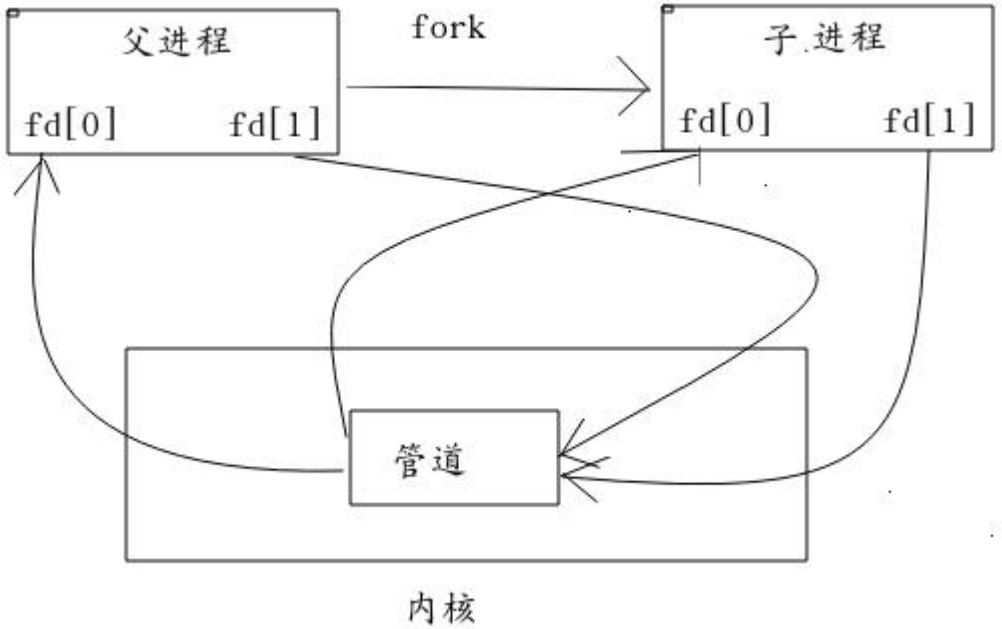
管道

1．建立一个管道

```
int pipe(int fd[2]);
//成功，返回0 ，失败，返回-1
```

2．管道结构



管道

3．管道读写特性

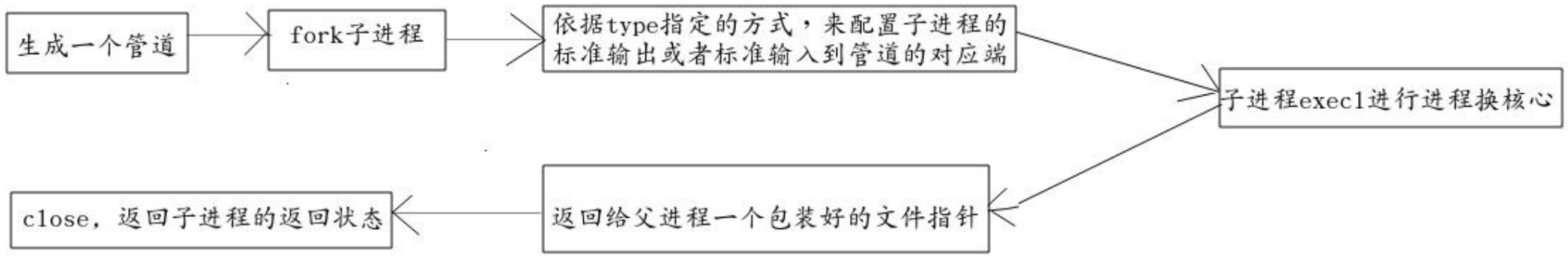
	有进程对应方式open	无进程对应方式open
open RD	打开	阻塞
open WR	打开	阻塞

	对端开	对端关
read	有数据读出来，无数据等待	有数据读，无数据返回 0
write	满，就等待，空闲，就写入	触发SIGPIPE, 返回-1

管道

4．基于管道的扩展_popen、pclose

```
FILE *popen(const char *cmdstring, const char *type);
//成功，返回文件指针，失败，返回NULL
int pclose(FILE *fp);
//成功，返回cmstring执行状态，出错，返回-1
```

