UDP *recvfrom*

As was the case with sending, the socket API provides several mechanisms for receiving a UDP datagram. We begin with a study of *recvfrom()* which, like *sendto(),* does not require the user to pass an address structure and does not support scatter/gather operations via the *iovec* mechanism. The *recvfrom()* function takes the following arguments.

fd	File (socket) descriptor.
ubuf	Pointer to a buffer to hold the received message.
size	The size of the above buffer.
addr	Pointer to a structure of type *struct sockaddr_in*. If not NULL and the socket is not connected, the address of the sender of the received message is returned here..
addr_len	A pointer to the size of the structure pointed to by *addr* . If non−zero, its value is changed to the actual size of the address.
flags	Can be used to modify the behaviour of the receive operation. Flags supported for UDP sockets include:

MSG_PEEK: Used to receive data, without dequeuing it. Thus, a subsequent call shall return the same data.

MSG_ERRQUEUE: Used to receive queued errors from socket error queue.

sys_recvfrom(), defined in net/socket.c, is the kernel function to which control is eventually passed from *sys_socketcall().* This function is defined in net/socket.c. Its parameters are those passed by the application to *recvfrom().* If an incoming packet is queued for the socket and successfuly copied to the user buffer, *sys_recvfrom* returns its length in bytes. A return value of less than or equal to zero is an indication that an error condition has been encountered and that no data has been returned.

```
asmlinkage long sys_recvfrom(int fd, void * ubuf,
                   size_t size, unsigned flags,
                   struct sockaddr *addr, int *addr_len)
{
    struct socket *sock;
    struct iovec iov;
    struct msghdr msg;
    char address[MAX_SOCK_ADDR];
    int err,err2;
```

The operation of the sockfd_lookup() function was described in the discussion of UDP sendto. It returns a pointer to struct socket structure corresponding to the fd passed in by the user. If the fd does not index a valid socket NULL is returned and the call fails.

```
    sock = sockfd_lookup(fd, &err);
    if (!sock)
        goto out;
```

Message structures

These structures, introduced in the previous section, are used in both send and receive operations. As with sendto(), the recvfrom() API does not support scatter−gather. Thus it is the responsibility of sys_recvfrom to construct the msghdr and iov.

struct msghdr is defined in include/linux/socket.h.

```
struct msghdr {
    void            *msg_name;
    int              msg_namelen;
    struct iovec    *msg_iov;
    __kernel_size_t msg_iovlen;
    void            *msg_control;
    __kernel_size_t msg_controllen;
    unsigned         msg_flags;
};
```

Functions of structure elements:

msg_name	A pointer to the appropriate struct sockaddr. For TCP/IP sockets this will always be struct sockaddr_in.
msg_namelen	The length of the name structure passed in. For TCP/IP sockets this should be sizeof(truct sockaddr_in).
msg_iov	A pointer to the IO vector.
msg_iovlen	The number of elements in the IO vector which is the number of disjoint fragments of memory comprising the message.
msg_control	A pointer to struct cmsghdr. We don't presently under stand the use of cmsgs.
msg_controllen	The size of the associated cmsg data.
msg_flags	These flags were documented on the first page of this section.

The struct iovec is defined in include/linux/uio.h.

```
struct iovec
{
        void *iov_base;
        kernel_size_t iov_len;
};
```

Functions of structure:

iov_base:	User space pointer to the input data buffer.
iov_len:	Size of buffer pointed to by iov_base. (in bytes)

After recovering the pointer to the struct socket, sys_recvfrom() fills in the struct msghdr and the struct iovec. Since the API supports no mechanism for the receipt of ancillary control data, such data is not collected.

```
        msg.msg_control=NULL;
        msg.msg_controllen=0;
```

Since the recvfrom() API also permits the caller to pass only a single continous input buffer, a simple I/O vector containing one data block is used.

```
        msg.msg_iovlen=1;
        msg.msg_iov=&iov;
```

The base pointer for the data block is set to point to the user space address of the data buffer.

```
        iov.iov_len=size;
        iov.iov_base=ubuf;
```

The name pointer contains the kernel space address of the local buffer in which the address of the sender of the message will be stored temporarily.

```
        msg.msg_name=address;
        msg.msg_namelen=MAX_SOCK_ADDR;
```

When a socket has O_NONBLOCK flag set, the application will not block(wait) if there is currently no data to receive.

```
        if (sock->file->f_flags & O_NONBLOCK)
             flags |= MSG_DONTWAIT;
```

The bulk of the work is done by sock_recvmsg(). If the value returned in err is positive, a packet has been successfully received.

```
    err = sock_recvmsg(sock, &msg, size, flags);
```

On return to sys_recvmsg() a positive value for err indicates success. If the argument addr, which contains the user space address, is not NULL, and the message was successfully received, the sender's address is returned to user space by the move_addr_to_user() function.

```
    if(err >= 0 && addr != NULL && msg.msg_namelen)
    {
        err2 = move_addr_to_user(address,
                    msg.msg_namelen, addr, addr_len);
    if(err2 < 0)
        err=err2;
    }
    sockfd_put(sock);
out:
    return err;
}
```

The sock_recvmsg() function, defined in net/socket.c, is the point at which kernel support for all of the recv*() APIs converges.

```
int sock_recvmsg(struct socket *sock, struct
    msghdr *msg, int size, int flags)
{
    struct scm_cookie scm;

    memset(&scm, 0, sizeof(scm));
```

