zhuanlan.zhihu.com

一文搞懂select、poll和epoll区别

49-61 minutes

---

0 epoll V.S select

- 住校时，你的朋友来找你：
- select版宿管阿姨，带着你的朋友挨个房间找，直到找到你
- epoll版阿姨，会先记下每位同学的房间号， 你的朋友来时，只需告诉你的朋友你住在哪个房间，无需亲自带着你朋友满大楼逐个房间找人

  如果来了10000个人，都要找自己住这栋楼的同学时，select版和epoll版宿管大妈，谁效率高？同理，高并发服务器中，轮询I/O是最耗时操作之一，epoll性能更高也是很明显。
- select的调用复杂度O(n)。如一个保姆照看一群孩子，如果把孩子是否需要尿尿比作网络I/O事件，select就像保姆挨个询问每个孩子：你要尿尿吗？若孩子回答是，保姆则把孩子拎出来放到另外一个地方。当所有孩子询问完之后，保姆领着这些要尿尿的孩子去上厕所（处理网络I/O事件）
- epoll机制下，保姆无需挨个询问孩子是否要尿尿，而是每个孩子若自己需要尿尿，主动站到事先约定好的地方，而保姆职责就是查看事先约定好的地方是否有孩子。若有小孩，则领着孩子去上厕所（网络事件处理）。因此，epoll的这种机制，能够高效的处理成千上万的并发连接，而且性能不会随着连接数增加而下降。

综上所述，select和epoll对比如下表所示

	select	epoll
性能	随着连接数增加，急剧下降。处理成千上万并发连接数时，性能很差。	随着连接数增加，性能基本上没有下降。处理成千上万并发连接时，性能很好。
连接数	连接数有限制，处理的最大连接数不超过1024。如果要处理超过1024个连接数，则需要修改FD_SETSIZE宏，并重新编译 。	连接数无限制。
内在处理机制	线性轮询	回调callback
开发复杂性	低	中

select单个进程可监视的fd数量受到限制，epoll和select都可实现同时监听多个I/O事件的状态。

- select 基于轮询机制
- epoll基于os支持的I/O通知机制。epoll支持水平触发和边沿触发两种模式。

select通过设置或检查存放fd标志位的数据结构进行下一步处理，这带来缺点：

- 单个进程可监视的fd数量被限制，即能监听端口的数量有限 单个进程能打开的最大连接数由FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32^32，同理64位机器上FD_SETSIZE为32^64），当然也可对其修改，然后重新编译内核，但性能可能受影响，这需要进一步测试。一般该数和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认1024个，64位默认2048。

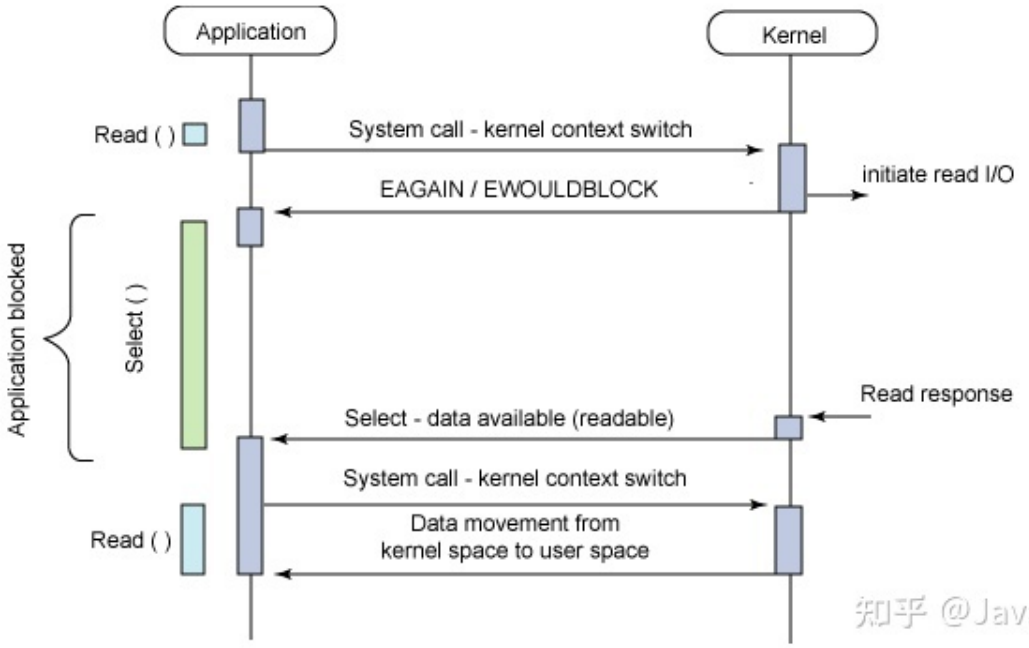
```
javaedge@root:/proc/sys/fs$ cat /proc/sys/fs/file-max
9223372036854775807
```

```
/* commonly an fd_set represents 256 FDs */
#ifndef FD_SETSIZE
#define FD_SETSIZE     256
#endif
```

