Unix进程间通信总结（IPC）

进程通信的主要目的

1. 数据传输:一个进程需要将它的数据发送给另一个进程，发送的数据量在几个字节到几M之间
2. 共享数据:多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该可以看到修改
3. 通知事件:一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某个事件（比如子进程销毁通知父进程）
4. 资源共享:多个进程共享某种资源，需要内核提供锁和同步机制
5. 进程控制:有时候有些进程需要控制其他进程，需要拦截其他进程的所有陷入和异常（比如gdb debug进程），并且需要及时知道该进程的状态改变

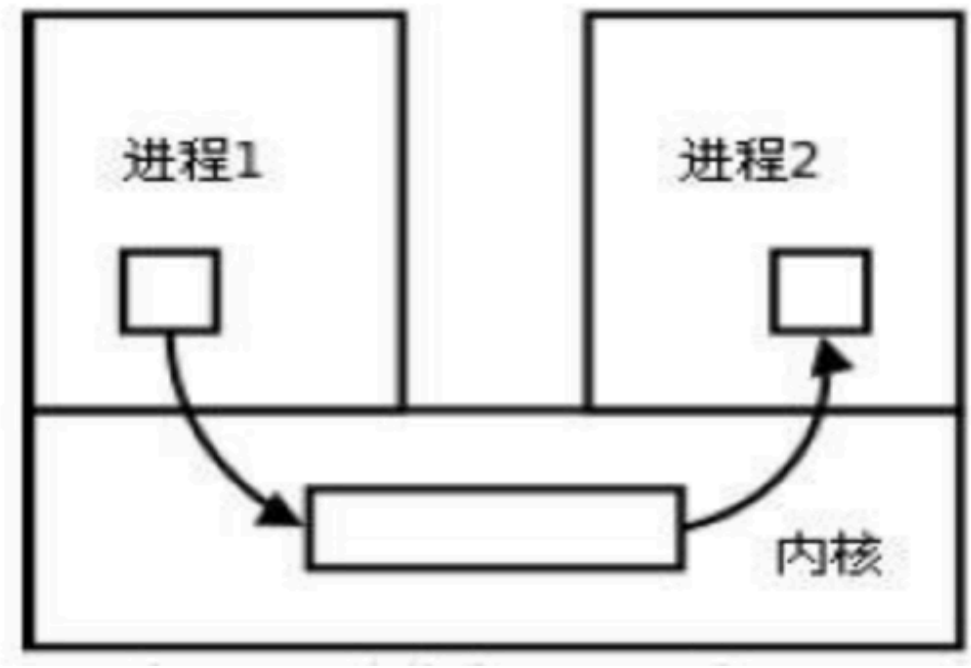
管道（无名管道）

管道历史上是半双工的，为了可移植性不能假定管道支持全双工

管道只能在具有公共祖先的两个进程间进行通信,通常管道由一个进程创建，然后该进程调用fork以后，父子进程可以共享管道，管道存储在内核中

创建一个管道 `int pipe(int fd[2])`

一般来说， `fd[0]` 仅为只读打开， `fd[1]` 仅为写打开， `fd[1]` 的输出是 `fd[0]` 的写入,如下为父子进程的管道示意图:



具体操作见如下代码：

```
int main(int argc,char *argv[]){
    // fd[0] to read,fd[1] to write
    int32_t fd[2];
    pid_t pid;
    std::string buff;
    const size_t MAX_SIZE = 1024;
    if(pipe(fd)){
        std::cerr<<"pipe error"<<std::endl;
        return -1;
    }

    if((pid = fork()) < 0)
        std::cerr<<"fork error"<<std::endl;
    else if(pid > 0){  //father process
        close(fd[0]); // close read fd
        std::string msg = "hello son!";
        write(fd[1],msg.data(), strlen(msg.data()));
    }else{  //child process
        close(fd[1]);
        read(fd[0], const_cast<char *>(buff.data()),MAX_SIZE);
        std::cout<<buff.data()<<std::endl;
    }

    return 0;
}
```

命名管道（FIFO）

Unfortunately，也是半双工,默认为阻塞模式
相关函数：成功返回0，失败返回-1

```
#include <sys/stat.h>
int mkfifo(const char *path,mode_t mode); //create fifo pipe
int mkfifoat(int fd,const char *path,mode_t mode);
```

命名管道不相关的进程也可以通信

发送方具体代码：

```
int main(int argc,char *argv[]){
    const std::string pipePath = "/Users/qiuyang/Desktop/fifo";
    int32_t fd = open(pipePath.data(),O_RDONLY);
    std::string buff;
    const size_t MAX_SIZE = 1024;

    if(fd < 0){
        std::cerr<<"open fd error"<<std::endl;
        return -1;
    }

    while(true){
        if(read(STDIN_FILENO, const_cast<char *>(buff.data()),MAX_SIZE) < 0){
            std::cerr<<"read error"<<std::endl;
            return -1;
        }

        if(write(fd,buff.data(),strlen(buff.data())) < 0){
            std::cerr<<"write error"<<std::endl;
            return -1;
        }
    }
    return 0;
}
```

