阻塞IO和非阻塞IO

1．什么叫阻塞IO?什么是非阻塞IO?

阻塞与非阻塞都是表示的函数的特性，并不是一种具有实体的结构，阻塞IO表示在遇到需要无限制的等待下去[读无数据，写无空间，文件锁已被加锁，管道特性，套接字特性等]的读写操作的时候就等下去，非阻塞IO表示遇到这种情况直接返回出错，并不会导致系统的阻塞。

2．如何实现非阻塞IO?

给文件描述符设置O_NONBLOCK标示

记录锁

1. 为什么我们需要记录锁？

   防止多方读写一个文件造成时序竟态

2. 记录锁的功能

   当一个进程正在读写一个文件的时候，为了防止其他用户破坏数据，可以对文件或者文件的一部分加上记录锁。

3. fcntl记录锁

   ```
   int fcntl(int fd, int cmd, strict flock *fp);
   ```

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

1. 多进程之间读共享，写独占， 一个进程之间新的锁会覆盖旧的锁[不会产生自锁现象]
2. 锁可以越过尾端加锁，但是不可以反向越过头部
3. l_len = 0; 不仅意味着加锁到文件尾部，还意味着后续尾部增加的字符也会被纳入锁的范围
4. 加读锁，文件必须读打开，加写锁，文件必须写打开

记录锁

F_SETLK (struct flock *)
Acquire a lock (when l_type is **F_RDLCK** or **F_WRLCK**) or release a lock (when l_type is **F_UNLCK**) on the bytes specified by the l_whence, l_start, and l_len fields of lock. If a conflicting lock is held by another process, this call returns -1 and sets errno to **EACCES** or **EAGAIN**. (The error returned in this case differs across implementations, so POSIX requires a portable application to check for both errors.)

F_SETLKW (struct flock *)
As for **F_SETLK**, but if a conflicting lock is held on the file, then wait for that lock to be released. If a signal is caught while waiting, then the call is interrupted and (after the signal handler has returned) returns immediately (with return value -1 and errno set to **EINTR**; see **signal**(7)).

F_GETLK (struct flock *)
On input to this call, lock describes a lock we would like to place on the file. If the lock could be placed, **fcntl**() does not actually place it, but returns **F_UNLCK** in the l_type field of lock and leaves the other fields of the structure unchanged.

If one or more incompatible locks would prevent this lock being placed, then **fcntl**() returns details about one of those locks in the l_type, l_whence, l_start, and l_len fields of lock. If the conflicting lock is a traditional (process-associated) record lock, then the l_pid field is set to the PID of the process holding that lock. If the conflicting lock is an open file description lock, then l_pid is set to -1. Note that the returned information may already be out of date by the time the caller inspects it.

记录锁

锁的隐含继承和释放

1. 锁与进程和文件两者相关联
   当进程结束，会释放建立的锁
   无论一个描述符何时关闭，该进程通过这一描述符引用的文件上的任何一把锁都会释放
2. fork产生的子进程不继承父进程设置的锁

3. exec后，如果没有设置文件描述符的close_on_exec，新程序将继承原有的锁

在同一个进程内部，只要两个文件描述符指向同一个文件表项，那么关闭任何一个，都会释放该进程在文件上持有的任何锁

记录锁

建议性锁和强制性锁

建议性锁是针对君子而言，就是可以访问该文件的所有进程可以按照一致的方法处理记录锁，这样的进程叫君子，一系列进程叫合作进程。对于懂的合作的君子来说，建议性锁就已经足够了。

