How to XDP (with Snabb)

Max Rottenkolber <max@mr.gy>

Tuesday, 21 January 2020

Table of Contents

Intro

Networking in userspace is becoming mainstream. Recent Linux kernels ship with a feature called *eXpress Data Path* (XDP) that intends to consolidate kernel-bypass packet processing applications with the kernel's networking stack. XDP provides a new kind of network socket that allows userspace applications to do almost direct I/O with network device drivers while maintaining integration with the kernel's native network stack.

To Snabb XDP looks like just another network I/O device. In fact the XDP ABI/API is so similar to a typical hardware interfaces that it feels a bit like driving a NIC, and not by accident. I think it is fair to describe XDP as a generic interface to Linux network drivers that also

has a virtual backend it integrates with: the Linux network stack. In that sense, XDP can be seen as a more hardware centric sibling of `AF_PACKET`.

I can think of two reasons why XDP might be interesting for Snabb users:

- While the Snabb community prefers NICs with open hardware interface specifications there are a lot of networking devices out there that lack open specifications. Vendors often provide drivers for their devices to the Linux kernel. XDP could be a performant way to use Linux drivers for hardware that supports it.
- The XDP slow-path (software backend) seems to be ~10 times faster than `AF_PACKET`. (Oh, and there is also this eBPF based filtering capability.) XDP might be a good option for users that want to offload parts of the traffic to the kernel.

What follows is a step by step recipe for driving an XDP socket written from the point of view of a Snabb app. However, it should be generally applicable, and can for some maybe even serve as the documentation I wish I had when writing the *Snabb XDP app* (https://github.com/eugeneia/snabb/tree/xdp-app2/src/apps/xdp) (for a full picture, I encourage you to read its source).

Note: this article describes how to use the kernel system call interface directly, and how to assemble BPF by hand. There is another way to do all this using *libbpf* (ships with the kernel) and an eBPF compiler backend such as the one provided by LLVM. The examples in this article are LuaJIT code and use *ljsyscall* (http://myriabit.com/ljsyscall/) extensively.

Creating an XDP socket

First thing we need is to request a new `AF_XDP` type socket from Linux. This is done using the `socket(2)` system call. With ljsyscall this is as easy as

```
sock = assert(S.socket('xdp', 'raw'))
```

The resulting file descriptor references the object we are going to converse about with the kernel. However, it is not at all a regular socket that we `read(2)` from or `write(2)` to. I/O with XDP sockets is done using descriptor rings as with a typical hardware NIC. These descriptor ring buffers carry pointers to packet buffers. The packet buffers need to be allocated somewhere.

In XDP speak this memory region is called *UMEM* which is currently required to be contiguous and mapped into the virtual address space of the process, aligned to the page size. The region is then subdivided into a number of chunks each used to store a single packet. The chunk size is configurable, but can not be larger than the size of a page.

```
page_size = S.getpagesize()
hunk_size = page_size
num_chunks = 200000

-- Allocate UMEM
umem_size = chunk_size * num_chunks
umem_backing = ffi.new("char[?]", umem_size + page_size)
umem_addr = ffi.cast("uintptr_t", umem_backing)
umem = ffi.cast("char*", lib.align(umem_addr, page_size))
```

XDP requires us to register the memory region used for packet buffers with the kernel using the `XDP_UMEM_REG` socket option. Currently, a process can only register a single UMEM region for each socket, but you can use the same UMEM region for multiple sockets!

```
xdp_umem_reg_t = ffi.typeof[[
   struct {
      void *   addr; /* Start of packet data area */
      uint64_t len;  /* Length of packet data area */
      uint32_t chunk_size;
      uint32_t headroom;
      uint32_t flags; /* Not available in 4.19 */
   } __attribute__((packed))]]
```

The argument to `XDP_UMEM_REG` has some obvious fields (`addr`—the virtual address of the beginning of the UMEM region; `len`—its total length; `chunk_size`—the size of individual packet buffers), and some less obvious:

- `headroom`—this one might be irrelevant to you, but it is crucial for Snabb. It specifies a region in bytes that the kernel will leave empty when filling UMEM chunks with packets. If you need to store metadata in front of packets, or have adjustable packet beginnings (i.e, for prepending headers to packets) this is where you need to account for it.