For UDP and other sockets of family AF_INET, this indirect call is to inet_recvmsg(). As with sendto() the scm cookie is not used.

```
    size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
```

If there was no error, scm_receive() is called to processes any control message data. Recall that control messages are used in the passing of sockets among unrelated processes.

```
    if (size >= 0)
        scm_recv(sock, msg, &scm, flags);

    return size;
}
```

The function inet_recvmsg() is defined in net/ipv4/af_inet.c.

```
int inet_recvmsg(struct socket *sock, struct msghdr
    *msg, int size, int flags, struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;
    int addr_len = 0;
    int err;
```

An indirect call is made to udp_recvmsg. Note that argument the "scm " is not used by inet_recvmsg regardless of the transport protocol that is in use and that the flags are repartitioned into the no block flag and the flags originally passed in by the user.

```
    err = sk->prot->recvmsg(sk, msg, size,
        flags&MSG_DONTWAIT, flags&~MSG_DONTWAIT, &addr_len);
```

If no error occured then the address length which was filled in by udp_recvmsg() is copied back to the msg->msg_namelen field.

```
    if (err >= 0)
         msg->msg_namelen = addr_len;
    return err;
}
```

The udp_recvmsg() function is defined in net/ipv4/udp.c.

```
int udp_recvmsg(struct sock *sk, struct msghdr *msg,
             int len, int noblock, int flags, int *addr_len)
{
```

The variable sin is declared to be a pointer of struct sockaddr_in type and set to the value of msg->msg_name.

```
    struct sockaddr_in *sin = (struct sockaddr_in *)
                                  msg->msg_name;
    struct sk_buff *skb;
    int copied, err;
```

Set length of address to size of struct sockaddr_in. No wonder they don't use very many comments in this code.

```
/*
 *      Check any passed addresses
 */
    if (addr_len)
         *addr_len=sizeof(*sin);
```

If MSG_ERRQUEUE is specified in flags, data is received from the error queue of the socket.
What are the operations in this case --- we need to check out ip_recv_error.

```
    if (flags & MSG_ERRQUEUE)
        return ip_recv_error(sk, msg, len);
```

A single UDP packet is retreived from the receive queue associated with the struct sock by skb_recv_datagram().

```
    skb = skb_recv_datagram(sk, flags, noblock, &err);
```

The skb_recv_datagram() function is defined in net/core/datagram.c.

```
struct sk_buff *skb_recv_datagram(struct sock *sk,
    unsigned flags, int noblock, int *err)
{
    int error;
    struct sk_buff *skb;
    long timeo;
```

This function sock_error returns value of err flag in sk (struct sock) and clears it.
Here we need to understand how and when the err flag might come to be set..

```
/*  Caller is allowed not to check sk->err before
    skb_recv_datagram() */
    error = sock_error(sk);
    if (error)
        goto no_packet;
```

The function sock_error simply returns the negative of the last value to have been stored in sk->err.

```
static inline int sock_error(struct sock *sk)
{
    int err=xchg(&sk->err,0);
    return -err;
}
```

Continuing in skb_recv_datagram, the value returned by sock_rcvtimeo determines the time to wait (in ticks) for data, if the received packet queue is presently empty.

```
      timeo = sock_rcvtimeo(sk, noblock);
```

sock_rcvtimeo is defined in include/net/sock.h. sk->rcvtimeo is set to default value of MAX_SCHEDULE_TIMEOUT by sys_socket. It is the maximum value of an unsigned long type. The units are specified in 10 msec jiffies, but this is effectively a wait forever.

```
static inline long sock_rcvtimeo(struct sock *sk,
                                            int  noblock)
{
      return noblock ? 0 : sk->rcvtimeo;
}
```

On return to skb_recv_datagram() the main receive loop is entered. Exit from the loop will occur when:

a datagram has been successfully received
a timeout occurs (either instantly or after a very long wait)
an error occurs
a signal is received

```
      do {
      /*   Again only user level code calls this
           function, so nothing interrupt level
           will suddenly eat the receive_queue.

           Look at current nfs client by the way...
           However, this function was corrent in
           any case. 8)
       */
```

If MSG_PEEK is specified in flags, skb_peek() is called. It is passed a pointer to the receive queue header. If the receive queue is non–empty it returns a pointer to the first sk buff without dequeuing it from receive queue.

```
    if (flags & MSG_PEEK)
    {
         unsigned long cpu_flags;

         spin_lock_irqsave(
              &sk->receive_queue.lock,cpu_flags);
         skb = skb_peek(&sk->receive_queue);
```

```
struct sk_buff_head {
/* These two members must be first. */
    struct sk_buff   * next;
    struct sk_buff   * prev;

    __u32           qlen;
    spinlock_t      lock;
};
```

1

The skb_peek() function is defined in include/linux/skbuff.h. Note that sk_buff_head and sk_buff pointers are used interchangably in line 402. This (bad) practice works correctly because the first two elements of the sk_buff_head structure are the same as those of the sk_buff. If the next pointer points back to the header, the list is empty and NULL is returned.

```
400 static inline struct sk_buff *skb_peek(struct
                                 sk_buff_head *list_)
401 {
402     struct sk_buff *list =
                 ((struct sk_buff *)list_)->next;
403     if (list == (struct sk_buff *)list_)
404          list = NULL;
405     return list;
406 }
```