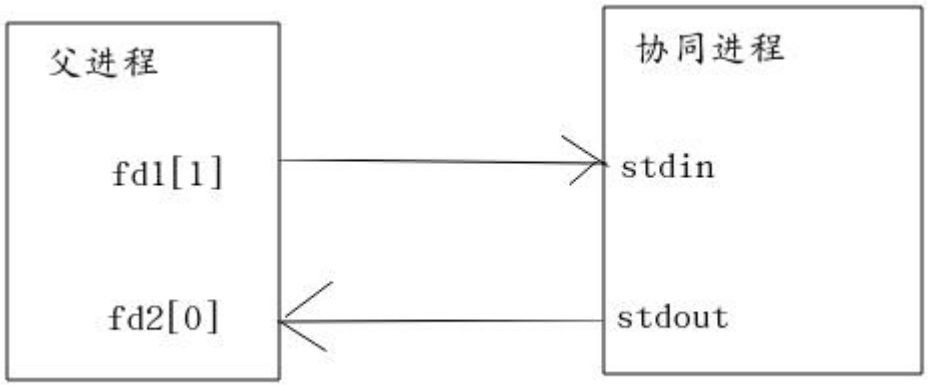


形象来看，该函数使得你可以将一个进程当做文件来操作

管道_协同进程

1．什么是协同进程？

其标准输入和标准输出均通过管道与其他进程相连



FIFO

1. 什么是FIFO?

FIFO是一个文件，其特性类似于管道

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

int mkfifo(const char *pathname, mode_t mode);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int mkfifoat(int dirfd, const char *pathname, mode_t mode);
```

//成功，返回0,出错， 返回-1
//mode参数与open函数中的一致

FIFO

3. FIFO读写特性，因为FIFO用open打开，就可以指定其中的O_NONBLOCK，以非阻塞模式打开，此时特性变化

阻塞模式	有进程对应方式open	无进程对应方式open
open RD	打开	阻塞
open WR	打开	阻塞

阻塞模式	对端开	对端关
read	有数据读出来，无数据等待	有数据读，无数据返回 0
write	满，就等待，空闲，就写入	触发SIGPIPE, 返回-1

非阻塞模式	有进程对应方式open	无进程对应方式open
open RD	打开	返回
open WR	打开	返回-1, 并设置errno为ENXIO

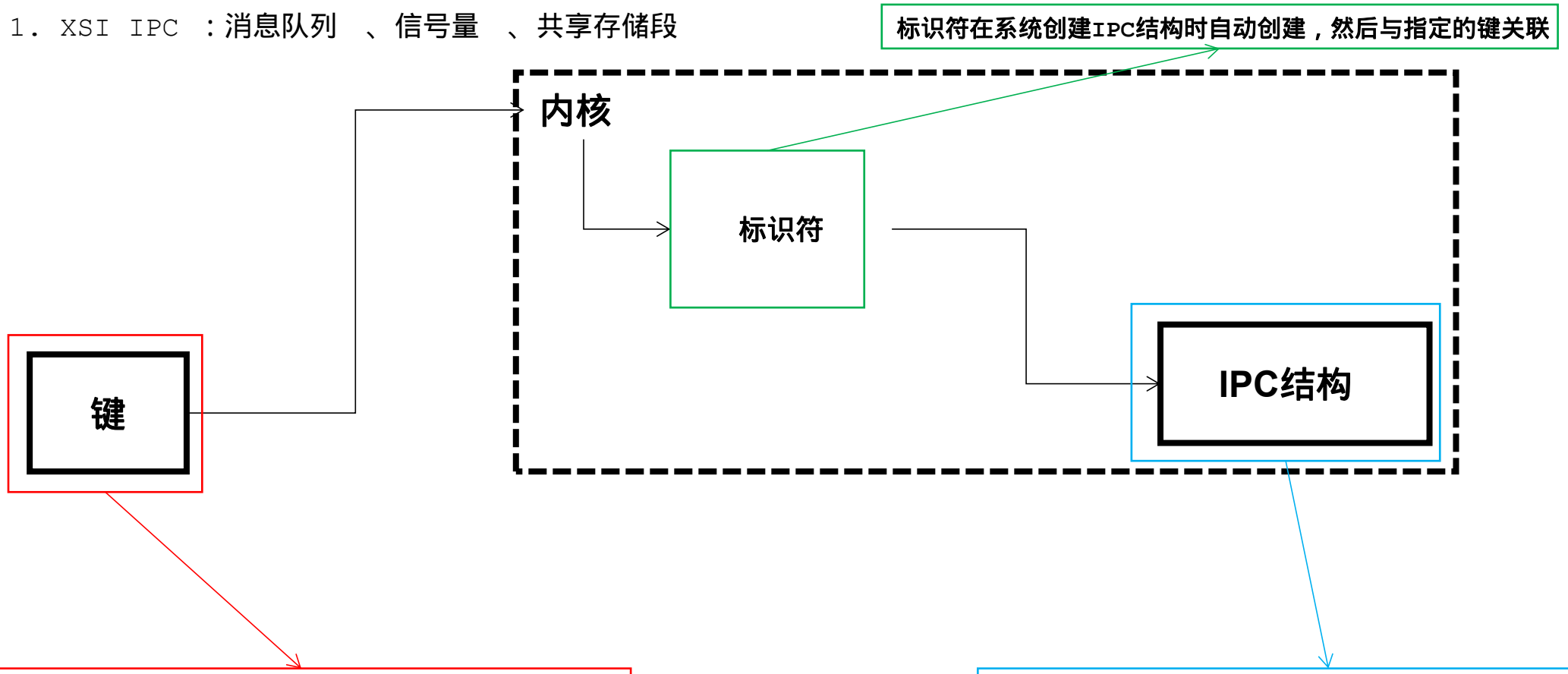
非阻塞模式	对端开	对端关
read	有数据读，无数据返回 0 ,设置errno为EAGAIN	有数据读，无数据返回 0
write	满，返回 0 ，设置 errno=EAGAIN，空闲，就写入	触发SIGPIPE, 返回-1

不管是pipe还是fifo,在写入的字节数小于PIPE_BUF的最大字节数的时候，都承诺保证原子化写入，不会产生多个进程写入数据的杂糅。

也就是说，在pipe和fifo的阻塞模式中，写操作因为没有空间而阻塞并不代表管道完全没有空间，而是管道剩余空间小于输入的的数据的长度。

XSI_IPC 结构

1. XSI IPC ：消息队列 、信号量 、共享存储段



如何生成一个键，可以自己定义一个不冲突的长整型作为键，
也可以利用下面函数生成键

