Packet MMAP

Abstract

This file documents the mmap() facility available with the PACKET socket interface. This type of sockets is used for

i. capture network traffic with utilities like tcpdump,
ii. transmit network traffic, or any other that needs raw access to network interface.

Howto can be found at:

https://sites.google.com/site/packetmmap/

Please send your comments to
- Ulisses Alonso CamarÃ³ <uaca@i.hate.spam.alumni.uv.es>
- Johann Baudy

Why use PACKET_MMAP

Non PACKET_MMAP capture process (plain AF_PACKET) 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).

On 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. CPU IRQ pinning of your network interface card can also be an advantage.

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.

Packet MMAP support was integrated into libpcap around the time of version 1.3.0; TPACKET_V3 support was added in version 1.5.0

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 = 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).

Similarly as without PACKET_MMAP, it is possible to use one socket for capture and transmission. This can be done by mapping the allocated RX and TX buffer ring with a single mmap() call. See "Mapping and use of the circular buffer (ring)".

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.
```

Socket creation and destruction is also straight forward, and is done the same way as in capturing described in the previous paragraph:

```
int fd = socket(PF_PACKET, mode, 0);
```

The protocol can optionally be 0 in case we only want to transmit via this socket, which avoids an expensive call to packet_rcv(). In this case, you also need to bind(2) the TX_RING with sll_protocol = 0 set. Otherwise, htons(ETH_P_ALL) or any other protocol, for example.

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;
   ...

   strscpy_pad (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 = htons(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: https://sites.google.com/site/packetmmap/
```

By default, the user should put data at:

```
frame base + TPACKET_HDRLEN - sizeof(struct sockaddr_ll)
```

So, whatever you choose for the socket mode (SOCK_DGRAM or SOCK_RAW), the beginning of the user data will be at:

```
frame base + TPACKET_ALIGN(sizeof(struct tpacket_hdr))
```

If you wish to put user data at a custom offset from the beginning of the frame (for payload alignment with SOCK_RAW mode for instance) you can set tp_net (with SOCK_DGRAM) or tp_mac (with SOCK_RAW). In order to make this work it must be enabled previously with setsockopt() and the PACKET_TX_HAS_OFF option.

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 across 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.

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

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.

To use one socket for capture and transmission, the mapping of both the RX and TX buffer ring has to be done with one call to mmap:

```
...
setsockopt(fd, SOL_PACKET, PACKET_RX_RING, &foo, sizeof(foo));
setsockopt(fd, SOL_PACKET, PACKET_TX_RING, &bar, sizeof(bar));
...
rx_ring = mmap(0, size * 2, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
tx_ring = rx_ring + size;
```

RX must be the first as the kernel maps the TX ring memory right after the RX one.

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          (1 << 1)
#define TP_STATUS_LOSING        (1 << 2)
#define TP_STATUS_CSUMNOTREADY  (1 << 3)
#define TP_STATUS_CSUM_VALID    (1 << 7)
```

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 that 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.

TP_STATUS_CSUM_VALID	This flag indicates that at least the transport header checksum of the packet has been already validated on the kernel side. If the flag is not set then we are free to check the checksum by ourselves provided that TP_STATUS_CSUMNOTREADY is also not set.

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);
```

What TPACKET versions are available and when to use them?

```
int val = tpacket_version;
setsockopt(fd, SOL_PACKET, PACKET_VERSION, &val, sizeof(val));
getsockopt(fd, SOL_PACKET, PACKET_VERSION, &val, sizeof(val));
```

where 'tpacket_version' can be TPACKET_V1 (default), TPACKET_V2, TPACKET_V3.

