本篇文章是对select、poll、epoll之间的区别进行了详细的分析介绍。需要的朋友参考下

linux提供了select、poll、epoll接口来实现IO复用，三者的原型如下所示，本文从参数、实现、性能等方面对三者进行对比。

复制代码代码如下:

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);  
int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);  
int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);

select、poll、epoll\_wait参数及实现对比  
**1．**select的第一个参数nfds为fdset集合中最大描述符值加1，fdset是一个位数组，其大小限制为\_\_FD\_SETSIZE（1024），位数组的每一位代表其对应的描述符是否需要被检查。  
  
select的第二三四个参数表示需要关注读、写、错误事件的文件描述符位数组，这些参数既是输入参数也是输出参数，可能会被内核修改用于标示哪些描述符上发生了关注的事件。所以每次调用select前都需要重新初始化fdset。  
  
timeout参数为超时时间，该结构会被内核修改，其值为超时剩余的时间。  
   
select对应于内核中的sys\_select调用，sys\_select首先将第二三四个参数指向的fd\_set拷贝到内核，然后对每个被SET的描述符调用进行poll，并记录在临时结果中（fdset），如果有事件发生，select会将临时结果写到用户空间并返回；当轮询一遍后没有任何事件发生时，如果指定了超时时间，则select会睡眠到超时，睡眠结束后再进行一次轮询，并将临时结果写到用户空间，然后返回。  
  
select返回后，需要逐一检查关注的描述符是否被SET（事件是否发生）。  
  
**2．**poll与select不同，通过一个pollfd数组向内核传递需要关注的事件，故没有描述符个数的限制，pollfd中的events字段和revents分别用于标示关注的事件和发生的事件，故pollfd数组只需要被初始化一次。  
  
poll的实现机制与select类似，其对应内核中的sys\_poll，只不过poll向内核传递pollfd数组，然后对pollfd中的每个描述符进行poll，相比处理fdset来说，poll效率更高。  
  
poll返回后，需要对pollfd中的每个元素检查其revents值，来得指事件是否发生。  
  
**3．**epoll通过epoll\_create创建一个用于epoll轮询的描述符，通过epoll\_ctl添加/修改/删除事件，通过epoll\_wait检查事件，epoll\_wait的第二个参数用于存放结果。  
  
epoll与select、poll不同，首先，其不用每次调用都向内核拷贝事件描述信息，在第一次调用后，事件信息就会与对应的epoll描述符关联起来。另外epoll不是通过轮询，而是通过在等待的描述符上注册回调函数，当事件发生时，回调函数负责把发生的事件存储在就绪事件链表中，最后写到用户空间。  
  
epoll返回后，该参数指向的缓冲区中即为发生的事件，对缓冲区中每个元素进行处理即可，而不需要像poll、select那样进行轮询检查。  
  
**select、poll、epoll\_wait性能对比**  
select、poll的内部实现机制相似，性能差别主要在于向内核传递参数以及对fdset的位操作上，另外，select存在描述符数的硬限制，不能处理很大的描述符集合。这里主要考察poll与epoll在不同大小描述符集合的情况下性能的差异。  
  
测试程序会统计在不同的文件描述符集合的情况下，1s内poll与epoll调用的次数。统计结果如下，从结果可以看出，对poll而言，每秒钟内的系统调用数目虽集合增大而很快降低，而epoll基本保持不变，具有很好的扩展性。

|  |  |  |
| --- | --- | --- |
| 描述符集合大小 | poll | epoll |
| 1 | 331598 | 258604 |
| 10 | 330648 | 297033 |
| 100 | 91199 | 288784 |
| 1000 | 27411 | 296357 |
| 5000 | 5943 | 288671 |
| 10000 | 2893 | 292397 |
| 25000 | 1041 | 285905 |
| 50000 | 536 | 293033 |
| 100000 | 224 | 285825 |

**一、连接数**  
我本人也曾经在项目中用过select和epoll,对于select，感触最深的是linux下select最大数目限制(windows 下似乎没有限制)，每个进程的select最多能处理FD\_SETSIZE个FD(文件句柄)，  
如果要处理超过1024个句柄，只能采用多进程了。  
常见的使用slect的多进程模型是这样的： 一个进程专门accept，成功后将fd通过unix socket传递给子进程处理，父进程可以根据子进程负载分派。曾经用过1个父进程+4个子进程 承载了超过4000个的负载。  
这种模型在我们当时的业务运行的非常好。epoll在连接数方面没有限制，当然可能需要用户调用API重现设置进程的资源限制。  
  
**二、IO差别**  
**1、select的实现**  
这段可以结合linux内核代码描述了，我使用的是2.6.28，其他2.6的代码应该差不多吧。  
**先看看select:**  
select系统调用的代码在fs/Select.c下，