- `flags`—at the time of writing there was only one flag available, `XDP_UMEM_UNALIGNED_CHUNK_FLAG` which probably lets you have unaligned (to page size?) chunk sizes. Did not use, did not test.

Anyway, to register the UMEM region with the socket we need to do a `setsockopt(2)` system call:

```
-- Register UMEM.
umem_reg = ffi.new(
   xdp_umem_reg_t,
   { addr = umem,
     len = umem_size,
     chunk_size = chunk_size,
     headroom = packet.default_headroom + packet_overhead }
)
assert(sock:setsockopt('xdp', 'xdp_umem_reg',
                       umem_reg, ffi.sizeof(umem_reg)))
```

Now we have allocated a UMEM region, and informed the kernel that we will use it with our XDP socket. Next in line are the descriptor rings used with the socket.

Mapping the descriptor rings

Each XDP socket uses four descriptor rings:

- `XDP_UMEM_FILL_RING`—for providing UMEM chunks for use by the kernel. The kernel will read packet buffers off this ring, fill them with incoming packets, and put them on the...
- `XDP_RX_RING`—descriptors filled with incoming packets end up on this ring. We read them off the ring to receive packets (and the associated UMEM chunks can be reused).
- `XDP_TX_RING`—we put descriptors on this ring to submit packets for transmission. Once the kernel is done with the packet it will put the associated UMEM on the...
- `XDP_UMEM_COMPLETION_RING`—the kernel puts UMEM chunks it no longer needs on this ring. Chunks we read off that ring can be reused.

Unlike the UMEM region which we allocated ourselves, the kernel is in charge of allocating the descriptor rings associated with the socket. But first, we tell the kernel how large they should be via `setsockopt(2)` calls.

```
xdp_ring_ndesc = 2048 -- Number of descriptors in ring.

local ndesc = ffi.new("int[1]", xdp_ring_ndesc)
assert(sock:setsockopt('xdp', 'xdp_rx_ring',
                       ndesc, ffi.sizeof(ndesc)))
assert(sock:setsockopt('xdp', 'xdp_tx_ring',
                       ndesc, ffi.sizeof(ndesc)))
assert(sock:setsockopt('xdp', 'xdp_umem_fill_ring',
                       ndesc, ffi.sizeof(ndesc)))
assert(sock:setsockopt('xdp', 'xdp_umem_completion_ring',
                       ndesc, ffi.sizeof(ndesc)))
```

We can now map the memory region containing each individual descriptor ring into our virtual address space. But first, we ask the kernel for the layout of the mappings by querying the socket's `XDP_MMAP_OFFSETS` using a `getsockopt(2)` system call.

```
xdp_ring_offset_t = ffi.typeof[[
   struct {
      uint64_t producer;
      uint64_t consumer;
      uint64_t desc;
      uint64_t flags; /* Not available in 4.19 */
   } __attribute__((packed))]]
xdp_mmap_offsets_t = ffi.typeof([[
   struct {
      $ rx,
        tx,
        fr,  /* Fill */
        cr;  /* Completion */
   } __attribute__((packed))]],
   xdp_ring_offset_t)
layouts = ffi.new(xdp_mmap_offsets_t)
sock:getsockopt('xdp', 'xdp_mmap_offsets',
                layouts, ffi.sizeof(layouts))
```

Note: this can fail if the kernel does not support the `flags` field. In this case you will have to retry without the field and do without ring flags (`kernel_has_ring_flags = false`).

Phew that is a handful. Now we are armed with the layouts for each ring. Each layout tells us the offsets of the fields of each ring. We can almost map the rings at this point. But first, you need to know that there are two different types of ring: `XDP_RX_RING` and `XDP_TX_RING` carry full packet descriptors, but `XDP_UMEM_FILL_RING` and `XDP_UMEM_COMPLETION_RING` only carry a reference to an UMEM chunk. The full packet descriptor looks like this:

```
xdp_desc_t = ffi.typeof[[
   struct {
      uint64_t addr;
      uint32_t len;
      uint32_t options;
   } __attribute__((packed))]]
```

Note: `XDP_UMEM_FILL_RING` and `XDP_UMEM_COMPLETION_RING` only carry the `addr` field.

- `addr`—offset from the beginning of the socket's UMEM region that points into a chunk containing packet data

- `len`—length of the packet
- `options`—reserved for future use? (I did not find a use for it.)

