Linux Networking and Network Devices APIs

Table of Contents

Chapter 1. Linux Networking

Table of Contents

Networking Base Types

Name

enum sock_type — Socket types

Synopsis

```
enum sock_type {
  SOCK_STREAM,
  SOCK_DGRAM,
  SOCK_RAW,
  SOCK_RDM,
  SOCK_SEQPACKET,
  SOCK_DCCP,
  SOCK_PACKET
};
```

Constants

SOCK_STREAM

stream (connection) socket

SOCK_DGRAM

datagram (conn.less) socket

SOCK_RAW

raw socket

SOCK_RDM

reliably-delivered message

SOCK_SEQPACKET

sequential packet socket

SOCK_DCCP

Datagram Congestion Control Protocol socket

SOCK_PACKET

linux specific way of getting packets at the dev level. For writing rarp and other similar things on the user level.

Description

When adding some new socket type please grep ARCH_HAS_SOCKET_TYPE include/asm-* /socket.h, at least MIPS overrides this enum for binary compat reasons.

Name

struct socket — general BSD socket

Synopsis

```
struct socket {
  socket_state state;
  short type;
  unsigned long flags;
  struct fasync_struct * fasync_list;
  wait_queue_head_t wait;
  struct file * file;
  struct sock * sk;
  const struct proto_ops * ops;
};
```

Members

state

socket state (SS_CONNECTED, etc)

type

socket type (SOCK_STREAM, etc)

flags

socket flags (SOCK_ASYNC_NOSPACE, etc)

fasync_list

Asynchronous wake up list

wait

wait queue for several uses

file

File back pointer for gc

sk

internal networking protocol agnostic socket representation

ops

protocol specific socket operations

Socket Buffer Functions

Name

struct skb_shared_hwtstamps — hardware time stamps

Synopsis

```
struct skb_shared_hwtstamps {
  ktime_t hwtstamp;
  ktime_t syststamp;
};
```

Members

hwtstamp

hardware time stamp transformed into duration since arbitrary point in time

syststamp

hwtstamp transformed to system time base

Description

Software time stamps generated by `ktime_get_real` are stored in skb->tstamp. The relation between the different kinds of time

stamps is as follows

syststamp and tstamp can be compared against each other in arbitrary combinations. The accuracy of a syststamp/tstamp/“syststamp from other device” comparison is limited by the accuracy of the transformation into system time base. This depends on the device driver and its underlying hardware.

hwtstamps can only be compared against other hwtstamps from the same device.

This structure is attached to packets as part of the skb_shared_info. Use `skb_hwtstamps` to get a pointer.

Name

struct skb_shared_tx — instructions for time stamping of outgoing packets

Synopsis

```
struct skb_shared_tx {
  struct {unnamed_struct};
```

```
  __u8 flags;
};
```

Members

{unnamed_struct}

anonymous

flags

all shared_tx flags

Description

These flags are attached to packets as part of the skb_shared_info. Use `skb_tx` to get a pointer.

Name

struct sk_buff — socket buffer

Synopsis

```
struct sk_buff {
  struct sk_buff * next;
  struct sk_buff * prev;
  struct sock * sk;
  ktime_t tstamp;
  struct net_device * dev;
  unsigned long _skb_dst;
#ifdef CONFIG_XFRM
  struct sec_path * sp;
#endif
  char cb[48];
  unsigned int len;
  unsigned int data_len;
  __u16 mac_len;
  __u16 hdr_len;
  union {unnamed_union};
  __u32 priority;
  __u8 local_df:1;
  __u8 cloned:1;
  __u8 ip_summed:2;
  __u8 nohdr:1;
  __u8 nfctinfo:3;
  __u8 pkt_type:3;
  __u8 fclone:2;
  __u8 ipvs_property:1;
  __u8 peeked:1;
  __u8 nf_trace:1;
  __be16 protocol;
  void (* destructor) (struct sk_buff *skb);
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
```

```
  struct nf_conntrack * nfct;
  struct sk_buff * nfct_reasm;
#endif
#ifdef CONFIG_BRIDGE_NETFILTER
  struct nf_bridge_info * nf_bridge;
#endif
  int iif;
  __u16 queue_mapping;
#ifdef CONFIG_NET_SCHED
  __u16 tc_index;
#ifdef CONFIG_NET_CLS_ACT
  __u16 tc_verd;
#endif
#endif
#ifdef CONFIG_IPV6_NDISC_NODETYPE
  __u8 ndisc_nodetype:2;
#endif
#ifdef CONFIG_NET_DMA
  dma_cookie_t dma_cookie;
#endif
#ifdef CONFIG_NETWORK_SECMARK
  __u32 secmark;
#endif
  __u32 mark;
  __u16 vlan_tci;
  sk_buff_data_t transport_header;
  sk_buff_data_t network_header;
  sk_buff_data_t mac_header;
  sk_buff_data_t tail;
  sk_buff_data_t end;
  unsigned char * head;
  unsigned char * data;
  unsigned int truesize;
  atomic_t users;
};
```

Members

next

Next buffer in list

prev

Previous buffer in list

sk

Socket we are owned by

tstamp

Time we arrived

dev

Device we arrived on/are leaving by

_skb_dst

destination entry

sp

the security path, used for xfrm

cb[48]

Control buffer. Free for use by every layer. Put private vars here

len

Length of actual data

data_len

Data length

mac_len

Length of link layer header

hdr_len

writable header length of cloned skb

{unnamed_union}

anonymous

priority

Packet queueing priority

local_df

allow local fragmentation

cloned

Head may be cloned (check refcnt to be sure)

ip_summed

Driver fed us an IP checksum

nohdr

Payload reference only, must not modify header

nfctinfo

Relationship of this skb to the connection

pkt_type

Packet class

fclone

skbuff clone status

ipvs_property

skbuff is owned by ipvs

peeked

this packet has been seen already, so stats have been done for it, don't do them again

nf_trace

netfilter packet trace flag

protocol

Packet protocol from driver

destructor

Destruct function

nfct

Associated connection, if any

nfct_reasm

netfilter conntrack re-assembly pointer

nf_bridge

Saved data about a bridged frame - see br_netfilter.c

iif

ifindex of device we arrived on

queue_mapping

Queue mapping for multiqueue devices

tc_index

Traffic control index

tc_verd

traffic control verdict

ndisc_nodetype

router type (from link layer)

dma_cookie

a cookie to one of several possible DMA operations done by skb DMA functions

secmark

security marking

mark

Generic packet mark

vlan_tci

vlan tag control information

transport_header

Transport layer header

network_header

Network layer header

mac_header

Link layer header

tail

Tail pointer

end

End pointer

head

Head of buffer

data

Data head pointer

truesize

Buffer size

users

User count - see {datagram,tcp}.c

Name

skb_queue_empty — check if a queue is empty

Synopsis

```
int skb_queue_empty (list);

const struct sk_buff_head *  list;
```

Arguments

list

queue head

Description

Returns true if the queue is empty, false otherwise.

Name

skb_queue_is_last — check if skb is the last entry in the queue

Synopsis

```
bool skb_queue_is_last (list,
                        skb);

const struct sk_buff_head *  list;
const struct sk_buff *       skb;
```

Arguments

list

queue head

skb

buffer

Description

Returns true if *skb* is the last buffer on the list.

Name

skb_queue_is_first — check if skb is the first entry in the queue

Synopsis

```
bool skb_queue_is_first (list,
                         skb);

const struct sk_buff_head * list;
const struct sk_buff *      skb;
```

Arguments

list

queue head

skb

buffer

Description

Returns true if *skb* is the first buffer on the list.

Name

skb_queue_next — return the next packet in the queue

Synopsis

```
struct sk_buff * skb_queue_next (list,
                                 skb);

const struct sk_buff_head * list;
const struct sk_buff *      skb;
```

Arguments

list

queue head

skb

current buffer

Description

Return the next packet in *list* after *skb*. It is only valid to call this if skb_queue_is_last evaluates to false.

Name

skb_queue_prev — return the prev packet in the queue

Synopsis

```
struct sk_buff * skb_queue_prev (list,
                                 skb);

const struct sk_buff_head *  list;
const struct sk_buff *       skb;
```

Arguments

list

queue head

skb

current buffer

Description

Return the prev packet in *list* before *skb*. It is only valid to call this if skb_queue_is_first evaluates to false.

Name

skb_get — reference buffer

Synopsis

```
struct sk_buff * skb_get (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to reference

Description

Makes another reference to a socket buffer and returns a pointer to the buffer.

Name

skb_cloned — is the buffer a clone

Synopsis

```
int skb_cloned (skb);

const struct sk_buff *  skb;
```

Arguments

skb

buffer to check

Description

Returns true if the buffer was generated with `skb_clone` and is one of multiple shared copies of the buffer. Cloned buffers are shared data so must not be written to under normal circumstances.

Name

skb_header_cloned — is the header a clone

Synopsis

```
int skb_header_cloned (skb);

const struct sk_buff *  skb;
```

Arguments

skb

buffer to check

Description

Returns true if modifying the header part of the buffer requires the data to be copied.

Name

skb_header_release — release reference to header

Synopsis

```
void skb_header_release (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to operate on

Description

Drop a reference to the header part of the buffer. This is done by acquiring a payload reference. You must not read from the header part of skb->data after this.

Name

skb_shared — is the buffer shared

Synopsis

```
int skb_shared (skb);

const struct sk_buff *  skb;
```

Arguments

skb

buffer to check

Description

Returns true if more than one person has a reference to this buffer.

Name

skb_share_check — check if buffer is shared and if so clone it

Synopsis

```
struct sk_buff * skb_share_check (skb,
                                  pri);

struct sk_buff *  skb;
gfp_t             pri;
```

Arguments

skb

buffer to check

pri

priority for memory allocation

Description

If the buffer is shared the buffer is cloned and the old copy drops a reference. A new clone with a single reference is returned. If the buffer is not shared the original buffer is returned. When being called from interrupt status or with spinlocks held pri must be GFP_ATOMIC.

NULL is returned on a memory allocation failure.

