sock结构和socket结构详细解释

2008-10-17 14:34

```
//**************************************************************************
/* 1、每一个打开的文件、socket等等都用一个file数据结构代表，这样文件和socket就通过inode->u(union)中的各个成员来区别：
struct inode {
.....
union {
struct ext2_inode_info ext2_i;
struct ext3_inode_info ext3_i;
struct socket socket_i;
.....
} u; };
2、每个socket数据结构都有一个sock数据结构成员，sock是对socket的扩充，两者一一对应，socket->sk指向对应的sock，sock->socket
指向对应的socket；
3、socket和sock是同一事物的两个侧面，为什么不把两个数据结构合并成一个呢？这是因为socket是inode结构中的一部分，即把inode结
构内部的一个union用作socket结构。由于插口操作的特殊性，这个数据结构中需要有大量的结构成分，如果把这些成分全部放到socket
结构中，则inode结构中的这个union就会变得很大，从而inode结构也会变得很大，而对于其他文件系统这个union是不需要这么大的，
所以会造成巨大浪费，系统中使用inode结构的数量要远远超过使用socket的数量，故解决的办法就是把插口分成两部分，把与文件系
统关系密切的放在socket结构中，把与通信关系密切的放在另一个单独结构sock中；
*/

struct socket
{
socket_state state; // 该state用来表明该socket的当前状态
typedef enum {
SS_FREE = 0, /* not allocated */
SS_UNCONNECTED, /* unconnected to any socket */
SS_CONNECTING, /* in process of connecting */
SS_CONNECTED, /* connected to socket */
SS_DISCONNECTING /* in process of disconnecting */
} socket_state;
```

```
unsigned long flags; //该成员可能的值如下，该标志用来设置socket是否正在忙碌
#define SOCK_ASYNC_NOSPACE 0
#define SOCK_ASYNC_WAITDATA 1
#define SOCK_NOSPACE 2
struct proto_ops *ops; //依据协议邦定到该socket上的特定的协议族的操作函数指针，例如IPv4 TCP就是inet_stream_ops
struct inode *inode; //表明该socket所属的inode
struct fasync_struct *fasync_list; //异步唤醒队列
struct file *file; //file回指指针
struct sock *sk; //sock指针
wait_queue_head_t wait; //sock的等待队列，在TCP需要等待时就sleep在这个队列上
short type; //表示该socket在特定协议族下的类型例如SOCK_STREAM,
unsigned char passcred; //在TCP分析中无须考虑
};

//****************************************************************************

struct sock {
/* socket用来对进入的包进行匹配的5大因素 */
__u32 daddr; // dip，Foreign IPv4 addr
__u32 rcv_saddr; // 记录套接字所绑定的地址 Bound local IPv4 addr
__u16 dport; // dport
unsigned short num; /* 套接字所在的端口号, 端口号小于1024的为特权端口, 只有特权用户才能绑定,当用户指定的端口号为零时, 系统将提供一个未分配的用户端口，如果对于raw socket的话，该num又可以用来
保存socket(int family, int type, int protocol)中的protocol，而不是端口号了；在bind时候，会首先
将邦定的源端口号赋予该成员，最终sport成员从该成员出获取源端口号*/
int bound_dev_if; // Bound device index if != 0

/* 主hash链，系统已分配的端口用tcp_hashinfo.__tcp_bhash来索引, 索引槽结构为tcp_bind_hashbucket, 端口绑定结构用tcp_bind_bucket描述,
它包含指向绑定到该端口套接字的指针(owners), 套接字的sk->prev指针指向该绑定结构
*/
struct sock *next;
struct sock **pprev;
/* sk->bind_next和sk->bind_pprev用来描述绑定到同一端口的套接字，例如http服务器 */
struct sock *bind_next;
struct sock **bind_pprev;
struct sock *prev;

volatile unsigned char state, zapped; // Connection state，zapped在TCP分析中无须考虑
__u16 sport; // 源端口，见num

unsigned short family; // 协议族，例如PF_INET
unsigned char reuse; // 地址是否可重用，只有RAW才使用
unsigned char shutdown; // 判断该socket连接在某方向或者双向方向上都已经关闭
#define SHUTDOWN_MASK 3
#define RCV_SHUTDOWN 1
#define SEND_SHUTDOWN 2
```

atomic_t refcnt; // 引用计数
socket_lock_t lock; // 锁标志， 每个socket都有一个自旋锁，该锁在用户上下文和软中断处理时提供了同步机制
typedef struct {
spinlock_t slock;
unsigned int users;
wait_queue_head_t wq;
} socket_lock_t;
wait_queue_head_t *sleep; // Sock所属线程的自身休眠等待队列
struct dst_entry *dst_cache; // 目的地的路由缓存
rwlock_t dst_lock; // 为该socket赋dst_entry值时的锁

