```
返回-EAGAINstruct seq_file {
        char *buf;
        size_tsize;
        size_tfrom;
        size_tcount;
        size_t pad_until;
        loff_tindex;
        loff_tread_pos;
        u64 version;
        struct mutex lock;
        const struct seq_operations *op;
        int poll_event;
#ifdef CONFIG_USER_NS
        struct user_namespace *user_ns;
#endif
        void *private;
};
```

[buf, buf + size)是seq_file 分配的memory。

初始只分配PAGE_SIZE size,但如果要输出的内容装不下one page，则seq_file会在原来size的基础上加一倍大小申请。

```
        if (!m->buf) {
                m->buf = seq_buf_alloc(m->size = PAGE_SIZE);
                if (!m->buf)
                        return -ENOMEM;
        }
```

```
                error = m->op->show(m, p);
                if (error < 0)
                        break;
                if (unlikely(error)) {
                        error = 0;
                        m->count = 0;
                }
                if (seq_overflow(m))
                        goto Eoverflow;

......

Eoverflow:
        m->op->stop(m, p);
        kvfree(m->buf);
        m->count = 0;
        m->buf = seq_buf_alloc(m->size <<= 1);
        return !m->buf ? -ENOMEM : -EAGAIN;
```

如果在.show()的过程中，通过seq_printf()写入的内容造成buf overflow，则会释放原来的buf，然后把buf size加倍。

这里有个注意的地方，释放掉原来buf中内容前并不会保留原来通过.show() callback循环调用生成的内容。当扩大buf后，seq_file会重新读取这些内容。

比如,原来有item 0 to item 9公10个iteratable item要输出.

第一次读取是这样的

item = seq_file.start()

if(item)

{

  seq_file.show() // 往buf中写入

  item = seq_file.next()

}

seq_file.stop()

如果在item 6时，seq_file.show()把buf撑满了，这时seq_file会调用seq_file.stop()，虽然实际上并没有枚举完所有item，

但seq_file.stop()必须与seq_file.start()匹配调用。然后释放原buf。

在扩大一倍buf后，seq_file会再次启动，从item 0重新开始枚举,就象上面的枚举没有发生过一样。

由此可见，为了输出整个virtual file的内容，buf实际上要象输出的内容一样大！

如果输出内容非常多，是否会令kernel失败呢？

目前seq_file这么处理的。

```
1.  static void *seq_buf_alloc(unsigned long size)
2.  {
3.          void *buf;
4.
5.          buf = kmalloc(size, GFP_KERNEL | __GFP_NOWARN);
6.          if (!buf && size > PAGE_SIZE)
7.                  buf = vmalloc(size);
8.          return buf;
9.  }
```