接收方具体代码:

```
int main(int argc,char *argv[]){
    int32_t fd;
    std::string buff;
    const size_t MAX_SIZE = 1024;
    const std::string pipePath = "/Users/qiuyang/Desktop/fifo";
    if(mkfifo(pipePath.data(),0777)) {
        std::cerr << "pipe error" << std::endl;
        return -1;
    }

    if((fd = open(pipePath.data(),O_RDONLY))){
        std::cerr<<"open file error"<<std::endl;
        return -1;
    }

    while(true){
        if(read(fd, const_cast<char *>(buff.data()),MAX_SIZE) < 0){
            std::cerr<<"read error"<<std::endl;
            return -1;
        }

        std::cout<<buff.data()<<std::endl;
    }

    return 0;
}
```

创建的fifo管道**使用前后大小为0**，说明内核没有真正的创建文件

具有同步和阻塞的问题,每一个数据块有一个最大长度

消息队列

消息队列是消息的链接表，存储在内核中，由消息队列标识符标识
相关函数:

```
#include <sys/msg.h>
int msgget(key_t key,int msgflg); //create msg queue, IPC_CREATE
key_t ftok(const char *fname,int id);
int msgsnd(int msqid,const void *msgp,size_t msgsz,int msgflg); //send msg
int msgrcv(int msqid,const void *msgp,size_t msgsz,long int msgtype,int msgflg);
int msgctl(int msqig,int command,struct msgid_ds *buf);
```

结构体msgp的定义:

```
struct msgbuf{
    long mtype; //msg type,positive number,must be long type
    char mtext[len]; //content of msg,not include mtype length
};
```

消息正文的长度是可变的，长度可以是1字节也可以是512字节或者更长，这解释了发送消息是需要指定消息的长度，消息正文mtext不受其长度定义1的限制，但是存在最大长度定义在usr/include/linux/msg.h中

```
#define MSGMAX 8192 /* max size of message (bytes) */
```

在msgsnd函数中msgflag用于控制当前队列满或队列消息到达系统范围的控制时将要发生的事情
在msgrcv函数中，msgtype可以实现一个简单的优先级
1. 如果msgtype为0，那么就接收消息队列中的第一条消息
2. 如果msgtype大于0，那么就接收队列中第一条与msgtype相同的消息（可以实现分发机制）
3. 如果msgtype小于0，n吗就接收队列中第一条消息类型大于等于msgtype绝对值的消息
msgtype用于控制队列中没有类型的信息可以接收时将做什么

msgctl函数用于控制消息队列，与共享内存shmctl函数类似，msgid*ds是指向消息队列模式和访问权限的结构体，command是要采取的动作，可以取三个值 1. IPC*STAT:把msgid*dsj结构中的数据设置为消息队列的当前关联值，即用消息队列的当前关联值覆盖msgid*ds的值 2. IPC*SET:如果进程有足够的权限，就把消息队列的当前关联值设置为msgid*ds结构体中的值 3. IPC_RMID:删除消息队列

```
struct msgid_ds{
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
}
```

发送端:

```
while(true){
    msgBuff.msgType = 1; // add msg type
    if(msgsnd(msgId,msgBuff,MAX_SIZE,0) < 0){
        std::cerr<<"send msg error"<<std::endl;
        break;
    }
}
```

接收端代码:

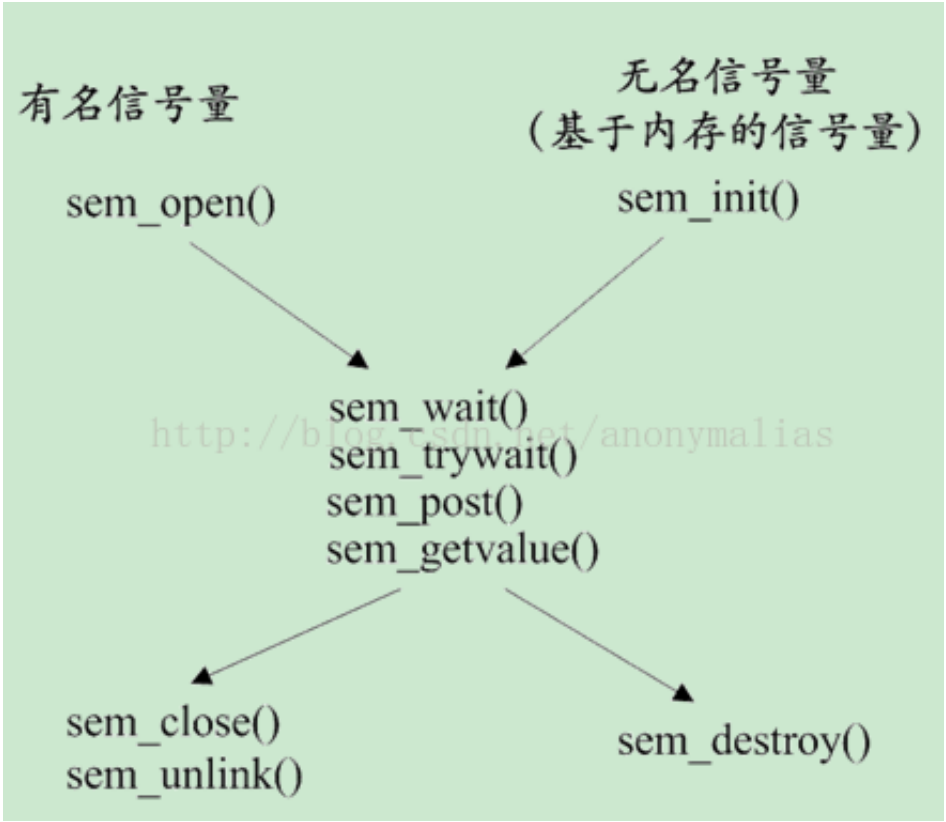
```
const int32_t MAX_SIZE = 1024;
typedef struct{
    long int msgType;
    char msg[MAX_SIZE];
}MsgBuff;

int main(int argc,char *argv[]){
    MsgBuff msgBuff;
    long int msgType = 0;//default recv the first msg in msgq
    key_t msgId = msgget((key_t)1234,0777|IPC_CREATE);
    if(msgId < 0){
        std::cerr<<"get msgq error"<<std::endl;
        return -1;
    }

    while(true){
        if(msgrcv(msgId,(void *)msgBuff,MAX_SIZE,0) < 0){
            std::cerr<<"recv msg error"<<std::endl;
            break;
        }

        std::cout<<"recv msg "<<msgBuff.msg<<std::endl;
    }

    if(msgctl(msgId,IPC_RMID,0) < 0){
        std::cerr<<"rm msgq failed"<<std::endl;
        return -1;
    }

    return 0;
}
```

信号量

信号量的设计目的与之前几种进程间通信方式有所不同，它主要终于进程或线程之间的同步
信号量是一个计数器，用于为多个线程或进程提供对共享数据的访问
信号量有两种：System V信号量，和新版本的Posix信号量，各自的需要包含的头文件也不同，分别为sys/sem.h和semaphore.h,前者常用于进程间通信，后者常用于线程间通信，当然也可以进程间
值得一提的是在《UNIX网络编程 卷2：进程间通信》的前言中作者这样写道：**POSIX IPC函数是大势所趋，因为他们比System V中的相应部分更具有优势...?**

Systm V	POSIX
semctl()	sem_getvalue()
semget()	sem_post()
semop()	sem_timedwait()
	sem_trywait()
	sem_wait()
	sem_destroy()
	sem_init()
	sem_close()
	sem_open()
	sem_unlink()

POSIX信号量有两种：**有名信号量**和**匿名信号量**，有名信号量可以用于进程间同步，匿名信号量如果想进行进程间通信，需要放在共享内存中，否则进程间是不可见的，相对应的函数如图：



有名信号量

有名信号量打开操作函数有两个版本如下：（如何实现重载的?）

```
sem_t *sem_open(const char *name,int oflag);
sem_t *sem_open(const char *name,int oflag,mode_t mode,unsigned int value);
```

name的作用同以上所有有名操作，oflag可以取三个值:
1. 0表示打开一个已存在的有名信号量
2. O_*CREATE表示不存在这个有名信号量就创建,此时需要指定mode和value这也解释了为什么需要两个版本的open函数*
*3. O*CREATE | O_EXCL表示如果信号量已存在就返回错误

mode同打开文件的权限操作，value表示信号量的初始值

有名信号量的关闭操作函数如下：

```
int sem_close(sem_t *sem);
int sem_uplink(const char *name);
```

close用于关闭打开的信号量，但是不能销毁信号量，换句话说，信号量保存着状态，其他进程打开这个信号量之后，仍然可以拿到之前的值,这也从侧面解释了为什么需要两个版本的open函数，uplink函数用于真正销毁这个有名信号量，但注意必须是在所有打开该信号量的进程都调用了close函数之后

P、V操作（荷兰语:Proberen，意为：尝试,Verhogen，意为：增加）

```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem,const struct timespec *abs_timeout)
int sem_getvalue(sem_t *sem,int *sval);
```

V操作只有一个版本，对信号量执行+1操作，如果有进程或线程等待该信号量，唤醒该进程或线程
P操作有三个版本，普通的wait版本，也是阻塞版本，trywait版本是非阻塞版本，如果没有等到该信号量，就会直接返回，并返回错误标识,timedwait可以等待一个由abs_timeout指示的时间，超时后，返回一个错误标识EAGAIN

getvalue返回当前信号量的值

有名信号量存在**继承**行为，意思是父进程打开了某个信号量，子进程可以直接使用该信号量

匿名信号量

匿名信号量的创建和销毁函数如下：

```
int sem_init(sem_t *sem,int pshared,unsigned int value);
int sem_destory(sem_t *sem);
```