```
key_t ftok(const char *path, int id);
//成功，返回键，失败，返回-1
//依据路径名和指定一个项目id变换为一个键
//两者相同，保证生成的键相同
```

1. 生成新的IPC结构
megget/semget/shmget: 指定key为不冲突的新key或者为IPC_PRIVATE,同时确保flag设置IPC_CREAT.

2. 引用一个现有的IPC结构
key必须为一个现有的key值，且flag不能被设置为IPC_CREAT.

3. 如果创建的时候，不确定key是不是冲突，可以flag增加IPC_EXCL, 这样冲突会进行错误返回 EEXIST

XSI_IPC 结构

2．IPC结构拥有的ipc_perm结构，规定了结构的权限和所有者

```
struct ipc_perm
  {
    __key_t __key;      /* Key.  */
    __uid_t uid;      /* Owner's user ID.  */
    __gid_t gid;      /* Owner's group ID.  */
    __uid_t cuid;     /* Creator's user ID.  */
    __gid_t cgid;     /* Creator's group ID.  */
    unsigned short int mode;    /* Read/write permission.
    unsigned short int __pad1;
    unsigned short int __seq;   /* Sequence number.  */
    unsigned short int __pad2;
    __syscall_ulong_t __glibc_reserved1;
    __syscall_ulong_t __glibc_reserved2;
  };
```

XSI_IPC 结构

3. IPC结构的特性

IPC 结构在内核系统中，独立与单一进程起作用，不随着某个进程的结束而删除。

IPC不存在于文件系统中，只能使用提供的系统调用来操作

IPC没有文件描述符，不方便使用IO多路转接的手段

XSI_IPC 消息队列

1. 消息队列的本质：链表

2. 消息队列的相关动态信息

```
struct msqid_ds
{
  struct ipc_perm msg_perm; /* structure describing operation permission */
  __time_t msg_stime;       /* time of last msgsnd command */
  __time_t msg_rtime;       /* time of last msgrcv command */
  __time_t msg_ctime;       /* time of last change */
  msgqnum_t msg_qnum;       /* number of messages currently on queue */
  msglen_t msg_qbytes;      /* max number of bytes allowed on queue */
  __pid_t msg_lspid;        /* pid of last msgsnd() */
  __pid_t msg_lrpid;        /* pid of last msgrcv() */
};
```

XSI_IPC 消息队列

3. 创建一个消息队列