如果size大于one page，则从vmalloc区域分配。因为kmalloc分配的是物理上连续的kernel memory，

而vmalloc分配的只是virtual address连续的memory,这样如果申请的size比较大，以不会浪费宝贵的

物理上连续的kernel memory，而是用物理上完全不连着的kernel memory来拼出virtual address连续

的kernel memory。这对seq_file功能没有丝毫影响。

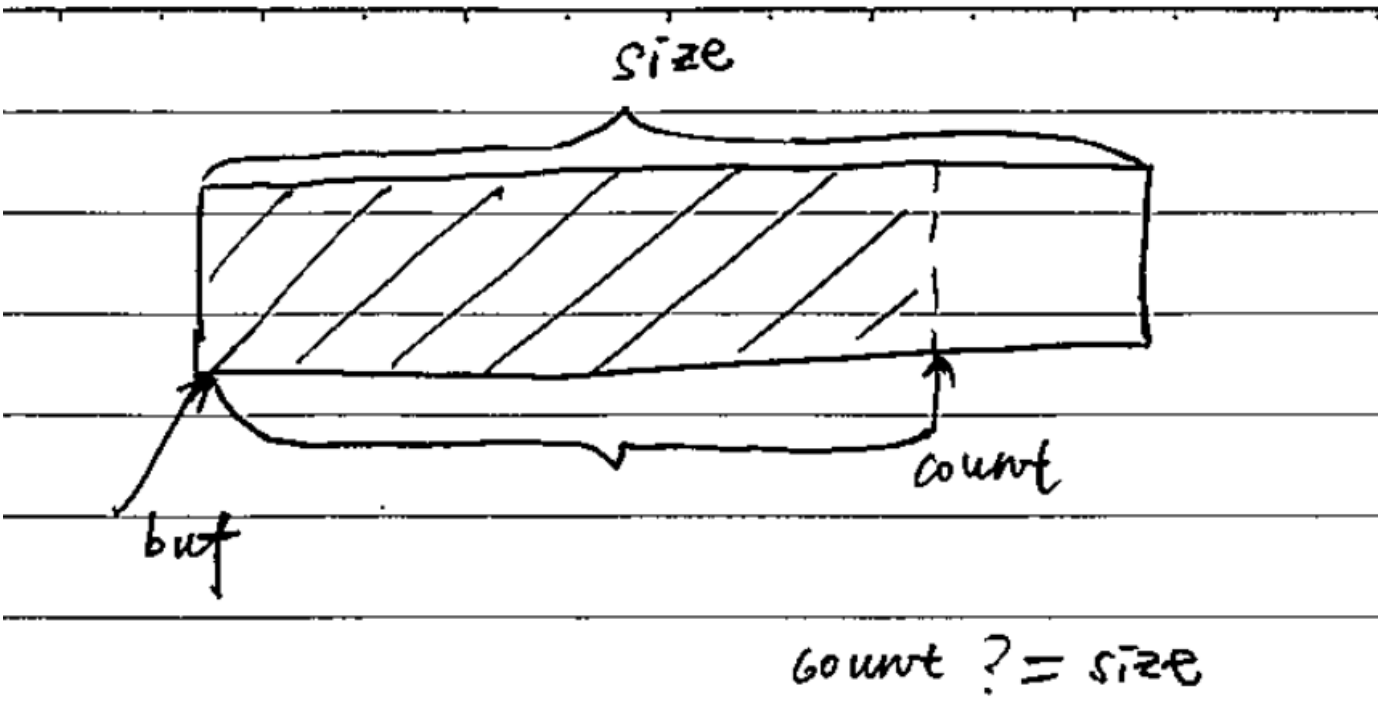
另外如果连vmalloc也分配失败，则会返回ENOMEM，而不是崩溃。即只是对virtual file的access失败，

读取不出内容而已。

return -ENOMEM;

---

count应该是buf中通过.show() callback已经fill的内容。




```
/*
 * seq_files have a buffer which can may overflow. When this happens a larger
 * buffer is reallocated and all the data will be printed again.
 * The overflow state is true when m->count == m->size.
 */
static bool seq_overflow(struct seq_file *m)
{
        return m->count == m->size;
}
```

通过判断m->count是否等于m->size来判断show的内容是否填满了buf

```
/**
 * seq_get_buf - get buffer to write arbitrary data to
 * @m: the seq_file handle
 * @bufp: the beginning of the buffer is stored here
 *
 * Return the number of bytes available in the buffer, or zero if
 * there's no space.
 */
static inline size_t seq_get_buf(struct seq_file *m, char **bufp)
{
        BUG_ON(m->count > m->size);
        if (m->count < m->size)
                *bufp = m->buf + m->count;
        else
                *bufp = NULL;

        return m->size - m->count;
}
```

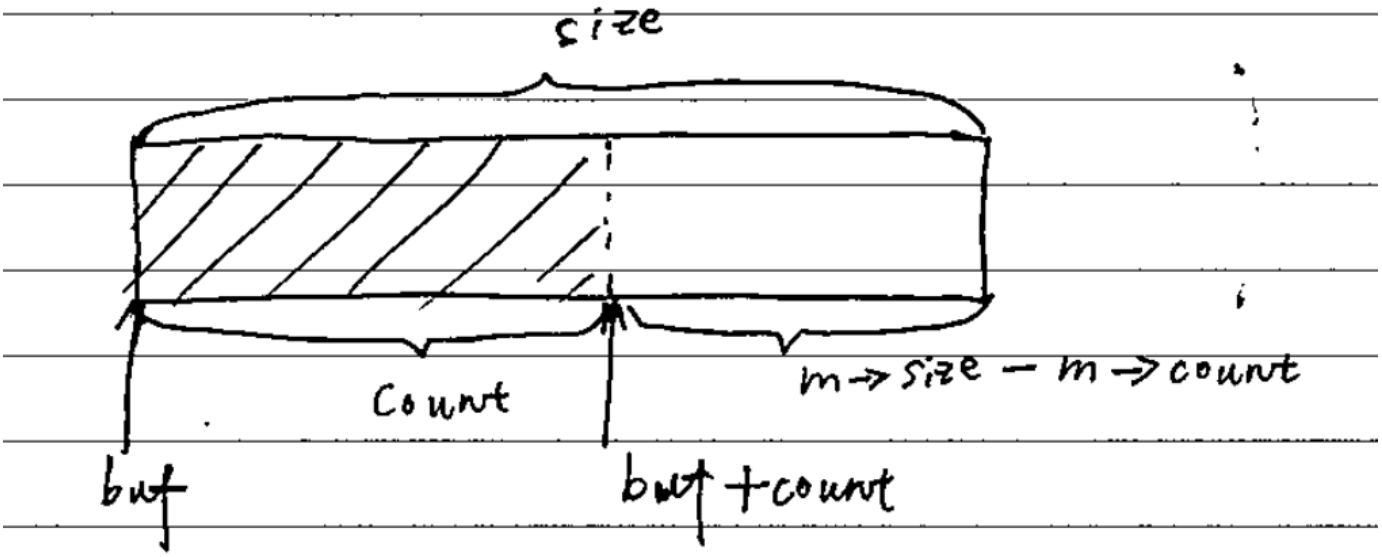
返回buf中可以写入的首地址和还可以写入多少字节。

```
*bufp = m->buf + m->count;
```