匿名信号量的init函数用于初始化，sem必须是一个指向预先分配好内存空间的sem_t的变量指针，所以匿名信号量也叫基于内存的信号量，pshared为0表示该信号量不在共享内存中，不能用于进程间共享，不为0时可以用于进程间同步，value为信号量的初始值

注意：

> 1.对一个已经初始化的匿名信号量再次进行初始化是未定义行为！
> 2.destory一个有线程阻塞在其上的信号量也是未定义行为！

匿名信号量的继承取决于信号量在内存中的存放位置,如果无名信号量没有存储在共享内存区，那么子进程会得到父进程信号量的一份拷贝，相反如果存储在共享内存区，则父子进程可以共享该信号量

有名信号量测试代码如下:

```
int main(int argc,char *argv[]){
    const std::string SHM_PATH = "test";
    sem_t *p_sem = sem_open(SHM_PATH.data(),O_CREAT,0666,5);
    if(p_sem == nullptr){
        std::cerr<<"sem open error"<<std::endl;
        return -1;
    }

    int val = 0;
    sem_getvalue(p_sem,&val);
    std::cout<<"before fork parent semaphore num "<<val<<std::endl;

    if(fork() == 0){
        sem_wait(p_sem);
        sem_getvalue(p_sem,&val);
        std::cout<<"child semaphore num "<<val<<std::endl;
        return -1;
    }

    sem_getvalue(p_sem,&val);
    std::cout<<"after fork parent semaphore num "<<val<<std::endl;
    sem_unlink(SHM_PATH.data());
    return 0;
}
```

信号

信号是通过软中断的方式进行处理的，进程之间可以通过系统调用kill来发送信号，内核也可以因为内部时间而给进程发送信号，通知进程发生了某种事件

```
#include <signal.h>
void (*singal(int sig,void(*func)(int)))(int);
```

上述函数的意思是捕捉值为sig的信号，然后使用func指向的函数进行处理，signal作为一种可调用对象使用时可以直接向函数一样调用,此外func可以设置为SIG*IGN,SIG*DFL分别代表忽略信号，和进行默认处理

具体事例代码如下:

```
void helloWorld(int sig){
    std::cout<<"catch sig number "<<sig<<std::endl;
    std::cout<<"in hello world func"<<std::endl;
}

int main(int argc,char *argv[]){
    signal(SIGINT,helloWorld);
    while(1){
        usleep(100);
        std::cout<<"waiting SIGINT"<<std::endl;
    }

    return 0;
}
```

注意:

> 信号的值因系统的不同可能会不同，但是信号的代码一般都固定

信号名称	说明
SIGABORT	*进程异常终止
SIGALRM	超时警告
SIGFPE	*浮点运算异常
SIGHUP	连接挂断
SIGILL	*非法指令
SIGINT	终端中断
SIGKILL	终止进程(此信号不能被捕获或忽略)
SIGPIPE	向无读进程的管道写数据
SIGQUIT	终端退出
SIGSEGV	*无效内存段访问
SIGTERM	终止
SIGUSER1	用户定义信号1
SIGUSER2	用户定义信号2

如果进程对上述信号没有做任何捕获处理，那么进程在收到上述任何信号后会直接退出

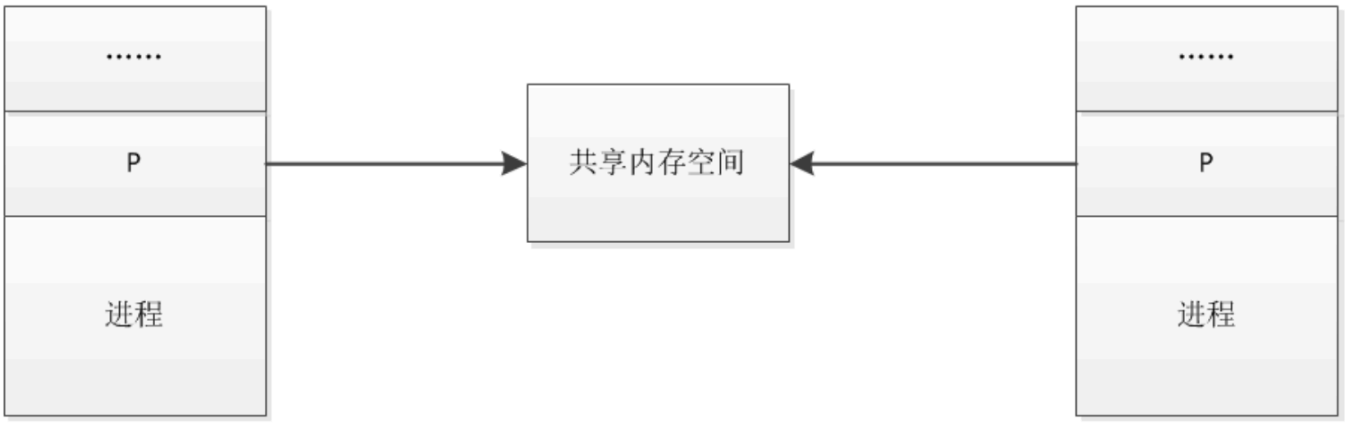
信号发送的相关函数如下:

```
#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid,int sig);
```