Name

skb_unshare — make a copy of a shared buffer

Synopsis

```
struct sk_buff * skb_unshare (skb,
                              pri);

struct sk_buff *  skb;
gfp_t             pri;
```

Arguments

skb

buffer to check

pri

priority for memory allocation

Description

If the socket buffer is a clone then this function creates a new copy of the data, drops a reference count on the old copy and returns the new copy with the reference count at 1. If the buffer is not a clone the original buffer is returned. When called with a spinlock held or from interrupt state *pri* must be GFP_ATOMIC

NULL is returned on a memory allocation failure.

Name

skb_peek —

Synopsis

```
struct sk_buff * skb_peek (list_);

struct sk_buff_head *  list_;
```

Arguments

*list_*

list to peek at

Description

Peek an sk_buff. Unlike most other operations you _MUST_ be careful with this one. A peek leaves the

buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns NULL for an empty list or a pointer to the head element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

Name

skb_peek_tail —

Synopsis

```
struct sk_buff * skb_peek_tail (list_);

struct sk_buff_head *  list_;
```

Arguments

*list_*

list to peek at

Description

Peek an sk_buff. Unlike most other operations you _MUST_ be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns NULL for an empty list or a pointer to the tail element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

Name

skb_queue_len — get queue length

Synopsis

```
__u32 skb_queue_len (list_);

const struct sk_buff_head *  list_;
```

Arguments

*list_*

list to measure

Description

Return the length of an sk_buff queue.

Name

__skb_queue_head_init — initialize non-spinlock portions of sk_buff_head

Synopsis

```
void __skb_queue_head_init (list);

struct sk_buff_head *  list;
```

Arguments

list

queue to initialize

Description

This initializes only the list and queue length aspects of an sk_buff_head object. This allows to initialize the list aspects of an sk_buff_head without reinitializing things like the spinlock. It can also be used for on-stack sk_buff_head objects where the spinlock is known to not be used.

Name

skb_queue_splice — join two skb lists, this is designed for stacks

Synopsis

```
void skb_queue_splice (list,
                       head);

const struct sk_buff_head *  list;
struct sk_buff_head *        head;
```

Arguments

list

the new list to add

head

the place to add it in the first list

Name

skb_queue_splice_init — join two skb lists and reinitialise the emptied list

Synopsis

```
void skb_queue_splice_init (list,
                            head);

struct sk_buff_head *  list;
struct sk_buff_head *  head;
```

Arguments

list

the new list to add

head

the place to add it in the first list

Description

The list at *list* is reinitialised

Name

skb_queue_splice_tail — join two skb lists, each list being a queue

Synopsis

```
void skb_queue_splice_tail (list,
                            head);

const struct sk_buff_head *  list;
struct sk_buff_head *        head;
```

Arguments

list

the new list to add

head

the place to add it in the first list

Name

skb_queue_splice_tail_init — join two skb lists and reinitialise the emptied list

Synopsis

```
void skb_queue_splice_tail_init (list,
                                 head);

struct sk_buff_head * list;
struct sk_buff_head * head;
```

Arguments

list

the new list to add

head

the place to add it in the first list

Description

Each of the lists is a queue. The list at *list* is reinitialised

---

Name

__skb_queue_after — queue a buffer at the list head

Synopsis

```
void __skb_queue_after (list,
                        prev,
                        newsk);

struct sk_buff_head * list;
struct sk_buff *      prev;
```

```
struct sk_buff *        newsk;
```

Arguments

list

list to use

prev

place after this buffer

newsk

buffer to queue

Description

Queue a buffer int the middle of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

Name

skb_headroom — bytes at buffer head

Synopsis

```
unsigned int skb_headroom (skb);

const struct sk_buff *  skb;
```

Arguments

skb

buffer to check

Description

Return the number of bytes of free space at the head of an sk_buff.

Name

skb_tailroom — bytes at buffer end

Synopsis

```
int skb_tailroom (skb);

const struct sk_buff *  skb;
```

Arguments

skb

buffer to check

Description

Return the number of bytes of free space at the tail of an sk_buff

---

Name

skb_reserve — adjust headroom

Synopsis

```
void skb_reserve (skb,
                  len);

struct sk_buff *  skb;
int               len;
```

Arguments

skb

buffer to alter

len

bytes to move

Description

Increase the headroom of an empty sk_buff by reducing the tail room. This is only allowed for an empty buffer.

Name

pskb_trim_unique — remove end from a paged unique (not cloned) buffer

Synopsis

```
void pskb_trim_unique (skb,
                       len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to alter

len

new length

Description

This is identical to pskb_trim except that the caller knows that the skb is not cloned so we should never get an error due to out- of-memory.

Name

skb_orphan — orphan a buffer

Synopsis

```
void skb_orphan (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to orphan

Description

If a buffer currently has an owner then we call the owner's destructor function and make the *skb* unowned. The buffer continues to exist but is no longer charged to its former owner.

Name

__dev_alloc_skb — allocate an skbuff for receiving

Synopsis

```
struct sk_buff * __dev_alloc_skb (length,
                                  gfp_mask);

unsigned int  length;
gfp_t         gfp_mask;
```

Arguments

length

length to allocate

gfp_mask

get_free_pages mask, passed to alloc_skb

Description

Allocate a new sk_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory.

Name

netdev_alloc_skb — allocate an skbuff for rx on a specific device

Synopsis

```
struct sk_buff * netdev_alloc_skb (dev,
                                   length);

struct net_device *  dev;
unsigned int         length;
```

Arguments

dev

network device to receive on

length

length to allocate

Description

Allocate a new sk_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

Name

netdev_alloc_page — allocate a page for ps-rx on a specific device

Synopsis

```
struct page * netdev_alloc_page (dev);

struct net_device *  dev;
```

Arguments

dev

network device to receive on

Description

Allocate a new page node local to the specified device.

NULL is returned if there is no free memory.

Name

skb_clone_writable — is the header of a clone writable

Synopsis

```
int skb_clone_writable (skb,
                        len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to check

len

length up to which to write

Description

Returns true if modifying the header part of the cloned buffer does not requires the data to be copied.

Name

skb_cow — copy header of skb when it is required

Synopsis

```
int skb_cow (skb,
             headroom);

struct sk_buff *  skb;
unsigned int      headroom;
```

Arguments

skb

buffer to cow

headroom

needed headroom

Description

If the skb passed lacks sufficient headroom or its data part is shared, data is reallocated. If reallocation

fails, an error is returned and original skb is not changed.

The result is skb with writable area skb->head...skb->tail and at least *headroom* of space at head.

Name

skb_cow_head — skb_cow but only making the head writable

Synopsis

```
int skb_cow_head (skb,
                  headroom);

struct sk_buff *  skb;
unsigned int      headroom;
```

Arguments

skb

buffer to cow

headroom

needed headroom

Description

This function is identical to skb_cow except that we replace the skb_cloned check by skb_header_cloned. It should be used when you only need to push on some header and do not need to modify the data.

Name

skb_padto — pad an skbuff up to a minimal size

Synopsis

```
int skb_padto (skb,
               len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to pad

len

minimal length

Description

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error.

Name

skb_linearize — convert paged skb to linear one

Synopsis

```
int skb_linearize (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to linarize

Description

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

Name

skb_linearize_cow — make sure skb is linear and writable

Synopsis

```
int skb_linearize_cow (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to process

Description

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

Name

skb_postpull_rcsum — update checksum for received skb after pull

Synopsis

```
void skb_postpull_rcsum (skb,
                         start,
                         len);

struct sk_buff *  skb;
const void *      start;
unsigned int      len;
```

Arguments

skb

buffer to update

start

start of data before pull

len

length of data pulled

Description

After doing a pull on a received packet, you need to call this to update the CHECKSUM_COMPLETE checksum, or set ip_summed to CHECKSUM_NONE so that it can be recomputed from scratch.

Name

pskb_trim_rcsum — trim received skb and update checksum

Synopsis

```
int pskb_trim_rcsum (skb,
                     len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to trim

len

new length

Description

This is exactly the same as pskb_trim except that it ensures the checksum of received packets are still valid after the operation.

Name

skb_get_timestamp — get timestamp from a skb

Synopsis

```
void skb_get_timestamp (skb,
                        stamp);

const struct sk_buff *  skb;
struct timeval *        stamp;
```

Arguments

skb

skb to get stamp from

stamp

pointer to struct timeval to store stamp in

Description

Timestamps are stored in the skb as offsets to a base timestamp. This function converts the offset back to a struct timeval and stores it in stamp.

Name

skb_checksum_complete — Calculate checksum of an entire packet

Synopsis

```
__sum16 skb_checksum_complete (skb);

struct sk_buff *  skb;
```

Arguments

skb

packet to process

Description

This function calculates the checksum over the entire packet plus the value of skb->csum. The latter can be used to supply the checksum of a pseudo header as used by TCP/UDP. It returns the checksum.

For protocols that contain complete checksums such as ICMP/TCP/UDP, this function can be used to verify that checksum on received packets. In that case the function should return zero if the checksum is correct. In particular, this function will return zero if skb->ip_summed is CHECKSUM_UNNECESSARY which indicates that the hardware has already verified the correctness of the checksum.

Name

struct sock_common — minimal network layer representation of sockets

Synopsis

```
struct sock_common {
  union {unnamed_union};
  atomic_t skc_refcnt;
  unsigned int skc_hash;
  unsigned short skc_family;
  volatile unsigned char skc_state;
  unsigned char skc_reuse;
  int skc_bound_dev_if;
```

```
  struct hlist_node skc_bind_node;
  struct proto * skc_prot;
#ifdef CONFIG_NET_NS
  struct net * skc_net;
#endif
};
```

Members

{unnamed_union}

anonymous

skc_refcnt

reference count

skc_hash

hash value used with various protocol lookup tables

skc_family

network address family

skc_state

Connection state

skc_reuse

SO_REUSEADDR setting

skc_bound_dev_if

bound device index if != 0

skc_bind_node

bind hash linkage for various protocol lookup tables

skc_prot

protocol handlers inside a network family

skc_net

reference to the network namespace of this socket

Description

This is the minimal network layer representation of sockets, the header for struct sock and struct

inet_timewait_sock.

Name

struct sock — network layer representation of sockets

Synopsis