/* sock的收发都是要占用内存的，即发送缓冲区和接收缓冲区。 系统对这些内存的使用是有限制的。 通常，每个sock都会从配额里
预先分配一些，这就是forward_alloc， 具体分配时：
1）比如收到一个skb，则要计算到rmem_alloc中，并从forward_alloc中扣除。 接收处理完成后（如用户态读取），则释放skb，并利
用tcp_rfree()把该skb的内存反还给forward_alloc。
2）发送一个skb，也要暂时放到发送缓冲区，这也要计算到wmem_queued中，并从forward_alloc中扣除。真正发送完成后，也释放
skb，并反还forward_alloc。 当从forward_alloc中扣除的时候，有可能forward_alloc不够，此时就要调用tcp_mem_schedule()来增
加forward_alloc，当然，不是随便想加就可以加的，系统对整个TCP的内存使用有总的限制，即sysctl_tcp_mem[3]。也对每个sock
的内存使用分别有限制，即sysctl_tcp_rmem[3]和sysctl_tcp_wmem[3]。只有满足这些限制（有一定的灵活性），forward_alloc才
能增加。 当发现内存紧张的时候，还会调用tcp_mem_reclaim()来回收forward_alloc预先分配的配额。
*/
int rcvbuf; // 接受缓冲区的大小（按字节）
int sndbuf; // 发送缓冲区的大小（按字节）
atomic_t rmem_alloc; // 接受队列中存放的数据的字节数
atomic_t wmem_alloc; // 发送队列中存放的数据的字节数
int wmem_queued; // 所有已经发送的数据的总字节数
int forward_alloc; // 预分配剩余字节数

struct sk_buff_head receive_queue; // 接受队列
struct sk_buff_head write_queue; // 发送队列
atomic_t omem_alloc; // 在TCP分析中无须考虑 * "o" is "option" or "other" */

__u32 saddr; /* 指真正的发送地址，这里需要注意的是，rcv_saddr是记录套接字所绑定的地址，其可能是广播或者多播，对于我们要发送的包来说，只能使用接口的IP地址，而不能使用广播或者多播地址 */
unsigned int allocation; // 分配该sock之skb时选择的模式，GFP_ATOMIC还是GFP_KERNEL等等

volatile char dead, // tcp_close.tcp_listen_stop.inet_sock_release调用sock_orphan将该值置1，表示该socket已经和进程分开，变成孤儿
done, // 用于判断该socket是否已经收到 fin，如果收到则将该成员置1
urginline, // 如果该值被设置为1，表示将紧急数据放于普通数据流中一起处理，而不在另外处理
keepopen, // 是否启动保活定时器
linger, // lingertime一起，指明了close()后保留的时间

```
destroy, // 在TCP分析中无须考虑
no_check, // 是否对发出的skb做校验和，仅对UDP有效
broadcast, // 是否允许广播，仅对UPD有效
bsdism; // 在TCP分析中无须考虑
unsigned char debug; // 在TCP分析中无须考虑
unsigned char rcvtstamp; // 是否将收到skb的时间戳发送给app
unsigned char use_write_queue; // 在init中该值被初始化为1
unsigned char userlocks; // 包括如下几种值的组合，从而改变收包等操作的执行顺序
#define SOCK_SNDBUF_LOCK 1
#define SOCK_RCVBUF_LOCK 2
#define SOCK_BINDADDR_LOCK 4
#define SOCK_BINDPORT_LOCK 8
int route_caps; // 指示本sock用到的路由的信息
int proc; // 保存用户线程的pid
unsigned long lingertime; // lingertime一起，指明了close()后保留的时间
int hashent; // 存放4元的hash值
struct sock *pair; // 在TCP分析中无须考虑

struct { //当sock被锁定时，收到的数据先放在这里
struct sk_buff *head;
struct sk_buff *tail;
} backlog;

rwlock_t callback_lock; // sock相关函数内部操作的保护锁
struct sk_buff_head error_queue; // 错误报文的队列，很少使用
struct proto *prot; // 例如指向tcp_prot

union { // 私有TCP相关数据保存
struct tcp_opt af_tcp;
.....
} tp_pinfo;

int err, // 保存各种错误，例如ECONNRESET Connection reset by peer，从而会影响到后续流程的处理
err_soft; // 保存各种软错误，例如EPROTO Protocol error，从而会影响到后续流程的处理
unsigned short ack_backlog; // 当前已经accept的数目
unsigned short max_ack_backlog; // 最大可accept的数目
__u32 priority; /* Packet queueing priority，Used to set the TOS field. Packets with a higher priority may be proc
essed first, depending on the device’ s queueing discipline. See SO_PRIORITY */
unsigned short type; // 例如SOCK_STREAM，SOCK_DGRAM或者SOCK_RAW等
unsigned char localroute; // Route locally only if set – set by SO_DONTROUTE option.
unsigned char protocol; // socket(int family, int type, int protocol)中的protocol
struct ucred peercred; // 在TCP分析中无须考虑
int rcvlowat; /* 声明在开始发送 数据 (SO_SNDLOWAT) 或正在接收数据的用户 (SO_RCVLOWAT) 传递数据之
前缓冲区内的最小字节数. 在 Linux 中这两个值是不可改变的, 固定为 1 字节. */
long rcvtimeo; // 接收时的超时设定, 并在超时时报错
long sndtimeo; // 发送时的超时设定, 并在超时时报错

union { // 私有inet相关数据保存
```

```
struct inet_opt af_inet;
.....
} protinfo;

/* the timer is used for SO_KEEPALIVE (i.e. sending occasional keepalive probes to a remote site – by default, set to 2 hours in
stamp is simply the time that the last packet was received. */
struct timer_list timer;
struct timeval stamp;
struct socket *socket; // 对应的socket
void *user_data; // 私有数据，在TCP分析中无须考虑

/* The state_change operation is called whenever the status of the socket is changed. Similarly, data_ready is called
when data have been received, write_space when free memory available for writing has increased and error_report
when an error occurs, backlog_rcv when socket locked, putting skb to backlog, destruct for release this sock*/
void (*state_change)(struct sock *sk);
void (*data_ready)(struct sock *sk,int bytes);
void (*write_space)(struct sock *sk);
void (*error_report)(struct sock *sk);
int (*backlog_rcv) (struct sock *sk, struct sk_buff *skb);
void (*destruct)(struct sock *sk);
};

//*************************************************************************
```