To map a ring we need to know its size (`mmap(2)` needs to be given the size of the mapping). This differs depending on whether a ring is a receive/transmit or a fill/completion ring. We can do this by summing the `desc` field's offset (because we just happen to know that its always the last field?) with the descriptor size multiplied by the number of descriptors. For example for the receive ring calculate the size like so

```
rx_ring_size = layouts.rx.desc
   + ffi.sizeof(desc_t)*xdp_ring_ndesc
```

. . . and for the fill ring we would calculate it as follows.

```
fr_ring_size = layouts.fr.desc
   + ffi.sizeof("uintptr_t")*xdp_ring_ndesc
```

Alright, finally we can map the rings. To map a specific ring we call `mmap(2)` on the socket file descriptor with a constant offset specific to the individual rings. These offsets are defined by the kernel:

- `XDP_PGOFF_RX_RING`: 0x000000000—for the receive ring
- `XDP_PGOFF_TX_RING`: 0x080000000—for the transmit ring
- `XDP_UMEM_PGOFF_FILL_RING`: 0x100000000—for the fill ring
- `XDP_UMEM_PGOFF_COMPLETION_RING`: 0x180000000—for the completion ring

For example, to map the receive ring one would then do something like this:

```
local prot = "read, write"
local flags = "shared, populate"
rx_ring = assert(S.mmap(nil, rx_ring_size,
                        prot, flags,
                        sock, 0x000000000ULL))
```

Given the ring's layout obtained previously we can now construct pointers to the fields of the ring—I will explain what how to use these later.

```
rx_producer = ffi.cast("uint32_t *",
                       rx_ring + layouts.rx.producer)
rx_consumer = ffi.cast("uint32_t *",
                       rx_ring + layouts.rx.consumer)
if kernel_has_ring_flags then
   rx_flags = ffi.cast("uint32_t *",
                       rx_ring + layouts.rx.flags)
end
rx_desc = ffi.cast("xdp_desc_t *",
                   rx_ring + layouts.rx.desc)
```

Note: if this was a fill or completion ring the type for the descriptors would be `uintptr_t *` instead.

Binding an XDP socket to an interface

This one is fairly easy. All we need do to is bind the socket using an XDP-specific socket address. It looks like this:

```
sockaddr_xdp_t = ffi.typeof[[
   struct {
      uint16_t family;
      uint16_t flags;
      uint32_t ifindex;
      uint32_t queue_id;
      uint32_t shared_umem_fd;
   } __attribute__((packed))]]
```

- `family`—this one must be `AF_XDP` (44)
- `flags`—various flags (we could set the second lowest bit here to force XDP to use zero-copy mode, i.e. go fast, or bail if the interface does not support going fast)
- `ifindex`—numeric identifier of the interface to bind to (as obtained by `if_nametoindex(3)`)
- `queue_id`—the queue to receive packets from (this is as important as it is redundant, see *Forwarding packets from a queue to an XDP socket*)

- `shared_umem_fd`—you can pass a XDP socket file descriptor here to share a single set of fill/completion rings among multiple XDP sockets (`XDP_SHARED_UMEM` flag needs to be set)

We will get by with just setting the `family`, `ifindex`, and `queue_id` flags.

```
ifname = "eth0"
queue = 0 -- first queue

-- Bind socket to interface
local sa = ffi.new(
   sockaddr_xdp_t,
   { family = 44,
     ifindex = S.util.if_nametoindex(ifname),
     queue_id = queue }
)
assert(sock:bind(sa, ffi.sizeof(sa)))
```

Forwarding packets from a queue to an XDP socket

We're almost done aren't we? Not so fast!

While we did specify a queue when binding the socket, that alone does not cause packets from that queue to be delivered to the socket. Currently, a packet can only ever be delivered to an XDP socket if it is forwarded to that socket by the eBPF program attached to the interface. A BPF program attached to an interface is also called an XDP program.

There can be only one XDP program per interface, and we might want to forward packets on different queues to different XDP sockets. This implies that the BPF program needs to multiplex incoming packets onto their respective destination sockets. To pull that off we need to employ a "BPF map", a kernel-managed data structure used to exchange data between a BPF program and userspace. So to get a packet onto an XDP socket in three easy steps we need to

- Create a BPF map that associates queues to XDP sockets
- Assemble a BPF program that forwards packets using the BPF map
- Attach our BPF program to the target interface

Creating a BPF map

BPF-related operations on Linux are performed using the `bpf(2)` system call. To create a map that can store XDP socket file descriptors we use the command code `BPF_MAP_CREATE` with map type `BPF_MAP_TYPE_XSKMAP`. We also need to supply the sizes of the key and value types, and the total number of map entries.