Note that the user count of the buffer is incremented here. This presumably occurs because when the buffer is eventually returned to the peeker, the count will be decremented, and, since the buffer still resides on the queue, we don't want it deleted! However, this is speculation, not fact, at the moment.

```
         if(skb!=NULL)
              atomic_inc(&skb->users);
         spin_unlock_irqrestore(
                      &sk->receive_queue.lock,cpu_flags);
```

If the MSG_PEEK flag is not specified, skb_dequeue is called. If the queue is non−empty, skb_dequeue will remove the first sk_buff from the list and return a pointer to it. Otherwise it will return NULL. Queue management is not via standard Linux list structures and the received packet queue is rooted at sk−>receive_queue which is an element of type sk_buff_head.

```
    } else
        skb = skb_dequeue(&sk->receive_queue);
```

The skb_dequeue() function is defined in include/linux/skbuff.h. It calls __skb_dequeue after obtaining the list's associated lock.

```
static inline struct sk_buff *skb_dequeue(struct
                             sk_buff_head *list)
{
    long flags;
    struct sk_buff *result;

    spin_lock_irqsave(&list->lock, flags);
    result = __skb_dequeue(list);
    spin_unlock_irqrestore(&list->lock, flags);
    return result;
}
```

The __skb_dequeue() function does the work of actually removing an sk_buff from the receive queue. Since the sk_buff_head structure starts with the same link pointers as an actual sk_buff structure, it can masquerade as a list element as is done via the cast in line 564. In line 564 prev is set to point to the sk_buff_head. Then in line 565, the local variable next receives the value of the next pointer in the sk_buff_head. The test in line 567 checks to see if the next pointer still points to the sk_buff_head. If so the list was empty. If not the first element is removed from the list and its link fields are zeroed.

```
560 static inline struct sk_buff *__skb_dequeue(struct
                                sk_buff_head *list)
561 {
562     struct sk_buff *next, *prev, *result;
563
564     prev = (struct sk_buff *) list;
565     next = prev->next;
566     result = NULL;
567     if (next != prev) {
568          result = next;
569          next = next->next;
570          list->qlen--;
571          next->prev = prev;
572          prev->next = next;
573          result->next = NULL;
574          result->prev = NULL;
575          result->list = NULL;
576     }
577     return result;
578 }
```

On return to skb_recv_datagram(), if a pointer to an sk_buff is received, it is returned to udp_recvmsg().

```
        if (skb)
            return skb;
```

If the time to wait is zero, return NULL.

```
    /* User doesn’t want to wait */
        error = -EAGAIN;
        if (!timeo)
            goto no_packet;
```

Otherwise, wait for data arrival.

```
    } while (wait_for_packet(sk, err, &timeo) ==  0);

    return NULL;

no_packet:
    *err = error;
    return NULL;
}
```

The wait_for_packet() function is defined in net/core/datagram.c. For reasons not fully understood at the moment it eschews the use of the kernel service routines designed to provide sleep/wakeup services and implements them internally.

```
static int wait_for_packet(struct sock * sk, int *err,
                                        long *timeo_p)
{
    int error;

    DECLARE_WAITQUEUE(wait, current);
```

```
#define DECLARE_WAITQUEUE(name, tsk)                \
   wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

The current process sets its state to TASK_INTERRUPTIBLE and adds itself to the queue of waiting processes. A significant amount of additional processing occurs before the process actually goes to sleep though. sk->sleep is of type wait_queue_head_t.

```
    __set_current_state(TASK_INTERRUPTIBLE);
    add_wait_queue_exclusive(sk->sleep, &wait);
```

The err flag of the struct sock is checked for any errors.

```
    /* Socket errors? */
    error = sock_error(sk);
    if (error)
        goto out_err;
```

The receive queue is tested for still empty. If not a jump is taken to the end of the function.

```
    if (!skb_queue_empty(&sk->receive_queue))
        goto ready;
```

See if the shutdown flag of the sk (struct sock) has been set indicating that some manner of receive close is in progress.

```
    /* Socket shut down? */
    if (sk->shutdown & RCV_SHUTDOWN)
        goto out_noerr;
```

Since, a SOCK_DGRAM type socket is connectionless, we always get past this if–statement. Recall that this function is __skb_dequeue() which can be used by protocols other than UDP.

```
/*  Sequenced packets can come disconnected.
    If so we report the problem */
    error = -ENOTCONN;
    if (connection_based(sk) &&
              !(sk->state==TCP_ESTABLISHED
              ||sk->state==TCP_LISTEN))
        goto out_err;
```

The connection_based() function is defined in net/core/datagram.c.

```
static inline int connection_based(struct sock *sk)
{
    return (sk->type==SOCK_SEQPACKET ||
              sk->type==SOCK_STREAM);
}
```

The signal_pending() function checks the sigpending flag of struct task_struct. If this flag is set, an error is returned.

```
    /* handle signals */
    if (signal_pending(current))
        goto interrupted;
```

signal_pending is defined in include/linux/sched.h.

```
static inline int signal_pending(struct task_struct *p)
{
    return (p->sigpending != 0);
}
```

Finally schedule_timeout() is called to give up control to the scheduler. Control will not return here until a packet is received, a timeout occurs, or a signal is received.