其中pid为发送给哪个进程的进程号，sig为信号值，该函数也可以给进程调用该函数的进程自身发送信号

共享内存

共享内存主要是针对多进程间大量数据的通信设计的，理论上是同一主机内进程间通信最快的方式



多个进程在使用此共享内存空间时，必须建立进程空间和共享内存空间的连接，即将共享内存空间挂载到内存中，共享内存由一个进程开辟之后，其他任何进程都可以挂载

注意：

> 共享内存并不会随着进程的退出而消失，所以最后不使用共享内存时，需要手动删除，一个小Tip就是可以把共享内存的删除放在类的析构函数里，这样随着主函数的退出，类会自动执行析构函数，就不会忘记删除共享内存啦
> ftok函数第一个参数一定要是一个存在的路径,Unix实现中是把路径的文件索引拿出来，然后加上ftok的第二个参数

共享内存的相关管理函数如下:

```
#include <sys/shm.h>
int shmget(key_t key,size_t size,int shmflg);
int shmctl(int shmid,int cmd,struct shmid_ds *buf);
void *shmat(int shmid,const void *shmaddr,int shmflg);
int shmdt(const void *shmaddr);
```

shmget用于创建一个共享内存空间，key参数同上述类似创建的函数参数，size为大小，shmflag同上述所有创建操作flag
shmctl函数用于对共享内存进行各种操作，包括读取、获取共享内存状态，以及删除操作

Macro	No	Description	Return
IPC_RMID	0	删除	0
IPC_SET	1	设置ipc_perm参数	0
IPC_STAT	2	设置ipc_perm参数	
SHM_LOCK	11	锁定共享内存段	0
SHM_UNLOCK	12	解锁共享内存段	0

shmat用于将共享内存挂载到进程的空间中，shmid为shmget的返回值，第二个参数一个设置为0让系统默认分配，第三个参数为权限位，可以设置读写权限
shmdt用于将进程和共享内存分离，注意只是与共享内存没有联系，而不是删除共享内存，参数为共享内存的首地址