Ljsyscall has various wrappers for wrestling with `bpf(2)` (also, the branch the Snabb XDP driver extends the coverage of these wrappers), and those are what we will use here for the examples. The wrappers hide some arcana, but the calls should be fairly easy to translate.

```
local klen, vlen = ffi.sizeof("int"), ffi.sizeof("int")
local nentries = 128
sockmap = S.bpf_map_create('xskmap', klen, vlen, nentries)
```

Heads up: this call can fail with a permission error on some kernels. The solution I dug up is to increase the `RLIMIT_MEMLOCK` limit using the `setrlimit(2)` system call until it works.

```
local lim = assert(S.getrlimit('memlock'))
assert(S.setrlimit('memlock', {cur=lim.cur*2, max=lim.max*2}))
```

Workaround from https://github.com/xdp-project/xdp-tutorial/issues/63

The map we just created maps integers to XDP socket file descriptors. To associate a socket with a queue we use the command code `BPF_MAP_UPDATE_ELEM`. As the key we pass in our queue number, and as the value we supply the file descriptor of our socket.

```
assert(S.bpf_map_op('map_update_elem', sockmap,
                    ffi.new("int[1]", queue),
                    ffi.new("int[1]", sock:getfd())))
```

Assembling a BPF program

The the Linux kernel community seems to intend you to create BPF programs is by writing the program in a subset of C which is then to be compiled by Clang and emitted as a BPF program by a LLVM backend. In the spirit of generating code at runtime, and to avoid a big complex dependency we are going to skip past that and do some hot needling.

BPF is a fairly simple fantasy instruction set. The kernel is capable of loading and executing programs encoded in this instruction set, and it defines a single eBPF instruction like so:

```
bpf_ins = ffi.typeof[[
   struct {
      uint8_t op;       /* opcode */
      uint8_t dst:4;    /* dest register */
      uint8_t src:4;    /* source register */
      int16_t off;      /* signed offset */
      int32_t imm;      /* signed immediate constant */
   } __attribute__((packed))
]]
```

Knowing that the kernel will accept a sequence of these instructions as a BPF program, and armed with a considerable set of constants defined by the kernel to fill the instruction fields with, it is imaginable to assemble them by hand.

```
{ op=bor(c.ALU, a.MOV, s.K), dst=3, imm=0 }
```

This instruction loads the constant zero into the third register. To get the right opcode we really just need to bitwise-or a bunch of constants (operand class, ALU mode, source mode, ...). Here, `op` is `0x04|0xb0|0x00` in C syntax. See *include/uapi/linux/bpf.h*

We can totally do this. We really just need to understand the BPF calling convention, the special context passed to XDP programs, and some BPF built-ins we need to call. BPF is a register machine: call arguments are passed in registers 1..*N*, and the return value must be placed in register zero. XDP programs are special in the sense that they get passed a `struct xdp_md *` context in register one, and can invoke certain built-ins used for forwarding packets. Calling built-ins such as `redirect_map` (which we will use) follows the generic calling convention.

```
function bpf_asm (insn)
   return ffi.typeof("$[?]", bpf_ins)(#insn, insn)
end
```

Assemble a sequence of BPF instructions.

Below is the shortest program I could come up with that does the job. You could go crazy too, and do way more elaborate filtering of packets here (e.g., sending some to the kernel networking stack etc.).

```
local insns = bpf_asm{
   -- r3 = XDP_ABORTED
   { op=bor(c.ALU, a.MOV, s.K), dst=3, imm=0 },
   -- r2 = ((struct xdp_md *)ctx)->rx_queue_index
   { op=bor(c.LDX, f.W, m.MEM), dst=2, src=1, off=16 },
   -- r1 = sockmap
   { op=bor(c.LD, f.DW, m.IMM), dst=1, src=s.MAP_FD,
     imm=sockmap:getfd() },
   { imm=0 }, -- nb: upper 32 bits of 64-bit (DW) immediate
   -- r0 = redirect_map(r1, r2, r3)
   { op=bor(c.JMP, j.CALL), imm=fn.redirect_map },
   -- EXIT:
   { op=bor(c.JMP, j.EXIT) }
}
```

Note that we are embedding our BPF map into the program by doing a special load with its file descriptor as the source operand. We can load the queue index of the packet via the `struct xdp_md *` context look up the associated XDP socket in our map. Kernel magic.

