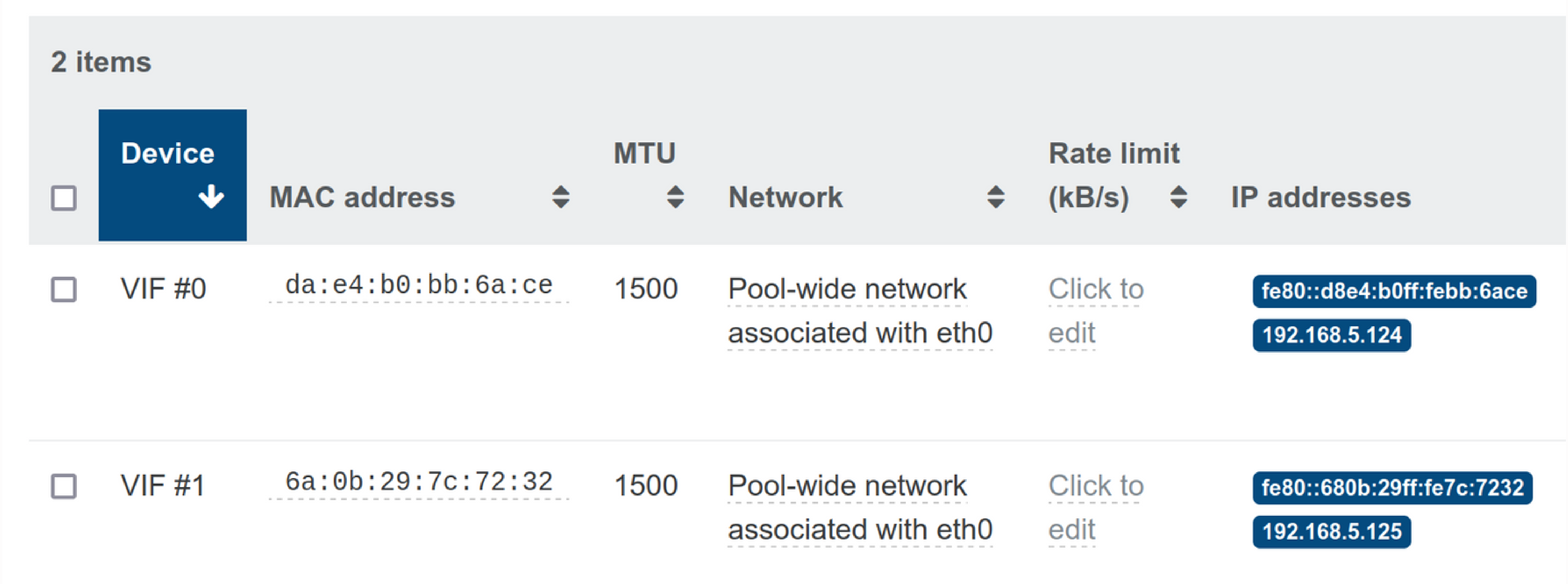
Created a ubuntu 20 LTS VM with 2 Network interfaces



2 items

Device	MAC address	MTU	Network	Rate limit (kB/s)	IP addresses
VIF #0	da:e4:b0:bb:6a:ce	1500	Pool-wide network associated with eth0	Click to edit	fe80::d8e4:b0ff:febb:6ace 192.168.5.124
VIF #1	6a:0b:29:7c:72:32	1500	Pool-wide network associated with eth0	Click to edit	fe80::680b:29ff:fe7c:7232 192.168.5.125

Vif0 (rx) Vif0 (tx) Vif1 (rx) Vif1 (tx)

```
root@ubuntu:/home/ubuntu/test/libbpf-0.4.0/src# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.124  netmask 255.255.255.0  broadcast 192.168.5.255
        inet6 fe80::d8e4:b0ff:febb:6ace  prefixlen 64  scopeid 0x20<link>
        ether da:e4:b0:bb:6a:ce  txqueuelen 1000  (Ethernet)
        RX packets 142057  bytes 205610725 (205.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 65942  bytes 5212758 (5.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.125  netmask 255.255.255.0  broadcast 192.168.5.255
        inet6 fe80::680b:29ff:fe7c:7232  prefixlen 64  scopeid 0x20<link>
        ether 6a:0b:29:7c:72:32  txqueuelen 1000  (Ethernet)
        RX packets 174386  bytes 274380382 (274.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 106259  bytes 8658892 (8.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task we're going to perform:

1. Writing a program to drop all packets.
2. Building and viewing a BPF(Berkeley Packet Filter) object.
3. Loading a BPF object.
4. Show information on a running BPF object.
5. Unloading a BPF object.

Step 1: Install development environment

Install the required packages using the following code:

