<https://blog.csdn.net/weiyuefei/article/details/52242924>

**0.等待队列**

在Linux内核中等待队列有很多用途，可用于中断处理、进程同步及定时。我们在这里只说，进程经常必须等待某些事件的发生。等待队列实现了在事件上的条件等待: 希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制全。因此，等待队列表示一组睡眠的进程，当某一条件为真时，由内核唤醒它们。

等待队列由循环链表实现，由等待队列头（wait\_queue\_head\_t）和等待队列项（wait\_queue）组成，其元素（等待队列项）包含指向进程描述符的指针。每个等待队列都有一个等待队列头(wait queue head),等待队列头是一个类型为wait\_queue\_head\_t的数据结构

定义等待队列头（相关内容可以在linux/include/wait.h中找到）

等待队列头结构体的定义：

struct wait\_queue\_head {

　　spinlock\_t  lock;          //自旋锁变量，用于在对等待队列头

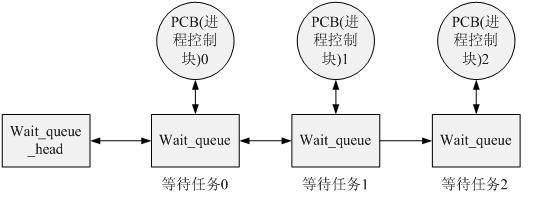
　　struct list\_head task\_list;  // 指向等待队列的list\_head

};

typedef struct \_\_wait\_queue\_head  wait\_queue\_head\_t;

使用等待队列时首先需要定义一个wait\_queue\_head，这可以通过DECLARE\_WAIT\_QUEUE\_HEAD宏来完成，这是静态定义的方法。该宏会定义一个wait\_queue\_head，并且初始化结构中的锁以及等待队列。

     Linux中等待队列的实现思想如下图所示，当一个任务需要在某个wait\_queue\_head上睡眠时，将自己的进程控制块信息封装到wait\_queue中，然后挂载到wait\_queue的链表中，执行调度睡眠。当某些事件发生后，另一个任务（进程）会唤醒wait\_queue\_head上的某个或者所有任务，唤醒工作也就是将等待队列中的任务设置为可调度的状态，并且从队列中删除。



（2）等待队列中存放的是在执行设备操作时不能获得资源而挂起的进程

定义等待对列：

struct wait\_queue {

　　unsigned int flags;  //prepare\_to\_wait()里有对flags的操作，查看以得出其含义

      #define WQ\_FLAG\_EXCLUSIVE        0x01 //一个常数,在prepare\_to\_wait()用于修改flags的值

          void \* private          //通常指向当前任务控制块

          wait\_queue\_func\_t func;    //唤醒阻塞任务的函数 ，决定了唤醒的方式

　　struct list\_head task\_list;    // 阻塞任务链表

};

typedef struct \_\_wait\_queue          wait\_queue\_t;

poll实现分析

**1.select/poll缺点**

 select/poll的缺点在于：  
     1.每次调用时要重复地从用户态读入参数。  
     2.每次调用时要重复地扫描文件描述符。  
     3.每次在调用开始时，要把当前进程放入各个文件描述符的等待队列。在调用结束后，又把进程从各个等待队列中删除。

**2. 内核实现**

**2.1 主要数据结构：**

(1) struct poll\_table\_entry {

        struct file  filp;

        wait\_queue\_t wait;//内部有一个指针指向一个进程

        wait\_queue\_head\_t   wait\_address;//等待队列头部（等待队列有多个wait\_queue\_t组成，通过双链表连接）

};

(2) struct poll\_table\_page {

        struct poll\_table\_page   next;

        struct poll\_table\_entry   entry;

        struct poll\_table\_entry entries[0];

};

(3) struct poll\_wqueues {

       poll\_table pt;//一个函数指针，通常指向\_\_pollwait或null

       struct poll\_table\_page \* table;

       int error;

};

(4) struct poll\_list {

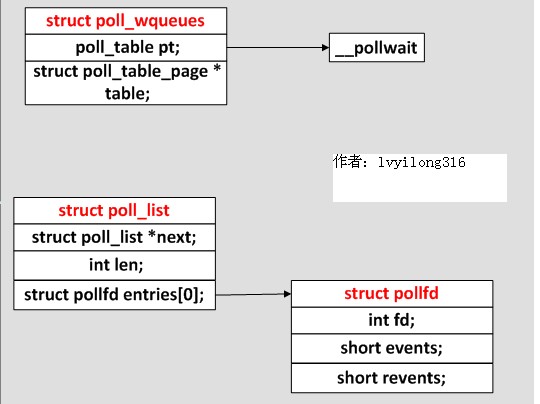
        struct poll\_list \*next;//按内存页连接，因为kmalloc有申请数据限制

        int len;//用户空间传入fd的数量

        struct pollfd entries[0];//存放用户空间存入的数据

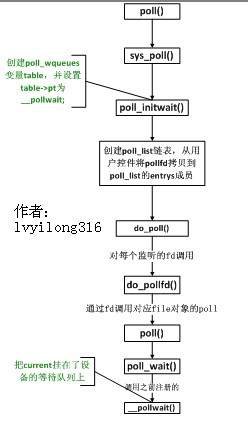
};

typedef void (\*poll\_queue\_proc)(struct file \*, wait\_queue\_head\_t \*, struct poll\_table\_struct \*);  
 typedef struct poll\_table  struct {  
     poll\_queue\_proc qproc;  
 } poll\_table;