TPACKET_V1:

- Default if not otherwise specified by setsockopt(2)
- RX_RING, TX_RING available

TPACKET_V1 --> TPACKET_V2:

- Made 64 bit clean due to unsigned long usage in TPACKET_V1 structures, thus this also works on 64 bit kernel with 32 bit userspace and the like
- Timestamp resolution in nanoseconds instead of microseconds
- RX_RING, TX_RING available
- VLAN metadata information available for packets (TP_STATUS_VLAN_VALID,

TP_STATUS_VLAN_TPID_VALID), in the tpacket2_hdr structure:

- TP_STATUS_VLAN_VALID bit being set into the tp_status field indicates that the tp_vlan_tci field has valid VLAN TCI value
- TP_STATUS_VLAN_TPID_VALID bit being set into the tp_status field indicates that the tp_vlan_tpid field has valid VLAN TPID value

- How to switch to TPACKET_V2:

1. Replace struct tpacket_hdr by struct tpacket2_hdr
2. Query header len and save
3. Set protocol version to 2, set up ring as usual
4. For getting the sockaddr_ll, use `(void *)hdr + TPACKET_ALIGN(hdrlen)` instead of `(void *)hdr + TPACKET_ALIGN(sizeof(struct tpacket_hdr))`

TPACKET_V2 --> TPACKET_V3:

- Flexible buffer implementation for RX_RING:

1. Blocks can be configured with non-static frame-size
2. Read/poll is at a block-level (as opposed to packet-level)
3. Added poll timeout to avoid indefinite user-space wait on idle links
4. Added user-configurable knobs:

4.1 block::timeout 4.2 tpkt_hdr::sk_rxhash

- RX Hash data available in user space
- TX_RING semantics are conceptually similar to TPACKET_V2; use tpacket3_hdr instead of tpacket2_hdr, and TPACKET3_HDRLEN instead of TPACKET2_HDRLEN. In the current implementation, the tp_next_offset field in the tpacket3_hdr MUST be set to zero, indicating that the ring does not hold variable sized frames. Packets with non-zero values of tp_next_offset will be dropped.

AF_PACKET fanout mode

In the AF_PACKET fanout mode, packet reception can be load balanced among processes. This also works in combination with mmap(2) on packet sockets.

Currently implemented fanout policies are:

- PACKET_FANOUT_HASH: schedule to socket by skb's packet hash
- PACKET_FANOUT_LB: schedule to socket by round-robin
- PACKET_FANOUT_CPU: schedule to socket by CPU packet arrives on
- PACKET_FANOUT_RND: schedule to socket by random selection
- PACKET_FANOUT_ROLLOVER: if one socket is full, rollover to another
- PACKET_FANOUT_QM: schedule to socket by skbs recorded queue_mapping

Minimal example code by David S. Miller (try things like "./test eth0 hash", "./test eth0 lb", etc.):

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/types.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#include <unistd.h>

#include <linux/if_ether.h>
#include <linux/if_packet.h>

#include <net/if.h>

static const char *device_name;
static int fanout_type;
static int fanout_id;

#ifndef PACKET_FANOUT
# define PACKET_FANOUT                  18
# define PACKET_FANOUT_HASH             0
```

```
# define PACKET_FANOUT_LB           1
#endif

static int setup_socket(void)
{
        int err, fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
        struct sockaddr_ll ll;
        struct ifreq ifr;
        int fanout_arg;

        if (fd < 0) {
                perror("socket");
                return EXIT_FAILURE;
        }

        memset(&ifr, 0, sizeof(ifr));
        strcpy(ifr.ifr_name, device_name);
        err = ioctl(fd, SIOCGIFINDEX, &ifr);
        if (err < 0) {
                perror("SIOCGIFINDEX");
                return EXIT_FAILURE;
        }

        memset(&ll, 0, sizeof(ll));
        ll.sll_family = AF_PACKET;
        ll.sll_ifindex = ifr.ifr_ifindex;
        err = bind(fd, (struct sockaddr *) &ll, sizeof(ll));
        if (err < 0) {
                perror("bind");
                return EXIT_FAILURE;
        }

        fanout_arg = (fanout_id | (fanout_type << 16));
        err = setsockopt(fd, SOL_PACKET, PACKET_FANOUT,
                         &fanout_arg, sizeof(fanout_arg));
        if (err) {
                perror("setsockopt");
                return EXIT_FAILURE;
        }

        return fd;
}

static void fanout_thread(void)
{
        int fd = setup_socket();
        int limit = 10000;

        if (fd < 0)
                exit(fd);

        while (limit-- > 0) {
                char buf[1600];
                int err;

                err = read(fd, buf, sizeof(buf));
                if (err < 0) {
                        perror("read");
                        exit(EXIT_FAILURE);
                }
                if ((limit % 10) == 0)
                        fprintf(stdout, "(%d) \n", getpid());
        }