```
struct sock {
  struct sock_common __sk_common;
#define sk_node                 __sk_common.skc_node
#define sk_nulls_node           __sk_common.skc_nulls_node
#define sk_refcnt               __sk_common.skc_refcnt
#define sk_copy_start           __sk_common.skc_hash
#define sk_hash                 __sk_common.skc_hash
#define sk_family               __sk_common.skc_family
#define sk_state                __sk_common.skc_state
#define sk_reuse                __sk_common.skc_reuse
#define sk_bound_dev_if         __sk_common.skc_bound_dev_if
#define sk_bind_node            __sk_common.skc_bind_node
#define sk_prot                 __sk_common.skc_prot
#define sk_net                  __sk_common.skc_net
  unsigned int sk_shutdown:2;
  unsigned int sk_no_check:2;
  unsigned int sk_userlocks:4;
  unsigned int sk_protocol:8;
  unsigned int sk_type:16;
  int sk_rcvbuf;
  socket_lock_t sk_lock;
  struct sk_backlog;
  wait_queue_head_t * sk_sleep;
  struct dst_entry * sk_dst_cache;
#ifdef CONFIG_XFRM
  struct xfrm_policy * sk_policy[2];
#endif
  rwlock_t sk_dst_lock;
  atomic_t sk_rmem_alloc;
  atomic_t sk_wmem_alloc;
  atomic_t sk_omem_alloc;
  int sk_sndbuf;
  struct sk_buff_head sk_receive_queue;
  struct sk_buff_head sk_write_queue;
#ifdef CONFIG_NET_DMA
  struct sk_buff_head sk_async_wait_queue;
#endif
  int sk_wmem_queued;
  int sk_forward_alloc;
  gfp_t sk_allocation;
  int sk_route_caps;
  int sk_gso_type;
  unsigned int sk_gso_max_size;
  int sk_rcvlowat;
  unsigned long sk_flags;
  unsigned long sk_lingertime;
  struct sk_buff_head sk_error_queue;
  struct proto * sk_prot_creator;
```

```
  rwlock_t sk_callback_lock;
  int sk_err;
  int sk_err_soft;
  atomic_t sk_drops;
  unsigned short sk_ack_backlog;
  unsigned short sk_max_ack_backlog;
  __u32 sk_priority;
  struct ucred sk_peercred;
  long sk_rcvtimeo;
  long sk_sndtimeo;
  struct sk_filter * sk_filter;
  void * sk_protinfo;
  struct timer_list sk_timer;
  ktime_t sk_stamp;
  struct socket * sk_socket;
  void * sk_user_data;
  struct page * sk_sndmsg_page;
  struct sk_buff * sk_send_head;
  __u32 sk_sndmsg_off;
  int sk_write_pending;
#ifdef CONFIG_SECURITY
  void * sk_security;
#endif
  __u32 sk_mark;
  void (* sk_state_change) (struct sock *sk);
  void (* sk_data_ready) (struct sock *sk, int bytes);
  void (* sk_write_space) (struct sock *sk);
  void (* sk_error_report) (struct sock *sk);
  int (* sk_backlog_rcv) (struct sock *sk,struct sk_buff *skb);
  void (* sk_destruct) (struct sock *sk);
};
```

Members

__sk_common

shared layout with inet_timewait_sock

sk_shutdown

mask of SEND_SHUTDOWN and/or RCV_SHUTDOWN

sk_no_check

SO_NO_CHECK setting, wether or not checkup packets

sk_userlocks

SO_SNDBUF and SO_RCVBUF settings

sk_protocol

which protocol this socket belongs in this network family

sk_type

socket type (SOCK_STREAM, etc)

sk_rcvbuf

size of receive buffer in bytes

sk_lock

synchronizer

sk_backlog

always used with the per-socket spinlock held

sk_sleep

sock wait queue

sk_dst_cache

destination cache

sk_policy[2]

flow policy

sk_dst_lock

destination cache lock

sk_rmem_alloc

receive queue bytes committed

sk_wmem_alloc

transmit queue bytes committed

sk_omem_alloc

"o“ is ”option“ or ”other"

sk_sndbuf

size of send buffer in bytes

sk_receive_queue

incoming packets

sk_write_queue

Packet sending queue

sk_async_wait_queue

DMA copied packets

sk_wmem_queued

persistent queue size

sk_forward_alloc

space allocated forward

sk_allocation

allocation mode

sk_route_caps

route capabilities (e.g. `NETIF_F_TSO`)

sk_gso_type

GSO type (e.g. `SKB_GSO_TCPV4`)

sk_gso_max_size

Maximum GSO segment size to build

sk_rcvlowat

`SO_RCVLOWAT` setting

sk_flags

`SO_LINGER` (l_onoff), `SO_BROADCAST`, `SO_KEEPALIVE`, `SO_OOBINLINE` settings, `SO_TIMESTAMPING` settings

sk_lingertime

`SO_LINGER` l_linger setting

sk_error_queue

rarely used

sk_prot_creator

sk_prot of original sock creator (see ipv6_setsockopt, IPV6_ADDRFORM for instance)

sk_callback_lock

used with the callbacks in the end of this struct

sk_err

last error

sk_err_soft

errors that don't cause failure but are the cause of a persistent failure not just 'timed out'

sk_drops

raw/udp drops counter

sk_ack_backlog

current listen backlog

sk_max_ack_backlog

listen backlog set in `listen`

sk_priority

`SO_PRIORITY` setting

sk_peercred

`SO_PEERCRED` setting

sk_rcvtimeo

`SO_RCVTIMEO` setting

sk_sndtimeo

`SO_SNDTIMEO` setting

sk_filter

socket filtering instructions

sk_protinfo

private area, net family specific, when not using slab

sk_timer

sock cleanup timer

sk_stamp

time stamp of last packet received

sk_socket

Identd and reporting IO signals

sk_user_data

RPC layer private data

sk_sndmsg_page

cached page for sendmsg

sk_send_head

front of stuff to transmit

sk_sndmsg_off

cached offset for sendmsg

sk_write_pending

a write to stream socket waits to start

sk_security

used by security modules

sk_mark

generic packet mark

sk_state_change

callback to indicate change in the state of the sock

sk_data_ready

callback to indicate there is data to be processed

sk_write_space

callback to indicate there is bf sending space available

sk_error_report

callback to indicate errors (e.g. `MSG_ERRQUEUE`)

sk_backlog_rcv

callback to process the backlog

sk_destruct

called at sock freeing time, i.e. when all refcnt == 0

Name

sk_filter_release —

Synopsis

```
void sk_filter_release (fp);

struct sk_filter *  fp;
```

Arguments

fp

filter to remove

Description

Remove a filter from a socket and release its resources.

Name

sk_wmem_alloc_get — returns write allocations

Synopsis

```
int sk_wmem_alloc_get (sk);

const struct sock *  sk;
```

Arguments

sk

socket

Description

Returns sk_wmem_alloc minus initial offset of one

Name

sk_rmem_alloc_get — returns read allocations

Synopsis

```
int sk_rmem_alloc_get (sk);

const struct sock *  sk;
```

Arguments

sk

socket

Description

Returns sk_rmem_alloc

Name

sk_has_allocations — check if allocations are outstanding

Synopsis

```
int sk_has_allocations (sk);

const struct sock *  sk;
```

Arguments

sk

socket

Description

Returns true if socket has write or read allocations

Name

sk_has_sleeper — check if there are any waiting processes

Synopsis

```
int sk_has_sleeper (sk);

struct sock *  sk;
```

Arguments

sk

socket

Description

Returns true if socket has waiting processes

The purpose of the sk_has_sleeper and sock_poll_wait is to wrap the memory barrier call. They were added due to the race found within the tcp code.

Consider following tcp code paths

CPU1 CPU2

sys_select receive packet ... ... __add_wait_queue update tp->rcv_nxt ... ... tp->rcv_nxt check sock_def_readable ... { schedule ... if (sk->sk_sleep && waitqueue_active(sk->sk_sleep)) wake_up_interruptible(sk->sk_sleep) ... }

The race for tcp fires when the __add_wait_queue changes done by CPU1 stay in its cache, and so does the tp->rcv_nxt update on CPU2 side. The CPU1 could then endup calling schedule and sleep forever if there are no more data on the socket.

The sk_has_sleeper is always called right after a call to read_lock, so we can use smp_mb__after_lock barrier.

Name

sock_poll_wait — place memory barrier behind the poll_wait call.

Synopsis

```
void sock_poll_wait (filp,
                     wait_address,
                     p);

struct file *        filp;
wait_queue_head_t *  wait_address;
poll_table *         p;
```

Arguments

filp

file

wait_address

socket wait queue

p

poll_table

Description

See the comments in the sk_has_sleeper function.

Name

sk_eat_skb — Release a skb if it is no longer needed

Synopsis

```
void sk_eat_skb (sk,
                 skb,
                 copied_early);

struct sock *     sk;
struct sk_buff *  skb;
int               copied_early;
```

Arguments

sk

socket to eat this skb from

skb

socket buffer to eat

copied_early

flag indicating whether DMA operations copied this data early

Description

This routine must be called with interrupts disabled or with the socket locked so that the sk_buff queue operation is ok.

Name

sockfd_lookup — Go from a file number to its socket slot

Synopsis

```
struct socket * sockfd_lookup (fd,
                               err);

int    fd;
int *  err;
```

Arguments

fd

file handle

err

pointer to an error code return

Description

The file handle passed in is locked and the socket it is bound too is returned. If an error occurs the err pointer is overwritten with a negative errno code and NULL is returned. The function checks for both invalid handles and passing a handle which is not a socket.

On a success the socket object pointer is returned.

Name

sock_release — close a socket

Synopsis

```
void sock_release (sock);

struct socket *  sock;
```

Arguments

sock

socket to close

Description

The socket is released from the protocol stack if it has a release callback, and the inode is then released if the socket is bound to an inode not a file.

Name

sock_register — add a socket protocol handler

Synopsis

```
int sock_register (ops);

const struct net_proto_family *  ops;
```

Arguments

ops

description of protocol

Description

This function is called by a protocol handler that wants to advertise its address family, and have it linked into the socket interface. The value ops->family coresponds to the socket system call protocol family.

Name

sock_unregister — remove a protocol handler

Synopsis

