
+ ABSTRACT

This file documents the mmap() facility available with the PACKET
socket interface on 2.4 and 2.6 kernels. This type of sockets is used for
capture network traffic with utilities like tcpdump or any other that needs
raw access to network interface.

You can find the latest version of this document at:
    http://pusa.uv.es/~ulisses/packet_mmap/

Howto can be found at:
    http://wiki.gnu-log.net (packet_mmap)

Please send your comments to
    Ulisses Alonso Camaró <uaca@i.hate.spam.alumni.uv.es>
    Johann Baudy <johann.baudy@gnu-log.net>

+ Why use PACKET_MMAP

In Linux 2.4/2.6 if PACKET_MMAP is not enabled, the capture process is very
inefficient. It uses very limited buffers and requires one system call
to capture each packet, it requires two if you want to get packet's
timestamp (like libpcap always does).

In the other hand PACKET_MMAP is very efficient. PACKET_MMAP provides a size
configurable circular buffer mapped in user space that can be used to either
send or receive packets. This way reading packets just needs to wait for them,
most of the time there is no need to issue a single system call. Concerning
transmission, multiple packets can be sent through one system call to get the
highest bandwidth.
By using a shared buffer between the kernel and the user also has the benefit
of minimizing packet copies.

It's fine to use PACKET_MMAP to improve the performance of the capture and
transmission process, but it isn't everything. At least, if you are capturing
at high speeds (this is relative to the cpu speed), you should check if the
device driver of your network interface card supports some sort of interrupt
load mitigation or (even better) if it supports NAPI, also make sure it is
enabled. For transmission, check the MTU (Maximum Transmission Unit) used and
supported by devices of your network.

+ How to use mmap() to improve capture process

From the user standpoint, you should use the higher level libpcap library, which
is a de facto standard, portable across nearly all operating systems
including Win32.

Said that, at time of this writing, official libpcap 0.8.1 is out and doesn't
include
support for PACKET_MMAP, and also probably the libpcap included in your

distribution.

I'm aware of two implementations of PACKET_MMAP in libpcap:

```
    http://pusa.uv.es/~ulisses/packet_mmap/  (by Simon Patarin, based on libpcap 0.6.2)
    http://public.lanl.gov/cpw/              (by Phil Wood, based on lastest libpcap)
```

The rest of this document is intended for people who want to understand
the low level details or want to improve libpcap by including PACKET_MMAP
support.

+ How to use mmap() directly to improve capture process

From the system calls stand point, the use of PACKET_MMAP involves
the following process:

```
[setup]     socket() -------> creation of the capture socket
            setsockopt() ---> allocation of the circular buffer (ring)
                              option: PACKET_RX_RING
            mmap() ---------> mapping of the allocated buffer to the
                              user process

[capture]   poll() ---------> to wait for incoming packets

[shutdown]  close() --------> destruction of the capture socket and
                              deallocation of all associated
                              resources.
```

socket creation and destruction is straight forward, and is done
the same way with or without PACKET_MMAP:

```
int fd;

fd= socket(PF_PACKET, mode, htons(ETH_P_ALL))
```

where mode is SOCK_RAW for the raw interface were link level
information can be captured or SOCK_DGRAM for the cooked
interface where link level information capture is not
supported and a link level pseudo-header is provided
by the kernel.

The destruction of the socket and all associated resources
is done by a simple call to close(fd).

Next I will describe PACKET_MMAP settings and its constraints,
also the mapping of the circular buffer in the user process and
the use of this buffer.

+ How to use mmap() directly to improve transmission process

Transmission process is similar to capture as shown below.

```
[setup]          socket() -------> creation of the transmission socket
                 setsockopt() ---> allocation of the circular buffer (ring)
                                   option: PACKET_TX_RING
                 bind() ---------> bind transmission socket with a network
interface
                 mmap() ---------> mapping of the allocated buffer to the
                                   user process

[transmission]   poll() ---------> wait for free packets (optional)
                 send() ---------> send all packets that are set as ready in
                                   the ring
                                   The flag MSG_DONTWAIT can be used to return
                                   before end of transfer.

[shutdown]  close() --------> destruction of the transmission socket and
                              deallocation of all associated resources.
```

Binding the socket to your network interface is mandatory (with zero copy) to know the header size of frames used in the circular buffer.

