Sheet 3

sock structure
/*
 * INET  An implementation of the TCP/IP protocol suite for the LINUX
 *  operating system.  INET is implemented using the BSD Socket
 *  interface as the means of communication with the user level.
 *  Definitions for the AF_INET socket handler.
 *  Version: @(#)sock.h  1.0.4  05/13/93
 *  Authors:  Ross Biro, <bir7@leland.Stanford.Edu>
 *            Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *            Corey Minyard <wf-rch!minyard@relay.EU.net>
 *            Florian La Roche <flla@stud.uni-sb.de>
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version
 *  2 of the License, or (at your option) any later version.
 */

THIS IS A COMPLEX DATA STRUCTURE, WHICH CONTAINS STUFF THAT DOESN'T REALLY BELONG HERE, BUT WHICH IS
HERE FOR HISTORICAL REASONS. I HAVE CHANGED THE ORDER OF THIS SLIGHTLY SO THAT IT IS MORE LOGICAL AND I
HAVE DELETED QUITE A LOT OF IMPORTANT STUFF (e.g all the locking code and much of the TCP related code), TO SHOW
THE BASIC STRUCTURE MORE CLEARLY.
This structure is initialised in the following sequence:

At the end of net/ipv4/af_inet.c you will see a call to `module_init(inet_init)`. As described in the definition of `include/linux/init.h::module_init`, this is a marker for a driver initialisation point, which is called when the kernel boots or when the module is loaded.

`net/ipv4/af_inet.c::inet_init` calls `net/socket.c::sock_register`. This latter routine is called by all protocol handlers that want to advertise their address family. It creates one entry per address family in `net/socket.c::net_families[family]` of type `include/linux/net.h::net_proto_family`. This has a field `create` which is used to create a socket of that given family type. In this case, this routine is set to point to `net/ipv4/af_inet.c::inet_create`.

Socket creation: `net/socket.c::sock_create` -> calls `create` on the appropriate network family. In our case, this will call through to the `net/ipv4/af_inet.c::inet_create`, as stored above. That initialises the sock datastructure, partly directly and partly by calling `net/core/sock.c::sock_init_data`.

```c
struct sock { 

/* Socket demultiplex comparisons on incoming packets. */
__u32    daddr;  /* Foreign IPv4 addr */
__u32    rcv_saddr; /* Bound local IPv4 addr */
```

The following are the source and destination information that must be entered into each IP packet. There appear to be two sender addresses. `rcv_saddr` is the one used by hash lookups, and `saddr` is used for transmit. In the BSD API these are almost always the same.
The next and prev components link sockets with the same hash value in the various socket hash tables. So, for example, in net/ipv4/udp.c you find a definition of udp_hash, which is hashed on a port number. This is an open hash table of struct socks which use linked lists, linked on the next and pprev values below.

TCP uses both the next and pprev fields above and the bind_next and bind_pprev and prev fields below for local binding TCP hash as well as for fast bind/connect.

In our case this will be PF_INET

```c
#include <net/sock.h>

__u32    saddr;  /* Sending source */
__u16    dport;  /* Destination port */
__u16    sport;  /* Source port */

unsigned short  num;   /* Local port */

/* Main hash linkage for various protocol lookup tables. */
struct sock   *next;
struct sock   **pprev;
struct sock   *bind_next;
struct sock   **bind_pprev;
struct sock   *prev;

unsigned short  family;  /* Address family */
unsigned short  type;
```

Operation vector for the protocol with which this socket is associated. In this case, can be `include/net/ipv4/tcp_ipv4.c::tcp_prot`, `net/ipv4/udp.c::udp_prot`, or `net/ipv4/raw.c::raw_prot`

```c
struct proto *prot;
```

In our case `include/linux/in,h::IPPROTO_TCP`, `include/linux/in,h::IPPROTO_UDP`, or `include/linux/in,h::IPPROTO_IP`

```c
unsigned char protocol;
```

State is dependent on protocol – main use is to drive TCP protocol state machine e.g. look for the enum with `TCP_ESTABLISHED` in it in `include/linux/tcp.h`

```c
volatile unsigned char state; /* Connection state */
```

Used when waiting for something to happen with this socket, e.g. waiting for connect in `net/ipv4/af_inet.c::inet_wait_for_connect`, `net/ipv4/tcp.c::wait_for_tcp_connect` and waiting for memory as in `net/ipv4/tcp.c::wait_for_tcp_memory`

```c
wait_queue_head_t *sleep; /* Sock wait queue */
struct dst_entry *dst_cache; /* Destination cache */
```

Packet queues. Note that there is also an error_queue, which I removed, but it's rarely used. See, for example, `net/ipv4/udp.c::udp_queue_rcv_skb` in which a call is made to `include/net/sock.h::sock_queue_rcv_skb`. You can see the write queue in use in `net/ipv4/tcp_output.c::tcp_send_skb`

```c
struct sk_buff_head receive_queue; /* Incoming packets */
struct sk_buff_head write_queue; /* Packet sending queue */
```
Space allocation variables.