- 对socket是线性扫描，即轮询，效率较低： 仅知道有I/O事件发生，却不知哪几个流，只会无差异轮询所有流，找出能读/写数据的流进行操作。同时处理的流越多，无差别轮询时间越长 - O(n)。

  当socket较多时，每次select都要通过遍历FD_SETSIZE个socket，不管是否活跃，这会浪费很多CPU时间。若能给 socket 注册某个回调函数，当他们活跃时，自动完成相关操作，即可避免轮询，这就是**epoll与kqueue。**

1.1 调用过程




```
int poll(struct pollfd *fds, int nfds, int timeout)
{
int ret = sys_poll(fds, nfds, timeout);

if (ret < 0) {
    SET_ERRNO(-ret);
    ret = -1;
}
return ret;
```

```
}

asmlinkage long sys_poll(struct pollfd * ufds, unsigned int
nfds, long timeout)
{
int i, j, fdcount, err;
struct pollfd **fds;
struct poll_wqueues table, *wait;
int nchunks, nleft;

/* Do a sanity check on nfds ... */
if (nfds  NR_OPEN)
    return -EINVAL;

// 2. 注册回调函数__pollwait
poll_initwait(&table);
wait = &table;
if (!timeout)
    wait = NULL;

err = -ENOMEM;
fds = NULL;
if (nfds != 0) {
    fds = (struct pollfd **)kmalloc(
        (1 + (nfds - 1) / POLLFD_PER_PAGE) * sizeof(struct
pollfd *),
        GFP_KERNEL);
    if (fds == NULL)
        goto out;
}
```

```
nchunks = 0;
nleft = nfds;
while (nleft  POLLFD_PER_PAGE) { /* allocate complete
PAGE_SIZE chunks */
    fds[nchunks] = (struct pollfd
*)__get_free_page(GFP_KERNEL);
    if (fds[nchunks] == NULL)
        goto out_fds;
    nchunks++;
    nleft -= POLLFD_PER_PAGE;
}
if (nleft) { /* allocate last PAGE_SIZE chunk, only nleft
elements used */
    fds[nchunks] = (struct pollfd
*)__get_free_page(GFP_KERNEL);
    if (fds[nchunks] == NULL)
        goto out_fds;
}

err = -EFAULT;
for (i=0; i < nchunks; i++)
    //
    if (copy_from_user(fds[i], ufds + i*POLLFD_PER_PAGE,
PAGE_SIZE))
        goto out_fds1;
if (nleft) {
    if (copy_from_user(fds[nchunks], ufds +
nchunks*POLLFD_PER_PAGE,
            nleft * sizeof(struct pollfd)))
```

```
        goto out_fds1;
}

fdcount = do_poll(nfds, nchunks, nleft, fds, wait,
timeout);

/* OK, now copy the revents fields back to user space. */
for(i=0; i < nchunks; i++)
    for (j=0; j < POLLFD_PER_PAGE; j++, ufds++)
        __put_user((fds[i] + j)->revents, &ufds->revents);
if (nleft)
    for (j=0; j < nleft; j++, ufds++)
        __put_user((fds[nchunks] + j)->revents, &ufds-
revents);

err = fdcount;
if (!fdcount && signal_pending(current))
    err = -EINTR;

out_fds1:
if (nleft)
    free_page((unsigned long)(fds[nchunks]));
out_fds:
for (i=0; i < nchunks; i++)
    free_page((unsigned long)(fds[i]));
if (nfds != 0)
    kfree(fds);
out:
poll_freewait(&table);
return err;
```

```
}

static int do_poll(unsigned int nfds, unsigned int nchunks,
unsigned int nleft,
struct pollfd *fds[], struct poll_wqueues *wait, long
timeout)
{
int count;
poll_table* pt = &wait->pt;

for (;;) {
    unsigned int i;

    set_current_state(TASK_INTERRUPTIBLE);
    count = 0;
    for (i=0; i < nchunks; i++)
        do_pollfd(POLLFD_PER_PAGE, fds[i], &pt, &count);
    if (nleft)
        do_pollfd(nleft, fds[nchunks], &pt, &count);
    pt = NULL;
    if (count || !timeout || signal_pending(current))
        break;
    count = wait->error;
    if (count)
        break;
    timeout = schedule_timeout(timeout);
}
current->state = TASK_RUNNING;
return count;
}
```