There we have our XDP program. All that left to do is load it with the `bpf(2)` command code `BPF_PROG_LOAD` and make sure we load it as `BPF_PROG_TYPE_XDP`. Good news: in case you messed up the BPF program I found that the error log produced by `BPF_PROG_LOAD` is really detailed and helpful!

```
prog, err, log = S.bpf_prog_load(
   'xdp',
   insns, ffi.sizeof(insns) / ffi.sizeof(bpf.ins),
   "Apache 2.0"
 )
 if prog then
    return prog
 else
    error(tostring(err).."\n"..log)
 end
```

Attaching a BPF program to an interface

Almost there! In the last step of our setup we need to tell the kernel to run our BPF program on packets incoming on our given interface. To do that we need to use the `netlink(7)` facility which entails opening

a special kind of socket, crafting a message that expresses our desire, sending it to the kernel via the netlink socket, and finally reading back the kernel's response.

First we create the socket using the `socket(2)` system call with address family `AF_NETLINK`, and type `SOCK_RAW` since we will send a datagram. The `NETLINK_ROUTE` protocol allows us to update the link (interface) settings. We also enable extend extended ACK reporting in the hope of obtaining more detailed error messages in case something goes wrong. (Truthfully, I didn't have any luck with that, might be that my kernel was too old to support extended ACKs for XDP settings at the time of writing?)

```
local netlink = assert(S.socket('netlink', 'raw', 'route'))
local SOL_NETLINK = 270
local NETLINK_EXT_ACK = 11
local ext_ack_on = ffi.new("int[1]", 1)
assert(netlink:setsockopt(SOL_NETLINK, NETLINK_EXT_ACK,
                          ext_ack_on, ffi.sizeof(ext_ack_on)))
```

Up next is crafting a datagram that will turn the right knob. Every netlink request has common header which is in our case followed by an `ifinfomsg` that selects the interface by index, followed by two attributes that say "XDP, nested attribute follows!" and "Here is a file descriptor to an XDP program!" respectively. The whole structure looks like this:

```
netlink_set_link_xdp_request_t = ffi.typeof[[
   struct {
     struct { /* nlmsghdr */
       uint32_t nlmsg_len; /* Length */
       uint16_t nlmsg_type; /* Content */
       uint16_t nlmsg_flags; /* Flags */
       uint32_t nlmsg_seq; /* Sequence # */
       uint32_t nlmsg_pid; /* Port ID */
     } nh;
     struct { /* ifinfomsg */
       unsigned char ifi_family;
       unsigned char __ifi_pad;
       unsigned short ifi_type; /* ARPHRD_* */
       int ifi_index; /* Link index */
       unsigned ifi_flags; /* IFF_* flags */
       unsigned ifi_change; /* IFF_* mask */
     } ifinfo;
     struct { /* nlattr */
       uint16_t nla_len;
       uint16_t nla_type;
     } xdp;
     struct { /* nlattr */
       uint16_t nla_len;
       uint16_t nla_type;
       int32_t fd;
     } xdp_fd;
   }__attribute__((packed))]]
```

For what it's worth: I would not have known what kind of request to send if I had not read the libbpf sources. As far as I know this is not documented anywhere at this point?

Luckily, a bunch of those fields we can leave zeroed. There is no shortage of fields we have to fill with magic values though. Some of those constants are in ljsyscall already, others can be looked up in the kernel sources. The operands should be clear however: we select an interface by index obtained via `if_nametoindex(3)`, and pass the file descriptor that represents our loaded BPF program. Oh, and there a multiple length fields which you really need to get right!

```
local IFLA_XDP = 43
local IFLA_XDP_FD = 1
local IFLA_XDP_FLAGS = 3
local request = ffi.new(
   netlink_set_link_xdp_request_t,
   { nh = { nlmsg_flags = bor(S.c.NLM_F.REQUEST, S.c.NLM_F.ACK),
            nlmsg_type = S.c.RTM.SETLINK },
     ifinfo = { ifi_family = S.c.AF.UNSPEC,
                ifi_index = S.util.if_nametoindex(ifname) },
     xdp = { nla_type = bor(bits{ NLA_F_NESTED=15 }, IFLA_XDP) },
     xdp_fd = { nla_type = IFLA_XDP_FD,
                fd = prog:getfd() } }
)
request.nh.nlmsg_len = ffi.sizeof(request)
request.xdp.nla_len = ffi.sizeof(request.xdp)
   + ffi.sizeof(request.xdp_fd)
request.xdp_fd.nla_len = ffi.sizeof(request.xdp_fd)
assert(netlink:send(request, ffi.sizeof(request)))
local response = assert(S.nl.read(netlink, nil, nil, true))
if response.error then
   error("NETLINK responded with error: "
            ..tostring(response.error))
end
```