```
void sock_unregister (family);

int  family;
```

Arguments

family

protocol family to remove

Description

This function is called by a protocol handler that wants to remove its address family, and have it unlinked from the new socket creation.

If protocol handler is a module, then it can use module reference counts to protect against new references. If protocol handler is not a module then it needs to provide its own protection in the ops->create routine.

Name

skb_over_panic — private function

Synopsis

```
void skb_over_panic (skb,
                     sz,
                     here);

struct sk_buff *  skb;
int               sz;
void *            here;
```

Arguments

skb

buffer

sz

size

here

address

Description

Out of line support code for `skb_put`. Not user callable.

Name

skb_under_panic — private function

Synopsis

```
void skb_under_panic (skb,
                      sz,
                      here);

struct sk_buff *  skb;
int               sz;
void *            here;
```

Arguments

skb

buffer

sz

size

here

address

Description

Out of line support code for `skb_push`. Not user callable.

Name

__alloc_skb — allocate a network buffer

Synopsis

```
struct sk_buff * __alloc_skb (size,
                              gfp_mask,
                              fclone,
                              node);

unsigned int  size;
gfp_t         gfp_mask;
int           fclone;
int           node;
```

Arguments

size

size to allocate

gfp_mask

allocation mask

fclone

allocate from fclone cache instead of head cache and allocate a cloned (child) skb

node

numa node to allocate memory on

Description

Allocate a new sk_buff. The returned buffer has no headroom and a tail room of size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is NULL.

Buffers may only be allocated from interrupts using a *gfp_mask* of GFP_ATOMIC.

Name

__netdev_alloc_skb — allocate an skbuff for rx on a specific device

Synopsis

```
struct sk_buff * __netdev_alloc_skb (dev,
                                     length,
                                     gfp_mask);

struct net_device *  dev;
unsigned int         length;
gfp_t                gfp_mask;
```

Arguments

dev

network device to receive on

length

length to allocate

gfp_mask

get_free_pages mask, passed to alloc_skb

Description

Allocate a new sk_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory.

Name

dev_alloc_skb — allocate an skbuff for receiving

Synopsis

```
struct sk_buff * dev_alloc_skb (length);

unsigned int  length;
```

Arguments

length

length to allocate

Description

Allocate a new sk_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

Name

__kfree_skb — private function

Synopsis

```
void __kfree_skb (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer

Description

Free an sk_buff. Release anything attached to the buffer. Clean the state. This is an internal helper function. Users should always call kfree_skb

---

Name

kfree_skb — free an sk_buff

Synopsis

```
void kfree_skb (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to free

Description

Drop a reference to the buffer and free it if the usage count has hit zero.

Name

consume_skb — free an skbuff

Synopsis

```
void consume_skb (skb);

struct sk_buff *  skb;
```

Arguments

skb

buffer to free

Description

Drop a ref to the buffer and free it if the usage count has hit zero Functions identically to kfree_skb, but kfree_skb assumes that the frame is being dropped after a failure and notes that

---

Name

skb_recycle_check — check if skb can be reused for receive

Synopsis

```
int skb_recycle_check (skb,
                       skb_size);

struct sk_buff *  skb;
int               skb_size;
```

Arguments

skb

buffer

skb_size

minimum receive buffer size

Description

Checks that the skb passed in is not shared or cloned, and that it is linear and its head portion at least as large as skb_size so that it can be recycled as a receive buffer. If these conditions are met, this function does any necessary reference count dropping and cleans up the skbuff as if it just came from `__alloc_skb`.

Name

skb_morph — morph one skb into another

Synopsis

```
struct sk_buff * skb_morph (dst,
                            src);
```

```
struct sk_buff *  dst;
struct sk_buff *  src;
```

Arguments

dst

the skb to receive the contents

src

the skb to supply the contents

Description

This is identical to skb_clone except that the target skb is supplied by the user.

The target skb is returned upon exit.

Name

skb_clone — duplicate an sk_buff

Synopsis

```
struct sk_buff * skb_clone (skb,
                            gfp_mask);

struct sk_buff *  skb;
gfp_t             gfp_mask;
```

Arguments

skb

buffer to clone

gfp_mask

allocation priority

Description

Duplicate an sk_buff. The new one is not owned by a socket. Both copies share the same packet data but not structure. The new buffer has a reference count of 1. If the allocation fails the function returns NULL otherwise the new buffer is returned.

If this function is called from an interrupt gfp_mask must be GFP_ATOMIC.

Name

skb_copy — create private copy of an sk_buff

Synopsis

```
struct sk_buff * skb_copy (skb,
                           gfp_mask);

const struct sk_buff *  skb;
gfp_t                   gfp_mask;
```

Arguments

skb

buffer to copy

gfp_mask

allocation priority

Description

Make a copy of both an sk_buff and its data. This is used when the caller wishes to modify the data and needs a private copy of the data to alter. Returns NULL on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

As by-product this function converts non-linear sk_buff to linear one, so that sk_buff becomes completely private and caller is allowed to modify all the data of returned buffer. This means that this function is not recommended for use in circumstances when only header is going to be modified. Use pskb_copy instead.

Name

pskb_copy — create copy of an sk_buff with private head.

Synopsis

```
struct sk_buff * pskb_copy (skb,
                            gfp_mask);

struct sk_buff *  skb;
gfp_t             gfp_mask;
```

Arguments

skb

buffer to copy

gfp_mask

allocation priority

Description

Make a copy of both an sk_buff and part of its data, located in header. Fragmented data remain shared. This is used when the caller wishes to modify only header of sk_buff and needs private copy of the header to alter. Returns NULL on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

Name

pskb_expand_head — reallocate header of sk_buff

Synopsis

```
int pskb_expand_head (skb,
                      nhead,
                      ntail,
                      gfp_mask);

struct sk_buff *  skb;
int               nhead;
int               ntail;
gfp_t             gfp_mask;
```

Arguments

skb

buffer to reallocate

nhead

room to add at head

ntail

room to add at tail

gfp_mask

allocation priority

Description

Expands (or creates identical copy, if nhead and ntail are zero) header of skb. sk_buff itself is not changed. sk_buff MUST have reference count of 1. Returns zero in the case of success or error, if expansion failed. In the last case, sk_buff is not changed.

All the pointers pointing into skb header may change and must be reloaded after call to this function.

Name

skb_copy_expand — copy and expand sk_buff

Synopsis

```
struct sk_buff * skb_copy_expand (skb,
                                  newheadroom,
                                  newtailroom,
                                  gfp_mask);

const struct sk_buff *  skb;
int                     newheadroom;
int                     newtailroom;
gfp_t                   gfp_mask;
```

Arguments

skb

buffer to copy

newheadroom

new free bytes at head

newtailroom

new free bytes at tail

gfp_mask

allocation priority

Description

Make a copy of both an sk_buff and its data and while doing so allocate additional space.

This is used when the caller wishes to modify the data and needs a private copy of the data to alter as well as more space for new fields. Returns NULL on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass GFP_ATOMIC as the allocation priority if this function is called from an interrupt.

Name

skb_pad — zero pad the tail of an skb

Synopsis

```
int skb_pad (skb,
             pad);

struct sk_buff *  skb;
int               pad;
```

Arguments

skb

buffer to pad

pad

space to pad

Description

Ensure that a buffer is followed by a padding area that is zero filled. Used by network drivers which may DMA or transfer data beyond the buffer end onto the wire.

May return error in out of memory cases. The skb is freed on error.

Name

skb_put — add data to a buffer

Synopsis

```
unsigned char * skb_put (skb,
                         len);
```

```
struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to use

len

amount of data to add

Description

This function extends the used data area of the buffer. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

Name

skb_push — add data to the start of a buffer

Synopsis

```
unsigned char * skb_push (skb,
                          len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to use

len

amount of data to add

Description

This function extends the used data area of the buffer at the buffer start. If this would exceed the total buffer headroom the kernel will panic. A pointer to the first byte of the extra data is returned.

Name

skb_pull — remove data from the start of a buffer

Synopsis

```
unsigned char * skb_pull (skb,
                          len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to use

len

amount of data to remove

Description

This function removes data from the start of a buffer, returning the memory to the headroom. A pointer to the next data in the buffer is returned. Once the data has been pulled future pushes will overwrite the old data.

Name

skb_trim — remove end from a buffer

Synopsis

```
void skb_trim (skb,
               len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to alter

len

new length

Description

Cut the length of a buffer down by removing data from the tail. If the buffer is already under the length specified it is not modified. The skb must be linear.

Name

__pskb_pull_tail — advance tail of skb header

Synopsis

```
unsigned char * __pskb_pull_tail (skb,
                                  delta);

struct sk_buff *  skb;
int               delta;
```

Arguments

skb

buffer to reallocate

delta

number of bytes to advance tail

Description

The function makes a sense only on a fragmented sk_buff, it expands header moving its tail forward and copying necessary data from fragmented part.

sk_buff MUST have reference count of 1.

Returns NULL (and sk_buff does not change) if pull failed or value of new tail of skb in the case of success.

All the pointers pointing into skb header may change and must be reloaded after call to this function.

Name

skb_store_bits — store bits from kernel buffer to skb

Synopsis

```
int skb_store_bits (skb,
                    offset,
                    from,
                    len);

struct sk_buff *  skb;
int               offset;
const void *      from;
int               len;
```

Arguments

skb

destination buffer

offset

offset in destination

from

source buffer

len

number of bytes to copy

Description

Copy the specified number of bytes from the source buffer to the destination skb. This function handles all the messy bits of traversing fragment lists and such.

Name

skb_dequeue — remove from the head of the queue

Synopsis

```
struct sk_buff * skb_dequeue (list);

struct sk_buff_head *  list;
```

Arguments

list

list to dequeue from

Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The head item is returned or NULL if the list is empty.

Name

skb_dequeue_tail — remove from the tail of the queue

Synopsis

```
struct sk_buff * skb_dequeue_tail (list);

struct sk_buff_head *  list;
```

Arguments

list

list to dequeue from

Description

Remove the tail of the list. The list lock is taken so the function may be used safely with other locking list functions. The tail item is returned or NULL if the list is empty.

Name

skb_queue_purge — empty a list

Synopsis

```
void skb_queue_purge (list);

struct sk_buff_head *  list;
```

Arguments

list

list to empty

Description

Delete all buffers on an sk_buff list. Each buffer is removed from the list and one reference dropped. This function takes the list lock and is atomic with respect to other list locking functions.

Name

skb_queue_head — queue a buffer at the list head

Synopsis

