ)
 471             timeout = (unsigned long)(timeout*HZ+999)/1000+1;
 472         else /* Negative or overflow */
 473             timeout = MAX_SCHEDULE_TIMEOUT;
 474     }
 475
 476     poll_initwait(&table);
```

其中poll_initwait较为关键，从字面上看，应该是初始化变量table，注意此处table在整个执行poll的过程中是很关键的变量。
而struct poll_table其实就只包含了一个函数指针：
[fs/poll.h]

```
  16 /*
  17  * structures and helpers for f_op->poll implementations
  18  */
  19 typedef void (*poll_queue_proc)(struct file *, wait_queue_head_t *, struct poll_table_struct *);
  20
  21 typedef struct poll_table_struct {
  22     poll_queue_proc qproc;
  23 } poll_table;
```

现在我们来看看poll_initwait到底在做些什么
[fs/select.c]

```
  57 void __pollwait(struct file *filp, wait_queue_head_t *wait_address, poll_table *p);
```

```
 58
 59 void poll_initwait(struct poll_wqueues *pwq)
 60 {
 61     &(pwq->pt)->qproc = __pollwait; /*此行已经被我"翻译"了，方便观看*/
 62     pwq->error = 0;
 63     pwq->table = NULL;
 64 }
```

很明显，poll_initwait的主要动作就是把table变量的成员poll_table对应的回调函数置为__pollwait。这个__pollwait不仅是poll系统调用需要，select系统调用也一样是用这个__pollwait，说白了，这是个操作系统的异步操作的"御用"回调函数。当然了，epoll没有用这个，它另外新增了一个回调函数，以达到其高效运转的目的，这是后话，暂且不表。

我们先不讨论__pollwait的具体实现，还是继续看sys_poll:

[fs/select.c -->sys_poll]

```
 478     head = NULL;
 479     walk = NULL;
 480     i = nfds;
 481     err = -ENOMEM;
 482     while(i!=0) {
 483         struct poll_list *pp;
 484         pp = kmalloc(sizeof(struct poll_list)+
 485                 sizeof(struct pollfd)*
 486                 (i>POLLFD_PER_PAGE?POLLFD_PER_PAGE:i),
 487                     GFP_KERNEL);
 488         if(pp==NULL)
 489             goto out_fds;
 490         pp->next=NULL;
 491         pp->len = (i>POLLFD_PER_PAGE?POLLFD_PER_PAGE:i);
 492         if (head == NULL)
 493             head = pp;
 494         else
 495             walk->next = pp;
 496
 497         walk = pp;
 498         if (copy_from_user(pp->entries, ufds + nfds-i,
 499                 sizeof(struct pollfd)*pp->len)) {
 500             err = -EFAULT;
 501             goto out_fds;
 502         }
 503         i -= pp->len;
 504     }
 505     fdcount = do_poll(nfds, head, &table, timeout);
```

这一大堆代码就是建立一个链表，每个链表的节点是一个page大小（通常是4k），这链表节点由一个指向struct poll_list的指针掌控，而众多的struct pollfd就通过struct_list的entries成员访问。上面的循环就是把用户态的struct pollfd拷进这些entries里。通常用户程序的poll调用就监控几个fd，所以上面这个链表通常也就只需要一个节点，即操作系统的一页。但是，当用户传入的fd很多时，由于poll系统调用每次都要把所有struct pollfd拷进内核，所以参数传递和页分配此时就成了poll系统调用的性能瓶颈。

最后一句do_poll，我们跟进去：

[fs/select.c-->sys_poll()-->do_poll()]

```
 395 static void do_pollfd(unsigned int num, struct pollfd * fdpage,
 396     poll_table ** pwait, int *count)
 397 {
 398     int i;
```