```
    *timeo_p = schedule_timeout(*timeo_p);
```

This is the standard wakeup action. Restore the task state and remove if from the wait queue.

```
ready:
    current->state = TASK_RUNNING;
    remove_wait_queue(sk->sleep, &wait);
    return 0;

interrupted:
    error = sock_intr_errno(*timeo_p);
```

sock_intr_errno is defined in include/net/sock.h.

```
/*  Alas, with timeout socket operations are
    not restartable. Compare this to poll().
*/
static inline int sock_intr_errno(long timeo)
{
    return timeo == MAX_SCHEDULE_TIMEOUT ?
                          -ERESTARTSYS : -EINTR;
}
```

```
out_err:
    *err = error;
out:
    current->state = TASK_RUNNING;
    remove_wait_queue(sk->sleep, &wait);
    return error;
out_noerr:
    *err = 0;
    error = 1;
    goto out;
}
```

schedule_timeout is defined in kernel/sched.c.

```
/*
     schedule_timeout - sleep until timeout
     @timeout: timeout value in jiffies

Make the current task sleep until @timeout jiffies
have elapsed. The routine will return immediately
unless the current task state has been set (see
set_current_state()).

You can set the task state as follows -

%TASK_UNINTERRUPTIBLE - at least @timeout jiffies
are guaranteed to pass before the routine returns.
The routine will return 0

%TASK_INTERRUPTIBLE - the routine may return early
if a signal is delivered to the current task. In this
 case the remaining time in jiffies will be  returned, or 0
if the timer expired in time

The current task state is guaranteed to be
TASK_RUNNING when this routine returns.

Specifying a @timeout value of
MAX_SCHEDULE_TIMEOUT will schedule the CPU away
without a bound on the timeout. In this case the
return value will be %MAX_SCHEDULE_TIMEOUT.

In all cases the return value is guaranteed to be
non-negative.
*/

signed long schedule_timeout(signed long timeout)
{
    struct timer_list timer;
    unsigned long expire;

```

The above comment clearly says that if timeout value is MAX_SCHEDULE_TIMEOUT, the current process waits for data indefinitely. schedule is called to schedule any processes contending for CPU.

```
switch (timeout)
{
case MAX_SCHEDULE_TIMEOUT:
     schedule();
     goto out;
default:
 /*
Another bit of PARANOID. Note that the retval
will be 0 since no piece of kernel is
supposed to do a check for a negative retval
of schedule_timeout() (since it should never
happens anyway). You just have the printk()
that will tell you if something is gone wrong
and where.
 */
     if (timeout < 0)
     {
          printk(KERN_ERR "schedule_timeout: wrong
               timeout "
                    "value %lx from %p\n",timeout,
                     builtin_return_address(0));
          current->state = TASK_RUNNING;
          goto out;
     }
}
```

If a timeout value less than default max value was specified, a timer is initialized and added to list of timers maintained by kernel and schedule() is invoked.

```
expire = timeout + jiffies;

init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;

add_timer(&timer);
schedule();
del_timer_sync(&timer);
```

process_timeout() is defined in kernel/sched.c. When the above timer expires, this function wakes up current process.

```
static void process_timeout(unsigned long __data)
{
        struct task_struct * p = (struct task_struct *)
                                   __data;

        wake_up_process(p);
}
```

When an event wakes the current process, update time remaining to wait and return to skb_recv_datagram to check if an sk_buff was received. If so, a pointer to it is returned to udp_recvmsg. Otherwise, the process either goes to sleep again or returns a NULL value to udp_recvmsg based on the time remaining to wait.

```
        timeout = expire - jiffies;

out:
        return timeout < 0 ? 0 : timeout;
}
```

Back in udp_recvmsg a test is made to see if an sk_buff pointer was returned by skb_recv_datagram() and if not a jump to an error exit pointe is made.

```
        if (!skb)
             goto out;
```

If a buffer pointer was returned it is necessary to copy the data back to user space and release the buffer . Since skb->len includes the length of the UDP header at this point (but no longer the MAC or IP header) , copied denotes length of user data.

```
        copied = skb->len - sizeof(struct udphdr);
```

If data to be returned to user exceeds size of buffer provided, adjust the length downward to fit and set an appropriate flag to indicate that the message was truncated.

```
        if (copied > len) {
             copied = len;
             msg->msg_flags |= MSG_TRUNC;
        }
```

As with send there are multiple copy mechanisms that depend upon the need to validate the UDP checksum.

```
    if (skb->ip_summed==CHECKSUM_UNNECESSARY) {
```

If check summing is not required, the skb_copy_datagram_iovec() copies the data to the user space buffer described by the iov.

```
        err = skb_copy_datagram_iovec(skb,
                  sizeof(struct udphdr),
                  msg->msg_iov, copied);
    } else if (msg->msg_flags & MSG_TRUNC) {
```

If it was necessary to truncate the message and a checksum is required. It is first necessary to call __udp_checksum_complete() to verify UDP checksum over the entire packet. In case of a checksum error, the sk_buff is freed and an error returned. In case of success only the part of the packet that will fit into the buffer provided is copied.