Heads up! When this failed I really had not much remedy besides reading kernel sources. The error code does not tell you too much.

Packet I/O

We've done it! We can receive packets now! But first we need to put chunks on the fill ring so the kernel can fill them with incoming packets. First thing we need to do is carve out a chunk from our UMEM region.

```
umem_used = 0
function alloc_chunk ()
   assert(umem_used + chunk_size <= umem_size,
          "Ran out of UMEM!.")
   local chunk = umem + umem_used
   umem_used = umem_used + chunk_size
   return chunk
end
```

Remember that the descriptor rings carry offsets relative to the beginning of the UMEM region. We will have to convert from pointers into our virtual address space to relative offsets, and vice versa.

```
function to_umem (ptr)
   return ffi.cast("uintptr_t", ptr) - ffi.cast("uintptr_t", umem)
end

function from_umem (offset)
   return umem + offset
end
```

The single-producer, single-consumer rings exposed by XDP follow the *Array + two unmasked indices* (https://www.snellman.net/blog/archive/2016-12-13-ring-buffers/) design. Meaning we have an array of descriptors, and two unsigned 32-bit cursors that track the read and write position respectively. When we increment the cursors we let them exceed the array boundaries, and when they would overflow we let them wrap around in the modulo 32-bit range. To translate the cursors to indices into the array we mask off the extra bits—the array's size must be a power of two!

Assume we mapped the fill ring to `fr_producer`, `fr_consumer` (of type `uint32_t *`), and `fr_desc` (of type `uintptr_t *`). We can get the number of entries in the fill ring like this:

```
function fr_entries ()
   return tobit(fr_producer[0] - fr_consumer[0])
end
```

Note: `tobit` truncates its argument into the 32-bit range.

We are the producer of the fill ring. To supply fresh chunks to the kernel we could write a loop like this:

```
while fr_entries() < xdp_ring_ndesc do
   local idx = band(fr_producer[0], xdp_ring_ndesc - 1)
   fr_desc[idx] = to_umem(alloc_chunk())
   fr_producer[0] = tobit(fr_producer[0] + 1)
end
```

Note: `band` is bitwise *and*.

Now the kernel, being the fill ring's consumer, can read chunks off the ring and put them on the receive ring. On the receive ring the roles are swapped: the kernel acts as producer and we act as the consumer. We could receive a packet like so:

```
function receive ()
   local idx = band(rx_consumer[0], xdp_ring_ndesc - 1)
   local data = from_umem(rx_desc[idx].addr)
   local len = rx_desc[idx].len
   rx_consumer[0] = tobit(rx_consumer[0] + 1)
   return data, len
end
```

Note: the chunk used to received a packet is now ours again. We can modify it and put it on the transmit ring, or recycle it, and put it back on the fill ring or potentially a freelist of some sort that we manage.

The transmit path works in almost the same way, just opposite in direction. We produce descriptors on the transmit ring, and then we might have to kick the kernel to wake up to consume the queued packets. If the kernel supports ring flags (`kernel_has_ring_flags == true`) we can check if kicking is necessary. Otherwise, we just have to assume it is. Kicking the kernel is done by sending an empty message over the socket via a non-blocking `sendto(2)` system call.

```
if (not kernel_has_ring_flags) or
   band(tx_flags[0], bits{XDP_RING_NEED_WAKEUP=1})
then
   sock:sendto(nil, 0, 'dontwait', nil, 0)
end
```

Note: if the first bit in the transmit ring's flags field is set the kernel needs a kick. The result of `sendto(2)` tells you if transmit has been successful.