但是如果有进程绕过记录锁而去文件中破坏数据，那就要采用强制性锁来实现。

强制性锁会使得内核检查每一个open,read和write， 验证调用进程是否违背了正在访问文件上的某一把锁。

如何实现强制性锁：关闭组执行位，打开设置组ID位

IO多路转接

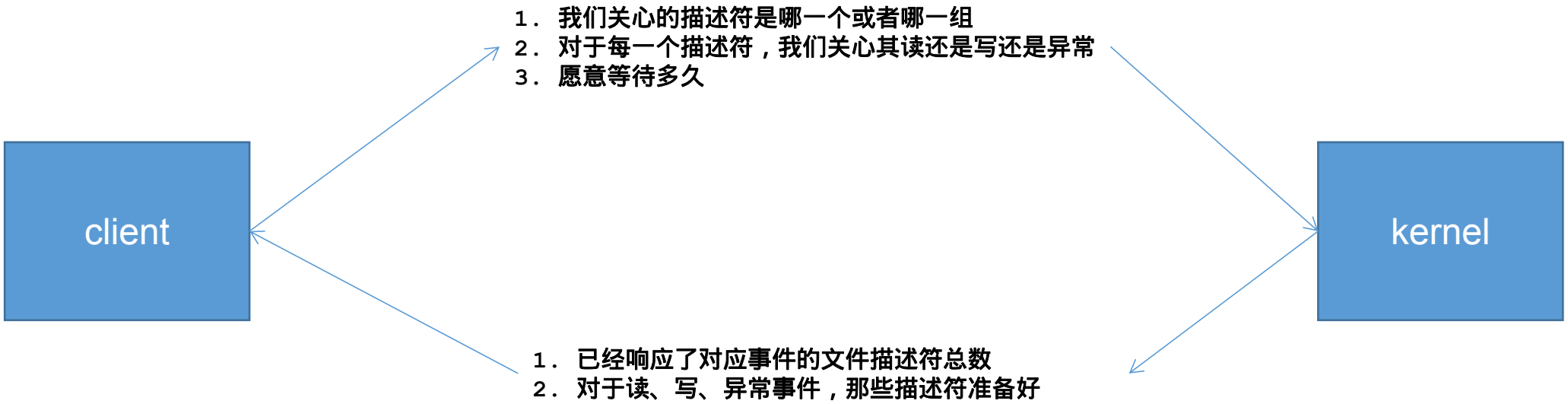
1．为什么需要IO多路转接？

当我们读取多个时间上没有明显先后顺序的来源的输入的时候，怎么在时间层面上统筹兼顾，采用传统的多个进程和多线程处理手段无疑增加了编程的复杂性，所以我们需要IO多路转接。在这项技术中，我们可以把感兴趣的文件描述符统统扔到一张表里，然后让内核去检测，当这张表里有文件描述符有数据到来的时候，内核就会返回，并且告知我们哪一个文件描述符有数据。

IO多路转接_select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```



IO多路转接_select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

1. select返回值：
   - 0 代表没有文件描述符准备好，所有的描述符集被清空
   - -1 代表等待的过程中，被信号中断，所有描述符集不进行操作
   - n>0 代表已经准备好的事件个数

2. 遇到文件尾，则select认为该描述符是可读的。

IO多路转接_pselect

```
int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
```

1. 采用更高精度的超时定时
2. 增加信号屏蔽功能

IO多路转接_poll

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

POSIX异步IO

```
struct aiocb
{
  int aio_fildes;          /* File desriptor.  */
  int aio_lio_opcode;              /* Operation to be performed.  */
  int aio_reqprio;                 /* Request priority offset.  */
  volatile void *aio_buf; /* Location of buffer.  */
  size_t aio_nbytes;               /* Length of transfer.  */
  struct sigevent aio_sigevent;    /* Signal number and value.  */
  __off_t aio_offset;              /* File offset.  */
};
```

异步IO的核心数据结构

异步IO事件结束后，怎么通知应用程序

散布读和聚集写

要写的目的文件或者读的源文件

结构数组　　数组元素数目

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
//出错，返回-1，否则，返回已读或者已写的字节数

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

存储映射IO

1. 什么是存储映射IO?

将一个磁盘文件映射到存储空间的一个缓冲区[堆栈之间的一个区域， 依据实现有差异]上，然后用存取内存数据的操作来对文件进行读写

2. 如何实现？

```
void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);
//成功，返回映射区的起始地址，出错，返回MAP_FAILED

//addr： 映射到虚拟空间的地址，可以用户指定，但是更合适的做法是设置为NULL， 由内核来指定， 必须与虚拟存储页长度
//fd: 指的是要被映射文件的描述符,mmap之前。文件必须打开，且至少有读权限
//len: 代表要映射的字节数
//off: 代表映射的起始位置在文件中的偏移量，必须是虚拟存储页的倍数
//prot: 映射区的保护要求，其不能大于文件描述符open的访问权限
```

The prot argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either ***PROT_NONE*** *or the bitwise OR of one or more of the following flags:*

PROT_EXEC *Pages may be executed.*

PROT_READ *Pages may be read.*

PROT_WRITE *Pages may be written.*

PROT_NONE *Pages may not be accessed.*

存储映射IO

The flags argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in flags:

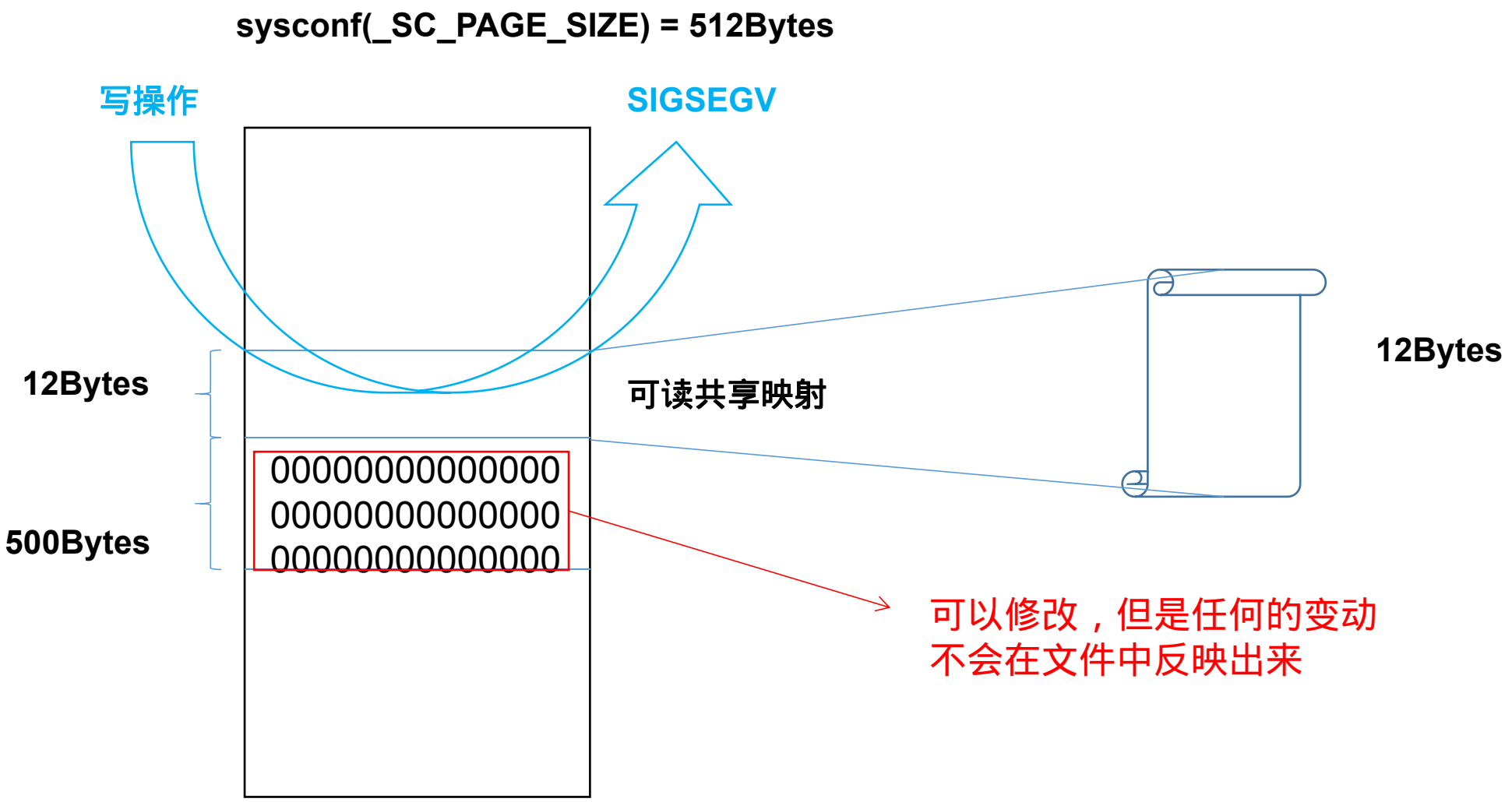
MAP_SHARED

Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of ***msync****(2).)*

MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the ***mmap****() call are visible in the mapped region.*

存储映射IO



如果在映射了之后，文件被截断，此时，你去映射区中访问被截断的映射部分，那就会引发段错误

子进程可以通过fork继承存储映射区，exec则会抹掉

存储映射IO

更改现有映射区的权限

```
int mprotect(void *addr, size_t len, int prot);
//成功，返回0,失败，返回-1
```

如果flag指定的是MAP_SHARED, 在内存中修改后，并不会立刻同步到磁盘文件中，需要靠系统的脏页冲洗算法的调度，也可以调用下面函数，进行手工冲洗：

```
int msync(void *addr, size_t len, int flags);
//成功，返回0,  失败，返回-1
```

*The flags argument should specify exactly one of **MS_ASYNC** and **MS_SYNC**, and may additionally include the **MS_INVALIDATE** bit. These bits have the following meanings:*

MS_ASYNC
Specifies that an update be scheduled, but the call returns immediately.

MS_SYNC
Requests an update and waits for it to complete.

MS_INVALIDATE
Asks to invalidate other mappings of the same file (so that they can be updated with the fresh values just written).

存储映射IO

关闭映射存储区

1. 当进程结束后，会自动解除存储映射区的映射

2. `int munmap(void *addr, size_t len); //succ return 0, else return -1`