```
399
400     for (i = 0; i < num; i++) {
401         int fd;
402         unsigned int mask;
403         struct pollfd *fdp;
404
405         mask = 0;
406         fdp = fdpage+i;
407         fd = fdp->fd;
408         if (fd >= 0) {
409             struct file * file = fget(fd);
410             mask = POLLNVAL;
411             if (file != NULL) {
412                 mask = DEFAULT_POLLMASK;
413                 if (file->f_op && file->f_op->poll)
414                     mask = file->f_op->poll(file, *pwait);
415                 mask &= fdp->events | POLLERR | POLLHUP;
416                 fput(file);
417             }
418             if (mask) {
419                 *pwait = NULL;
420                 (*count)++;
421             }
422         }
423         fdp->revents = mask;
424     }
425 }
426
427 static int do_poll(unsigned int nfds,  struct poll_list *list,
428             struct poll_wqueues *wait, long timeout)
429 {
430     int count = 0;
431     poll_table* pt = &wait->pt;
432
433     if (!timeout)
434         pt = NULL;
435
436     for (;;) {
437         struct poll_list *walk;
438         set_current_state(TASK_INTERRUPTIBLE);
439         walk = list;
440         while(walk != NULL) {
441             do_pollfd( walk->len, walk->entries, &pt, &count);
442             walk = walk->next;
443         }
444         pt = NULL;
445         if (count || !timeout || signal_pending(current))
446             break;
447         count = wait->error;
448         if (count)
449             break;
450         timeout = schedule_timeout(timeout); /* 让current挂起，别的进程跑，timeout到了
以后再回来运行current*/
```

```
451     }
452     __set_current_state(TASK_RUNNING);
453     return count;
454 }
```

注意438行的set_current_state和445行的signal_pending，它们两句保障了当用户程序在调用poll后挂起时，发信号可以让程序迅速推出poll调用，而通常的系统调用是不会被信号打断的。
纵览do_poll函数，主要是在循环内等待，直到count大于0才跳出循环，而count主要是靠do_pollfd函数处理。
注意标红的440-443行，当用户传入的fd很多时（比如1000个），对do_pollfd就会调用很多次，poll效率瓶颈的另一原因就在这里。
do_pollfd就是针对每个传进来的fd，调用它们各自对应的poll函数，简化一下调用过程，如下：

```
struct file* file = fget(fd);
file->f_op->poll ( file, &(table->pt));
```

如果fd对应的是某个socket，do_pollfd调用的就是网络设备驱动实现的poll；如果fd对应的是某个ext3文件系统上的一个打开文件，那do_pollfd调用的就是ext3文件系统驱动实现的poll。一句话，这个file->f_op->poll是设备驱动程序实现的，那设备驱动程序的poll实现通常又是什么样子呢？其实，设备驱动程序的标准实现是：调用poll_wait，即以设备自己的等待队列为参数（通常设备都有自己的等待队列，不然一个不支持异步操作的设备会让人很郁闷）调用struct poll_table的回调函数。
作为驱动程序的代表，我们看看socket在使用tcp时的代码：
[net/ipv4/tcp.c-->tcp_poll]