```
        if (__udp_checksum_complete(skb))
             goto csum_copy_err;

        err = skb_copy_datagram_iovec(skb,
        sizeof(struct udphdr),msg->msg_iov,
                  copied);
    } else {
```

In the final case a checksum is required and the entire packet is to be copied. Here skb_copy_and_csum_datagram_iovec() verifies checksum and copies data from sk buff to I/O vector.

```
        err = skb_copy_and_csum_datagram_iovec
                  (skb, sizeof(struct udphdr),
        msg->msg_iov);

        if (err == -EINVAL)
             goto csum_copy_err;
    }

    if (err)
        goto out_free;
```

The sock_recv_timestamp() function records the time stamp, when the sk_buff was received.

```
    sock_recv_timestamp(msg, sk, skb);
```

sock_recv_timestamp() is defined in include/net/sock.h.

```
static __inline__ void
sock_recv_timestamp(struct msghdr *msg, struct sock
                        *sk, struct sk_buff *skb)
{
```

rcvtstamp is a flag of struct sock. In the case of recvfrom() there is no provision for control messages and the put_cmsg() will do nothing.

```
    if (sk->rcvtstamp)
        put_cmsg(msg, SOL_SOCKET, SO_TIMESTAMP,
                        sizeof(skb->stamp), &skb->stamp);
    else
        sk->stamp = skb->stamp;
}
```

Assemble the sender address from elements of the sk_buff.

```
    /* Copy the address. */
    if (sin)
    {
        sin->sin_family = AF_INET;
        sin->sin_port = skb->h.uh->source;
        sin->sin_addr.s_addr = skb->nh.iph->saddr;
        memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
    }
```

Depending on control message flags specified, corresponding ancillary control data is collected by ip_cmsg_recv. However, sys_recvfrom does not get any such data.

```
    if (sk->protinfo.af_inet.cmsg_flags)
        ip_cmsg_recv(msg, skb);
    err = copied;

```

Free the sk_buff() and return.

```
out_free:
    skb_free_datagram(sk, skb);
out:
    return err;

```

In case of a check sum error, the sk_buff is also freed.

```
csum_copy_err:
    UDP_INC_STATS_BH(UdpInErrors);

    /* Clear queue. */
    if (flags & MSG_PEEK) {
         int clear = 0;
         spin_lock_irq(&sk->receive_queue.lock);
         if (skb == skb_peek(&sk->receive_queue)) {
              __skb_unlink(skb, &sk->receive_queue);
              clear = 1;
         }
         spin_unlock_irq(&sk->receive_queue.lock);
         if (clear)
              kfree_skb(skb);
    }

    skb_free_datagram(sk, skb);

    return -EAGAIN;
}
```

Transfering data from an sk_buff to an I/O vector

This procedure appears (and is) far more complex in the worst case than is actually the case in practice. The problem lies with the ‘‘unususual’’ implementation of the sk_buff. In the worst case, a single packet could consist of the following:

– an instance of the struct skbuff buffer header.
– a kmalloc’d ‘‘data’’ area allocated to hold the headers of the packet.
– up to 6 unmapped single page structures called fragments holding packet data
– a pointer to an additional sk_buff which may contain all 4 of these elements!

This possibility leads to a recursive implementation of checksumming and data movement code. Fortunately, in practice a UDP packet always consists of only:

– an instance of the struct skbuff buffer header.
– a kmalloc’d ‘‘data’’ area allocated holding both packet headers an data.

The ‘‘worst case’’ sk_buff structure is mapped by the struct skb_shared_info defined in include/linux/skbuff.h. When used, this structure resides at the end of the kmalloc’d data area and is pointed to by the end element of the struct sk_buff header.

```
/* This data is invariant across clones and lives at
 * the end of the header data, ie. at skb->end.
 */
struct skb_shared_info {
    atomic_t        dataref;
    unsigned int    nr_frags;
    struct sk_buff  *frag_list;
    skb_frag_t      frags[MAX_SKB_FRAGS];
};
```

Functions of structure elements:

frag_list	may contain zero a pointer to the next sk_buff in the chain. When is it used??
frags	is a six element array of skb_frag_t type. These are the unmapped single page entities.
nr_frags	denotes number of elements of frags array in use.

```
#define MAX_SKB_FRAGS 6

typedef struct skb_frag_struct skb_frag_t;

struct skb_frag_struct
{
    struct page *page;
    __u16 page_offset;
    __u16 size;
};
```

Functions of structure elements:

page	Pointer to a struct page.
offset	Offset in page from where data is stored.
size	Size of data.

The buffer header of type struct sk_buff contains two members, namely len and data_len, used to describe the length of the received packet. The skb->len field denotes length of the amount of data in the packet that remains to be processed. That is, it is initially set to the length of all headers and application data. As headers are removed as the packet is passed up the stack, the value of skb->len is decremented by the length of each network header removed. The value of skb->data_len is the amount of data that is held in fragments and in chained sk_buffs. It is used by TCP but appears to have no use in processing UDP packets.

The function skb_copy_datagram_iovec(), defined in net/core/datagram.c, is used to copy a UDP datagram when checksumming is not required. In this case the value of offset is the size of the UDP header and skb−>data_len is 0.

```
204 int skb_copy_datagram_iovec(const struct sk_buff *skb,
                 int offset, struct iovec *to, int len)
206 {
207     int i, copy;
```

It is the case that skb−>len includes the kmalloc'd stuff but that skb−>datalen includes only that which is in the appendicies. Thus start would be set here to the amount of data in the kmalloc'd part which would be just what is needed!