As capture, each frame contains two parts:

```
 --------------------
| struct tpacket_hdr | Header. It contains the status of
|                    | of this frame
|--------------------|
| data buffer        |
.                    .  Data that will be sent over the network interface.
.                    .
 --------------------
```

bind() associates the socket to your network interface thanks to sll_ifindex parameter of struct sockaddr_ll.

Initialization example:

```
struct sockaddr_ll my_addr;
struct ifreq s_ifr;
...

strncpy (s_ifr.ifr_name, "eth0", sizeof(s_ifr.ifr_name));

/* get interface index of eth0 */
ioctl(this->socket, SIOCGIFINDEX, &s_ifr);

/* fill sockaddr_ll struct to prepare binding */
my_addr.sll_family = AF_PACKET;
my_addr.sll_protocol = ETH_P_ALL;
my_addr.sll_ifindex =  s_ifr.ifr_ifindex;

/* bind socket to eth0 */
bind(this->socket, (struct sockaddr *)&my_addr, sizeof(struct sockaddr_ll));
```

A complete tutorial is available at: http://wiki.gnu-log.net/

+ PACKET_MMAP settings

To setup PACKET_MMAP from user level code is done with a call like

```
 - Capture process
     setsockopt(fd, SOL_PACKET, PACKET_RX_RING, (void *) &req, sizeof(req))
 - Transmission process
     setsockopt(fd, SOL_PACKET, PACKET_TX_RING, (void *) &req, sizeof(req))
```

The most significant argument in the previous call is the req parameter, this parameter must to have the following structure:

```
    struct tpacket_req
    {
        unsigned int    tp_block_size;  /* Minimal size of contiguous block */
        unsigned int    tp_block_nr;    /* Number of blocks */
        unsigned int    tp_frame_size;  /* Size of frame */
        unsigned int    tp_frame_nr;    /* Total number of frames */
    };
```

This structure is defined in /usr/include/linux/if_packet.h and establishes a circular buffer (ring) of unswappable memory.
Being mapped in the capture process allows reading the captured frames and related meta-information like timestamps without requiring a system call.

Frames are grouped in blocks. Each block is a physically contiguous region of memory and holds tp_block_size/tp_frame_size frames. The total number of blocks is tp_block_nr. Note that tp_frame_nr is a redundant parameter because

```
    frames_per_block = tp_block_size/tp_frame_size
```

indeed, packet_set_ring checks that the following condition is true

```
    frames_per_block * tp_block_nr == tp_frame_nr
```

Lets see an example, with the following values:

```
     tp_block_size= 4096
     tp_frame_size= 2048
     tp_block_nr  = 4
     tp_frame_nr  = 8
```

we will get the following buffer structure:

```
        block #1                 block #2
+---------+---------+    +---------+---------+
| frame 1 | frame 2 |    | frame 3 | frame 4 |
+---------+---------+    +---------+---------+

        block #3                 block #4
```

```
+---------+---------+    +---------+---------+
| frame 5 | frame 6 |    | frame 7 | frame 8 |
+---------+---------+    +---------+---------+
```

A frame can be of any size with the only condition it can fit in a block. A block
can only hold an integer number of frames, or in other words, a frame cannot be spawned accross two blocks, so there are some details you have to take into account when choosing the frame_size. See "Mapping and use of the circular buffer (ring)".

+ PACKET_MMAP setting constraints

In kernel versions prior to 2.4.26 (for the 2.4 branch) and 2.6.5 (2.6 branch), the PACKET_MMAP buffer could hold only 32768 frames in a 32 bit architecture or 16384 in a 64 bit architecture. For information on these kernel versions see http://pusa.uv.es/~ulisses/packet_mmap/packet_mmap.pre-2.4.26_2.6.5.txt

Block size limit

As stated earlier, each block is a contiguous physical region of memory. These memory regions are allocated with calls to the __get_free_pages() function. As the name indicates, this function allocates pages of memory, and the second argument is "order" or a power of two number of pages, that is
(for PAGE_SIZE == 4096) order=0 ==> 4096 bytes, order=1 ==> 8192 bytes,
order=2 ==> 16384 bytes, etc. The maximum size of a
region allocated by __get_free_pages is determined by the MAX_ORDER macro. More precisely the limit can be calculated as:

```
   PAGE_SIZE << MAX_ORDER

   In a i386 architecture PAGE_SIZE is 4096 bytes
   In a 2.4/i386 kernel MAX_ORDER is 10
   In a 2.6/i386 kernel MAX_ORDER is 11
```

So get_free_pages can allocate as much as 4MB or 8MB in a 2.4/2.6 kernel respectively, with an i386 architecture.