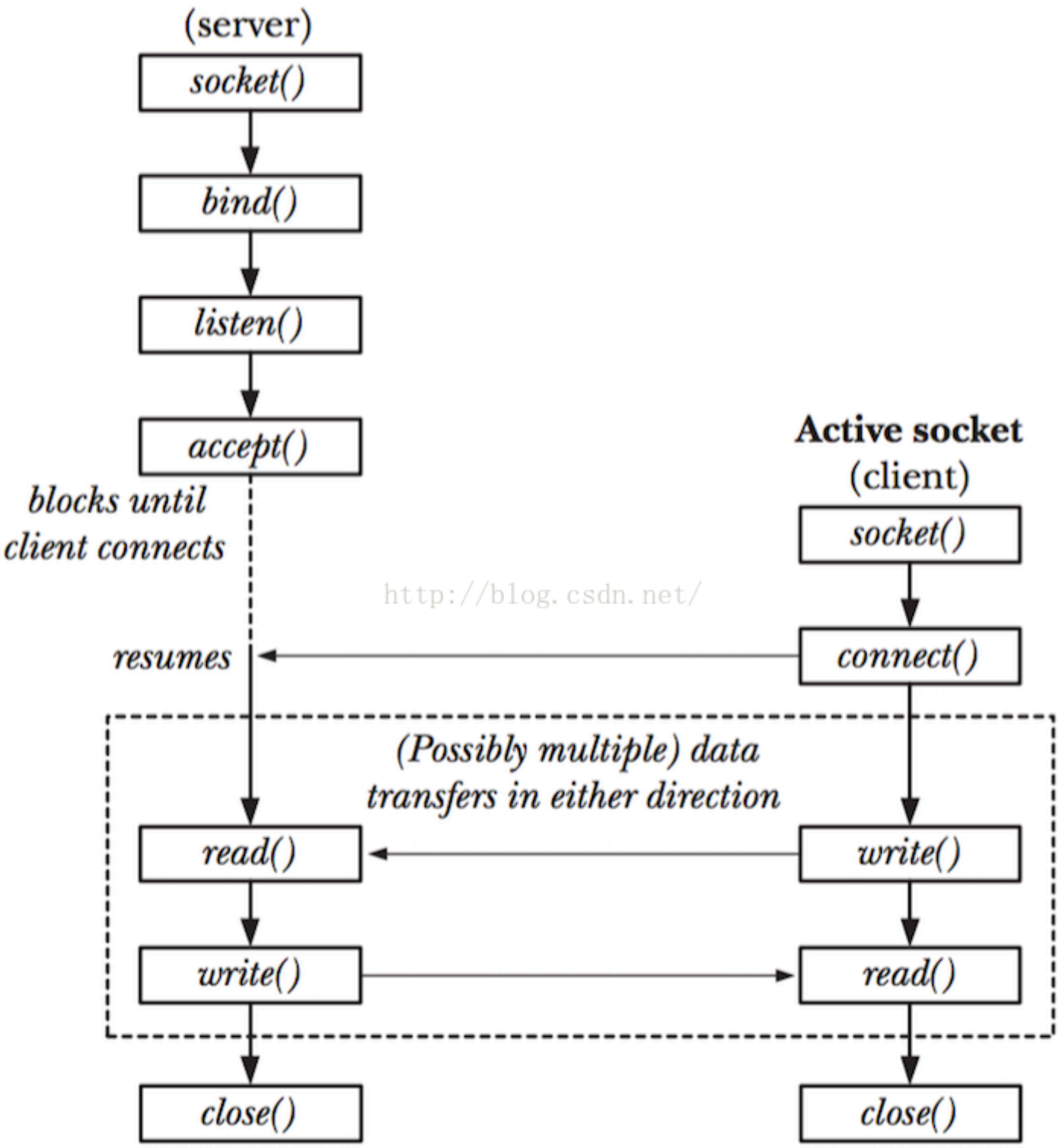
示例读取共享内存代码如下:

```
const size_t MAX_SIZE = 8192;
const size_t CANWRITE = 0;
const size_t CANREAD = 1;
typedef struct{
    int written;
    char text[MAX_SIZE];
}SharedBuff;

int main(int argc,char *argv[]){
    void *shm = nullptr;
    SharedBuff *pSharedBuff = nullptr;
    int shmid;

    shmid = shmget((key_t)1234, sizeof(SharedBuff),0666|IPC_CREAT);

    shm = shmat(shmid,0,0);
    pSharedBuff = (SharedBuff *)shm;

    while(true){
        if(pSharedBuff->written == CANREAD){
            std::cout<<pSharedBuff->text<<std::endl;
            pSharedBuff->written = CANWRITE;
        }else
            usleep(10);
    }

    shmdt(shm);
    shmctl(shmid,IPC_RMID,0);

    return 0;
}
```