```
void skb_queue_head (list,
                     newsk);

struct sk_buff_head *  list;
struct sk_buff *       newsk;
```

Arguments

list

list to use

newsk

buffer to queue

Description

Queue a buffer at the start of the list. This function takes the list lock and can be used safely with other locking sk_buff functions safely.

A buffer cannot be placed on two lists at the same time.

Name

skb_queue_tail — queue a buffer at the list tail

Synopsis

```
void skb_queue_tail (list,
                     newsk);
```

```
struct sk_buff_head *  list;
struct sk_buff *       newsk;
```

Arguments

list

list to use

newsk

buffer to queue

Description

Queue a buffer at the tail of the list. This function takes the list lock and can be used safely with other locking sk_buff functions safely.

A buffer cannot be placed on two lists at the same time.

Name

skb_unlink — remove a buffer from a list

Synopsis

```
void skb_unlink (skb,
                 list);

struct sk_buff *       skb;
struct sk_buff_head *  list;
```

Arguments

skb

buffer to remove

list

list to use

Description

Remove a packet from a list. The list locks are taken and this function is atomic with respect to other list locked calls

You must know what list the SKB is on.

Name

skb_append — append a buffer

Synopsis

```
void skb_append (old,
                 newsk,
                 list);

struct sk_buff *       old;
struct sk_buff *       newsk;
struct sk_buff_head *  list;
```

Arguments

old

buffer to insert after

newsk

buffer to insert

list

list to use

Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

Name

skb_insert — insert a buffer

Synopsis

```
void skb_insert (old,
                 newsk,
                 list);

struct sk_buff *       old;
```

```
struct sk_buff *       newsk;
struct sk_buff_head *  list;
```

Arguments

old

buffer to insert before

newsk

buffer to insert

list

list to use

Description

Place a packet before a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls.

A buffer cannot be placed on two lists at the same time.

Name

skb_split — Split fragmented skb to two parts at length len.

Synopsis

```
void skb_split (skb,
                skb1,
                len);

struct sk_buff *  skb;
struct sk_buff *  skb1;
const u32         len;
```

Arguments

skb

the buffer to split

skb1

the buffer to receive the second part

len

new length for skb

Name

skb_prepare_seq_read — Prepare a sequential read of skb data

Synopsis

```
void skb_prepare_seq_read (skb,
                           from,
                           to,
                           st);

struct sk_buff *        skb;
unsigned int            from;
unsigned int            to;
struct skb_seq_state *  st;
```

Arguments

skb

the buffer to read

from

lower offset of data to be read

to

upper offset of data to be read

st

state variable

Description

Initializes the specified state variable. Must be called before invoking `skb_seq_read` for the first time.

Name

skb_seq_read — Sequentially read skb data

Synopsis

```
unsigned int skb_seq_read (consumed,
                           data,
                           st);

unsigned int            consumed;
const u8 **             data;
struct skb_seq_state *  st;
```

Arguments

consumed

number of bytes consumed by the caller so far

data

destination pointer for data to be returned

st

state variable

Description

Reads a block of skb data at consumed relative to the lower offset specified to `skb_prepare_seq_read`. Assigns the head of the data block to data and returns the length of the block or 0 if the end of the skb data or the upper offset has been reached.

The caller is not required to consume all of the data returned, i.e. consumed is typically set to the number of bytes already consumed and the next call to `skb_seq_read` will return the remaining part of the block.

Note 1

The size of each block of data returned can be arbitary, this limitation is the cost for zerocopy seqeuental reads of potentially non linear data.

Note 2

Fragment lists within fragments are not implemented at the moment, state->root_skb could be replaced with a stack for this purpose.

Name

skb_abort_seq_read — Abort a sequential read of skb data

Synopsis

```
void skb_abort_seq_read (st);

struct skb_seq_state *  st;
```

Arguments

st

state variable

Description

Must be called if `skb_seq_read` was not called until it returned 0.

Name

skb_find_text — Find a text pattern in skb data

Synopsis

```
unsigned int skb_find_text (skb,
                            from,
                            to,
                            config,
                            state);

struct sk_buff *    skb;
unsigned int        from;
unsigned int        to;
struct ts_config *  config;
struct ts_state *   state;
```

Arguments

skb

the buffer to look in

from

search offset

to

search limit

config

textsearch configuration

state

uninitialized textsearch state variable

Description

Finds a pattern in the skb data according to the specified textsearch configuration. Use `textsearch_next` to retrieve subsequent occurrences of the pattern. Returns the offset to the first occurrence or UINT_MAX if no match was found.

Name

skb_append_datato_frags — append the user data to a skb

Synopsis

```
int skb_append_datato_frags (sk,
                             skb,
                             getfrag,
                             from,
                             length);

struct sock *  sk;
struct sk_buff
*              skb;
int (*         getfrag(void *from, char *to, int offset, int len, int odd, struct
               sk_buff *skb);
void *         from;
int            length;
```

Arguments

sk

sock structure

skb

skb structure to be appened with user data.

getfrag

call back function to be used for getting the user data

from

pointer to user message iov

length

length of the iov message

Description

This procedure append the user data in the fragment part of the skb if any page alloc fails user this procedure returns -ENOMEM

Name

skb_pull_rcsum — pull skb and update receive checksum

Synopsis

```
unsigned char * skb_pull_rcsum (skb,
                                len);

struct sk_buff *  skb;
unsigned int      len;
```

Arguments

skb

buffer to update

len

length of data pulled

Description

This function performs an skb_pull on the packet and updates the CHECKSUM_COMPLETE checksum. It should be used on receive path processing instead of skb_pull unless you know that the checksum difference is zero (e.g., a valid IP header) or you are setting ip_summed to CHECKSUM_NONE.

Name

skb_segment — Perform protocol segmentation on skb.

Synopsis

```
struct sk_buff * skb_segment (skb,
                              features);

struct sk_buff *  skb;
int               features;
```

Arguments

skb

buffer to segment

features

features for the output path (see dev->features)

Description

This function performs segmentation on the given skb. It returns a pointer to the first in a list of new skbs for the segments. In case of error it returns ERR_PTR(err).

Name

skb_cow_data — Check that a socket buffer's data buffers are writable

Synopsis

```
int skb_cow_data (skb,
                  tailbits,
                  trailer);

struct sk_buff *   skb;
int                tailbits;
struct sk_buff **  trailer;
```

Arguments

skb

The socket buffer to check.

tailbits

Amount of trailing space to be added

trailer

Returned pointer to the skb where the *tailbits* space begins

Description

Make sure that the data buffers attached to a socket buffer are writable. If they are not, private copies are made of the data buffers and the socket buffer is set to use these instead.

If *tailbits* is given, make sure that there is space to write *tailbits* bytes of data beyond current end of socket buffer. *trailer* will be set to point to the skb in which this space begins.

The number of scatterlist elements required to completely map the COW'd and extended socket buffer will be returned.

Name

skb_partial_csum_set — set up and verify partial csum values for packet

Synopsis

```
bool skb_partial_csum_set (skb,
                           start,
                           off);

struct sk_buff *  skb;
u16               start;
u16               off;
```

Arguments

skb

the skb to set

start

the number of bytes after skb->data to start checksumming.

off

the offset from start to place the checksum.

Description

For untrusted partially-checksummed packets, we need to make sure the values for skb->csum_start and skb->csum_offset are valid so we don't oops.

This function checks and sets those values and skb->ip_summed: if this returns false you should drop the packet.

Name

sk_alloc — All socket objects are allocated here

Synopsis

```
struct sock * sk_alloc (net,
                        family,
                        priority,
                        prot);

struct net *    net;
int             family;
gfp_t           priority;
struct proto *  prot;
```

Arguments

net

the applicable net namespace

family

protocol family

priority

for allocation (GFP_KERNEL, GFP_ATOMIC, etc)

prot

struct proto associated with this new sock instance

Name

sk_wait_data — wait for data to arrive at sk_receive_queue

Synopsis

```
int sk_wait_data (sk,
                  timeo);

struct sock *  sk;
long *         timeo;
```

Arguments

sk

sock to wait on

timeo

for how long

Description

Now socket state including sk->sk_err is changed only under lock, hence we may omit checks after joining wait queue. We check receive queue before `schedule` only as optimization; it is very likely that `release_sock` added new data.

Name

__sk_mem_schedule — increase sk_forward_alloc and memory_allocated

Synopsis

```
int __sk_mem_schedule (sk,
                       size,
                       kind);

struct sock *  sk;
int            size;
int            kind;
```

Arguments

sk

socket

size

memory size to allocate

kind

allocation type

Description

If kind is SK_MEM_SEND, it means wmem allocation. Otherwise it means rmem allocation. This function assumes that protocols which have memory_pressure use sk_wmem_queued as write buffer accounting.

Name

__sk_mem_reclaim — reclaim memory_allocated

Synopsis

```
void __sk_mem_reclaim (sk);

struct sock *  sk;
```

Arguments

sk

socket

Name

__skb_recv_datagram — Receive a datagram skbuff

Synopsis

```
struct sk_buff * __skb_recv_datagram (sk,
                                      flags,
                                      peeked,
                                      err);

struct sock *  sk;
unsigned       flags;
int *          peeked;
int *          err;
```

Arguments

sk

socket

flags

MSG_ flags

peeked

returns non-zero if this packet has been seen before

err

error code returned

Description

Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible races. This replaces identical code in packet, raw and udp, as well as the IPX AX.25 and Appletalk. It also finally fixes the long standing peek and read race for datagram sockets. If you alter this routine remember it must be re-entrant.

This function will lock the socket if a skb is returned, so the caller needs to unlock the socket in that case (usually by calling skb_free_datagram)

* It does not lock socket since today. This function is * free of race conditions. This measure should/can improve * significantly datagram socket latencies at high loads, * when data copying to user space takes lots of time. * (BTW I've just killed the last `cli` in IP/IPv6/core/netlink/packet * 8) Great win.) * --ANK (980729)

The order of the tests when we find no data waiting are specified quite explicitly by POSIX 1003.1g, don't change them without having the standard around please.

Name

skb_kill_datagram — Free a datagram skbuff forcibly

Synopsis