atomic_t rmem_alloc;  /* Receive queue bytes committed */
atomic_t wmem_alloc;  /* Transmit queue bytes committed */
atomic_t omem_alloc;  /* "o" is "option" or "other" */
int wmem_queued;  /* Persistent queue size */
int forward_alloc; /* Space allocated forward. */

Allocation is the priority with which memory is requested for this socket

unsigned int allocation;  /* Allocation mode */

Maximum amount of memory that can be requested for this socket when sending or receiving packets

int rcvbuf;  /* Size of receive buffer in bytes */
int sndbuf;  /* Size of send buffer in bytes */

A non zero value means that we are allowed to reuse port numbers for ports that are in the TIME_WAIT state.

unsigned char reuse;  /* SO_REUSEADDR setting */

This says something about the way we are shutting down.

unsigned char shutdown;

The volatile keyword is used when we have something that might change as a result of an external event, and where the compiler will reuse the physical address rather than optimising access. E.g. if my code looks like

A = sk->dead;
B = sk->dead;
then the compiler will do both dereferences. If dead was not volatile, the compiler would normally optimise this to
A = B = sk->dead i.e. it would only do one dereference of sk. This is not helpful if its value is changes by an external agency
in between A’s access and B’s. In any case, these are various options that can be set for a socket.

```
volatile char  dead, done, urginline, keepopen, linger, destroy,
no_check, broadcast, bsdism;
unsigned long  lingertime;
```

SO_TIMESTAMP option – if enabled then recvmsg returns a timestamp corresponding to when datagram was received.
```
unsigned char  rcvtstamp;
```

Says something about the features of the network device, like whether it can do the checksumming of TCP/UDP packets,
and whether it can DMA. Look for NETIF_F_* in include/linux/netdevice.h:net_device
```
int     route_caps;
```

The proc variable is used to contain a process or process group which will be sent a signal on receipt of out-of-band data
```
int     proc;
```

Used when we have peered sockets, such as with unix (local) sockets. See e.g. net/unix/af_unix.c
```
struct sock   *pair;
```

A process may ‘lock’ socket state so that it can’t be changed. In particular this means that it can’t be changed by bottom
half (interrupt driven) handlers i.e. arriving packets are blocked so we don’t get any new data or changes to the state here.
Whilst locked, bottom half processing can add packets to the backlog queue.
/* The backlog queue is special, it is always used with
 * the per-socket spinlock held and requires low latency
 * access. Therefore we special case its implementation.
 */

struct {
    struct sk_buff *head;
    struct sk_buff *tail;
} backlog;

TCP stuff – there's more stuff that I've deleted and some of the options described above only really apply to TCP

union {
    struct tcp_opt af_tcp;
    struct raw_opt tp_raw4;
} tp_pinfo;

int     hashent;

Error conditions
int     err, err_soft; /* Soft holds errors that don't
cause failure but are the cause
of a persistent failure not just
'timed out' */
backlog is the second parameter to the listen routine. It represents the maximum number of pending connections there can be. Here, max_ack_backlog is this number and ack_backlog is a count of the number of connections pending at any given time. The latter is manipulated using helper routines in include/net/tcp.h

```
unsigned short  max_ack_backlog;
unsigned short  ack_backlog;
```

Used to set the TOS field. Packets with a higher priority may be processed first, depending on the device’s queueing discipline. See SO_PRIORITY

```
__u32    priority;
```

Route locally only if set – set by SO_DONTROUTE option.

```
unsigned char  localroute;  /* Route locally only */
```

From SO_PEERCRED option

```
struct ucred   peercred;
```

From SO_RCVLOWAT

```
int     rcvlowat;
```

From SO_RCVTIMEO

```
long     rcvtimeo;
```

From SO_SNDTIMEO

```
long     sndtimeo;
```

Private data for each address family (truncated)

```
/* This is where all the private (optional) areas that don't
 * overlap will eventually live.
```
union {
    void    *destruct_hook;
    struct unix_opt  af_unix;
    #if defined(CONFIG_INET) || defined (CONFIG_INET_MODULE)
    struct inet_opt  af_inet;
    #endif
} protinfo;

Timer functions. You'll find a lot of useful timer stuff in include/linux/timer.h and kernel/timer.c In this case, the timer is used for SO_KEEPALIVE (i.e. sending occasional keepalive probes to a remote site – by default, set to 2 hours in include/net/tcp.h). stamp is simply the time that the last packet was received.

/* This part is used for the timeout functions. */
struct timer_list  timer;  /* This is the sock cleanup timer. */
struct timeval  stamp;

A backpointer to the enclosing include/linux/net.h::socket structure.

/* Identd and reporting IO signals */
struct socket  *socket;

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.

/* Callbacks */
include/net/sock.h

216    void (*state_change)(struct sock *sk);
217    void (*data_ready)(struct sock *sk, int bytes);
218    void (*write_space)(struct sock *sk);
219    void (*error_report)(struct sock *sk);
220
221    int (*backlog_rcv)(struct sock *sk, struct sk_buff *skb);
222
223    Get rid of the socket.
224    void (*destruct)(struct sock *sk);
225    );