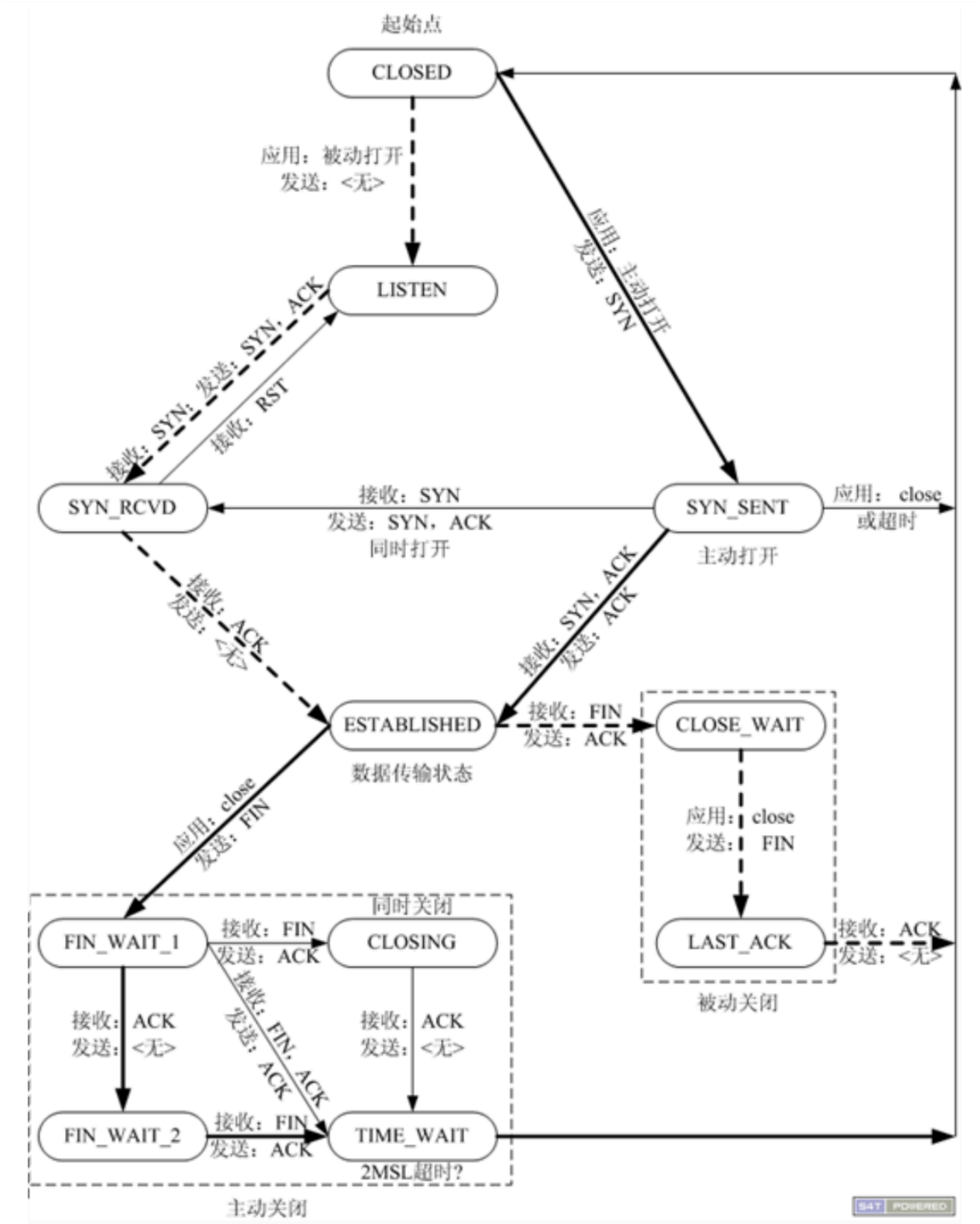
Unix网络套接字

在Unix环境套接字既有可以用于不同主机间通信的基于TCP/IP协议的套接字，也有只能在同一主机下的进程间通信的Unix域套接字，该部分内容太多详见《Unix网络编程 卷1》，这里只总结TCP在连接和断开的各种状态变化，以便加深理解

以下为TCP套接字创建到删除的全过程:

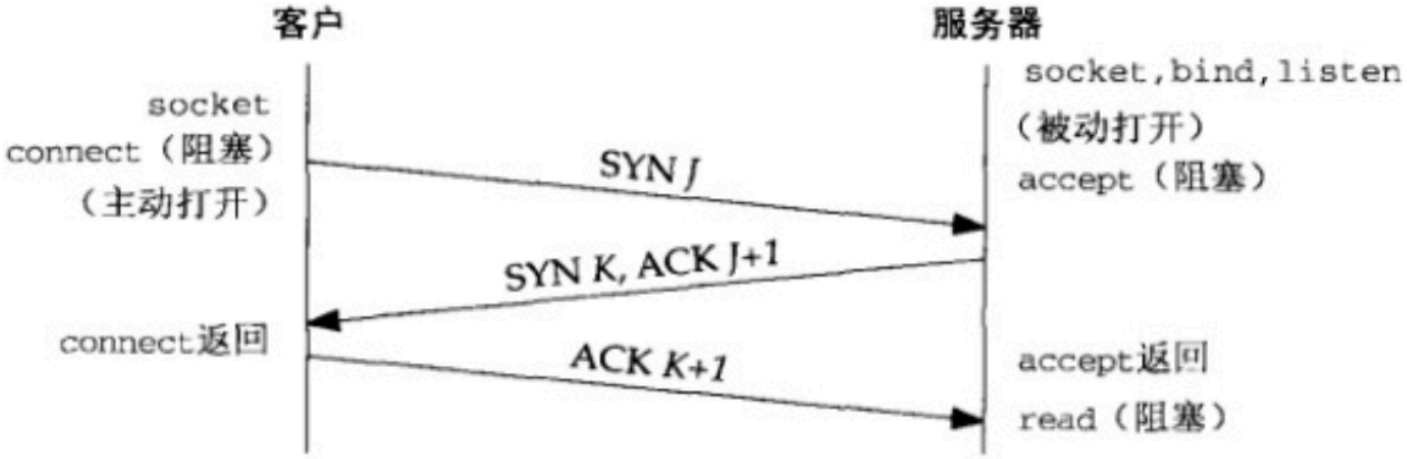


TCP从建立到关闭有以下各种状态:



建立过程:

1. 服务器首先依次调用socket、bind、listen、accecpt进入被动打开状态，此时状态机从CLOSED变为LISTEN状态，此时服务器阻塞在accept函数
2. 客户端调用socket、connection函数主动打开，并发送SYN数据包请求同步，进入SYN_SENT状态，此时客户端阻塞在connection函数
3. 服务器收到SYN数据包，随后也发送SYN数据包，并发送ACK给客户端，进入SYN__RCVD状态
4. 客户端收到服务器回复的SYN，以及ACK，随后回复一个ACK包告知服务器已经收到，进入到ESTABLISHED状态，此时connection函数返回
5. 服务器收到客户端的ACK包，进入ESTABLISHED状态,此时accept函数返回