```
208     int start = skb->len - skb->data_len;
```

The comment is misleading and apparently reflects the philosophy that the kmalloc'd part of the sk_buff structure is for storage of network header elements. What is actually happening in the case of UDP is that memcpy_toiovec() is being passed a pointer to the start of the user data along with the length of the user data. In the standard case (no fragments) the value of len will become 0 at line 216 and the function will return.

```
210     /* Copy header. */
211     if ((copy = start-offset) > 0) {
212          if (copy > len)
213               copy = len;
214          if (memcpy_toiovec(to, skb->data + offset, copy))
215               goto fault;
216          if ((len -= copy) == 0)
217               return 0;
218          offset += copy;
219     }
```

The memcpy_to_iovec() function is defined in net/core/iovec.c. It copies kernel data into an I/O vector. Note that as data is copied to the iovec, the len field of the element which is the recipient is decremented and the base pointer is incremented. This strategy makes it possible, albeit slightly inefficient, for callers that are passing multiple fragments of a packet to be copied to always just pass the base address of the iovec. Elements that have been previously filled will just be bypassed in the while loop because the if statement at line 88 will find that such elements have iov_len equal to 0.

```
82 int memcpy_toiovec(struct iovec *iov, unsigned char
                                    *kdata, int len)
83 {
84     int err = -EFAULT;
85
86     while(len>0)
87     {
88          if(iov->iov_len)
89          {
```

min_t returns minimum of two arguments.

```
90                    int copy = min_t(unsigned int,
                           iov->iov_len, len);
91                    if (copy_to_user(iov->iov_base, kdata,
                           copy))
92                         goto out;
93                    kdata+=copy;
```

Update available buffer space and base pointer of I/O vector.

```
94                    len-=copy;
95                    iov->iov_len-=copy;
96                    iov->iov_base+=copy;
97          }
98          iov++;
99     }
100    err = 0;
101 out:
102    return err;
103 }
```

If there do exist fragments skb_copy_datagram_iovec() will continue and copy data from page fragments into the I/O vector.

```
/*   Copy paged appendix. Hmm... why does this
     look so complicated? */
for (i=0; i<skb_shinfo(skb)->nr_frags; i++) {
```

skb_shinfo is defined in include/linux/skbuff.h. It simply returns a pointer to skb_shared_info structure that is pointed to by skb->end.

```
/* Internal */
#define skb_shinfo(SKB)
          ((struct skb_shared_info *)((SKB)->end))
```

```
     int end;

     BUG_TRAP(start <= offset+len);

```

In the first iteration of this loop start contains the offset from the start of the packet data (including UDP header) of the beginning of the paged appendix. Thus end is set to the offset of the 1st byte beyond the data in the paged appendix and copy is set to the amount of data in this element of the paged appendix.

```
     end = start + skb_shinfo(skb)->frags[i].size;
     if ((copy = end-offset) > 0) {
          int err;
          u8  *vaddr;
          skb_frag_t *frag =
          &skb_shinfo(skb)->frags[i];
          struct page *page = frag->page;

          if (copy > len)
             copy = len;
```

Get logical address of page corresponding to page. Copy data from fragment into I/O vector using memcpy_to_iovec.

```
          vaddr = kmap(page);
          err = memcpy_toiovec(to, vaddr +
                         frag->page_offset +
                         offset-start, copy);
          kunmap(page);
          if (err)
               goto fault;
          if (!(len -= copy))
               return 0;
          offset += copy;
     }
     start = end;
}
```

Finally, if there exist additional sk_buffs in the chain, the are processed via a recursive call to skb_copy_datagram_iovec().   This incredible structure is actually a tree of general degree.

```
    if (skb_shinfo(skb)->frag_list) {
         struct sk_buff *list;

         for (list = skb_shinfo(skb)->frag_list;
                   list; list=list->next) {
              int end;

              BUG_TRAP(start <= offset+len);

              end = start + list->len;
              if ((copy = end-offset) > 0) {
                   if (copy > len)
                        copy = len;
                   if (skb_copy_datagram_iovec(list,
                             offset-start, to, copy))
                        goto fault;

                   if ((len -= copy) == 0)
                        return 0;
                   offset += copy;
              }
              start = end;
         }
    }
    if (len == 0)
         return 0;

fault:
    return -EFAULT;
}
```

UDP checksum

The function __udp_checksum_complete() is defined in net/ipv4/udp.c and is used when a UDP datagram must be truncated. Its mission is to make sure that the entire datagram passes the checksum test. If so, then it is safe to return the truncated part to user space. It consists of call to skb_checksum() followed by a call to csum_fold() which converts the 32 bit checksum to a proper 16 bit one.

```
static __inline__ int __udp_checksum_complete(struct
                                      sk_buff *skb)
{
    return (unsigned short)csum_fold(skb_checksum(skb,
                   0, skb->len, skb->csum));
}
```

skb_checksum is defined in net/core/skbuff.c. It computes checksum on data present in sk buff using function csum_partial.