```
$ sudo dnf install clang llvm gcc libbpf libbpf-devel libxdp libxdp-devel xdp-tools bpft
ool kernel-headers
```

This command is for Rad Hat Enterprise Linux8 (RHEL8) so we have to modify this command to be used in ubuntu

- Command to perform this task on ubuntu 20

```
$ sudo apt install clang llvm libelf-dev libpcap-dev gcc-multilib
 build-essential
```

```
$ sudo apt install linux-tools-$(uname -r)
```

```
$ sudo apt install linux-headers-$(uname -r)
```

```
$ sudo apt install linux-tools-common linux-tools-generic
$ sudo apt install tcpdump
```

eBPF programs are written in CLang

Created a file xdp_drop.c

```
#include <linux/bpf.h>
#include </home/ubuntu/test/libbpf-0.4.0/src/bpf_helpers.h>

SEC("xdp_drop")
int xdp_drop_prog(struct xdp_md *ctx)
{
    return XDP_DROP;
}
```

this file is provided by the kernel-header package, which defines all the supported BPF helpers and xdp_actions like we've used XDP_DROP action

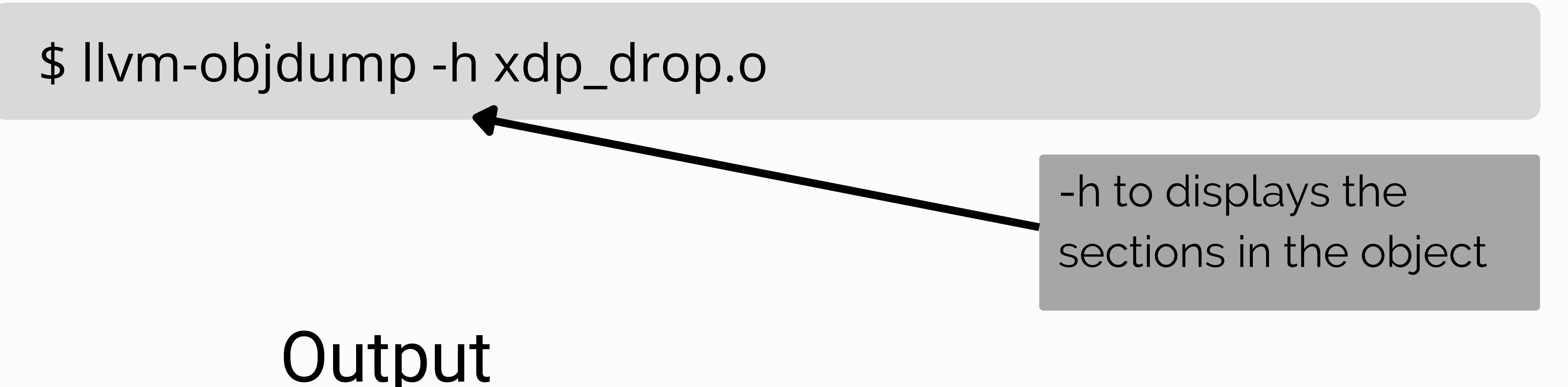
this module is not found by the compiler therefore manually downloaded from GitHub and provided the complete path to compiler

Build and dump the BPF object