1. 使用copy_from_user从用户空间拷贝fd_set到内核空间
2. 注册回调函数__pollwait

```
void poll_initwait(struct poll_wqueues *pwq)
{
        init_poll_funcptr(&pwq->pt, __pollwait);
        pwq->error = 0;
        pwq->table = NULL;
}
```


1. 遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock_poll，sock_poll根据情况会调用到tcp_poll，udp_poll或datagram_poll）
2. 以tcp_poll为例，核心实现就是__pollwait，即上面注册的回调函数
3. __pollwait，就是把current（当前进程）挂到设备的等待队列，不同设备有不同等待队列，如tcp_poll的等待队列是sk-sk_sleep（把进程挂到等待队列中，并不代表进程已睡眠）。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睡眠的进程，这时current便被唤醒。

```
void __pollwait(struct file *filp, wait_queue_head_t
*wait_address, poll_table *_p)
{
struct poll_wqueues *p = container_of(_p, struct
poll_wqueues, pt);
struct poll_table_page *table = p-table;

if (!table || POLL_TABLE_FULL(table)) {
    struct poll_table_page *new_table;

    new_table = (struct poll_table_page *)
__get_free_page(GFP_KERNEL);
```

```
    if (!new_table) {
        p->error = -ENOMEM;
        __set_current_state(TASK_RUNNING);
        return;
    }
    new_table->entry = new_table->entries;
    new_table->next = table;
    p->table = new_table;
    table = new_table;
}

/* 添加新节点 */
{
    struct poll_table_entry * entry = table->entry;
    table->entry = entry+1;
    get_file(filp);
    entry->filp = filp;
    entry->wait_address = wait_address;
    init_waitqueue_entry(&entry->wait, current);
    add_wait_queue(wait_address,&entry->wait);
}
}

static void do_pollfd(unsigned int num, struct pollfd *
fdpage,
poll_table ** pwait, int *count)
{
int i;

for (i = 0; i < num; i++) {
```

```
    int fd;
    unsigned int mask;
    struct pollfd *fdp;

    mask = 0;
    fdp = fdpage+i;
    fd = fdp-fd;
    if (fd = 0) {
        struct file * file = fget(fd);
        mask = POLLNVAL;
        if (file != NULL) {
            mask = DEFAULT_POLLMASK;
            if (file-f_op && file-f_op-poll)
                mask = file-f_op-poll(file, *pwait);
            mask &= fdp-events | POLLERR | POLLHUP;
            fput(file);
        }
        if (mask) {
            *pwait = NULL;
            (*count)++;
        }
    }
    fdp-revents = mask;
}
}
```

1. poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd_set赋值
2. 若遍历完所有fd，还没返回一个可读写的mask掩码，则调schedule_timeout是调用select的进程（也就是current）进入睡眠。

当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。若超过一定超时时间（schedule_timeout指定），还没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有无就绪的fd

3. 把fd_set从内核空间拷贝到用户空间

1.2 缺点

内核需要将消息传递到用户空间，都需要内核拷贝动作。需要维护一个用来存放大量fd的数据结构，使得用户空间和内核空间在传递该结构时复制开销大。

- 每次调用select，都需把fd集合从用户态拷贝到内核态，fd很多时开销就很大
- 同时，每次调用select都需在内核遍历传递进来的所有fd，fd很多时开销就很大
- select支持的文件描述符数量太小，默认最大支持1024个
- 主动轮询效率很低

2 poll

和select类似，只是描述fd集合的方式不同，poll使用`pollfd`结构而非select的`fd_set`结构。

```
struct pollfd {
int fd;
short events;
short revents;
};
```