如果满了，则返回NULL

```
return m->size - m->count;
```

还可以写入多少bytes。



read_pos为当前已经读过的内容在整个virtual file中的offset

```
ssize_t seq_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{
        struct seq_file *m = file->private_data;
        size_t copied = 0;
        loff_t pos;
        size_t n;
        void *p;
        int err = 0;

        mutex_lock(&m->lock);

        /*
         * seq_file->op->..m_start/m_stop/m_next may do special actions
         * or optimisations based on the file->f_version, so we want to
         * pass the file->f_version to those methods.
         *
         * seq_file->version is just copy of f_version, and seq_file
         * methods can treat it simply as file version.
         * It is copied in first and copied out after all operations.
         * It is convenient to have it as  part of structure to avoid the
         * need of passing another argument to all the seq_file methods.
         */
        m->version = file->f_version;

        /* Don't assume *ppos is where we left it */
        if (unlikely(*ppos != m->read_pos)) {                    ①
                while ((err = traverse(m, *ppos)) == -EAGAIN)    ②
                        ;
                if (err) {                                       ③
                        /* With prejudice... */
                        m->read_pos = 0;
                        m->version = 0;
                        m->index = 0;
                        m->count = 0;
                        goto Done;
                } else {
                        m->read_pos = *ppos;        ④
                }

        }

......
```

①

if (unlikely(*ppos != m->read_pos))

这里*ppos就是virtual file的file pointer，如果不等于seq_file当前读取到的在virtual file

中的position，则实际上要作废原来seq_file所有的读取，要重头开始读取，也就是traverse到指定的file pointer,即这里的*ppos。

从此可看出seq_file对随机读是极其极其极其低效的，它只适合sequence read，所以它的name是seq_file

的原因。

②

如果traverse()返回-EAGAIN，则要重新开始traverse().

```
1.          m->buf = seq_buf_alloc(m->size <<= 1);
2.          return !m->buf ? -ENOMEM : -EAGAIN;
```

返回-EAGAIN的原因是在移动file pointer到*ppos的过程中，发觉buf太小，要加大buf,并再次重头开始读取。

比如一开始buf为one page, 4K (4096)，而*ppos为 17K = 16K + 1K

则第一次traverse()，读了4K，发觉overflow了，释放buf，申请8K

对使用seq_file的client端而言

seq_file.start()

while(seq_file.next())

{

seq_file.show()　到fill 满4K的时候退出loop

}

seq_file.stop()

第二次traverse()，读了8K，发觉overflow了，释放buf，申请16K

```
seq_file.start()
while(seq_file.next())
{
   seq_file.show()   到fill 满16K的时候退出loop
}
seq_file.stop()
```

第三次traverse()，读了16K，发觉overflow了，释放buf，申请32K

```
seq_file.start()
while(seq_file.next())
{
   seq_file.show()   到fill 满17K的时候退出loop
}
seq_file.stop()
```

