第十四章 高级I/O

冯 晶 E-mail: gfeng@whu.edu.cn

1. 综述

- 非阻塞I/O
- 记录锁
- 系统V流机制
- I/O多路转接（select和poll函数）
- readv和writev函数
- 存储映照I/O（mmap）

2. 非阻塞I/O

- 阻塞I/O
  - 如果数据不存在，则读文件可能使调用者阻塞（管道，终端设备，网络设备）
  - 如果数据不能立即被接受，则写文件会使调用者阻塞
  - 打开文件也会被阻塞（打开一个终端设备需要等待调制解调器应答；以只写方式打开FIFO，则在没有其他进程已用读方式打开该FIFO时也需要等待）
  - 对已经加上强制性记录锁的文件进行读、写
  - 某些ioctl操作
  - 某些进程间通信

2. 非阻塞I/O

- 非阻塞I/O
  - 调用不会永远阻塞的I/O操作，例如open，read和write。如果这种操作不能完成，则立即出错返回。

- 设置非阻塞
  - 调用open获得描述符时可指定O_NONBLOCK标志
  - 已打开的描述符，调用fcntl打开O_NONBLOCK文件状态标志

```c
#include    <sys/types.h>
#include    <errno.h>
#include    <fcntl.h>
#include    "ourhdr.h"

char    buf[100000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    for (ptr = buf; ntowrite > 0; ) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */
    exit(0);
}
```

2. 非阻塞I/O


```
void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

2. 非阻塞I/O

值	含　义
EAGAIN	接收超时，或者套接字描述符设置为非阻塞，而此时没有数据
EAGAIN/EWOULDBLOCK	此 socket 使用了非阻塞模式，当前情况下没有可接收的连接
EBADF	描述符非法
ECONNABORTED	连接取消
EINTR	信号在合法连接到来之前打断了 accept 的系统调用
EINVAL	socket 没有侦听连接或者地址长度不合法
EMFILE	每个进程允许打开的文件描述符数量最大值已经到达
ENFILE	达到系统允许打开文件的总数量
ENOTSOCK	文件描述符是一个文件，不是 socket
EOPNOTSUPP	引用的 socket 不是流类型 SOCK_STREAM
EFAULT	参数 addr 不可写
ENOBUFS/ENOMEM	内存不足
EPROTO	协议错误
EPERM	防火墙不允许连接

3. 记录锁

- 记录锁（record locking）
  - 一个进程正在读或修改文件的某部分时，可以阻止其他进程修改同一文件区。

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */);
```

F_GETLK
F_SETLK或F_SETLKW

3. 记录锁

- 记录锁（record locking）
  - F_GETLK，决定由flockptr所描述的锁是否被另外一把锁所排斥（阻塞）。
  - F_SETLK，设置由flockptr所描述的锁。
  - F_SETLKW，是F_SETLK的阻塞版本，如果由于存在其他锁，则flockptr所要求的锁不能被创建，则进程睡眠。

