```c
/**
 *      seq_read -      ->read() method for sequential files.
 *      @file: the file to read from
 *      @buf: the buffer to read to
 *      @size: the maximum number of bytes to read
 *      @ppos: the current position in the file
 *
 *      Ready-made ->f_op->read()
 */
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
                }
        }

        /* 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) {                                 ⑤
                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;
        }
        /* we need at least one record in buffer */
        pos = m->index;                     ⑥
        p = m->op->start(m, &pos);
        while (1) {
                err = PTR_ERR(p);
                if (!p || IS_ERR(p))
                        break;
                err = m->op->show(m, p);
                if (err < 0)                    ⑦
                        break;
                if (unlikely(err))              ⑧
                        m->count = 0;
                if (unlikely(!m->count)) {      ⑨
                        p = m->op->next(m, p, &pos);
                        m->index = pos;
                        continue;
                }
                if (m->count < m->size)         ⑩
                        goto Fill;
                m->op->stop(m, p);              (A)
                kvfree(m->buf);
                m->count = 0;                   (B)
                m->buf = seq_buf_alloc(m->size <<= 1);         (C)
                if (!m->buf)
                        goto Enomem;
                m->version = 0;
                pos = m->index;             (D)
                p = m->op->start(m, &pos);  (E)
        }
        m->op->stop(m, p);
        m->count = 0;                   (F)
        goto Done;
Fill:
        /* they want more? let's try to get some more */
        while (m->count < size) {       (G)
```

```
                size_t offs = m->count;     (H)
                loff_t next = pos;          (I)
                p = m->op->next(m, p, &next);
                if (!p || IS_ERR(p)) {      (J)
                        err = PTR_ERR(p);
                        break;
                }
                err = m->op->show(m, p);
                if (seq_overflow(m) || err) {   (K)
                        m->count = offs;            (L)
                        if (likely(err <= 0))
                                break;
                }
                pos = next;                     (M)
        }
        m->op->stop(m, p);
        n = min(m->count, size);            (N)
        err = copy_to_user(buf, m->buf, n);
        if (err)
                goto Efault;
        copied += n;
        m->count -= n;
        if (m->count)                       (O)
                m->from = n;
        else
                pos++;                          (P)
        m->index = pos;
Done:
        if (!copied)                        (Q)
                copied = err;
        else {
                *ppos += copied;                (R)
                m->read_pos += copied;          (S)
        }
        file->f_version = m->version;
        mutex_unlock(&m->lock);
        return copied;
Enomem:
        err = -ENOMEM;
        goto Done;
Efault:
        err = -EFAULT;
        goto Done;
}
```

①

*ppos != m->read_pos

表示read system call的*ppos与seq_file中记录当前的pos不一致。由于是sequence file，不支持random access，所以需要traverse()来从头开始读取文件，以便把file pointer移动到read()指定的*ppos。