        fprintf(stdout, "%d: Received 10000 packets\n", getpid());

        close(fd);
        exit(0);
}

int main(int argc, char **argp)
{
        int fd, err;
        int i;

        if (argc != 3) {
                fprintf(stderr, "Usage: %s INTERFACE {hash|lb}\n", argp[0]);
                return EXIT_FAILURE;
        }

        if (!strcmp(argp[2], "hash"))
```

```
                fanout_type = PACKET_FANOUT_HASH;
        else if (!strcmp(argp[2], "lb"))
                fanout_type = PACKET_FANOUT_LB;
        else {
                fprintf(stderr, "Unknown fanout type [%s]\n", argp[2]);
                exit(EXIT_FAILURE);
        }

        device_name = argp[1];
        fanout_id = getpid() & 0xffff;

        for (i = 0; i < 4; i++) {
                pid_t pid = fork();

                switch (pid) {
                case 0:
                        fanout_thread();

                case -1:
                        perror("fork");
                        exit(EXIT_FAILURE);
                }
        }

        for (i = 0; i < 4; i++) {
                int status;

                wait(&status);
        }

        return 0;
}
```

AF_PACKET TPACKET_V3 example

AF_PACKET's TPACKET_V3 ring buffer can be configured to use non-static frame sizes by doing it's own memory management. It is based on blocks where polling works on a per block basis instead of per ring as in TPACKET_V2 and predecessor.

It is said that TPACKET_V3 brings the following benefits:

- ~15% - 20% reduction in CPU-usage
- ~20% increase in packet capture rate
- ~2x increase in packet density
- Port aggregation analysis
- Non static frame size to capture entire packet payload

So it seems to be a good candidate to be used with packet fanout.

Minimal example code by Daniel Borkmann based on Chetan Loke's lolpcap (compile it with gcc -Wall -O2 blob.c, and try things like "./a.out eth0", etc.):

```
/* Written from scratch, but kernel-to-user space API usage
* dissected from lolpcap:
*  Copyright 2011, Chetan Loke <loke.chetan@gmail.com>
*  License: GPL, version 2.0
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <poll.h>
#include <unistd.h>
#include <signal.h>
#include <inttypes.h>
#include <sys/socket.h>
#include <sys/mman.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>
#include <linux/ip.h>

#ifndef likely
# define likely(x)          __builtin_expect(!!(x), 1)
#endif
#ifndef unlikely
```

```
# define unlikely(x)                __builtin_expect(!!(x), 0)
#endif

struct block_desc {
        uint32_t version;
        uint32_t offset_to_priv;
        struct tpacket_hdr_v1 h1;
};

struct ring {
        struct iovec *rd;
        uint8_t *map;
        struct tpacket_req3 req;
};

static unsigned long packets_total = 0, bytes_total = 0;
static sig_atomic_t sigint = 0;

static void sighandler(int num)
{
        sigint = 1;
}

static int setup_socket(struct ring *ring, char *netdev)
{
        int err, i, fd, v = TPACKET_V3;
        struct sockaddr_ll ll;
        unsigned int blocksiz = 1 << 22, framesiz = 1 << 11;
        unsigned int blocknum = 64;

        fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
        if (fd < 0) {
                perror("socket");
                exit(1);
        }

        err = setsockopt(fd, SOL_PACKET, PACKET_VERSION, &v, sizeof(v));
        if (err < 0) {
                perror("setsockopt");
                exit(1);
        }

        memset(&ring->req, 0, sizeof(ring->req));
        ring->req.tp_block_size = blocksiz;
        ring->req.tp_frame_size = framesiz;
        ring->req.tp_block_nr = blocknum;
        ring->req.tp_frame_nr = (blocksiz * blocknum) / framesiz;
        ring->req.tp_retire_blk_tov = 60;
        ring->req.tp_feature_req_word = TP_FT_REQ_FILL_RXHASH;

        err = setsockopt(fd, SOL_PACKET, PACKET_RX_RING, &ring->req,
                         sizeof(ring->req));
        if (err < 0) {
                perror("setsockopt");
                exit(1);
        }