3. 记录锁

- 记录锁（record locking）

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
  off_t  l_start;   /* offset in bytes, relative to l_whence */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_len;     /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;     /* returned with F_GETLK */
};
```

l_len=0，表示开始直到最大可能位置

3. 记录锁

- 记录锁（record locking）
  - 不同类型锁之间的兼容性

区域当前有 \ 要求	读锁	写锁
无锁	可以	可以
一把或多把读锁	可以	拒绝
一把写锁	拒绝	拒绝

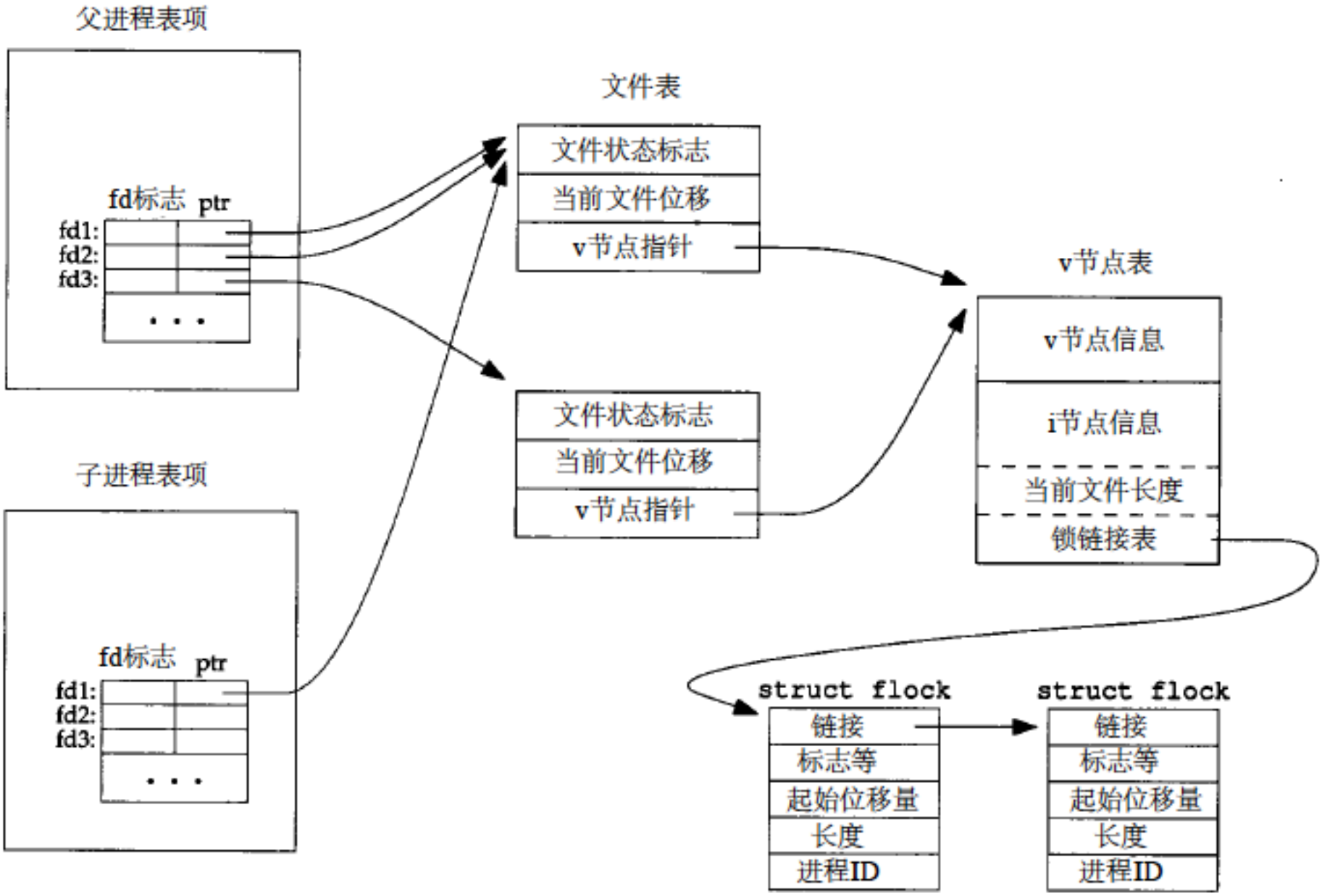
```c
#include    <sys/types.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    return( fcntl(fd, cmd, &lock) );
}

#define read_lock(fd,offset,whence,len) \
            lock_reg(fd,F_SETLK,F_RDLCK,offset,whence,len)
#define readw_lock(fd,offset,whence,len) \
            lock_reg(fd,F_SETLKW,F_RDLCK,offset,whence,len)
#define write_lock(fd,offset,whence,len) \
            lock_reg(fd,F_SETLK,F_WRLCK,offset,whence,len)
#define writew_lock(fd,offset,whence,len) \
            lock_reg(fd,F_SETLKW,F_WRLCK,offset,whence,len)
#define un_lock(fd,offset,whence,len) \
            lock_reg(fd,F_SETLK,F_UNLCK,offset,whence,len)
```

3. 记录锁

- 记录锁的自动继承和释放规则
  - 锁与进程、文件两方面有关。当进程终止时，释放它建立的所有锁；关闭一个描述符时，释放该描述所能访问的文件上的所有锁。
  - 有fork产生的子程序不继承父进程所设置的锁。
  - 在执行exec后，新程序可以继承原执行程序的锁