管理多个描述符也是进行轮询，根据描述符的状态进行处理，但

poll无最大文件描述符数量的限制。

poll和select同样存在一个缺点：包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，其开销也随着文件描述符数量增加而线性增大。

- 将用户态传入的数组拷贝到内核空间
- 然后查询每个fd对应设备状态：
- 若设备就绪 在设备等待队列中加入一项继续遍历
- 若遍历完所有fd后，都没发现就绪的设备 挂起当前进程，直到设备就绪或主动超时，被唤醒后它又再次遍历fd。这个过程经历多次无意义遍历。

无最大连接数限制，因其基于链表存储，缺点：

- 大量fd数组被整体复制于用户态和内核地址空间间，而不管是否有意义
- 若报告了fd后，没有被处理，则下次poll时会再次报告该fd

所以又有epoll模型。

3 epoll（基于Linux2.4.5）

epoll模型修改主动轮询为被动通知，当有事件发生时，被动接收通知。所以epoll模型注册套接字后，主程序可做其他事情，当事件发生时，接收到通知后再去处理。

可理解为**event poll**，epoll会把哪个流发生哪种I/O事件通知我们。所以epoll是事件驱动（每个事件关联fd），此时我们对这些流的操作都是有意义的。复杂度也降到O(1)。

```
# 事件参数描述链接到文件描述符fd的对象
struct epoll_event {
```

```
__u32 events;
__u64 data;
} EPOLL_PACKED;
```

3.1 触发模式

EPOLL LT和EPOLL ET两种：

- LT，默认的模式（水平触发） 只要该fd还有数据可读，每次 epoll_wait 都会返回它的事件，提醒用户程序去操作
- ET是“高速”模式（边缘触发）

```
public static final int EPOLLET = epollet();
```

只会提示一次，直到下次再有数据流入之前都不会再提示，无论fd中是否还有数据可读。所以ET模式下，read一个fd时，一定要把它的buffer读完，即读到read返回值小于请求值或遇到EAGAIN错误。

epoll使用“事件”就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似回调机制激活该fd，epoll_wait便可收到通知。

ET的意义

若用LT，系统中一旦有大量无需读写的就绪文件描述符，它们每次调用epoll_wait都会返回，这大大降低处理程序检索自己关心的就绪文件描述符的效率。 而采用ET，当被监控的文件描述符上有可读写事件发生时，epoll_wait会通知处理程序去读写。若这次没有把数据全部读写完(如读写缓冲区太小)，则下次调用epoll_wait时，它不会通知你，即只会通知你一次，直到该文件描述符上出现第二次可读写事件才通知你。这比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符。

3.2 优点

- 无最大并发连接的限制，能打开的FD上限远大于1024（1G内存能监听约10万个端口）
- 效率提升，不是轮询，不会随FD数目增加而效率下降。只有活跃可用的FD才会调用callback函数 即Epoll最大优点在于它只关心“活跃”连接，而跟连接总数无关，因此实际网络环境中，Epoll效率远高于select、poll
- 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。
- epoll通过内核和用户空间共享一块内存而实现

  表面上看epoll的性能最好，但在连接数少且都十分活跃情况下，select/poll性能可能比epoll好，毕竟epoll通知机制需要很多函数回调。

  epoll跟select都能提供多路I/O复用。在现在的Linux内核里有都能够支持，epoll是Linux所特有，而select则是POSIX所规定，一般os均有实现。