```
int msgget(key_t key, int msgflg);
```

//成功，返回消息队列标识符， 失败，返回-1

4. 修改和删除一个消息队列

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

//成功，返回 0 ，出错，返回-1
//删除消息队列后，并不会等待所有使用该消息队列的进程结束
//再删除，而是直接删除

IPC_STAT

Copy information from the kernel data structure associated with msqid into the msqid_ds structure pointed to by buf. The caller must have read permission on the message queue.

IPC_SET

*Write the values of some members of the msqid_ds structure pointed to by buf to the kernel data structure associated with this message queue, updating also its msg_ctime member. The following members of the structure are updated: msg_qbytes, msg_perm.uid, msg_perm.gid, and (the least significant 9 bits of) msg_perm.mode. The effective UID of the calling process must match the owner (msg_perm.uid) or creator (msg_perm.cuid) of the message queue, or the caller must be privileged. Appropriate privilege (Linux: the **CAP_SYS_RESOURCE** capability) is required to raise the msg_qbytes value beyond the system parameter **MSGMNB**.*

IPC_RMID

*Immediately remove the message queue, awakening all waiting reader and writer processes (with an error return and errno set to **EIDRM**). The calling process must have appropriate privileges or its effective user ID must be either that of the creator or owner of the message queue. The third argument to **msgctl**() is ignored in this case.*

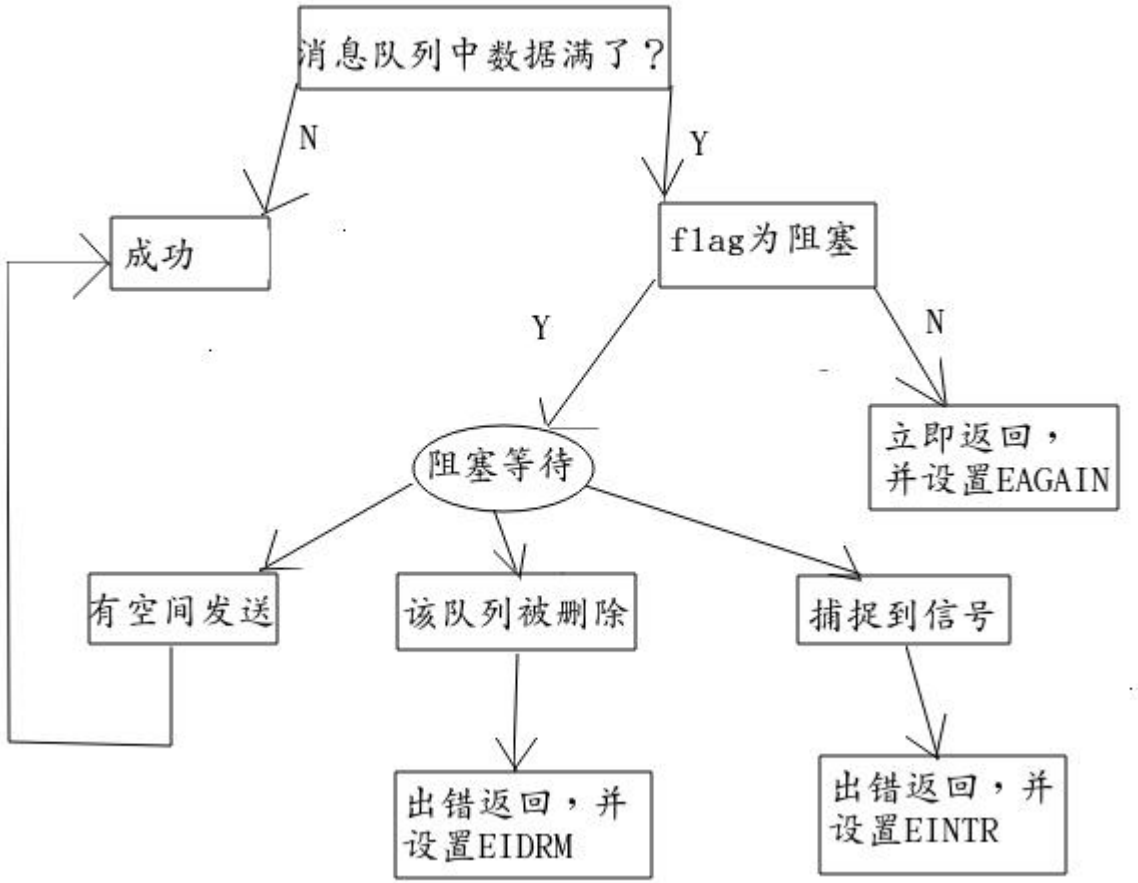
XSI_IPC 消息队列

5. 将一个消息插入消息队列