```
fd1=open(pathname, ...);
read_lock(fd1, ...);
fd2=dup(fd1);
close(fd2);
```

```
fd1=open(pathname, ...);
read_lock(fd1, ...);
fd2=open(pathname, ...);
close(fd2);
```


父进程表项
fd标志
ptr
fd1:
fd2:
fd3:
文件表
文件状态标志
当前文件位移
v节点指针
文件状态标志
当前文件位移
v节点指针
v节点表
v节点信息
i节点信息
当前文件长度
锁链接表
子进程表项
fd标志
ptr
fd1:
fd2:
fd3:
struct flock
链接
标志等
起始位移量
长度
进程ID
struct flock
链接
标志等
起始位移量
长度
进程ID

4. 流

- 流是系统V提供的构造内核设备驱动程序和网络协议包的一种通用方法
  - 系统V的终端界面
  - I/O多路复用中轮询函数的使用
  - 基于流管道和命名管道的实现

4. 流

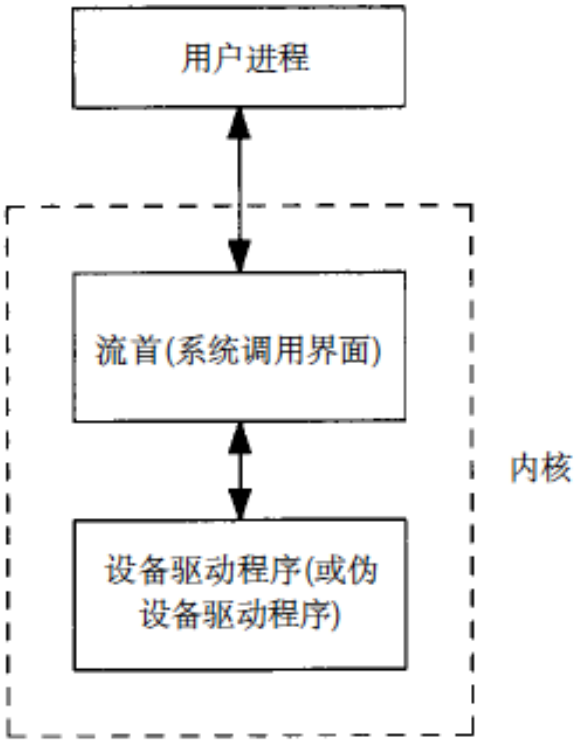


图12-5　一个简单流

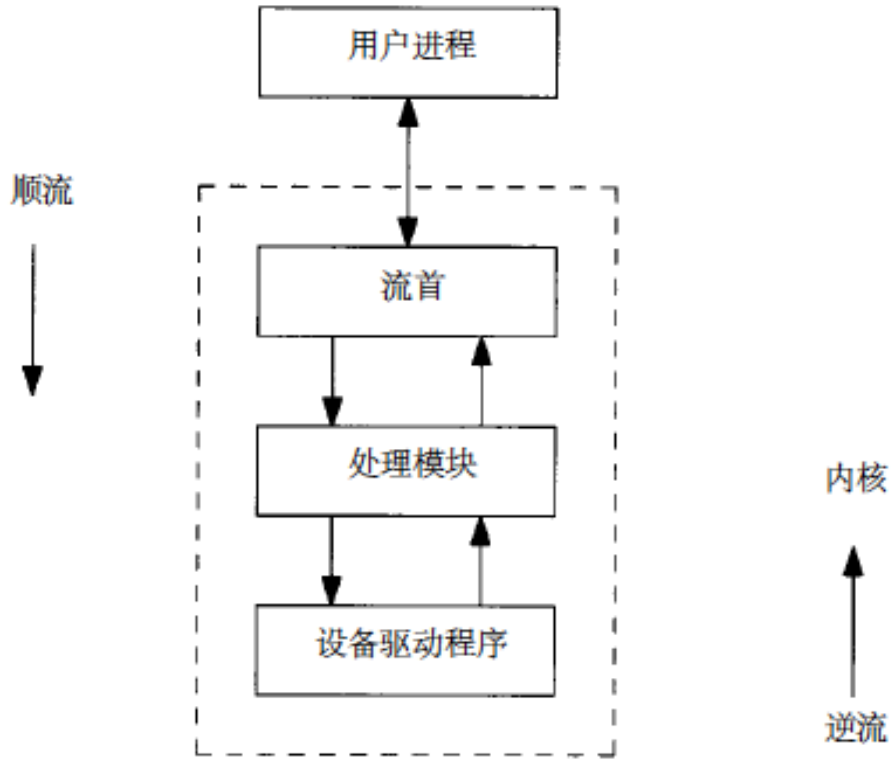


图12-6　具有处理模块的流

4. 流

- 流消息写入函数