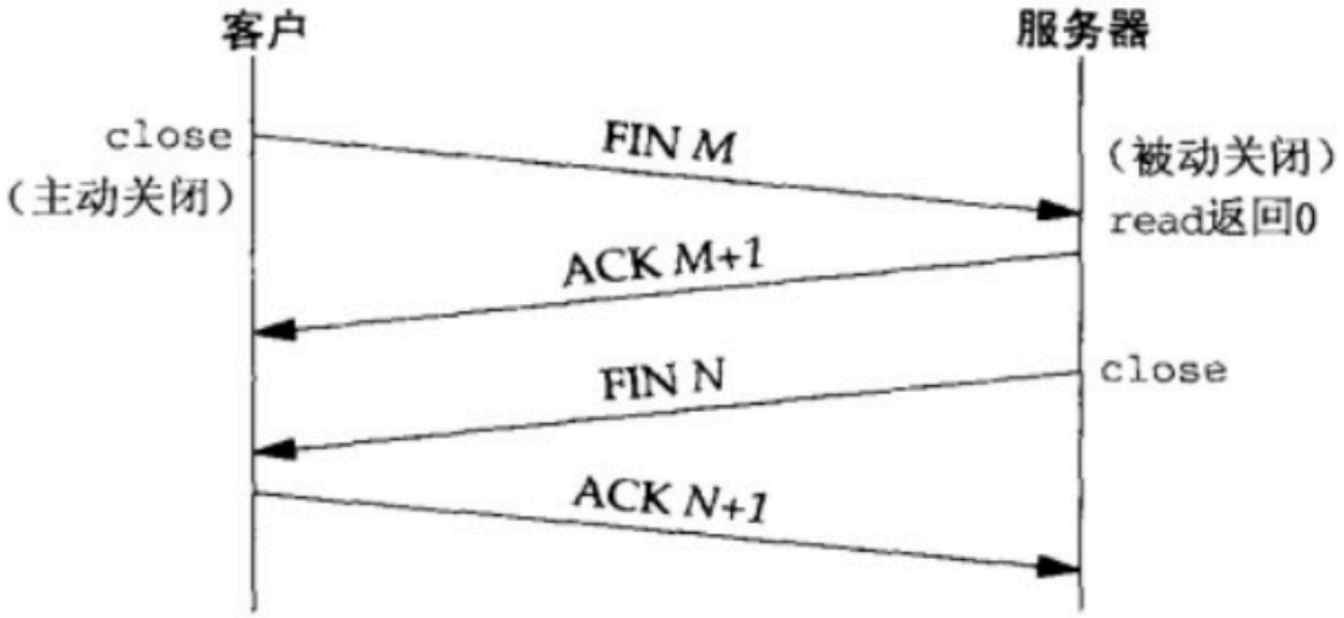
注意:

> 1. 在TCP连接建立的过程中，有一个极为罕见的同时打开现象，即对应上图中SYN*SENT状态到SYN*RCVD状态的转移，这个发生在TCP的两端同时发送SYN数据包，并在网络中错开，至今没有遇到过这种情况
> 2. 如果客户端在发送SYN数据包之后不行崩溃，客户端重启以后，客户端收到来自服务器的SYN+ACK，但是此时该数据包已经失效，客户端会回复RST数据包要求服务器重置，服务器会有SYN_RCVD状态回归到LISTEN状态，客户端回归CLOSED状态
> 3. 相对应的，如果服务器进程崩溃，那么客户端发送到服务器端后，此时重启以后的服务器TCP状态处于CLOSED状态，会相应一个RST数据包，发送给客户端，客户端会回归CLOSED状态，如果服务器还没有恢复，则客户端收不到SYN的相应，超时后也会自动回归CLOSED状态

断开连接

1. 主动关闭端通过调用close函数发送FIN数据包，由ESTABLISHED状态进入FIN*WAIT1状态，被动关闭端接收到FIN数据包，会回复这个数据包的ACK，进入CLOSE*WAIT状态
2. 主动关闭端收到FIN的ACK，进入FIN_WAIT2状态
3. 之前的被动关闭端调用close函数，发送FIN数据包给主动关闭端，进入LAST*ACK状态，主动关闭端收到这个FIN数据包，发送ACK给被动关闭端，被动关闭端回归到CLOSED状态，主动关闭端进入*

*TIME*WAIT状态，在2个MSL时间之后，回归CLOSED状态



注意:

1. 关闭过程同样存在同时关闭的现象，即两个FIN数据包在网络中同时被发送，此时发送双方会同时进入FIN*WAIT1状态，然后双方都收到ACK进入CLOSING状态，最后进入TIME*WAIT状态
2. 如果被动关闭端在收到FIN之后立即后立即调用close函数，会将FIN和ACK在一个数据包中发送给主动关闭方，此时主动关闭方会跳过FIN*WAIT2阶段，直接发送ACK数据包进入TIME*WAIT状态