```
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
//成功，返回0,失败，返回-1
//消息以尾插法插入消息队列
//flag表示是否为阻塞模式
```

The msgp argument is a pointer to a caller-defined structure of the following general form:

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1];    /* message data */
};
```



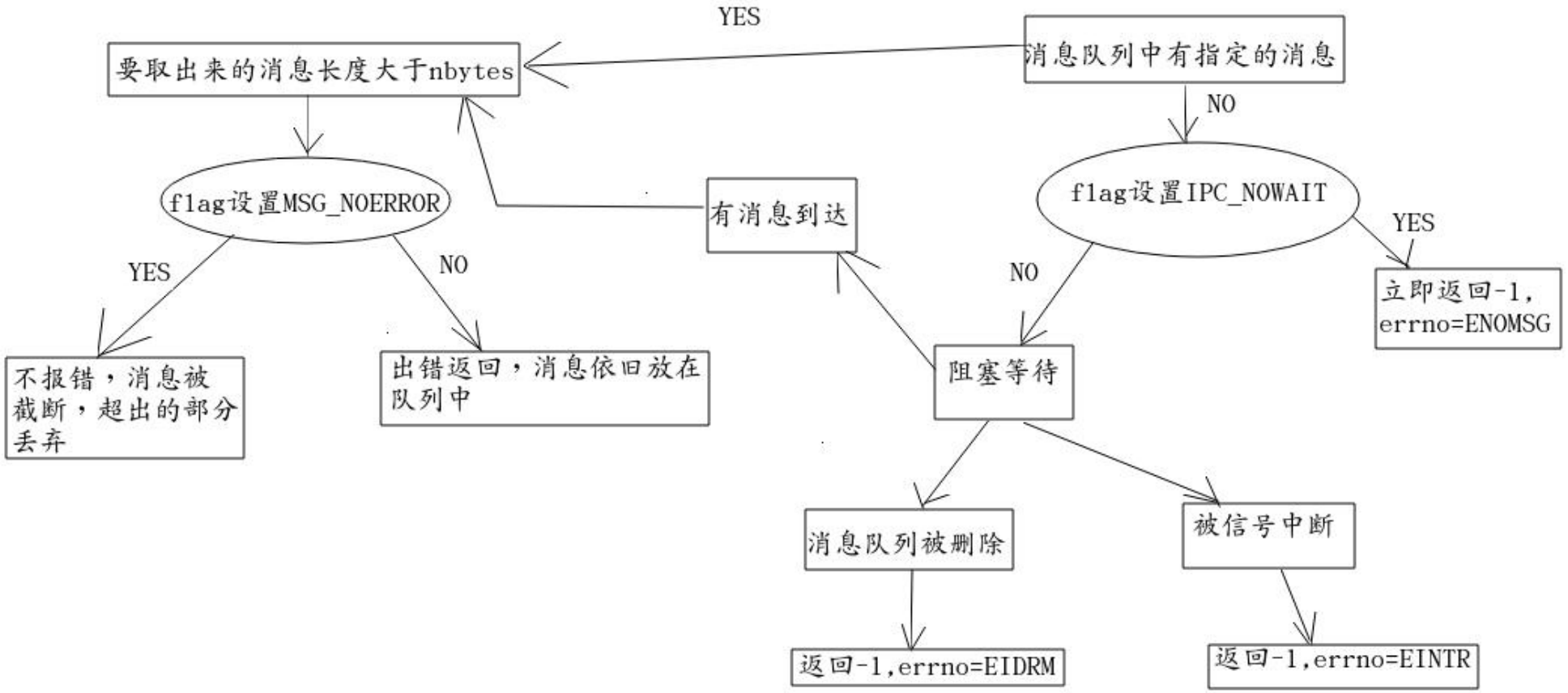
XSI_IPC 消息队列

6. 从队列中读取消息

```
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
//成功，返回消息数据部分的长度，出错，返回-1
//ptr指向取出来消息存放的位置
//nbytes存储的是缓冲区ptr的范围
//type存储要读取的消息信息
//flag设置函数的异常特性
```