```
#include <stropts.h>
int  putmsg(int filedes, const struct strbuf *ctlptr, const struct
     strbuf *dataptr, int flag);
int  putpmsg(int filedes, const struct strbuf *ctlptr, const struct
     strbuf *dataptr, int band, int flag);
```

```
struct strbuf
  int   maxlen;   /* size of buffer */
  int   len;      /* number of bytes currently in buffer */
  char *buf;      /* pointer to buffer */
};
```

4. 流

- 流ioctl操作
  - I_CANPUT

    int isastream(int filedes);

```
#include    <stropts.h>
#include    <unistd.h>

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

4. 流

- 流ioctl操作
  - I_LIST：返回流上所有模块的名字，包括驱动程序

```
int
main(int argc, char *argv[])
{
    int                 fd, i, nmods;
    struct str_list     list;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ( (fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s is not a stream", argv[1]);

            /* fetch number of modules */
    if ( (nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
        err_sys("I_LIST error for nmods");
    printf("#modules = %d\n", nmods);

            /* allocate storage for all the module names */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("calloc error");
    list.sl_nmods = nmods;

            /* and fetch the module names */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("I_LIST error for list");

            /* print the module names */
    for (i = 1; i <= nmods; i++)
        printf("  %s: %s\n", (i == nmods) ? "driver" : "module",
                                list.sl_modlist++);

    exit(0);
}
```

4. 流

- 流ioctl操作
  - I_GWRPOT：获取流写入方式
  - I_SWROPT：设置流写入方式

- 两个写入方式
  - SNDZERO：对管道和FIFO的0长写会造成顺流传送一个0长消息。
  - SNDPIPE：在流上已出错后，若调用write或putmsg，则向调用进程发送SIGPIPE信号。

4. 流

- 流消息读取函数

```
#include <stropts.h>
int  getmsg(int filedes, const struct strbuf *ctlptr, const struct
     strbuf *dataptr, int *flagptr);
int  getpmsg(int filedes, const struct strbuf *ctlptr, const struct
     strbuf *dataptr, int *bandptr, int *flagptr);
```

*flagptr==0，返回下一个消息
*flagptr==RS_HIPRI，只返回高优先权消息

4. 流

- 流ioctl操作
  - I_GRDOPT：获取流读取方式
  - I_SRDOPT：设置流读取方式

- 三种读方式
  - RNORM，普通，字节流方式，是默认方式
  - RMSGN，消息不删除方式
  - RMSGD，消息删除方式，某次读了消息的一部分，则整个消息被删除

4. 流

- 将标准输入复制到标准输出

```
#define BUFFSIZE    8192

int
main(void)
{
    int             n, flag;
    char            ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf   ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0;       /* return any message */
        if ( (n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
                        flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}
```

5. I/O多路转接

- select函数

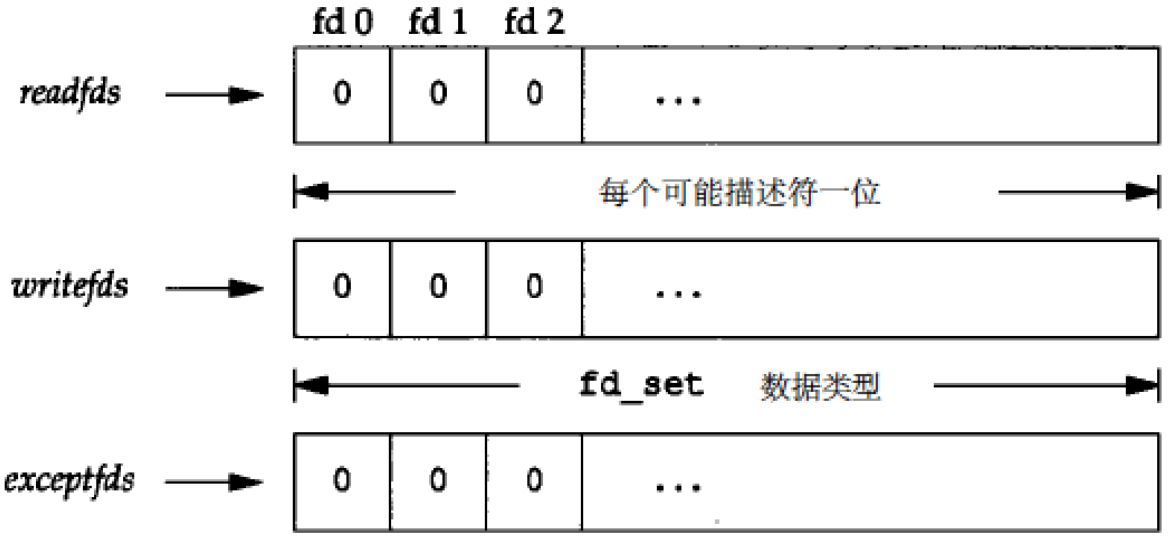
```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int  select (int maxfdpl,  fd_set *readfds, fd_set *writefds, fd_set
    *exceptfds, struct timeval *tvptr);
```