复制代码代码如下:

asmlinkage long sys\_select(int n, fd\_set \_\_user \*inp, fd\_set \_\_user \*outp,  
            fd\_set \_\_user \*exp, struct timeval \_\_user \*tvp)  
{  
    struct timespec end\_time, \*to = NULL;  
    struct timeval tv;  
    int ret;  
    if (tvp) {  
        if (copy\_from\_user(&tv, tvp, sizeof(tv)))  
            return -EFAULT;  
        to = &end\_time;  
        if (poll\_select\_set\_timeout(to,  
                tv.tv\_sec + (tv.tv\_usec / USEC\_PER\_SEC),  
                (tv.tv\_usec % USEC\_PER\_SEC) \* NSEC\_PER\_USEC))  
            return -EINVAL;  
    }  
    ret = core\_sys\_select(n, inp, outp, exp, to);  
    ret = poll\_select\_copy\_remaining(&end\_time, tvp, 1, ret);  
    return ret;  
}

前面是从用户控件拷贝各个fd\_set到内核空间，接下来的具体工作在core\_sys\_select中，  
core\_sys\_select->do\_select,真正的核心内容在do\_select里：

复制代码代码如下:

int do\_select(int n, fd\_set\_bits \*fds, struct timespec \*end\_time)  
{  
    ktime\_t expire, \*to = NULL;  
    struct poll\_wqueues table;  
    poll\_table \*wait;  
    int retval, i, timed\_out = 0;  
    unsigned long slack = 0;  
    rcu\_read\_lock();  
    retval = max\_select\_fd(n, fds);  
    rcu\_read\_unlock();  
    if (retval < 0)  
        return retval;  
    n = retval;  
    poll\_initwait(&table);  
    wait = &table.pt;  
    if (end\_time && !end\_time->tv\_sec && !end\_time->tv\_nsec) {  
        wait = NULL;  
        timed\_out = 1;  
    }  
    if (end\_time && !timed\_out)  
        slack = estimate\_accuracy(end\_time);  
    retval = 0;  
    for (;;) {  
        unsigned long \*rinp, \*routp, \*rexp, \*inp, \*outp, \*exp;  
        set\_current\_state(TASK\_INTERRUPTIBLE);  
        inp = fds->in; outp = fds->out; exp = fds->ex;  
        rinp = fds->res\_in; routp = fds->res\_out; rexp = fds->res\_ex;  
        for (i = 0; i < n; ++rinp, ++routp, ++rexp) {  
            unsigned long in, out, ex, all\_bits, bit = 1, mask, j;  
            unsigned long res\_in = 0, res\_out = 0, res\_ex = 0;  
            const struct file\_operations \*f\_op = NULL;  
            struct file \*file = NULL;  
            in = \*inp++; out = \*outp++; ex = \*exp++;  
            all\_bits = in | out | ex;  
            if (all\_bits == 0) {  
                i += \_\_NFDBITS;  
                continue;  
            }  
            for (j = 0; j < \_\_NFDBITS; ++j, ++i, bit <<= 1) {  
                int fput\_needed;  
                if (i >= n)  
                    break;  
                if (!(bit & all\_bits))  
                    continue;  
                file = fget\_light(i, &fput\_needed);  
                if (file) {  
                    f\_op = file->f\_op;  
                    mask = DEFAULT\_POLLMASK;  
                    if (f\_op && f\_op->poll)  
                        mask = (\*f\_op->poll)(file, retval ? NULL : wait);  
                    fput\_light(file, fput\_needed);  
                    if ((mask & POLLIN\_SET) && (in & bit)) {  
                        res\_in |= bit;  
                        retval++;  
                    }  
                    if ((mask & POLLOUT\_SET) && (out & bit)) {  
                        res\_out |= bit;  
                        retval++;  
                    }  
                    if ((mask & POLLEX\_SET) && (ex & bit)) {  
                        res\_ex |= bit;  
                        retval++;  
                    }  
                }  
            }  
            if (res\_in)  
                \*rinp = res\_in;  
            if (res\_out)  
                \*routp = res\_out;  
            if (res\_ex)  
                \*rexp = res\_ex;  
            cond\_resched();  
        }  
        wait = NULL;  
        if (retval || timed\_out || signal\_pending(current))  
            break;  
        if (table.error) {  
            retval = table.error;  
            break;  
        }  
        /\*  
         \* If this is the first loop and we have a timeout  
         \* given, then we convert to ktime\_t and set the to  
         \* pointer to the expiry value.  
         \*/  
        if (end\_time && !to) {  
            expire = timespec\_to\_ktime(\*end\_time);  
            to = &expire;  
        }  
        if (!schedule\_hrtimeout\_range(to, slack, HRTIMER\_MODE\_ABS))  
            timed\_out = 1;  
    }  
    \_\_set\_current\_state(TASK\_RUNNING);  
    poll\_freewait(&table);  
    return retval;  
}