  select和poll都只提供一个函数：select或poll函数。而epoll提供了三个函数：

3.2.1 epoll_create

创建一个epoll句柄

```
/*
 * It opens an eventpoll file descriptor by allocating
space for "maxfds"
 * file descriptors
 * kernel part of the userspace epoll_create(2)
```

```
 */
asmlinkage int sys_epoll_create(int maxfds)
{
  int error = -EINVAL, fd;
  unsigned long addr;
  struct inode *inode;
  struct file *file;
  struct eventpoll *ep;

  /*
   * eventpoll接口中不可能存储超过MAX_FDS_IN_EVENTPOLL的fd
   */
  if (maxfds > MAX_FDS_IN_EVENTPOLL)
    goto eexit_1;

  /*
   * Creates all the items needed to setup an eventpoll
file. That is,
   * a file structure, and inode and a free file
descriptor.
   */
  error = ep_getfd(&fd, &inode, &file);
  if (error)
    goto eexit_1;

  /*
   * 调用去初始化eventpoll file. 这和"open" file operation
callback一样，因为 inside
   * ep_getfd() we did what the kernel usually does before
invoking
```

```
   * corresponding file "open" callback.
   */
  error = open_eventpoll(inode, file);
  if (error)
    goto eexit_2;

  /* "private_data" 由open_eventpoll()设置 */
  ep = file->private_data;

  /* 分配页给event double buffer */
  error = ep_do_alloc_pages(ep, EP_FDS_PAGES(maxfds + 1));
  if (error)
    goto eexit_2;

  /*
   * 创建event double buffer的一个用户空间的映射，以避免当返回events给调用者时，内核到用户空间的内存复制
   */
  down_write(&current->mm->mmap_sem);
  addr = do_mmap_pgoff(file, 0, EP_MAP_SIZE(maxfds + 1),
PROT_READ,
           MAP_PRIVATE, 0);
  up_write(&current->mm->mmap_sem);
  error = PTR_ERR((void *) addr);
  if (IS_ERR((void *) addr))
    goto eexit_2;

  return fd;

eexit_2:
```

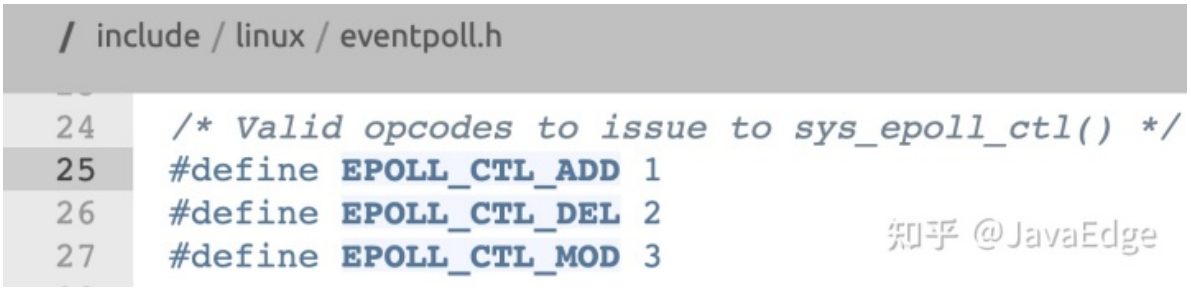
```
  sys_close(fd);
eexit_1:
  return error;
}
```

3.2.2 epoll_ctl

注册要监听的事件类型。

对于第一个缺点，epoll的解决方案在epoll_ctl.c。每次注册新事件到epoll句柄中时（在epoll_ctl中指定EPOLL_CTL_ADD），会把所有fd拷贝进内核，而非在epoll_wait时重复拷贝。epoll保证每个fd在整个过程中**只会拷贝一次**!

```
EPOLL_CTL_ADD：在文件描述符epfd所引用的epoll实例上注册目标文件描述符fd，并将事件事件与内部文件链接到fd
EPOLL_CTL_MOD：更改与目标文件描述符fd相关联的事件
EPOLL_CTL_DEL：从epfd引用的epoll实例中删除目标文件描述符fd。该事件将被忽略，并且可以为NULL
```




```
/*
 * 该方法实现了eventpoll file 的controller接口，可启用insertion/removal/change of file descriptors inside
 * the interest set. It represents the kernel part of the user spcae epoll_ctl(2).

 * @epfd: epoll_createa创建的用于eventpoll的fd
```

```
 * @op: 控制的命令类型
 * @fd: 要操作的文件描述符
 * @events:与fd相关的对象
 */
asmlinkage int sys_epoll_ctl(int epfd, int op, int fd,
unsigned int events)
{
  int error = -EBADF;
  struct file *file;
  struct eventpoll *ep;
  struct epitem *dpi;
  struct pollfd pfd;

  // 获取epfd对应的file实例
  file = fget(epfd);
  if (!file)
    goto eexit_1;

  /*
   * We have to check that the file structure underneath
the file descriptor
   * the user passed to us _is_ an eventpoll file.
   * 检查fd对应文件是否是一个eventpoll文件
   */
  error = -EINVAL;
  if (!IS_FILE_EPOLL(file))
    goto eexit_2;

  /*
   * At this point it is safe to assume that the
```

```
"private_data" contains
   * our own data structure.
   * 获取eventpoll文件中的私有数据，该数据是在epoll_create中创建的
   */
  ep = file->private_data;

  down_write(&ep->acsem);

  pfd.fd = fd;
  pfd.events = events | POLLERR | POLLHUP;
  pfd.revents = 0;

  // 在eventpoll中存储文件描述符信息的红黑树中查找指定的fd对应的epitem实例
  dpi = ep_find(ep, fd);

  error = -EINVAL;
  switch (op) {
  case EP_CTL_ADD:
    // 若要添加的fd不存在，则调用ep_insert()插入红黑树
    if (!dpi)
      error = ep_insert(ep, &pfd);
    else
      // 若已存在，则返回EEXIST错误
      error = -EEXIST;
    break;
  case EP_CTL_DEL:
    if (dpi)
      error = ep_remove(ep, dpi);
```

```
    else
      error = -ENOENT;
    break;
  case EP_CTL_MOD:
    if (dpi) {
      dpi->pfd.events = events;
      error = 0;
    } else
      error = -ENOENT;
    break;
  }

  up_write(&ep->acsem);

eexit_2:
  fput(file);
eexit_1:
  return error;
}
```