NULL，阻塞等待
0，立即返回
其它，超时前等待

```
struct  timeval{
    long   tv_sec;   /* seconds */
    long   tv_usec; /* and microseconds */
};
```

5. I/O多路转接

- select函数

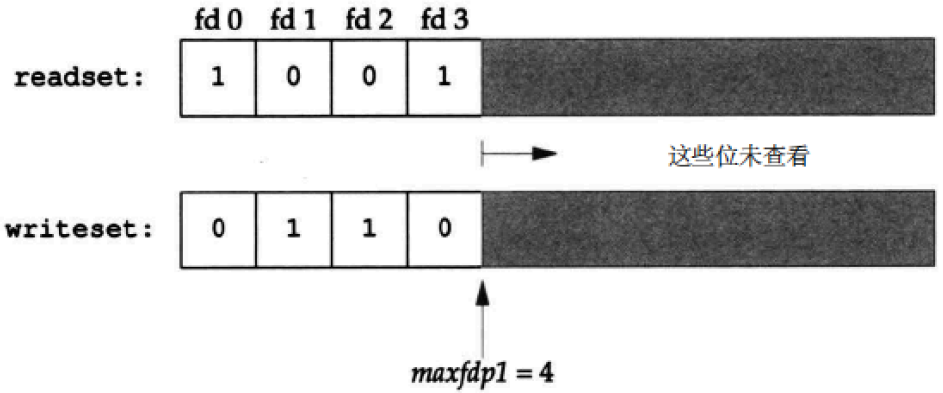



```
FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);   /* turn on bit for fd in fdset */
FD_CLR(int fd, fd_set *fdset);   /* turn off bit for fd in fdset */
FD_ISSET(int fd, fd_set *fdset); /* test bit for fd in fdset */
```

5. I/O多路转接

- select函数

```
fd_set readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);

FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);

select (4, &readset, &writeset, NULL, NULL);
```



5. I/O多路转接

- poll函数

INFTIM，阻塞等待
0，立即返回
>0，等待timeout毫秒

```
#include <stropts.h>
#include <poll.h>

int  poll(struct pollfd  fdarray[], unsigned long nfds, int timeout);
```

```
struct pollfd {
  int     fd;         /* file descriptor to check, or < 0 to ignore */
  short   events;     /* events of interest on fd */
  short   revents;    /* events that occurred on fd */
};
```

5. I/O多路转接

- poll函数

名　称	对events的输入	从revents得到的结果	说　明
POLLIN	•	•	可读除高优级外的数据，不阻塞
POLLRDNORM	•	•	可读普通(优先波段0 数据，不阻塞
POLLRDBAND	•	•	可读0优先波段数据，不阻塞
POLLPRI	•	•	可读高优先级数据，不阻塞
POLLOUT	•	•	可与普通数据，不阻塞
POLLWRNORM	•	•	与POLLOUT相同
POLLWRBAND	•	•	可写非0优先波段数据，不阻塞
POLLERR		•	已出错
POLLHUP		•	已挂起
POLLNVAL		•	此描述符并不引用一打开文件