```
329 unsigned int tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
330 {
331     unsigned int mask;
332     struct sock *sk = sock->sk;
333     struct tcp_opt *tp = tcp_sk(sk);
334
335     poll_wait(file, sk->sk_sleep, wait);
```

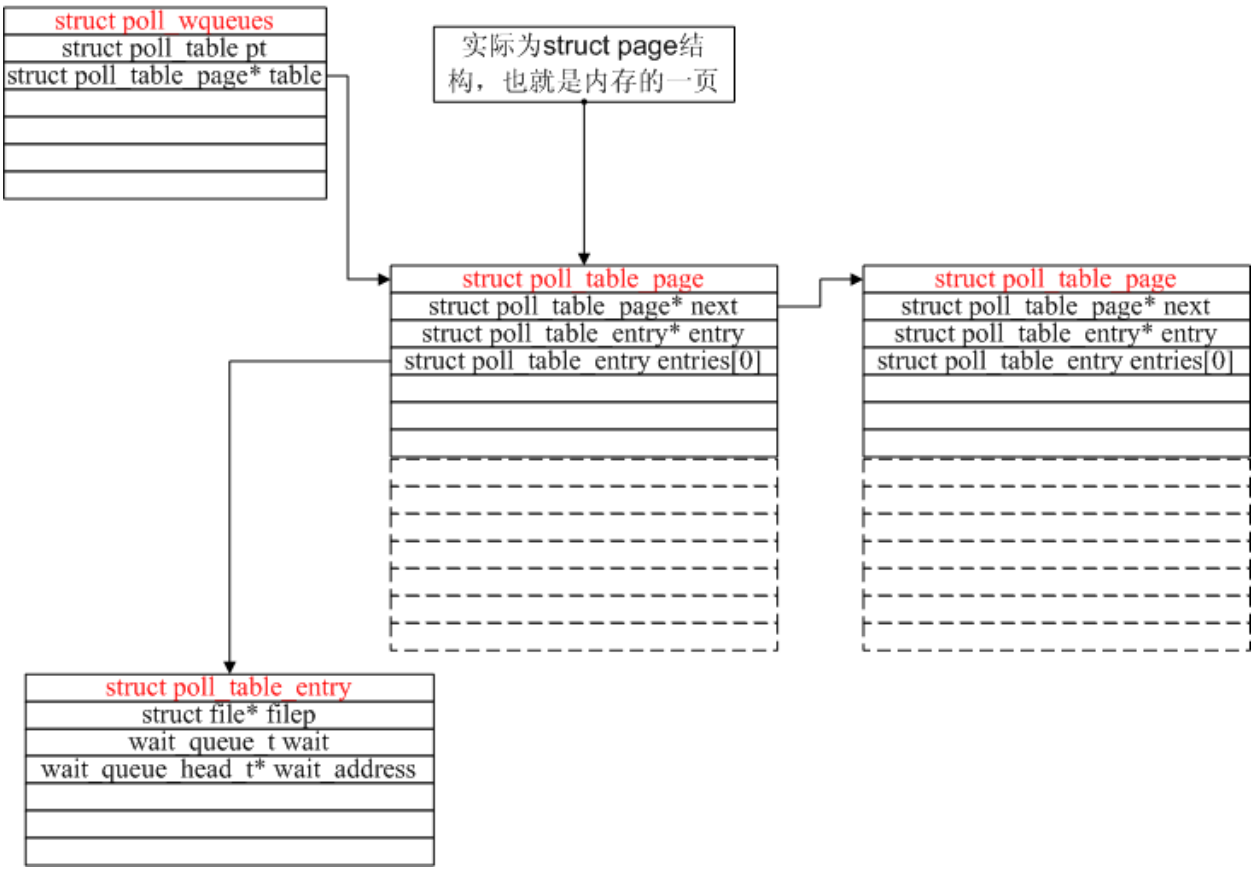
代码就看这些，剩下的无非就是判断状态、返回状态值，tcp_poll的核心实现就是poll_wait，而poll_wait就是调用struct poll_table对应的回调函数，那poll系统调用对应的回调函数就是__poll_wait，所以这里几乎就可以把tcp_poll理解为一个语句：

```
__poll_wait(file, sk->sk_sleep, wait);
```

由此也可以看出，每个socket自己都带有一个等待队列sk_sleep，所以上面我们所说的"设备的等待队列"其实不止一个。
这时候我们再看看__poll_wait的实现：
[fs/select.c-->__poll_wait()]

```
89 void __pollwait(struct file *filp, wait_queue_head_t *wait_address, poll_table *_p)
90 {
91     struct poll_wqueues *p = container_of(_p, struct poll_wqueues, pt);
92     struct poll_table_page *table = p->table;
93
94     if (!table || POLL_TABLE_FULL(table)) {
95         struct poll_table_page *new_table;
96
97         new_table = (struct poll_table_page *) __get_free_page(GFP_KERNEL);
98         if (!new_table) {
99             p->error = -ENOMEM;
100            __set_current_state(TASK_RUNNING);
101            return;
102        }
103        new_table->entry = new_table->entries;
104        new_table->next = table;
105        p->table = new_table;
```

```
106         table = new_table;
107     }
108
109     /* Add a new entry */
110     {
111         struct poll_table_entry * entry = table->entry;
112         table->entry = entry+1;
113         get_file(filp);
114         entry->filp = filp;
115         entry->wait_address = wait_address;
116         init_waitqueue_entry(&entry->wait, current);
117         add_wait_queue(wait_address,&entry->wait);
118     }
119 }
```



__poll_wait的作用就是创建了上图所示的数据结构（一次__poll_wait即一次设备poll调用只创建一个poll_table_entry），并通过struct poll_table_entry的wait成员，把current挂在了设备的等待队列上，此处的等待队列是wait_address，对应tcp_poll里的sk->sk_sleep。
现在我们可以回顾一下poll系统调用的原理了：先注册回调函数__poll_wait，再初始化table变量（类型为struct poll_wqueues)，接着拷贝用户传入的struct pollfd（其实主要是fd），然后轮流调用所有fd对应的poll（把current挂到各个fd对应的设备等待队列上）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上的进程，这时current便被唤醒了。current醒来后离开sys_poll的操作相对简单，这里就不逐行分析了。