这次总算成功了，seq_file中size = 32K。

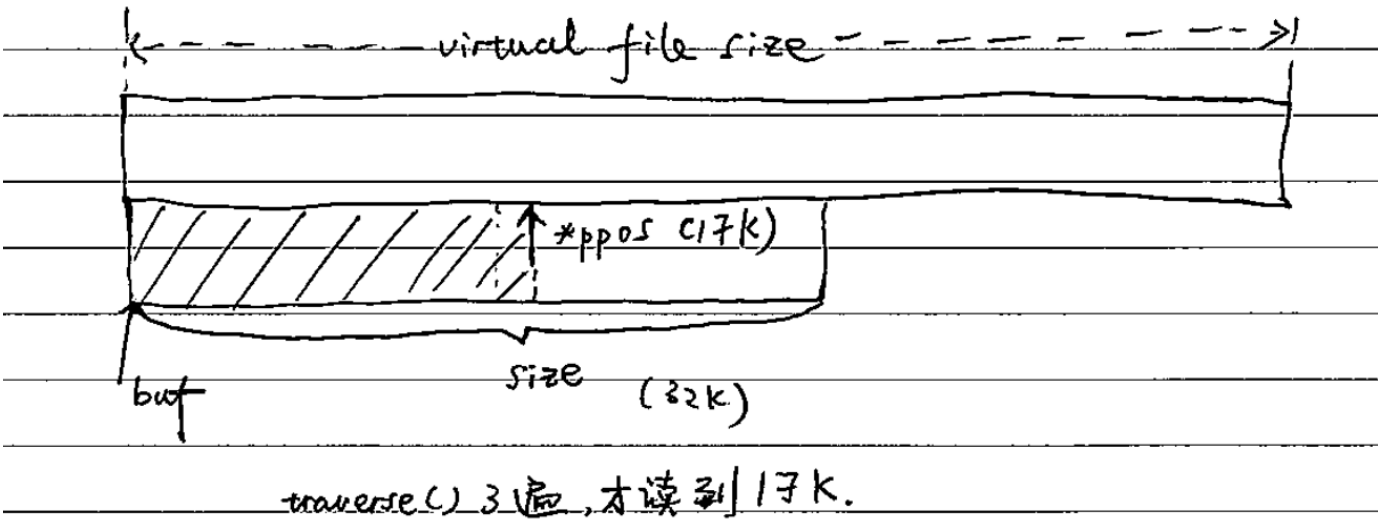
③

如果出错了，成功的话，traverse()返回0

④

m->read_pos = *ppos

反应了在virtual file中的position。



traverse() 3遍，才读到17K.