5. I/O多路转接

- poll函数
  - 产生SIGPOLL信号的条件

常　数	说　明
S_INPUT	非高优先级的消息已到达
S_RDNORM	一普通消息已到达
S_RDBAND	一0优先波段消息已到达
S_BANDURG	若此常数说明为S_RDBAND，则当一非0优先波段消息到达时产生SIGURG信号而非SIGPOLL
S_HIPRI	一高优先级消息已到达
S_OUTPUT	写队列不再满
S_WRNORM	与S_OUTPUT一样
S_WRBAND	可发送一非0优先波段消息
S_MSG	包含SIGPOLL信号的流信号消息已到达
S_ERROR	M_ERROR消息已到达
S_HANGUP	M_HANGUP消息已到达

6. 散布读和聚集写

- readv函数和writev函数

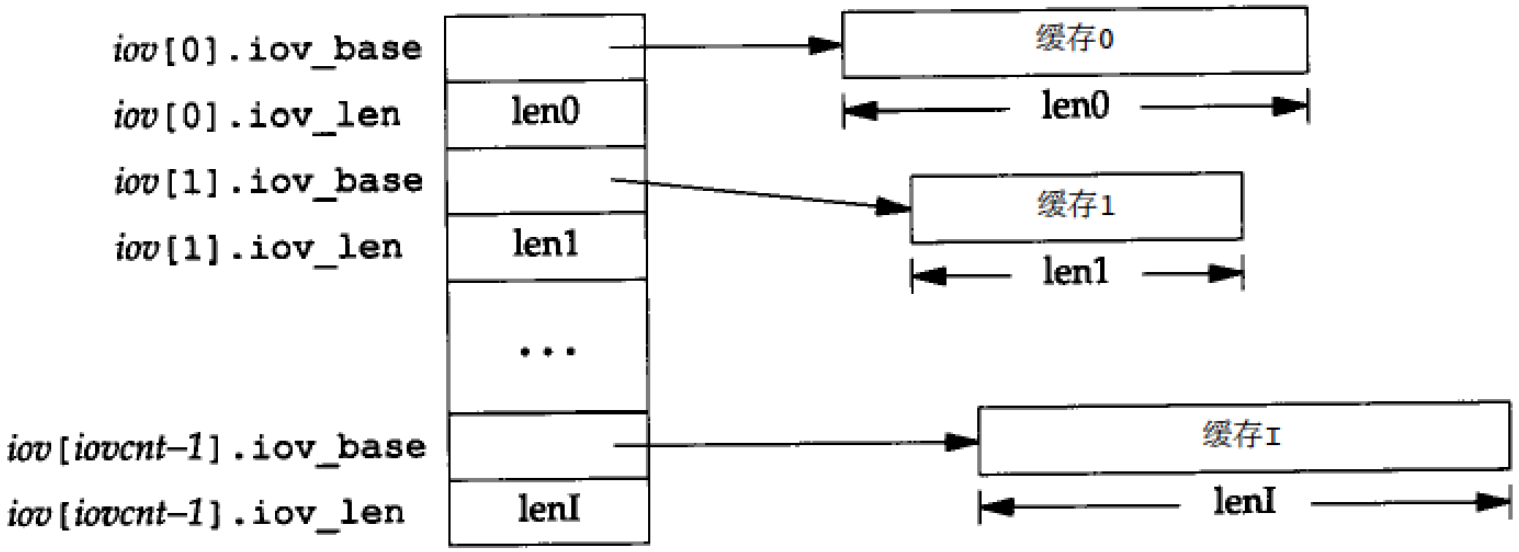
```
#include <sys/types.h>
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovet iov[], int iovcnt);
ssize_t writev(int filedes, const struct iovet iov[], int iovcnt);
```

```
struct iovec {
  void    *iov_base;    /* starting address of buffer */
  size_t   iov_len;     /* size of buffer */
};
```

*iov*数组中的元素数由*iovcnt*说明。

6. 散布读和聚集写

- iovec结构



7. readn和writen

- readv函数和writev函数

```
ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t writen(int filedes, void *buff, size_t nbytes);
```

7. readn和writen

- readv函数和writev函数

```
ssize_t                         /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
    size_t      nleft;
    ssize_t     nwritten;
    const char  *ptr;

    ptr = vptr; /* can't do pointer arithmetic on void* */
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0)
            return(nwritten);       /* error */

        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n);
}
```

7. readn和writen

- readv函数和writev函数

```
ssize_t                         /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char    *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0)
            return(nread);      /* error, return < 0 */
        else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}
```