```
int skb_kill_datagram (sk,
                       skb,
                       flags);

struct sock *     sk;
struct sk_buff *  skb;
unsigned int      flags;
```

Arguments

sk

socket

skb

datagram skbuff

flags

MSG_ flags

Description

This function frees a datagram skbuff that was received by skb_recv_datagram. The flags argument must match the one used for skb_recv_datagram.

If the MSG_PEEK flag is set, and the packet is still on the receive queue of the socket, it will be taken off the queue before it is freed.

This function currently only disables BH when acquiring the sk_receive_queue lock. Therefore it must not be used in a context where that lock is acquired in an IRQ context.

It returns 0 if the packet was removed by us.

Name

skb_copy_datagram_iovec — Copy a datagram to an iovec.

Synopsis

```
int skb_copy_datagram_iovec (skb,
                             offset,
                             to,
                             len);

const struct sk_buff *  skb;
int                     offset;
struct iovec *          to;
int                     len;
```

Arguments

skb

buffer to copy

offset

offset in the buffer to start copying from

to

io vector to copy to

len

amount of data to copy from buffer to iovec

Note

the iovec is modified during the copy.

Name

skb_copy_datagram_const_iovec — Copy a datagram to an iovec.

Synopsis

```
int skb_copy_datagram_const_iovec (skb,
                                   offset,
                                   to,
                                   to_offset,
                                   len);

const struct sk_buff *  skb;
int                     offset;
const struct iovec *    to;
int                     to_offset;
int                     len;
```

Arguments

skb

buffer to copy

offset

offset in the buffer to start copying from

to

io vector to copy to

to_offset

offset in the io vector to start copying to

len

amount of data to copy from buffer to iovec

Description

Returns 0 or -EFAULT.

Note

the iovec is not modified during the copy.

Name

skb_copy_datagram_from_iovec — Copy a datagram from an iovec.

Synopsis

```
int skb_copy_datagram_from_iovec (skb,
                                  offset,
                                  from,
                                  from_offset,
                                  len);

struct sk_buff *      skb;
int                   offset;
const struct iovec *  from;
int                   from_offset;
int                   len;
```

Arguments

skb

buffer to copy

offset

offset in the buffer to start copying to

from

io vector to copy to

from_offset

offset in the io vector to start copying from

len

amount of data to copy to buffer from iovec

Description

Returns 0 or -EFAULT.

Note

the iovec is not modified during the copy.

Name

skb_copy_and_csum_datagram_iovec — Copy and checkum skb to user iovec.

Synopsis

```
int skb_copy_and_csum_datagram_iovec (skb,
                                      hlen,
                                      iov);

struct sk_buff *  skb;
int               hlen;
struct iovec *    iov;
```

Arguments

skb

skbuff

hlen

hardware length

iov

io vector

Description

Caller _must_ check that skb will fit to this iovec.

Returns

0 - success. -EINVAL - checksum failure. -EFAULT - fault during copy. Beware, in this case iovec can be modified!

Name

datagram_poll — generic datagram poll

Synopsis

```
unsigned int datagram_poll (file,
                            sock,
                            wait);

struct file *    file;
struct socket *  sock;
poll_table *     wait;
```

Arguments

file

file struct

sock

socket

wait

poll table

Datagram poll

Again totally generic. This also handles sequenced packet sockets providing the socket receive queue is only ever holding data ready to receive.

Note

when you _don't_ use this routine for this protocol, and you use a different write policy from `sock_writeable` then please supply your own write_space callback.

Name

sk_stream_write_space — stream socket write_space callback.

Synopsis

```
void sk_stream_write_space (sk);

struct sock *  sk;
```

Arguments

sk

socket

FIXME

write proper description

Name

sk_stream_wait_connect — Wait for a socket to get into the connected state

Synopsis

```
int sk_stream_wait_connect (sk,
                            timeo_p);

struct sock *  sk;
long *         timeo_p;
```

Arguments

sk

sock to wait on

timeo_p

for how long to wait

Description

Must be called with the socket locked.

Name

sk_stream_wait_memory — Wait for more memory for a socket

Synopsis

```
int sk_stream_wait_memory (sk,
                           timeo_p);

struct sock *  sk;
long *         timeo_p;
```

Arguments

sk

socket to wait for memory

timeo_p

for how long

Socket Filter

Name

sk_filter — run a packet through a socket filter

Synopsis

```
int sk_filter (sk,
               skb);

struct sock *     sk;
struct sk_buff *  skb;
```

Arguments

sk

sock associated with sk_buff

skb

buffer to filter

Description

Run the filter code and then cut skb->data to correct size returned by sk_run_filter. If pkt_len is 0 we toss packet. If skb->len is smaller than pkt_len we keep whole skb->data. This is the socket level wrapper to sk_run_filter. It returns 0 if the packet should be accepted or -EPERM if the packet should be tossed.

Name

sk_run_filter — run a filter on a socket

Synopsis

```
unsigned int sk_run_filter (skb,
```

filter,
flen);

```
struct sk_buff *      skb;
struct sock_filter *  filter;
int                   flen;
```

Arguments

skb

buffer to run the filter on

filter

filter to apply

flen

length of filter

Description

Decode and apply filter instructions to the skb->data. Return length to keep, 0 for none. skb is the data we are filtering, filter is the array of filter instructions, and len is the number of filter blocks in the array.

Name

sk_chk_filter — verify socket filter code

Synopsis

int **sk_chk_filter** (*filter*,
flen);

```
struct sock_filter *  filter;
int                   flen;
```

Arguments

filter

filter to verify

flen

length of filter

Description

Check the user's filter code. If we let some ugly filter code slip through kaboom! The filter must contain no references or jumps that are out of range, no illegal instructions, and must end with a RET instruction.

All jumps are forward as they are not signed.

Returns 0 if the rule set is legal or -EINVAL if not.

Generic Network Statistics

Name

struct gnet_stats_basic — byte/packet throughput statistics

Synopsis

```
struct gnet_stats_basic {
  __u64 bytes;
  __u32 packets;
};
```

Members

bytes

number of seen bytes

packets

number of seen packets

Name

struct gnet_stats_rate_est — rate estimator

Synopsis

```
struct gnet_stats_rate_est {
  __u32 bps;
  __u32 pps;
};
```

Members

bps

current byte rate

pps

current packet rate

Name

struct gnet_stats_queue — queuing statistics

Synopsis

```
struct gnet_stats_queue {
  __u32 qlen;
  __u32 backlog;
  __u32 drops;
  __u32 requeues;
  __u32 overlimits;
};
```

Members

qlen

queue length

backlog

backlog size of queue

drops

number of dropped packets

requeues

number of requeues

overlimits

number of enqueues over the limit

Name

struct gnet_estimator — rate estimator configuration

Synopsis

```
struct gnet_estimator {
  signed char interval;
  unsigned char ewma_log;
};
```

Members

interval

sampling period

ewma_log

the log of measurement window weight

Name

gnet_stats_start_copy_compat — start dumping procedure in compatibility mode

Synopsis

```
int gnet_stats_start_copy_compat (skb,
                                  type,
                                  tc_stats_type,
                                  xstats_type,
                                  lock,
                                  d);

struct sk_buff *    skb;
int                 type;
int                 tc_stats_type;
int                 xstats_type;
spinlock_t *        lock;
struct gnet_dump *  d;
```

Arguments

skb

socket buffer to put statistics TLVs into

type

TLV type for top level statistic TLV

tc_stats_type

TLV type for backward compatibility struct tc_stats TLV

xstats_type

TLV type for backward compatibility xstats TLV

lock

statistics lock

d

dumping handle

Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVS.

The dumping handle is marked to be in backward compatibility mode telling all `gnet_stats_copy_XXX` functions to fill a local copy of struct tc_stats.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

Name

gnet_stats_start_copy — start dumping procedure in compatibility mode

Synopsis

```
int gnet_stats_start_copy (skb,
                           type,
                           lock,
                           d);

struct sk_buff *    skb;
int                 type;
spinlock_t *        lock;
struct gnet_dump *  d;
```

Arguments

skb

socket buffer to put statistics TLVs into

type

TLV type for top level statistic TLV

lock

statistics lock

d

dumping handle

Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVS.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

Name

gnet_stats_copy_basic — copy basic statistics into statistic TLV

Synopsis

```
int gnet_stats_copy_basic (d,
                           b);

struct gnet_dump *                d;
struct gnet_stats_basic_packed *  b;
```

Arguments

d

dumping handle

b

basic statistics

Description

Appends the basic statistics to the top level TLV created by `gnet_stats_start_copy`.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

Name

gnet_stats_copy_rate_est — copy rate estimator statistics into statistics TLV

Synopsis

```
int gnet_stats_copy_rate_est (d,
                              r);

struct gnet_dump *           d;
struct gnet_stats_rate_est *  r;
```

Arguments

d

dumping handle

r

rate estimator statistics

Description

Appends the rate estimator statistics to the top level TLV created by `gnet_stats_start_copy`.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

Name

gnet_stats_copy_queue — copy queue statistics into statistics TLV

Synopsis

```
int gnet_stats_copy_queue (d,
                           q);

struct gnet_dump *         d;
struct gnet_stats_queue *  q;
```

Arguments

d

dumping handle

q

queue statistics

Description

Appends the queue statistics to the top level TLV created by `gnet_stats_start_copy`.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

Name

gnet_stats_copy_app — copy application specific statistics into statistics TLV

Synopsis

```
int gnet_stats_copy_app (d,
                         st,
                         len);

struct gnet_dump *  d;
void *              st;
int                 len;
```

Arguments

d

dumping handle

st

application specific statistics data

len

length of data

Description

Appends the application sepecific statistics to the top level TLV created by `gnet_stats_start_copy` and remembers the data for XSTATS if the dumping handle is in backward compatibility mode.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

Name

gnet_stats_finish_copy — finish dumping procedure

Synopsis

```
int gnet_stats_finish_copy (d);

struct gnet_dump *  d;
```

Arguments

d

dumping handle

Description

Corrects the length of the top level TLV to include all TLVs added by `gnet_stats_copy_XXX` calls. Adds the backward compatibility TLVs if `gnet_stats_start_copy_compat` was used and releases the statistics lock.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

Name

gen_new_estimator — create a new rate estimator

Synopsis