**2.2 poll系统调用函数关系总图**

 int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);



**3. 内核2.6.9 poll实现代码分析**

[fs/select.c -->sys\_poll]

 asmlinkage long sys\_poll(struct pollfd \_\_user \* ufds, unsigned int nfds, long timeout)  
 {  
 struct poll\_wqueues table;  
 struct poll\_list \*head;  
 struct poll\_list \*walk;  
 ……

 poll\_initwait(&table);

……

while(i!=0) {  
struct poll\_list \*pp;  
pp = kmalloc(sizeof(struct poll\_list)+ sizeof(struct pollfd)

\*(i>POLLFD\_PER\_PAGE?POLLFD\_PER\_PAGE:i), GFP\_KERNEL));  
if (head == NULL)  
head = pp;  
else  
walk->next = pp;  
walk = pp;  
if (copy\_from\_user(pp->entries, ufds + nfds-i,  
sizeof(struct pollfd)\*pp->len)) {  
err = -EFAULT;  
goto out\_fds;  
}

i -= pp->len;

}

/\*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的指针掌控每个poll\_list的entrys成员指向一个struct pollfd。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。\*/

fdcount = do\_poll(nfds, head, &table, timeout);  
}  
    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。

现在我们来看看poll\_initwait到底在做些什么  
 void \_\_pollwait(struct file \*filp, wait\_queue\_head\_t \*wait\_address, poll\_table \*p);  
 void poll\_initwait(struct poll\_wqueues \*pwq)  
 {

&(pwq->pt)->qproc = \_\_pollwait; /\*设置回调函数\*/

……

}  
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。  
     最后一句do\_poll，我们跟进去：

static int do\_poll(unsigned int nfds, struct poll\_list \*list,struct poll\_wqueues \*wait,

 long timeout)  
{  
   int count = 0;  
   poll\_table\* pt = &wait->pt;  
   for (;;) {  
   struct poll\_list \*walk;  
   set\_current\_state(TASK\_INTERRUPTIBLE);  
   walk = list;  
   while(walk != NULL) {  
   do\_pollfd( walk->len, walk->entries, &pt, &count);  
   walk = walk->next;  
    }  
   pt = NULL;  
   if (count || !timeout || signal\_pending(current))  
    break;  
    count = wait->error;  
    if (count)  
    break;  
    timeout = schedule\_timeout(timeout); /\* 让current挂起，别的进程跑，timeout到了  
以后再回来运行current\*/  
    }  
    \_\_set\_current\_state(TASK\_RUNNING);  
    return count;  
   }

注意set\_current\_state和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while循环，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：

[fs/select.c-->sys\_poll()-->do\_poll()]  
static void do\_pollfd(unsigned int num, struct pollfd \* fdpage, poll\_table \*\* pwait, int \*count)  
 {

……

struct file\* file = fget(fd);  
file->f\_op->poll（file, &(table->pt));

……  
 }

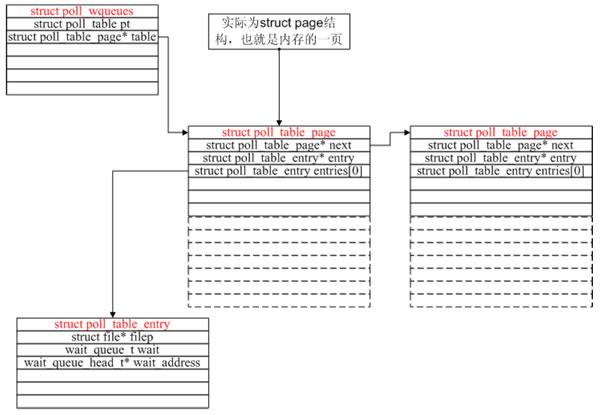
如果fd对应的是某个socket，do\_pollfd调用的就是网络设备驱动实现的poll；如果fd对应的是某个ext3文件系统上的一个打开文件，那do\_pollfd调用的就是ext3文件系统驱动实现的poll。一句话，这个file->f\_op->poll是设备驱动程序实现的，那设备驱动程序的poll实现通常又是什么样子呢？其实，设备驱动程序的标准实现是：调用poll\_wait，即以设备自己的等待队列为参数（通常设备都有自己的等待队列，不然一个不支持异步操作的设备会让人很郁闷）调用struct poll\_table的回调函数。  
作为驱动程序的代表，我们看看socket在使用tcp时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

……

 poll\_wait(file, sk->sk\_sleep, wait);  
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()]  
 void \_\_pollwait(struct file \*filp, wait\_queue\_head\_t \*wait\_address, poll\_table \*\_p)  
 {

……

}



   \_\_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）(瓶颈1)，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）(瓶颈2)。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。