②

在sequence virtual file中移动file pointer是非常低效和费力的，见traverse()的comments。

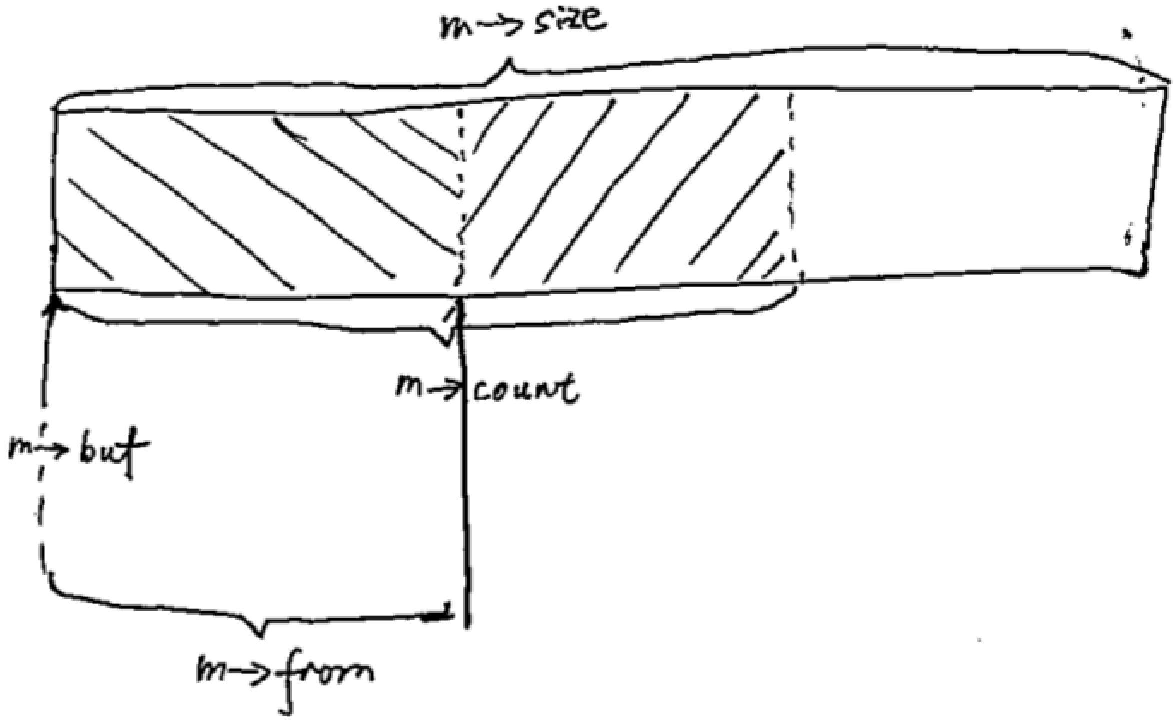
③

如果移动出错了，则reset seq_file变量m中的fields

④

travserse()移动file pointer正确，则令m->read_pos记录当前virtual file的position。

⑤



m->count > 0

表示m->buf[]中有内容。这里需要复制m->buf[]中内容到read buffer。

```
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;
```

如果read buffer size < m->count，则m->buf[]中还会留有部分内容。上图中[m->buf, m->buf + m->from)已经被复制到read buffer，

而[m->buf + m->from, m->buf + m->count)则还有效。

⑥

这里的m->index有可能是在traverse()后的值，即比如traverse()跳过了6个item，则m->start()

显然应该接受的参数是7，而不是依然是0。

⑦

err < 0表示m->op->show() fail，所以要break出while loop，并把err值做为错误值返回。

⑧

err非零，也表示m->op->show()出错了，所以令m->count = 0;这实际上就是令

[m->buf, m->buf + m->count)中的内容都无效了。

⑨

从这里一下，m->op->show()返回0,表示正常返回。但如果m->count为0，表示m->buf中是空的。

即上面的m->op->show()放回成功，但其实什么都没有产生。那就通过m->op->next()来enumerate

下一个item。

```
            m->index = pos;
            continue;
```

并且把下一个enumerate的index记录在m->index，然后就再次去.show()该新的item。

⑩

m->count < m->size

如果在m->op->show()以后，m->buf中的还没有填满整个buf,则跳转到去填满整个buf的逻辑。

(A)

运行到这里，表示[m->buf, m->buf + m->size)已经被充满。所以发起.stop(),暂停enumerate

item。

(B)

无效m->buf中的内容

(C)

m->buf[]空间扩大一倍

(D)

这里的m->index还保留着⑥处的值，即从⑥到(A)的整个.start(), .show(), .next(). stop

都白费了。

pos = m->index;

从原来老地方(old index)处再开始enumerate，但m->buf已经被扩大了。

(E)

再次开始新一轮enumeration。

紧接着⑥的while(1) loop的退出，只有两种情况。

1. 出错了

2. 在m->buf[]中存放了所有的enumeratable items,即m->op->start()或m->op->next()返回

NULL,表示没有进一步的输出了，才会退出。如果在此期间，由于m->buf[]空间耗尽，但还有输出，

则把m->buf[]扩大一倍，并从该while lop开始前的index开始enumerate。由此可见比较低效，

但也没办法，因为是sequence virtual file。

(F)

invalid m->buf[]中的内容

(G)

当m->count < m->size,即m->buf[]还由空闲，就会运行到此。

m->count < size,表示在m->buf[]中的内容少于read buffer.

(H)

offs记录下m->buf[]当前的内容的边界，以便在需要时恢复，比如(L)处

(I)

loff_t next = pos;

这里的pos记录的是上次m->op->next()对index的返回值，即下次希望读取的item的index

(J)

m->op->next()出错了

(K)

seq_overflow(m) || err

最新的m->op->show()填满了m->buf或出错了

(L)

m->count = offs;

恢复(H)处的m->buf[]中内容边界，因为???

(M)

与(I)处呼应，保持enumeration index

(N)

要复制到read buffer中的内容长度

(O)

m->count非零，表示m->buf[]中东西比read buffer要求的多

m->from = n;

这样从[m->buf, m->buf + n)已经被复制到read buffer，而[m->buf + n, m->buf + m->count)

还未被读取。m->from就记录了下次该从哪儿读取。

(P)

m->count为零的情况下，即在m->buf[]中的item都被读取了，下次枚举的item index应该从下

一个开始。

(Q)

copied为复制到read buffer的字节数，如果为零，表示出错了

(R)

没有出错

```
*ppos += copied;
```

更新virtual file的file pointer

(S)

```
m->read_pos += copied;
```

更新seq_file struct中记录的读取到的file pointer。