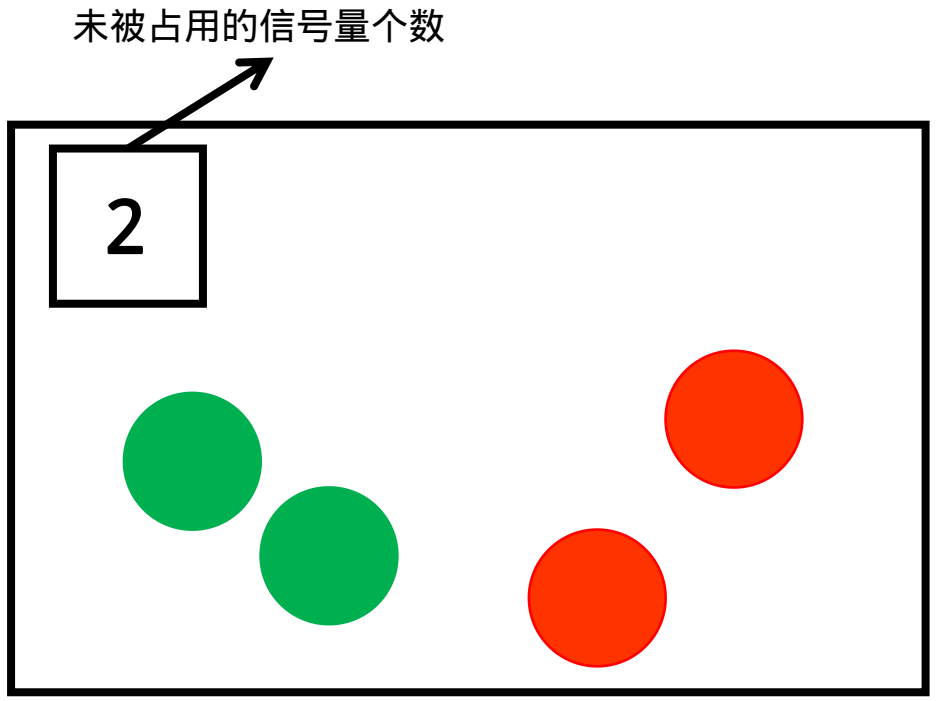
type == 0:返回队列中第一个消息
type > 0 :返回队列中消息类型为type的第一个消息
type < 0 :返回消息类型小于等于|type|的类型值最小的消息

XSI_IPC 消息队列


YES
消息队列中有指定的消息
要取出来的消息长度大于nbytes
NO
flag设置MSG_NOERROR
有消息到达
flag设置IPC_NOWAIT
YES
YES
NO
NO
立即返回-1，
errno=ENOMSG
不报错，消息被
截断，超出的部分
丢弃
出错返回，消息依旧放在
队列中
阻塞等待
消息队列被删除
被信号中断
返回-1,errno=EIDRM
返回-1,errno=EINTR

XSI_IPC 信号量

1. 信号量的本质：提供了计数功能的信号量值结构的集合，用于为多个进程提供对共享数据对象的访问



2. 信号量复杂性
   2.1 信号量并非是单个的非负值。而是一个集合，需要指定集合中信号量值的个数。
   2.2 信号量的创建和初始化不是原子操作
   2.3 信号量独立于单个进程，在进程结束后，信号量的特性

尚未被占用的信号量

被占用的信号量

XSI_IPC 共享存储

1. 共享存储：允许两个或者多个进程共享一个给定的存储区，是一种最快的IPC. 与文件映射IO的区别是，其没有相关的文件。

2. 共享存储的属性结构

```
struct shmid_ds
  {
    struct ipc_perm shm_perm;       /* operation permission struct */
    size_t shm_segsz;           /* size of segment in bytes */
    __time_t shm_atime;         /* time of last shmat() */
    __time_t shm_dtime;         /* time of last shmdt() */

    __time_t shm_ctime;         /* time of last change by shmctl() */

    __pid_t shm_cpid;           /* pid of creator */
    __pid_t shm_lpid;           /* pid of last shmop */
    shmatt_t shm_nattch;        /* number of current attaches */
  };
```

XSI_IPC 共享存储

3. 获得一个共享存储的引用

```
int shmget(key_t key, size_t size, int flag);
//成功，返回共享存储ID, 失败，返回-1
//新建共享存储的时候，结合前面的要求，size设置为锁需要的大小，新建完成后，内存内容被初始化为 0
//引用已经存在的共享存储的时候，结合前面的要求，size设置为 0
```