User space programs can include /usr/include/sys/user.h and /usr/include/linux/mmzone.h to get PAGE_SIZE MAX_ORDER declarations.

The pagesize can also be determined dynamically with the getpagesize (2) system call.

Block number limit

To understand the constraints of PACKET_MMAP, we have to see the structure used to hold the pointers to each block.

Currently, this structure is a dynamically allocated vector with kmalloc called pg_vec, its size limits the number of blocks that can be allocated.

```
    +---+---+---+---+
    | x | x | x | x |
    +---+---+---+---+
      |   |   |   |
      |   |   |   v
      |   |   v  block #4
      |   v  block #3
      v  block #2
     block #1
```

kmalloc allocates any number of bytes of physically contiguous memory from a pool of pre-determined sizes. This pool of memory is maintained by the slab allocator which is at the end the responsible for doing the allocation and hence which imposes the maximum memory that kmalloc can allocate.

In a 2.4/2.6 kernel and the i386 architecture, the limit is 131072 bytes. The predetermined sizes that kmalloc uses can be checked in the "size-<bytes>" entries of /proc/slabinfo

In a 32 bit architecture, pointers are 4 bytes long, so the total number of pointers to blocks is

```
     131072/4 = 32768 blocks
```

PACKET_MMAP buffer size calculator

Definitions:

```
<size-max>    : is the maximum size of allocable with kmalloc (see
/proc/slabinfo)
<pointer size>: depends on the architecture -- sizeof(void *)
<page size>   : depends on the architecture -- PAGE_SIZE or getpagesize (2)
<max-order>   : is the value defined with MAX_ORDER
<frame size>  : it's an upper bound of frame's capture size (more on this later)
```

from these definitions we will derive

```
        <block number> = <size-max>/<pointer size>
        <block size> = <pagesize> << <max-order>
```

so, the max buffer size is

```
        <block number> * <block size>
```

and, the number of frames be

```
        <block number> * <block size> / <frame size>
```

Suppose the following parameters, which apply for 2.6 kernel and an i386 architecture:

```
        <size-max> = 131072 bytes
        <pointer size> = 4 bytes
        <pagesize> = 4096 bytes
        <max-order> = 11
```

and a value for <frame size> of 2048 bytes. These parameters will yield

```
        <block number> = 131072/4 = 32768 blocks
        <block size> = 4096 << 11 = 8 MiB.
```

and hence the buffer will have a 262144 MiB size. So it can hold
262144 MiB / 2048 bytes = 134217728 frames

Actually, this buffer size is not possible with an i386 architecture.
Remember that the memory is allocated in kernel space, in the case of
an i386 kernel's memory size is limited to 1GiB.

All memory allocations are not freed until the socket is closed. The memory
allocations are done with GFP_KERNEL priority, this basically means that
the allocation can wait and swap other process' memory in order to allocate
the necessary memory, so normally limits can be reached.

Other constraints

If you check the source code you will see that what I draw here as a frame
is not only the link level frame. At the beginning of each frame there is a
header called struct tpacket_hdr used in PACKET_MMAP to hold link level's frame
meta information like timestamp. So what we draw here a frame it's really
the following (from include/linux/if_packet.h):

```
/*
   Frame structure:

   - Start. Frame must be aligned to TPACKET_ALIGNMENT=16
   - struct tpacket_hdr
   - pad to TPACKET_ALIGNMENT=16
   - struct sockaddr_ll
   - Gap, chosen so that packet data (Start+tp_net) aligns to
     TPACKET_ALIGNMENT=16
   - Start+tp_mac: [ Optional MAC header ]
   - Start+tp_net: Packet data, aligned to TPACKET_ALIGNMENT=16.
   - Pad to align to TPACKET_ALIGNMENT=16
 */
```

The following are conditions that are checked in packet_set_ring

```
   tp_block_size must be a multiple of PAGE_SIZE (1)
   tp_frame_size must be greater than TPACKET_HDRLEN (obvious)
   tp_frame_size must be a multiple of TPACKET_ALIGNMENT
   tp_frame_nr   must be exactly frames_per_block*tp_block_nr
```