上面的代码很多，其实真正关键的代码是这一句:

复制代码代码如下:

mask = (\*f\_op->poll)(file, retval ? NULL : wait);   
这个是调用文件系统的 poll函数，不同的文件系统poll函数自然不同，由于我们这里关注的是tcp连接，而socketfs的注册在 net/Socket.c里。  
register\_filesystem(&sock\_fs\_type);   
socket文件系统的函数也是在net/Socket.c里：  
static const struct file\_operations socket\_file\_ops = {  
    .owner =    THIS\_MODULE,  
    .llseek =    no\_llseek,  
    .aio\_read =    sock\_aio\_read,  
    .aio\_write =    sock\_aio\_write,  
    .poll =        sock\_poll,  
    .unlocked\_ioctl = sock\_ioctl,  
#ifdef CONFIG\_COMPAT  
    .compat\_ioctl = compat\_sock\_ioctl,  
#endif  
    .mmap =        sock\_mmap,  
    .open =        sock\_no\_open,    /\* special open code to disallow open via /proc \*/  
    .release =    sock\_close,  
    .fasync =    sock\_fasync,  
    .sendpage =    sock\_sendpage,  
    .splice\_write = generic\_splice\_sendpage,  
    .splice\_read =    sock\_splice\_read,  
};

从sock\_poll跟随下去，  
最后可以到 net/ipv4/tcp.c的  
unsigned int tcp\_poll(struct file \*file, struct socket \*sock, poll\_table \*wait)   
这个是最终的查询函数，  
也就是说select 的核心功能是调用tcp文件系统的poll函数，不停的查询，如果没有想要的数据，主动执行一次调度（防止一直占用cpu），直到有一个连接有想要的消息为止。  
从这里可以看出select的执行方式基本就是不同的调用poll,直到有需要的消息为止，如果select 处理的socket很多，这其实对整个机器的性能也是一个消耗。  
**2、epoll的实现**  
epoll的实现代码在 fs/EventPoll.c下，  
由于epoll涉及到几个系统调用，这里不逐个分析了，仅仅分析几个关键点，  
第一个关键点在  
static int ep\_insert(struct eventpoll \*ep, struct epoll\_event \*event,  
             struct file \*tfile, int fd)   
这是在我们调用sys\_epoll\_ctl 添加一个被管理socket的时候调用的函数，关键的几行如下：

复制代码代码如下:

epq.epi = epi;  
    init\_poll\_funcptr(&epq.pt, ep\_ptable\_queue\_proc);  
    /\*  
     \* Attach the item to the poll hooks and get current event bits.  
     \* We can safely use the file\* here because its usage count has  
     \* been increased by the caller of this function. Note that after  
     \* this operation completes, the poll callback can start hitting  
     \* the new item.  
     \*/  
    revents = tfile->f\_op->poll(tfile, &epq.pt);

这里也是调用文件系统的poll函数，不过这次初始化了一个结构，这个结构会带有一个poll函数的callback函数：ep\_ptable\_queue\_proc，  
在调用poll函数的时候，会执行这个callback，这个callback的功能就是将当前进程添加到 socket的等待进程上。

复制代码代码如下:

static void ep\_ptable\_queue\_proc(struct file \*file, wait\_queue\_head\_t \*whead,  
                 poll\_table \*pt)  
{  
    struct epitem \*epi = ep\_item\_from\_epqueue(pt);  
    struct eppoll\_entry \*pwq;  
    if (epi->nwait >= 0 && (pwq = kmem\_cache\_alloc(pwq\_cache, GFP\_KERNEL))) {  
        init\_waitqueue\_func\_entry(&pwq->wait, ep\_poll\_callback);  
        pwq->whead = whead;  
        pwq->base = epi;  
        add\_wait\_queue(whead, &pwq->wait);  
        list\_add\_tail(&pwq->llink, &epi->pwqlist);  
        epi->nwait++;  
    } else {  
        /\* We have to signal that an error occurred \*/  
        epi->nwait = -1;  
    }  
}

**注意到参数** whead 实际上是 sk->sleep，其实就是将当前进程添加到sk的等待队列里，当该socket收到数据或者其他事件触发时，会调用  
sock\_def\_readable 或者sock\_def\_write\_space 通知函数来唤醒等待进程，这2个函数都是在socket创建的时候填充在sk结构里的。  
从前面的分析来看，epoll确实是比select聪明的多、轻松的多，不用再苦哈哈的去轮询了。