XSI_IPC 共享存储

4. 对共享存储进行操作

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
//成功，返回0,失败，返回-1
```

IPC_STAT Copy information from the kernel data structure associated with shmid into the shmid_ds structure pointed to by buf. The caller must have read permission on the shared memory segment.

IPC_SET Write the values of some members of the shmid_ds structure pointed to by buf to the kernel data structure associated with this shared memory segment, updating also its shm_ctime member. The following fields can be changed: shm_perm.uid, shm_perm.gid, and (the least significant 9 bits of) shm_perm.mode. The effective UID of the calling process must match the owner (shm_perm.uid) or creator (shm_perm.cuid) of the shared memory segment, or the caller must be privileged.

IPC_RMID Mark the segment to be destroyed. The segment will actually be destroyed only after the last process detaches it (i.e., when the shm_nattch member of the associated structure shmid_ds is zero). The caller must be the owner or creator of the segment, or be privileged. The buf argument is ignored.

If a segment has been marked for destruction, then the (non-standard) **SHM_DEST** flag of the shm_perm.mode field in the associated data structure retrieved by **IPC_STAT** will be set.

The caller must ensure that a segment is eventually destroyed; otherwise its pages that were faulted in will remain in memory or swap.

在删除后，只有使用该存储段的最后一个进程终止或者与该段分离，才会删除该共享存储段。但是一旦调用删除，段的shmid将失去作用。

XSI_IPC 共享存储

5. 连接到共享存储

```
void *shmat(int shmid, void *addr, int flag);
//成功，返回指向共享存储段的指针，出错，返回-1
//addr指定为 0 ，表示使用默认地址，也是推荐的方式
//flag设置为 0 ，表示使用默认值，读写权限
//如果要设置读权限，flag设置为SHM_RDONLY
```

6. 与共享存储分离[断开连接]

```
int shmdt(const void *addr);
//成功，返回 0 ，出错，返回-1
//一个进程终止，自动与共享存储分离
```

XSI_IPC 共享存储

7. /dev/zero存储映射

/dev/zero采用mmap进行进程通信，为什么只能用在有血缘关系的进程中?
采用mmap映射一个普通文件的时候，有没有上面的限制

8. 匿名存储映射

匿名存储映射是基于/dev/zero的实现，只能用在有血缘关系的进程中

```
mmap(0, SIZE, PROT_READ, MAP_ANON | MAP_SHARED, -1, 0);
        //设置MAP_ANON标志
        //文件描述符设置为-1
```

POSIX 信号量

1. POSIX信号量的改进
   性能的增强
   摒弃信号量集合的设计
   当一个POSIX信号量删除后，所有在这个信号量上的操作能继续正常工作直到该信号量的最后一次引用被释放

2. POSIX未命名信号量的创建与销毁[属于单个进程的资源]

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
//成功，返回0， 出错，返回-1
//pshared=0表示只能在单个进程中使用，非零表示允许进程间共享
//value指定信号量的初始值

int sem_destory(semt_t *sem);
//成功，返回 0 ，失败，返回-1
//实际的销毁会延迟到信号量的最后一个引用线程结束

int sem_getvalue(sem_t *sem, int *valp);
//成功，返回0， 失败，返回-1
//得到信号量的值
```

POSIX 信号量

2. POSIX命名信号量的创建与销毁[属于系统的资源]

```
sem_t *sem_open(const char *name, int oflag, ..., /*mode_t mode, unsigned int value*/);
//成功，返回信号量指针，出错，返回SEM_FAILED

int sem_close(sem_t *sem);
//成功，返回 0 ，失败，返回-1
//释放本进程对于该信号量开辟的资源
//进程结束后不调用sem_close也会默认会释放资源
//并不会对信号量的值产生影响

int sem_unlink(const char *name);
//成功，返回 0 ，失败，返回-1
//从系统中销毁信号量，等到对该信号量的引用为 0 才会真正的销毁
```

POSIX 信号量

2. 信号量的+1和-1

```
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *tsptr);
        //如果信号量不为 0 ，那么时间不起作用，就算时间是过去的时间
//成功，返回0,出错，返回-1



int sem_post(sem_t *sem);
//成功，返回0， 失败，返回-1
//信号量 + 1
```