        ring->map = mmap(NULL, ring->req.tp_block_size * ring->req.tp_block_nr,
                         PROT_READ | PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd, 0);
        if (ring->map == MAP_FAILED) {
                perror("mmap");
                exit(1);
        }

        ring->rd = malloc(ring->req.tp_block_nr * sizeof(*ring->rd));
        assert(ring->rd);
        for (i = 0; i < ring->req.tp_block_nr; ++i) {
                ring->rd[i].iov_base = ring->map + (i * ring->req.tp_block_size);
                ring->rd[i].iov_len = ring->req.tp_block_size;
        }

        memset(&ll, 0, sizeof(ll));
        ll.sll_family = PF_PACKET;
        ll.sll_protocol = htons(ETH_P_ALL);
        ll.sll_ifindex = if_nametoindex(netdev);
        ll.sll_hatype = 0;
        ll.sll_pkttype = 0;
        ll.sll_halen = 0;

        err = bind(fd, (struct sockaddr *) &ll, sizeof(ll));
        if (err < 0) {
```

```
                perror("bind");
                exit(1);
        }

        return fd;
}

static void display(struct tpacket3_hdr *ppd)
{
        struct ethhdr *eth = (struct ethhdr *) ((uint8_t *) ppd + ppd->tp_mac);
        struct iphdr *ip = (struct iphdr *) ((uint8_t *) eth + ETH_HLEN);

        if (eth->h_proto == htons(ETH_P_IP)) {
                struct sockaddr_in ss, sd;
                char sbuff[NI_MAXHOST], dbuff[NI_MAXHOST];

                memset(&ss, 0, sizeof(ss));
                ss.sin_family = PF_INET;
                ss.sin_addr.s_addr = ip->saddr;
                getnameinfo((struct sockaddr *) &ss, sizeof(ss),
                            sbuff, sizeof(sbuff), NULL, 0, NI_NUMERICHOST);

                memset(&sd, 0, sizeof(sd));
                sd.sin_family = PF_INET;
                sd.sin_addr.s_addr = ip->daddr;
                getnameinfo((struct sockaddr *) &sd, sizeof(sd),
                            dbuff, sizeof(dbuff), NULL, 0, NI_NUMERICHOST);

                printf("%s -> %s, ", sbuff, dbuff);
        }

        printf("rxhash: 0x%x\n", ppd->hv1.tp_rxhash);
}

static void walk_block(struct block_desc *pbd, const int block_num)
{
        int num_pkts = pbd->h1.num_pkts, i;
        unsigned long bytes = 0;
        struct tpacket3_hdr *ppd;

        ppd = (struct tpacket3_hdr *) ((uint8_t *) pbd +
                                       pbd->h1.offset_to_first_pkt);
        for (i = 0; i < num_pkts; ++i) {
                bytes += ppd->tp_snaplen;
                display(ppd);

                ppd = (struct tpacket3_hdr *) ((uint8_t *) ppd +
                                               ppd->tp_next_offset);
        }

        packets_total += num_pkts;
        bytes_total += bytes;
}

static void flush_block(struct block_desc *pbd)
{
        pbd->h1.block_status = TP_STATUS_KERNEL;
}

static void teardown_socket(struct ring *ring, int fd)
{
        munmap(ring->map, ring->req.tp_block_size * ring->req.tp_block_nr);
        free(ring->rd);
        close(fd);
}

int main(int argc, char **argp)
{
        int fd, err;
        socklen_t len;
        struct ring ring;
        struct pollfd pfd;
        unsigned int block_num = 0, blocks = 64;
        struct block_desc *pbd;
        struct tpacket_stats_v3 stats;

        if (argc != 2) {
                fprintf(stderr, "Usage: %s INTERFACE\n", argp[0]);
                return EXIT_FAILURE;
        }
```