When the kernel sends the packets, it puts the used chunks on the completion ring. Chunks that we consume from the completion ring can be put back on the fill ring, or used to send the next batch of packets, or put on a freelist managed in userspace. In any case, you absolutely do need to make sure to fill the fill ring and drain completion ring. The receive and transmit paths will block if either of those rings would underflow or overflow.

If you want to go fast—you are using XDP, so chances are that you do—the examples above are *not* how you would do transmit and receive. Beyond being single-producer, single-consumer ring buffers, XDP

descriptor rings are designed to be efficient in multi-core CPUs. Specifically, they also follow the *MCRingBuffer* (https://www.cse.cuhk.edu.hk/~pclee/www/pubs/ancs09poster.pdf) design.

Consider this: the read and write cursors (as well as the descriptor arrays) of XDP rings are stored in memory, and different processors cores may have separate caches. So if one core updates a cursor it might invalidate the cache of another core. That core will then incur a cache miss when accessing the cursor. These cache misses are expensive in terms of latency so we really want to avoid them as much as possible. To that end we make sure that

- the cursors are on separate cache lines so that updating one does not affect the other (keyword: "false sharing")
- we do not read or write from the cursors unless absolutely necessary to minimize cache misses

If you were wondering why *Mapping the descriptor rings* was so contrived, it's because the size of a cache line depends on the CPU model. Hence, the memory layout of the rings can differ depending on the CPU. The `XDP_MMAP_OFFSETS` socket option can be queried to get the layout for you current CPU.

Additionally, we really do not want to read `rx_producer` and `rx_consumer` for every packet we receive or transmit. We can work around this by keeping local copies of the cursor, which synchronize with the shared cursors when necessary.

```
local rx_read, rx_write = 0, 0
```

Local copies of the receive ring's cursors.

Before reading a batch of packets we could query the shared producer cursor once to find out how many packets we can receive. Say we want to process up to one hundred packets at a time, so we also set an upper bound for the batch size.

```
local rx_batch = min(tobit(rx_producer[0] - rx_read), 100)
```

We can then receive the incoming packets in a tight loop without touching the shared producer and consumer cursors.

```
for _ = 1, rx_batch do
   local idx = band(rx_read, xdp_ring_ndesc - 1)
   local data = from_umem(rx_desc[idx].addr)
   local len = rx_desc[idx].len
   rx_read = tobit(rx_read + 1)
   -- do something with the packet here
end
```

And only once we are done with our batch of packets we update the shared consumer cursor. Note that x86_64 does not require a memory barrier here because it maintains *Total store order* (TSO) across cores. Other CPU architectures may require a memory barrier prior to updating the shared cursor.

```
rx_consumer[0] = rx_read
```

Deallocating the XDP socket

At the end of the day we might want to deallocate the XDP socket. Using the `close(2)` system call we can stop the flow of packets and deallocate the socket.

```
sock:close()
```

The next step is to unmap the the rings we mapped into our virtual address space. Before we do that we first might want to reclaim the chunks that are still on the rings. If you can get away with deallocating the whole UMEM region you do not need to bother. However, if you (like me) share the UMEM region with multiple stand-alone XDP sockets (i.e., with independent fill/completion rings), and need to know which chunks exactly remain on the rings and must bet put back onto a freelist, then things can get a little icky.

Turns out, we currently can not reliably inspect the kernel's internal read cursors for the transmit and fill rings. There is a hack that works for now, it goes as follows (hopefully, this will get easier in future kernels).

You keep two counters. One is incremented when you put a chunk on the fill ring, and decremented when read a packet off the receive ring. The other counter is incremented when you put a packet on the transmit ring, and decremented when you take a chunk off the completion ring. After closing the XDP socket you can then flush the receive and completion rings—e.g., consume all packets that remain on those rings.

When you update your counters accordingly you now know how many chunks are still in-flight on the fill and transmit rings. You can then undo the right amount of writes to the fill and transmit rings to reclaim your chunks. This hack depends on the assumption that the kernel...

- does not modify the rings after closing the XDP socket
- moves packets from the fill ring to the receive ring and from the transmit ring to the completion ring *in-order*
- does not clobber descriptors that have not yet moved to a receive or completion ring

Finally, you can unmap the rings using a `munmap(2)` system call. You might also want to close the file descriptors of the BPF program and map.

```
S.munmap(rx_ring, rx_ring_size)
```