3.2.3 epoll_wait

等待事件的产生。

对于第二个缺点，epoll解决方案不像select/poll每次都把current流加入fd对应的设备等待队列，而只在epoll_ctl时把current挂一遍（这一遍必不可少），并为每个fd指定一个回调函数。

当设备就绪，唤醒等待队列上的等待者时，就会调用该回调函数，而回调函数会把就绪fd加入一个就绪链表。

epoll_wait实际上就是在该就绪链表中查看有无就绪fd（利用

schedule_timeout()实现睡一会，判断一会的效果，和select实现中的第7步类似）。

```
/*
 * 实现eventpoll file的event wait接口
 * kernel part of the user space epoll_wait(2)
 *
 * @eptd 文件描述符
 * @events
 * @timeout
 */
asmlinkage int sys_epoll_wait(int epfd, struct pollfd const
**events, int timeout)
{
  int error = -EBADF;
  void *eaddr;
  struct file *file;
  struct eventpoll *ep;
  struct evpoll dvp;

  file = fget(epfd);
  if (!file)
    goto eexit_1;

  /*
   * We have to check that the file structure underneath
the file descriptor
   * the user passed to us _is_ an eventpoll file.
   */
  error = -EINVAL;
```

```
  if (!IS_FILE_EPOLL(file))
    goto eexit_2;

  ep = file->private_data;

  /*
   * It is possible that the user created an eventpoll file
by open()ing
   * the corresponding /dev/ file and he did not perform
the correct
   * initialization required by the old /dev/epoll
interface. This test
   * protect us from this scenario.
   */
  error = -EINVAL;
  if (!atomic_read(&ep->mmapped))
    goto eexit_2;

  dvp.ep_timeout = timeout;
  error = ep_poll(ep, &dvp);
  if (error > 0) {
    eaddr = (void *) (ep->vmabase + dvp.ep_resoff);
    if (copy_to_user(events, &eaddr, sizeof(struct pollfd
*)))
      error = -EFAULT;
  }

eexit_2:
  fput(file);
eexit_1:
```

```
  return error;
}
```

对于第三个缺点，epoll无此限制，其支持FD上限是最大可以打开文件的数目，一般远大于2048。1GB内存机器大约10万左右，具体数目可查看 cat /proc/sys/fs/file-max，这数目和系统内存关系很大。

4 总结

select，poll，epoll都是I/O多路复用机制，即能监视多个fd，一旦某fd就绪（读或写就绪），能够通知程序进行相应读写操作。 但select，poll，epoll本质都是**同步I/O**，因为他们都需在读写事件就绪后，自己负责进行读写，即该读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O实现会负责把数据从内核拷贝到用户空间。

select，poll需自己主动不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需调用epoll_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但它是设备就绪时，调用回调函数，把就绪fd放入就绪链表，并唤醒在epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但select和poll在“醒着”时要遍历整个fd集合，而epoll在“醒着”的时候只需判断就绪链表是否为空，节省大量CPU时间，这就是回调机制带来的性能提升。

select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，且把current往等待队列上挂也只挂一次（在epoll_wait开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少开销。

参考

- Linux下select/poll/epoll机制的比较

- https://www.cnblogs.com/anker/p/326