```
        signal(SIGINT, sighandler);

        memset(&ring, 0, sizeof(ring));
        fd = setup_socket(&ring, argp[argc - 1]);
        assert(fd > 0);

        memset(&pfd, 0, sizeof(pfd));
        pfd.fd = fd;
        pfd.events = POLLIN | POLLERR;
        pfd.revents = 0;

        while (likely(!sigint)) {
                pbd = (struct block_desc *) ring.rd[block_num].iov_base;

                if ((pbd->h1.block_status & TP_STATUS_USER) == 0) {
                        poll(&pfd, 1, -1);
                        continue;
                }

                walk_block(pbd, block_num);
                flush_block(pbd);
                block_num = (block_num + 1) % blocks;
        }

        len = sizeof(stats);
        err = getsockopt(fd, SOL_PACKET, PACKET_STATISTICS, &stats, &len);
        if (err < 0) {
                perror("getsockopt");
                exit(1);
        }

        fflush(stdout);
        printf("\nReceived %u packets, %lu bytes, %u dropped, freeze_q_cnt: %u\n",
            stats.tp_packets, bytes_total, stats.tp_drops,
            stats.tp_freeze_q_cnt);

        teardown_socket(&ring, fd);
        return 0;
}
```

PACKET_QDISC_BYPASS

If there is a requirement to load the network with many packets in a similar fashion as pktgen does, you might set the following option after socket creation:

```
int one = 1;
setsockopt(fd, SOL_PACKET, PACKET_QDISC_BYPASS, &one, sizeof(one));
```

This has the side-effect, that packets sent through PF_PACKET will bypass the kernel's qdisc layer and are forcedly pushed to the driver directly. Meaning, packet are not buffered, tc disciplines are ignored, increased loss can occur and such packets are also not visible to other PF_PACKET sockets anymore. So, you have been warned; generally, this can be useful for stress testing various components of a system.

On default, PACKET_QDISC_BYPASS is disabled and needs to be explicitly enabled on PF_PACKET sockets.

PACKET_TIMESTAMP

The PACKET_TIMESTAMP setting determines the source of the timestamp in the packet meta information for mmap(2)ed RX_RING and TX_RINGs. If your NIC is capable of timestamping packets in hardware, you can request those hardware timestamps to be used. Note: you may need to enable the generation of hardware timestamps with SIOCSHWTSTAMP (see related information from Documentation/networking/timestamping.rst).

PACKET_TIMESTAMP accepts the same integer bit field as SO_TIMESTAMPING:

```
int req = SOF_TIMESTAMPING_RAW_HARDWARE;
setsockopt(fd, SOL_PACKET, PACKET_TIMESTAMP, (void *) &req, sizeof(req))
```

For the mmap(2)ed ring buffers, such timestamps are stored in the `tpacket{,2,3}_hdr` structure's tp_sec and `tp_{n,u}sec` members. To determine what kind of timestamp has been reported, the tp_status field is binary or'ed with the following possible bits ...

```
TP_STATUS_TS_RAW_HARDWARE
TP_STATUS_TS_SOFTWARE
```

... that are equivalent to its `SOF_TIMESTAMPING_*` counterparts. For the RX_RING, if neither is set (i.e. PACKET_TIMESTAMP is not set), then a software fallback was invoked *within* PF_PACKET's processing code (less precise).

Getting timestamps for the TX_RING works as follows: i) fill the ring frames, ii) call sendto() e.g. in blocking mode, iii) wait for status

of relevant frames to be updated resp. the frame handed over to the application, iv) walk through the frames to pick up the individual hw/sw timestamps.

Only (!) if transmit timestamping is enabled, then these bits are combined with binary | with TP_STATUS_AVAILABLE, so you must check for that in your application (e.g. !(tp_status & (TP_STATUS_SEND_REQUEST | TP_STATUS_SENDING)) in a first step to see if the frame belongs to the application, and then one can extract the type of timestamp in a second step from tp_status)!

If you don't care about them, thus having it disabled, checking for TP_STATUS_AVAILABLE resp. TP_STATUS_WRONG_FORMAT is sufficient. If in the TX_RING part only TP_STATUS_AVAILABLE is set, then the tp_sec and tp_{n,u}sec members do not contain a valid value. For TX_RINGs, by default no timestamp is generated!

See include/linux/net_tstamp.h and Documentation/networking/timestamping.rst for more information on hardware timestamps.

Miscellaneous bits

- Packet sockets work well together with Linux socket filters, thus you also might want to have a look at Documentation/networking/filter.rst

THANKS

Jesse Brandeburg, for fixing my grammathical/spelling errors