```
/* Checksum skb data. */

unsigned int skb_checksum(const struct sk_buff *skb,
         int offset, int len, unsigned int csum)
{
    int i, copy;
    int start = skb->len - skb->data_len;
    int pos = 0;
```

Compute a partial checksum, csum , on UDP header and any data past it. Note that argument offset has the value zero.

```
    /* Checksum header. */
    if ((copy = start-offset) > 0) {
         if (copy > len)
              copy = len;
         csum = csum_partial(skb->data+offset, copy,
         csum);
```

Prototype of function csum_partial is specified in include/asm−i386/checksum.h. The function is defined in arch/i386/checksum.S. It is written in assembly language.

```
computes the checksum of a memory block at buff,  length len,
and adds in "sum" (32−bit) returns a 32−bit number suitable
for  feeding into itself or csum_tcpudp_magic this function
must be called with even lengths, except for the last
fragment, which may be odd it’s best to have buff aligned on
a 32−bit boundary
```

```
asmlinkage unsigned int csum_partial(const unsigned
     char * buff, int len, unsigned int sum);
```

```
          if ((len −= copy) == 0)
               return csum;
          offset += copy;
          pos = copy;
     }
```

On data in each fragment, compute partial checksum, csum2, using csum_partial. Add the two checksums, csum and csum2, using csum_block_add.

```
    for (i=0; i<skb_shinfo(skb)->nr_frags; i++) {
         int end;

         BUG_TRAP(start <= offset+len);

         end = start + skb_shinfo(skb)->frags[i].size;
         if ((copy = end-offset) > 0) {
              unsigned int csum2;
              u8 *vaddr;
              skb_frag_t *frag =
              &skb_shinfo(skb)->frags[i];

              if (copy > len)
                   copy = len;
              vaddr = kmap_skb_frag(frag);

              csum2 = csum_partial(vaddr +
              frag->page_offset +
                        offset-start, copy, 0);
              kunmap_skb_frag(vaddr);
              csum = csum_block_add(csum, csum2, pos);
              if (!(len -= copy))
                   return csum;
              offset += copy;
              pos += copy;
         }
         start = end;
    }
```

csum_block_add is defined as an inline function in include/net/checksum.h. It performs an adjustment to csum2, if csum1 is a partial checksum of an odd number of bytes of data. How does this work?

```
static inline unsigned int
csum_block_add(unsigned int csum, unsigned int csum2,
                                  int offset)
{
    if (offset&1)
         csum2 = ((csum2&0xFF00FF)<<8)
                   +((csum2>>8)&0xFF00FF);
    return csum_add(csum, csum2);
}
```

csum_add is defined as below. It combines two partial checksums.

```
static inline unsigned int csum_add(unsigned int csum,
                             unsigned int addend)
{
    csum += addend;
    return csum + (csum < addend);
}
```

On data present in each sk_ buff on fragment list, compute partial checksum, csum2, using csum_partial. Add the two checksums as done earlier.

```
    if (skb_shinfo(skb)->frag_list) {
         struct sk_buff *list;

         for (list = skb_shinfo(skb)->frag_list; list;
              list=list->next) {
              int end;

              BUG_TRAP(start <= offset+len);

              end = start + list->len;
              if ((copy = end-offset) > 0) {
                   unsigned int csum2;
                   if (copy > len)
                        copy = len;
                   csum2 = skb_checksum(list, offset-
                   start, copy, 0);
                   csum = csum_block_add(csum, csum2,
                        pos);
                        if ((len -= copy) == 0)
                        return csum;
                   offset += copy;
                   pos += copy;
              }
              start = end;
         }
    }
    if (len == 0)
         return csum;

    BUG();
    return csum;
}
```

csum_fold is defined as below. It folds a 32–bit checksum to a 16–bit value. How does this work?

```
/*
 *      Fold a partial checksum
 */

static inline unsigned int csum_fold(unsigned int sum)
{
         asm__("
         addl %1, %0
         adcl $0xffff, %0"
         "
         : "=r" (sum)
         : "r" (sum << 16), "" (sum &  0xffff0000)
         );
         return (~sum) >> 16;
}
```

skb_copy_and_csum_datagram_iovec is defined in net/core/datagram.c. It gets called when checksum is necessary and message has not been truncated.

```
int skb_copy_and_csum_datagram_iovec(const struct
    sk_buff *skb, int hlen, struct iovec *iov)
{
    unsigned int csum;
    int chunk = skb->len - hlen;

    /*   Skip filled elements. Pretty silly, look at
         memcpy_toiovec, though 8) */
    while (iov->iov_len == 0)
         iov++;

    if (iov->iov_len < chunk) {
```

We have covered this case, where the size of specified buffer is less than available data.

```
         if ((unsigned short)csum_fold(skb_checksum
              (skb, 0, chunk+hlen, skb->csum)))
            goto csum_error;
         if (skb_copy_datagram_iovec(skb, hlen, iov,
                   chunk))
            goto fault;
    } else {
```

Obtain checksum value of UDP header. skb_copy_and_csum_datagram gets called which performs both checksumming and copy of data.

```
         csum = csum_partial(skb->data, hlen,
                                   skb->csum);
         if (skb_copy_and_csum_datagram(skb, hlen,
              iov->iov_base, chunk, &csum))
            goto fault;
         if ((unsigned short)csum_fold(csum))
            goto csum_error;
         iov->iov_len -= chunk;
         iov->iov_base += chunk;
    }
    return 0;

csum_error:
    return -EINVAL;

fault:
    return -EFAULT;
}
```

skb_copy_and_csum_datagram is defined in net/core/datagram.c.