```
int gen_new_estimator (bstats,
                       rate_est,
                       stats_lock,
                       opt);

struct gnet_stats_basic_packed *  bstats;
struct gnet_stats_rate_est *      rate_est;
spinlock_t *                      stats_lock;
struct nlattr *                   opt;
```

Arguments

bstats

basic statistics

rate_est

rate estimator statistics

stats_lock

statistics lock

opt

rate estimator configuration TLV

Description

Creates a new rate estimator with bstats as source and rate_est as destination. A new timer with the interval specified in the configuration TLV is created. Upon each interval, the latest statistics will be read from bstats and the estimated rate will be stored in rate_est with the statistics lock grabed during this period.

Returns 0 on success or a negative error code.

NOTE

Called under rtnl_mutex

Name

gen_kill_estimator — remove a rate estimator

Synopsis

```
void gen_kill_estimator (bstats,
                         rate_est);

struct gnet_stats_basic_packed *  bstats;
struct gnet_stats_rate_est *      rate_est;
```

Arguments

bstats

basic statistics

rate_est

rate estimator statistics

Description

Removes the rate estimator specified by bstats and rate_est.

NOTE

Called under rtnl_mutex

Name

gen_replace_estimator — replace rate estimator configuration

Synopsis

```
int gen_replace_estimator (bstats,
                           rate_est,
                           stats_lock,
                           opt);

struct gnet_stats_basic_packed *  bstats;
struct gnet_stats_rate_est *      rate_est;
spinlock_t *                      stats_lock;
struct nlattr *                   opt;
```

Arguments

bstats

basic statistics

rate_est

rate estimator statistics

stats_lock

statistics lock

opt

rate estimator configuration TLV

Description

Replaces the configuration of a rate estimator by calling `gen_kill_estimator` and `gen_new_estimator`.

Returns 0 on success or a negative error code.

Name

gen_estimator_active — test if estimator is currently in use

Synopsis

```
bool gen_estimator_active (bstats,
                           rate_est);

const struct gnet_stats_basic_packed *  bstats;
const struct gnet_stats_rate_est *      rate_est;
```

Arguments

bstats

basic statistics

rate_est

rate estimator statistics

Description

Returns true if estimator is active, and false if not.

SUN RPC subsystem

Name

xdr_encode_opaque_fixed — Encode fixed length opaque data

Synopsis

```
__be32 * xdr_encode_opaque_fixed (p,
                                  ptr,
                                  nbytes);

__be32 *      p;
const void *  ptr;
unsigned int  nbytes;
```

Arguments

p

pointer to current position in XDR buffer.

ptr

pointer to data to encode (or NULL)

nbytes

size of data.

Description

Copy the array of data of length nbytes at ptr to the XDR buffer at position p, then align to the next 32-bit boundary by padding with zero bytes (see RFC1832).

Note

if ptr is NULL, only the padding is performed.

Returns the updated current XDR buffer position

Name

xdr_encode_opaque — Encode variable length opaque data

Synopsis

```
__be32 * xdr_encode_opaque (p,
                            ptr,
                            nbytes);

__be32 *      p;
const void *  ptr;
unsigned int  nbytes;
```

Arguments

p

pointer to current position in XDR buffer.

ptr

pointer to data to encode (or NULL)

nbytes

size of data.

Description

Returns the updated current XDR buffer position

Name

xdr_init_encode — Initialize a struct xdr_stream for sending data.

Synopsis

```
void xdr_init_encode (xdr,
                      buf,
                      p);

struct xdr_stream *  xdr;
struct xdr_buf *     buf;
__be32 *             p;
```

Arguments

xdr

pointer to xdr_stream struct

buf

pointer to XDR buffer in which to encode data

p

current pointer inside XDR buffer

Note

at the moment the RPC client only passes the length of our scratch buffer in the xdr_buf's header kvec. Previously this meant we needed to call `xdr_adjust_iovec` after encoding the data. With the new scheme, the xdr_stream manages the details of the buffer length, and takes care of adjusting the kvec length for us.

Name

xdr_reserve_space — Reserve buffer space for sending

Synopsis

```
__be32 * xdr_reserve_space (xdr,
                            nbytes);
```

```
struct xdr_stream *  xdr;
size_t               nbytes;
```

Arguments

xdr

pointer to xdr_stream

nbytes

number of bytes to reserve

Description

Checks that we have enough buffer space to encode 'nbytes' more bytes of data. If so, update the total xdr_buf length, and adjust the length of the current kvec.

Name

xdr_write_pages — Insert a list of pages into an XDR buffer for sending

Synopsis

```
void xdr_write_pages (xdr,
                      pages,
                      base,
                      len);

struct xdr_stream *  xdr;
struct page **       pages;
unsigned int         base;
unsigned int         len;
```

Arguments

xdr

pointer to xdr_stream

pages

list of pages

base

offset of first byte

len

length of data in bytes

Name

xdr_init_decode — Initialize an xdr_stream for decoding data.

Synopsis

```
void xdr_init_decode (xdr,
                      buf,
                      p);

struct xdr_stream *  xdr;
struct xdr_buf *     buf;
__be32 *             p;
```

Arguments

xdr

pointer to xdr_stream struct

buf

pointer to XDR buffer from which to decode data

p

current pointer inside XDR buffer

Name

xdr_inline_decode — Retrieve non-page XDR data to decode

Synopsis

```
__be32 * xdr_inline_decode (xdr,
                            nbytes);

struct xdr_stream *  xdr;
size_t               nbytes;
```

Arguments

xdr

pointer to xdr_stream struct

nbytes

number of bytes of data to decode

Description

Check if the input buffer is long enough to enable us to decode 'nbytes' more bytes of data starting at the current position. If so return the current pointer, then update the current pointer position.

Name

xdr_read_pages — Ensure page-based XDR data to decode is aligned at current pointer position

Synopsis

```
void xdr_read_pages (xdr,
                     len);

struct xdr_stream *  xdr;
unsigned int         len;
```

Arguments

xdr

pointer to xdr_stream struct

len

number of bytes of page data

Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + “len” bytes is moved into the XDR tail[].

Name

xdr_enter_page — decode data from the XDR page

Synopsis

```
void xdr_enter_page (xdr,
                     len);

struct xdr_stream *  xdr;
unsigned int         len;
```

Arguments

xdr

pointer to xdr_stream struct

len

number of bytes of page data

Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + “len” bytes is moved into the XDR tail[]. The current pointer is then repositioned at the beginning of the first XDR page.

Name

svc_print_addr — Format rq_addr field for printing

Synopsis

```
char * svc_print_addr (rqstp,
                       buf,
                       len);

struct svc_rqst *  rqstp;
char *             buf;
size_t             len;
```

Arguments

rqstp

svc_rqst struct containing address to print

buf

target buffer for formatted address

len

length of target buffer

Name

svc_reserve — change the space reserved for the reply to a request.

Synopsis

```
void svc_reserve (rqstp,
                  space);

struct svc_rqst *  rqstp;
int                space;
```

Arguments

rqstp

The request in question

space

new max space to reserve

Description

Each request reserves some space on the output queue of the transport to make sure the reply fits. This function reduces that reserved space to be the amount of space used already, plus *space*.

Name

svc_find_xprt — find an RPC transport instance

Synopsis

```
struct svc_xprt * svc_find_xprt (serv,
                                 xcl_name,
                                 af,
                                 port);

struct svc_serv *     serv;
const char *          xcl_name;
const sa_family_t     af;
const unsigned short  port;
```

Arguments

serv

pointer to svc_serv to search

xcl_name

C string containing transport's class name

af

Address family of transport's local address

port

transport's IP port number

Description

Return the transport instance pointer for the endpoint accepting connections/peer traffic from the specified transport class, address family and port.

Specifying 0 for the address family or port is effectively a wild-card, and will result in matching the first transport in the service's list that has a matching class name.

Name

svc_xprt_names — format a buffer with a list of transport names

Synopsis

```
int svc_xprt_names (serv,
                    buf,
                    buflen);

struct svc_serv *  serv;
char *             buf;
const int          buflen;
```

Arguments

serv

pointer to an RPC service

buf

pointer to a buffer to be filled in

buflen

length of buffer to be filled in

Description

Fills in *buf* with a string containing a list of transport names, each name terminated with '\n'.

Returns positive length of the filled-in string on success; otherwise a negative errno value is returned if an error occurs.

Name

xprt_register_transport — register a transport implementation

Synopsis

```
int xprt_register_transport (transport);

struct xprt_class *  transport;
```

Arguments

transport

transport to register

Description

If a transport implementation is loaded as a kernel module, it can call this interface to make itself known to the RPC client.

0

transport successfully registered -EEXIST: transport already registered -EINVAL: transport module being unloaded

---

Name

xprt_unregister_transport — unregister a transport implementation

Synopsis

```
int xprt_unregister_transport (transport);

struct xprt_class *  transport;
```

Arguments

transport

transport to unregister

0

transport successfully unregistered -ENOENT: transport never registered

---

Name

xprt_load_transport — load a transport implementation

Synopsis

```
int xprt_load_transport (transport_name);

const char *  transport_name;
```

Arguments

transport_name

transport to load

0

transport successfully loaded -ENOENT: transport module not available

---

Name

xprt_reserve_xprt — serialize write access to transports

Synopsis

```
int xprt_reserve_xprt (task);

struct rpc_task *  task;
```

Arguments

task

task that is requesting access to the transport

Description

This prevents mixing the payload of separate requests, and prevents transport connects from colliding with writes. No congestion control is provided.

Name

xprt_release_xprt — allow other requests to use a transport

Synopsis

```
void xprt_release_xprt (xprt,
                        task);

struct rpc_xprt *  xprt;
struct rpc_task *  task;
```

Arguments

xprt

transport with other tasks potentially waiting

task

task that is releasing access to the transport

Description

Note that “task” can be NULL. No congestion control is provided.

Name

xprt_release_xprt_cong — allow other requests to use a transport

Synopsis

```
void xprt_release_xprt_cong (xprt,
```

task);

```
struct rpc_xprt *  xprt;
struct rpc_task *  task;
```

Arguments

xprt

transport with other tasks potentially waiting

task

task that is releasing access to the transport

Description

Note that “task” can be NULL. Another task is awoken to use the transport if the transport's congestion window allows it.

Name

xprt_release_rqst_cong — housekeeping when request is complete

Synopsis

void **xprt_release_rqst_cong** (*task*);