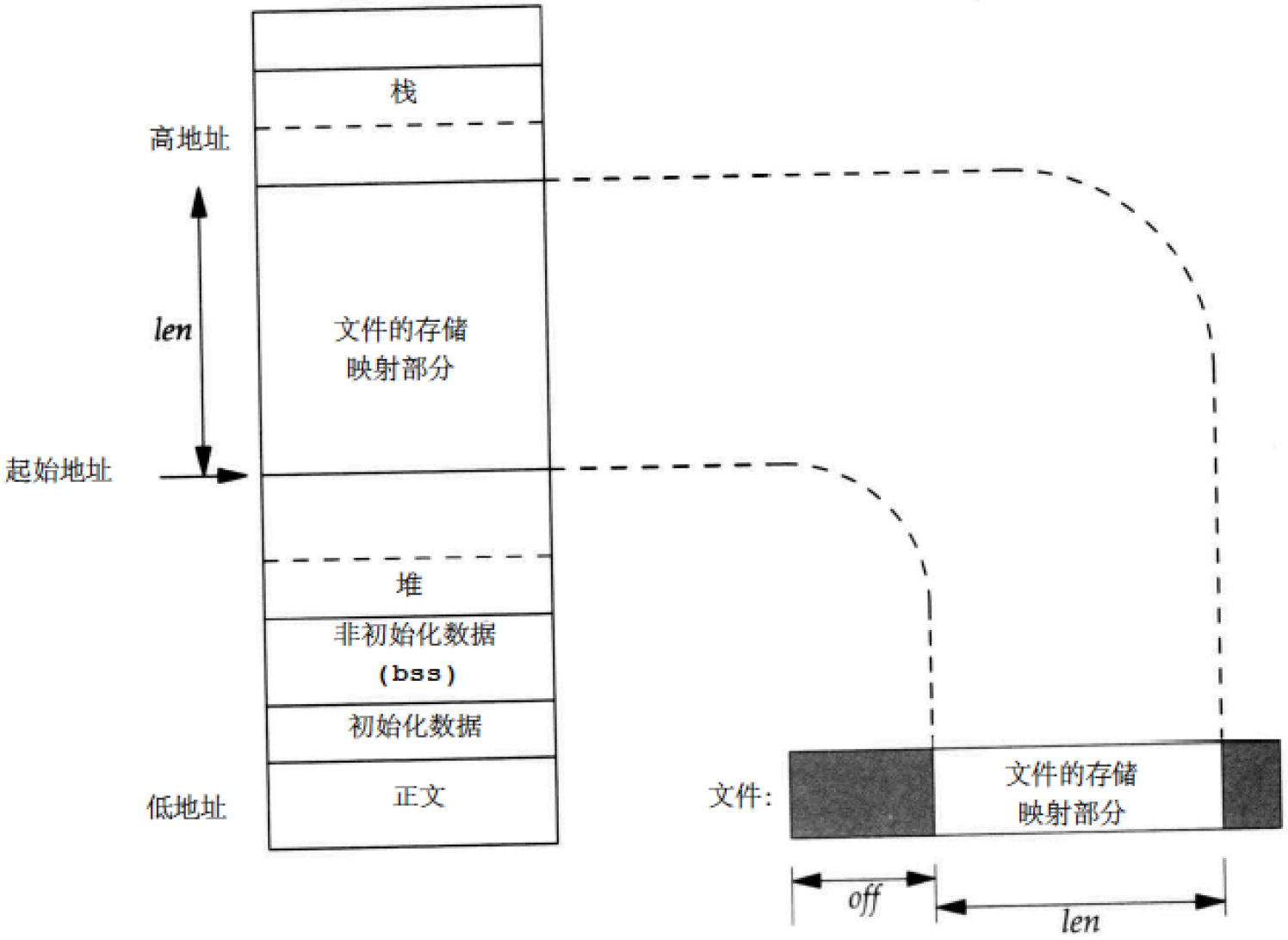
8. 存储映射I/O

- mmap函数

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap(caddr_t addr, size_t len, int prot, int flag, int
    filedes, off_t off);
```

prot	说　明
PROT_READ	区域可读
PROT_WRITE	区域可写
PROT_EXEC	区域可执行
PROT_NONE	区域可存取(4.3+BSD无)

8. 存储映射I/O


栈
高地址
len
文件的存储
映射部分
起始地址
堆
非初始化数据
(bss)
初始化数据
低地址
正文
文件:
文件的存储
映射部分
off
len