```
int skb_copy_and_csum_datagram(const struct sk_buff   *skb,
    int offset, u8 *to, int len, unsigned int *csump)
{
    int i, copy;
    int start = skb->len - skb->data_len;
    int pos = 0;

    /* Copy header. */
    if ((copy = start-offset) > 0) {
         int err = 0;
         if (copy > len)
              copy = len;
         *csump = csum_and_copy_to_user(skb->data
                           offset, to, copy, *csump,&err);
         if (err)
                   goto fault;
         if ((len -= copy) == 0)
              return 0;
         offset += copy;
         to += copy;
         pos = copy;
    }
```

csum_and_copy_to_user is defined in include/asm-i386/checksum.h.

```
/*
 *      Copy and checksum to user
 */
#define HAVE_CSUM_COPY_USER
static __inline__ unsigned int csum_and_copy_to_user
 (const char *src, char *dst, int len, int sum,
    int *err_ptr)
{
    if (access_ok(VERIFY_WRITE, dst, len))
         return csum_partial_copy_generic(src, dst,
         len, sum, NULL, err_ptr);

    if (len)
         *err_ptr = -EFAULT;

    return -1; /* invalid checksum */
}
```

Prototype of csum_partial_copy_generic is defined in include/asm–i386/checksum.h.

```
/*
     the same as csum_partial, but copies from src
while it checksums, and handles user-space pointer
exceptions correctly, when needed.

     here even more important to align src and dst on a
32-bit (or even better 64-bit) boundary
*/

asmlinkage unsigned int csum_partial_copy_generic(
const char *src, char *dst, int len, int sum,
     int *src_err_ptr, int *dst_err_ptr);
```

On return to skb_copy_and_csum_datagram repeat for of each fragment.

```
for (i=0; i<skb_shinfo(skb)->nr_frags; i++) {
     int end;

     BUG_TRAP(start <= offset+len);

     end = start + skb_shinfo(skb)->frags[i].size;
     if ((copy = end-offset) > 0) {
          unsigned int csum2;
          int err = 0;
          u8  *vaddr;
          skb_frag_t *frag =
          &skb_shinfo(skb)->frags[i];
          struct page *page = frag->page;

          if (copy > len)
               copy = len;
          vaddr = kmap(page);
          csum2 = csum_and_copy_to_user(vaddr +
          frag->page_offset +
                    offset-start, to, copy, 0,
          &err);
          kunmap(page);
          if (err)
               goto fault;
          *csump = csum_block_add(*csump, csum2,
          pos);
          if (!(len -= copy))
               return 0;
          offset += copy;
          to += copy;
          pos += copy;
     }
     start = end;
}
```

And then on data present in each sk buff on fragment list.

```
    if (skb_shinfo(skb)->frag_list) {
         struct sk_buff *list;

         for (list = skb_shinfo(skb)->frag_list; list;
              list=list->next) {
              int end;

              BUG_TRAP(start <= offset+len);

              end = start + list->len;
              if ((copy = end-offset) > 0) {
                   unsigned int csum2 = 0;
                   if (copy > len)
                        copy = len;
                   if (skb_copy_and_csum_datagram
                             (list, offset-start, to,
                        copy, &csum2))
                        goto fault;
                   *csump = csum_block_add(*csump,
                   csum2, pos);
                   if ((len -= copy) == 0)
                        return 0;
                   offset += copy;
                   to += copy;
                        pos += copy;
              }
              start = end;
         }
    }
    if (len == 0)
         return 0;

fault:
    return -EFAULT;
}
```

scm_rcv is defined in include/net/scm.h. It is called by sock_recvmsg after the return from the call to inet_recvmsg().

```
45 static __inline__ void scm_recv(struct socket *sock,
       struct msghdr *msg, struct scm_cookie *scm, int
       flags)
47 {
48     if (!msg->msg_control)
49     {
```

Since no control buffer was specified by sys_recvfrom, scm_destroy() is called. As shown below the call is a no−op here because scm−>fpl is NULL. The MSG_CTRUNC flag indicates that control data was discarded.

```
50          if (sock->passcred || scm->fp)
51               msg->msg_flags |= MSG_CTRUNC;
52          scm_destroy(scm);
53          return;
54     }
```

Because of the return on line 53, control can't possibly reach this point when called by recvfrom(). However, other callers might pass a msg_control pointer. In that case, if the passcred flag is set, any socket control data is copied into msg control buffer.

```
56     if (sock->passcred)
57          put_cmsg(msg, SOL_SOCKET, SCM_CREDENTIALS,
                         sizeof(scm->creds), &scm->creds);
59     if (!scm->fp)
60          return;
```

Any passed file descriptors are freed by scm_detach_fds.

```
62     scm_detach_fds(msg, scm);
63 }
```

The scm_destroy function is a wrapper for __scm_destroy.

```
void __scm_destroy(struct scm_cookie *scm)
{
    struct scm_fp_list *fpl = scm->fp;
    int i;

    if (fpl) {
        scm->fp = NULL;
        for (i=fpl->count-1; i>=0; i--)
            fput(fpl->fp[i]);
        kfree(fpl);
    }
}
```