```
struct rpc_task *  task;
```

Arguments

task

RPC request that recently completed

Description

Useful for transports that require congestion control.

Name

xprt_adjust_cwnd — adjust transport congestion window

Synopsis

```
void xprt_adjust_cwnd (task,
                       result);

struct rpc_task *  task;
int                result;
```

Arguments

task

recently completed RPC request used to adjust window

result

result code of completed RPC request

Description

We use a time-smoothed congestion estimator to avoid heavy oscillation.

Name

xprt_wake_pending_tasks — wake all tasks on a transport's pending queue

Synopsis

```
void xprt_wake_pending_tasks (xprt,
                              status);

struct rpc_xprt *  xprt;
int                status;
```

Arguments

xprt

transport with waiting tasks

status

result code to plant in each task before waking it

Name

xprt_wait_for_buffer_space — wait for transport output buffer to clear

Synopsis

```
void xprt_wait_for_buffer_space (task,
                                 action);

struct rpc_task *  task;
rpc_action         action;
```

Arguments

task

task to be put to sleep

action

function pointer to be executed after wait

Name

xprt_write_space — wake the task waiting for transport output buffer space

Synopsis

```
void xprt_write_space (xprt);

struct rpc_xprt *  xprt;
```

Arguments

xprt

transport with waiting tasks

Description

Can be called in a soft IRQ context, so xprt_write_space never sleeps.

Name

xprt_set_retrans_timeout_def — set a request's retransmit timeout

Synopsis

```
void xprt_set_retrans_timeout_def (task);

struct rpc_task *  task;
```

Arguments

task

task whose timeout is to be set

Description

Set a request's retransmit timeout based on the transport's default timeout parameters. Used by transports that don't adjust the retransmit timeout based on round-trip time estimation.

Name

xprt_disconnect_done — mark a transport as disconnected

Synopsis

```
void xprt_disconnect_done (xprt);

struct rpc_xprt *  xprt;
```

Arguments

xprt

transport to flag for disconnect

Name

xprt_lookup_rqst — find an RPC request corresponding to an XID

Synopsis

```
struct rpc_rqst * xprt_lookup_rqst (xprt,
                                    xid);

struct rpc_xprt *  xprt;
__be32             xid;
```

Arguments

xprt

transport on which the original request was transmitted

xid

RPC XID of incoming reply

Name

xprt_update_rtt — update an RPC client's RTT state after receiving a reply

Synopsis

```
void xprt_update_rtt (task);

struct rpc_task *  task;
```

Arguments

task

RPC request that recently completed

Name

xprt_complete_rqst — called when reply processing is complete

Synopsis

```
void xprt_complete_rqst (task,
                         copied);

struct rpc_task *  task;
int                copied;
```

Arguments

task

RPC request that recently completed

copied

actual number of bytes received from the transport

Description

Caller holds transport lock.

Name

rpc_wake_up — wake up all rpc_tasks

Synopsis

```
void rpc_wake_up (queue);

struct rpc_wait_queue *  queue;
```

Arguments

queue

rpc_wait_queue on which the tasks are sleeping

Description

Grabs queue->lock

Name

rpc_wake_up_status — wake up all rpc_tasks and set their status value.

Synopsis

```
void rpc_wake_up_status (queue,
                         status);

struct rpc_wait_queue *  queue;
int                      status;
```

Arguments

queue

rpc_wait_queue on which the tasks are sleeping

status

status value to set

Description

Grabs queue->lock

Name

rpc_malloc — allocate an RPC buffer

Synopsis

```
void * rpc_malloc (task,
                   size);

struct rpc_task *  task;
size_t             size;
```

Arguments

task

RPC task that will use this buffer

size

requested byte size

Description

To prevent rpciod from hanging, this allocator never sleeps, returning NULL if the request cannot be serviced immediately. The caller can arrange to sleep in a way that is safe for rpciod.

Most requests are 'small' (under 2KiB) and can be serviced from a mempool, ensuring that NFS reads and writes can always proceed, and that there is good locality of reference for these buffers.

In order to avoid memory starvation triggering more writebacks of NFS requests, we avoid using GFP_KERNEL.

Name

rpc_free — free buffer allocated via rpc_malloc

Synopsis

```
void rpc_free (buffer);

void *  buffer;
```

Arguments

buffer

buffer to free

Name

xdr_skb_read_bits — copy some data bits from skb to internal buffer

Synopsis

```
size_t xdr_skb_read_bits (desc,
                          to,
                          len);

struct xdr_skb_reader *  desc;
void *                   to;
size_t                   len;
```

Arguments

desc

sk_buff copy helper

to

copy destination

len

number of bytes to copy

Description

Possibly called several times to iterate over an sk_buff and copy data out of it.

Name

xdr_partial_copy_from_skb — copy data out of an skb

Synopsis

```
ssize_t xdr_partial_copy_from_skb (xdr,
                                   base,
                                   desc,
                                   copy_actor);

struct xdr_buf *         xdr;
unsigned int             base;
struct xdr_skb_reader *  desc;
xdr_skb_read_actor       copy_actor;
```

Arguments

xdr

target XDR buffer

base

starting offset

desc

sk_buff copy helper

copy_actor

virtual method for copying data

Name

csum_partial_copy_to_xdr — checksum and copy data

Synopsis

```
int csum_partial_copy_to_xdr (xdr,
                              skb);

struct xdr_buf *  xdr;
struct sk_buff *  skb;
```

Arguments

xdr

target XDR buffer

skb

source skb

Description

We have set things up such that we perform the checksum of the UDP packet in parallel with the copies into the RPC client iovec. -DaveM

Name

rpc_alloc_iostats — allocate an rpc_iostats structure

Synopsis

```
struct rpc_iostats * rpc_alloc_iostats (clnt);

struct rpc_clnt *  clnt;
```

Arguments

clnt

RPC program, version, and xprt

Name

rpc_free_iostats — release an rpc_iostats structure

Synopsis

```
void rpc_free_iostats (stats);

struct rpc_iostats *  stats;
```

Arguments

stats

doomed rpc_iostats structure

Name

rpc_queue_upcall —

Synopsis

```
int rpc_queue_upcall (inode,
                      msg);

struct inode *         inode;
struct rpc_pipe_msg *  msg;
```

Arguments

inode

inode of upcall pipe on which to queue given message

msg

message to queue

Description

Call with an *inode* created by `rpc_mkpipe` to queue an upcall. A userspace process may then later read the upcall by performing a read on an open file for this inode. It is up to the caller to initialize the fields of *msg* (other than *msg*->list) appropriately.

Name

rpc_mkpipe — make an rpc_pipefs file for kernel<->userspace communication

Synopsis

```
struct dentry * rpc_mkpipe (parent,
                            name,
                            private,
                            ops,
                            flags);

struct dentry *              parent;
const char *                 name;
void *                       private;
const struct rpc_pipe_ops *  ops;
int                          flags;
```

Arguments

parent

dentry of directory to create new “pipe” in

name

name of pipe

private

private data to associate with the pipe, for the caller's use

ops

operations defining the behavior of the pipe: upcall, downcall, release_pipe, open_pipe, and destroy_msg.

flags

rpc_inode flags

Description

Data is made available for userspace to read by calls to `rpc_queue_upcall`. The actual reads will result in calls to *ops*->upcall, which will be called with the file pointer, message, and userspace buffer to copy to.

Writes can come at any time, and do not necessarily have to be responses to upcalls. They will result in calls to *msg*->downcall.

The *private* argument passed here will be available to all these methods from the file pointer, via RPC_I(file->f_dentry->d_inode)->private.

Name

rpc_unlink — remove a pipe

Synopsis

```
int rpc_unlink (dentry);

struct dentry *  dentry;
```

Arguments

dentry

dentry for the pipe, as returned from rpc_mkpipe

Description

After this call, lookups will no longer find the pipe, and any attempts to read or write using preexisting opens of the pipe will return -EPIPE.

Name

rpcb_getport_sync — obtain the port for an RPC service on a given host

Synopsis

```
int rpcb_getport_sync (sin,
                       prog,
                       vers,
                       prot);

struct sockaddr_in *  sin;
u32                   prog;
u32                   vers;
int                   prot;
```

Arguments

sin

address of remote peer

prog

RPC program number to bind

vers

RPC version number to bind

prot

transport protocol to use to make this request

Description

Return value is the requested advertised port number, or a negative errno value.

Called from outside the RPC client in a synchronous task context. Uses default timeout parameters specified by underlying transport.

XXX

Needs to support IPv6

Name

rpcb_getport_async — obtain the port for a given RPC service on a given host

Synopsis

```
void rpcb_getport_async (task);

struct rpc_task *  task;
```

Arguments

task

task that is waiting for portmapper request

Description

This one can be called for an ongoing RPC request, and can be used in an async (rpciod) context.

Name

rpc_bind_new_program — bind a new RPC program to an existing client

Synopsis

```
struct rpc_clnt * rpc_bind_new_program (old,
                                        program,
                                        vers);

struct rpc_clnt *     old;
struct rpc_program *  program;
u32                   vers;
```

Arguments

old

old rpc_client

program

rpc program to set

vers

rpc program version

Description

Clones the rpc client and sets up a new RPC program. This is mainly of use for enabling different RPC programs to share the same transport. The Sun NFSv2/v3 ACL protocol can do this.

Name

rpc_run_task — Allocate a new RPC task, then run rpc_execute against it

Synopsis

```
struct rpc_task * rpc_run_task (task_setup_data);

const struct rpc_task_setup *  task_setup_data;
```

Arguments

task_setup_data

pointer to task initialisation data

Name

rpc_call_sync — Perform a synchronous RPC call

Synopsis

```
int rpc_call_sync (clnt,
                   msg,
                   flags);

struct rpc_clnt *           clnt;
const struct rpc_message *  msg;
int                         flags;
```

Arguments

clnt

pointer to RPC client

msg

RPC call parameters

flags

RPC call flags

Name

rpc_call_async — Perform an asynchronous RPC call

Synopsis

```
int rpc_call_async (clnt,
                    msg,
                    flags,
                    tk_ops,
```