Note that tp_block_size should be chosen to be a power of two or there will
be a waste of memory.

```
-------------------------------------------------------------------------------
+ Mapping and use of the circular buffer (ring)
-------------------------------------------------------------------------------

The mapping of the buffer in the user process is done with the conventional
mmap function. Even the circular buffer is compound of several physically
discontiguous blocks of memory, they are contiguous to the user space, hence
just one call to mmap is needed:

    mmap(0, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

If tp_frame_size is a divisor of tp_block_size frames will be
contiguously spaced by tp_frame_size bytes. If not, each
tp_block_size/tp_frame_size frames there will be a gap between
the frames. This is because a frame cannot be spawn across two
blocks.

At the beginning of each frame there is an status field (see
struct tpacket_hdr). If this field is 0 means that the frame is ready
to be used for the kernel, If not, there is a frame the user can read
and the following flags apply:

+++ Capture process:
     from include/linux/if_packet.h

     #define TP_STATUS_COPY          2
     #define TP_STATUS_LOSING        4
     #define TP_STATUS_CSUMNOTREADY  8


TP_STATUS_COPY        : This flag indicates that the frame (and associated
                        meta information) has been truncated because it's
                        larger than tp_frame_size. This packet can be
                        read entirely with recvfrom().

                        In order to make this work it must to be
                        enabled previously with setsockopt() and
                        the PACKET_COPY_THRESH option.

                        The number of frames than can be buffered to
                        be read with recvfrom is limited like a normal socket.
                        See the SO_RCVBUF option in the socket (7) man page.

TP_STATUS_LOSING      : indicates there were packet drops from last time
                        statistics where checked with getsockopt() and
                        the PACKET_STATISTICS option.

TP_STATUS_CSUMNOTREADY: currently it's used for outgoing IP packets which
                        its checksum will be done in hardware. So while
                        reading the packet we should not try to check the
                        checksum.

for convenience there are also the following defines:

     #define TP_STATUS_KERNEL        0
```

```
     #define TP_STATUS_USER          1
```

The kernel initializes all frames to TP_STATUS_KERNEL, when the kernel
receives a packet it puts in the buffer and updates the status with
at least the TP_STATUS_USER flag. Then the user can read the packet,
once the packet is read the user must zero the status field, so the kernel
can use again that frame buffer.

The user can use poll (any other variant should apply too) to check if new
packets are in the ring:

```
    struct pollfd pfd;

    pfd.fd = fd;
    pfd.revents = 0;
    pfd.events = POLLIN|POLLRDNORM|POLLERR;

    if (status == TP_STATUS_KERNEL)
        retval = poll(&pfd, 1, timeout);
```

It doesn't incur in a race condition to first check the status value and
then poll for frames.

++ Transmission process
Those defines are also used for transmission:

```
     #define TP_STATUS_AVAILABLE        0 // Frame is available
     #define TP_STATUS_SEND_REQUEST     1 // Frame will be sent on next send()
     #define TP_STATUS_SENDING          2 // Frame is currently in transmission
     #define TP_STATUS_WRONG_FORMAT     4 // Frame format is not correct
```

First, the kernel initializes all frames to TP_STATUS_AVAILABLE. To send a
packet, the user fills a data buffer of an available frame, sets tp_len to
current data buffer size and sets its status field to TP_STATUS_SEND_REQUEST.
This can be done on multiple frames. Once the user is ready to transmit, it
calls send(). Then all buffers with status equal to TP_STATUS_SEND_REQUEST are
forwarded to the network device. The kernel updates each status of sent
frames with TP_STATUS_SENDING until the end of transfer.
At the end of each transfer, buffer status returns to TP_STATUS_AVAILABLE.

```
    header->tp_len = in_i_size;
    header->tp_status = TP_STATUS_SEND_REQUEST;
    retval = send(this->socket, NULL, 0, 0);
```

The user can also use poll() to check if a buffer is available:
(status == TP_STATUS_SENDING)

```
    struct pollfd pfd;
    pfd.fd = fd;
    pfd.revents = 0;
    pfd.events = POLLOUT;
    retval = poll(&pfd, 1, timeout);
```

+ THANKS

Jesse Brandeburg, for fixing my grammathical/spelling errors