关于from field

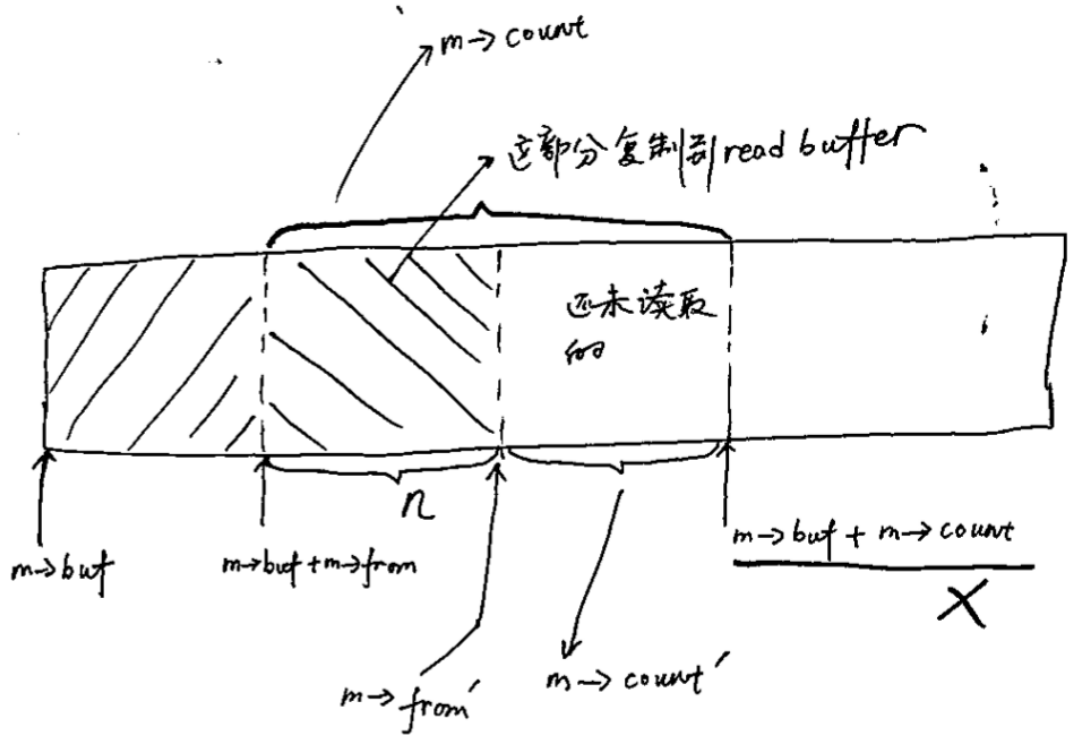
```
ssize_t seq_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{
        struct seq_file *m = file->private_data;
        size_t copied = 0;
        loff_t pos;
        size_t n;
        void *p;
        int err = 0;

        mutex_lock(&m->lock);

        /*
         * seq_file->op->..m_start/m_stop/m_next may do special actions
         * or optimisations based on the file->f_version, so we want to
         * pass the file->f_version to those methods.
         *
         * seq_file->version is just copy of f_version, and seq_file
         * methods can treat it simply as file version.
         * It is copied in first and copied out after all operations.
         * It is convenient to have it as  part of structure to avoid the
         * need of passing another argument to all the seq_file methods.
         */
        m->version = file->f_version;

        /* Don't assume *ppos is where we left it */
        if (unlikely(*ppos != m->read_pos)) {
                while ((err = traverse(m, *ppos)) == -EAGAIN)
                        ;
                if (err) {
                        /* With prejudice... */
                        m->read_pos = 0;
                        m->version = 0;
                        m->index = 0;
                        m->count = 0;
                        goto Done;
                } else {
                        m->read_pos = *ppos;
                }
        }

        /* grab buffer if we didn't have one */
        if (!m->buf) {
                m->buf = seq_buf_alloc(m->size = PAGE_SIZE);
                if (!m->buf)
                        goto Enomem;
        }
        /* if not empty - flush it first */
        if (m->count) {                         ①
                n = min(m->count, size);
                err = copy_to_user(buf, m->buf + m->from, n);
                if (err)
                        goto Efault;
                m->count -= n;
```

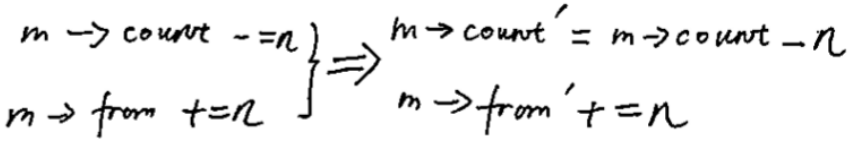
```
                m->from += n;
                size -= n;
                buf += n;
                copied += n;
                if (!m->count)
                        m->index++;
                if (!size)
                        goto Done;
        }

......
```

①

这里m->count > 0，表示m->buf[]中有内容。

则需要从m->buf[]中复制内容到read buffer。稍微有点复杂。





m->from + n = m->from' (new read position in m->buf[])

m->count' = m->count - n

这样整个[m->buf, m->buf + m->buf + m->size)的空间被分成了3部分

1. [m->buf, m->buf + m->from)

这是已经被读取的内容所占空间

2. [m->buf + m->from, m->buf + m->from + m->count)

这是还未被读取的内容所占空间

3. [m->buf + m->from + m->count, m->buf + m->size)

这是没有valid内容的空间

m->count的含义这张图比较准确。

---

index的含义

index用来记录当前seq_file读取到的item index，主要给seq_file->op->start()和seq_file->op->stop()使用。

seq_file假设读取的virtual file中的内容是enumeratable，即不是stream(流式)的。可以item_1, item_2,...,

item_n这样一个个数出来。有点象自然数和实数一样的感觉。

index就是用于计数这些item的！

void * (*start) (struct seq_file *m, loff_t *pos);

void * (*next) (struct seq_file *m, void *v, loff_t *pos);

这两个callback function中的loff_t *pos就是这里index的作用。传递指针是为了callback可以修改这个值。

比如start() callback function，*pos可以告诉它从哪个item开始enumerate,而不一定必定从0开始。

next() callback function可以修改*pos的值，这样下次在调用next()时，就可以接受到上次next()中的修改。

由于enumeration可能被中断，比如m->buf[]空间不够等，所以在seq_file structure中需要记录这个enumerate count。