```
$ clang -O2 -g -Wall -target bpf -c xdp_drop.c -o xdp_drop.o
```

-O to define output file

Using llvm-objdump to view ELF format after the build



llvm-objdump is used to know what a program does , if you don't have the source code

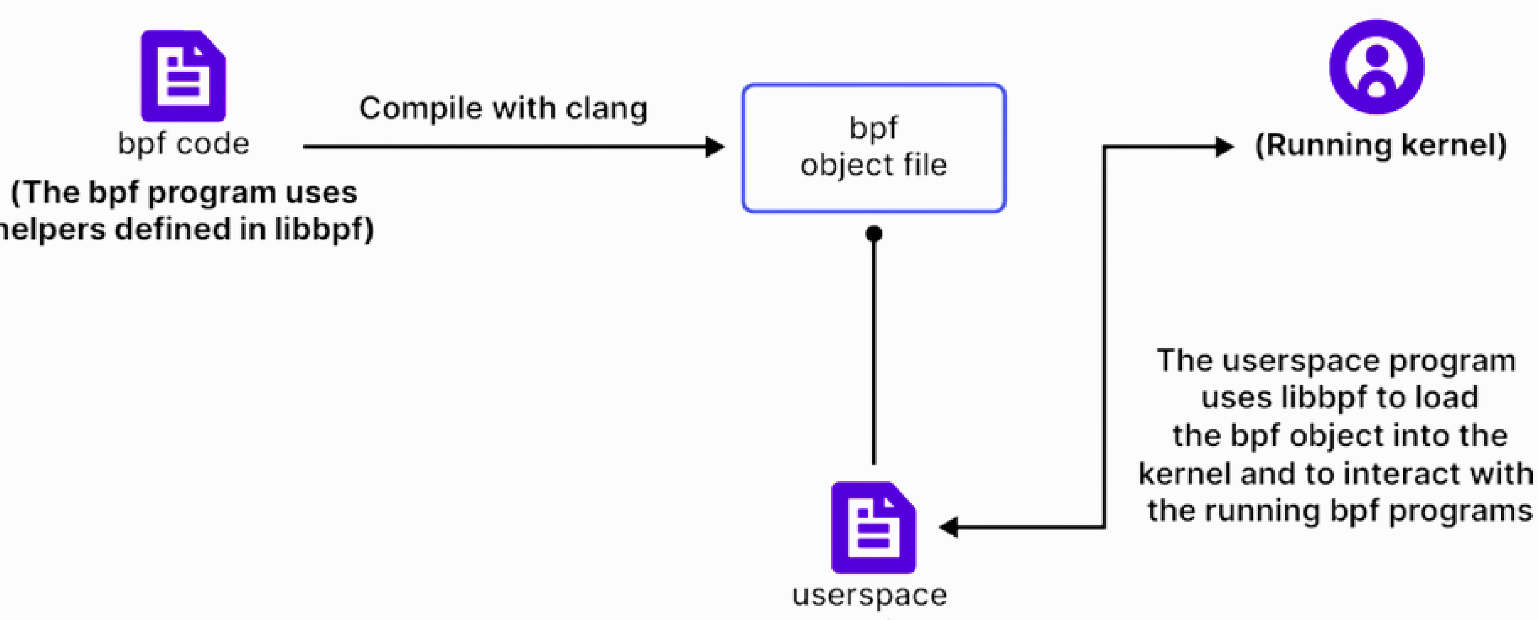
Output

```
root@ubuntu:/home/ubuntu/test/libbpf-0.4.0/src# llvm-objdump -h xdp_drop.o

xdp_drop.o:     file format ELF64-BPF

Sections:
Idx Name               Size     VMA              Type
  0                    00000000 0000000000000000
  1 .strtab            000000a3 0000000000000000
  2 .text              00000000 0000000000000000 TEXT
  3 xdp_drop           00000010 0000000000000000 TEXT
  4 .debug_str         000000f1 0000000000000000
  5 .debug_abbrev      0000008e 0000000000000000
  6 .debug_info        000000e5 0000000000000000
  7 .rel.debug_info    00000180 0000000000000000
  8 .BTF               0000016c 0000000000000000
  9 .BTF.ext           00000050 0000000000000000
 10 .rel.BTF.ext       00000020 0000000000000000
 11 .debug_frame       00000028 0000000000000000
 12 .rel.debug_frame   00000020 0000000000000000
 13 .debug_line        00000086 0000000000000000
 14 .rel.debug_line    00000010 0000000000000000
 15 .llvm_addrsig      00000001 0000000000000000
 16 .symtab            00000288 0000000000000000
```

FLow Diagram



Use llvm-objdump to view ELF format after the build

```
$ llvm-objdump -S -no-show-raw-insn xdp_drop.o
```

-S option displays the source interleaved with the disassembled object code

Output

```
root@ubuntu:/home/ubuntu/test/libbpf-0.4.0/src# llvm-objdump -S -no-show-raw-insn xdp_drop.o

xdp_drop.o:     file format ELF64-BPF


Disassembly of section xdp_drop:

0000000000000000 xdp_drop_prog:
;       return XDP_DROP;
       0:       r0 = 1
       1:       exit
```

Attaching the XDP program the device (lo) :

```
$ ip link set dev lo xdpgeneric obj xdp_pass_kern.o sec xdp
```

Listing the device via ip link show also shows the XDP info:

```
$ ip link show dev lo
```

Let's have a look at running BPF programs and activities on our device

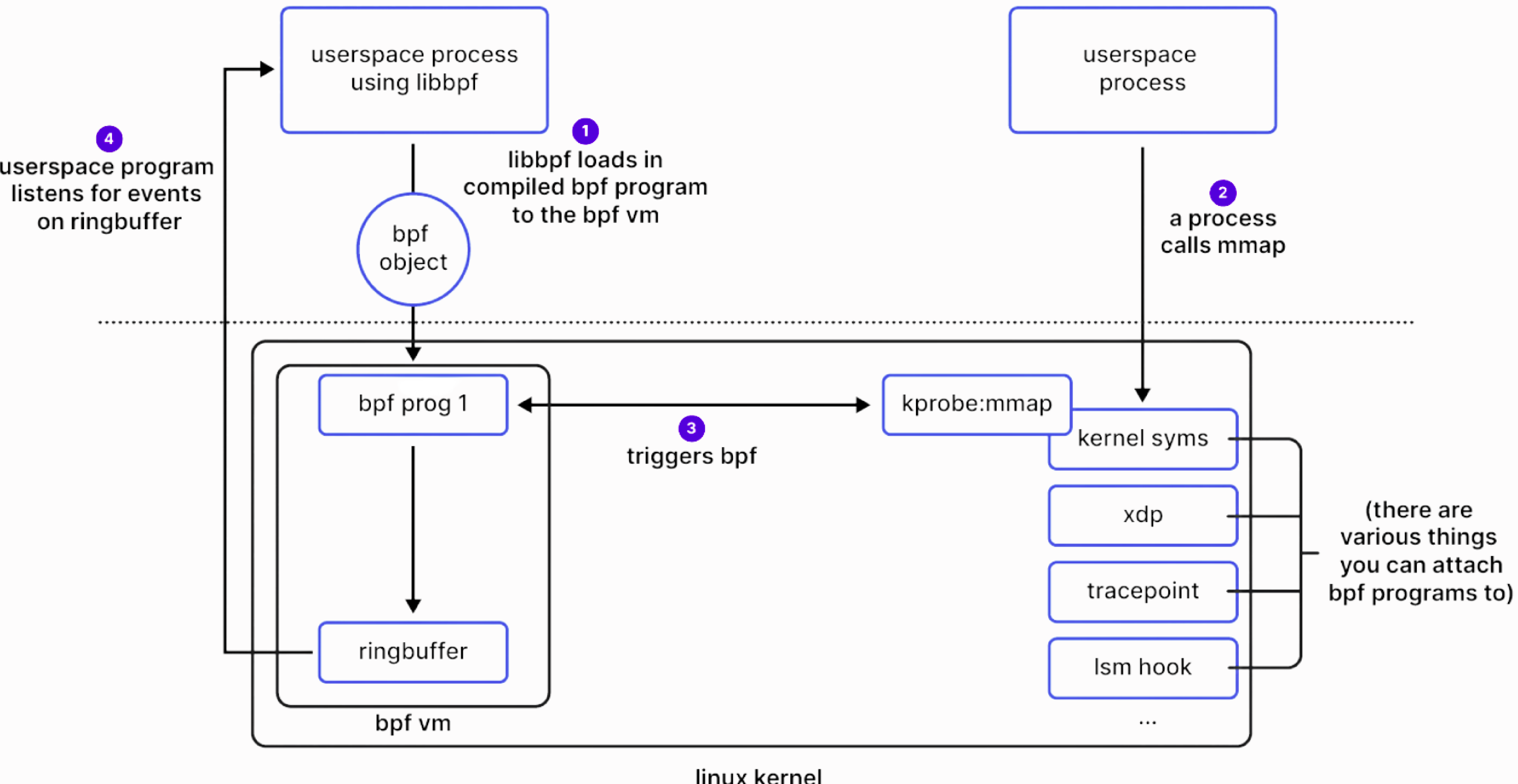
```
root@ubuntu:/home/ubuntu# sudo ip link show lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 xdpgeneric qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    prog/xdp id 82 tag 3b185187f1855c4c jited
```

The program that we've created is attached here

Running BPF programs and activity on our interface device

```
root@ubuntu:/home/ubuntu# sudo ip link show lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 xdpgeneric qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    prog/xdp id 82 tag 3b185187f1855c4c jited
root@ubuntu:/home/ubuntu# sudo bpftool prog show
68: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-05-21T07:42:33+0000  uid 0
        xlated 64B  jited 61B  memlock 4096B
69: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-05-21T07:42:33+0000  uid 0
        xlated 64B  jited 61B  memlock 4096B
70: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-05-21T07:42:33+0000  uid 0
        xlated 64B  jited 61B  memlock 4096B
71: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-05-21T07:42:33+0000  uid 0
        xlated 64B  jited 61B  memlock 4096B
72: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-05-21T07:42:33+0000  uid 0
        xlated 64B  jited 61B  memlock 4096B
73: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-05-21T07:42:33+0000  uid 0
        xlated 64B  jited 61B  memlock 4096B
82: xdp  name xdp_prog_simple  tag 3b185187f1855c4c  gpl
        loaded_at 2022-05-21T07:47:49+0000  uid 0
        xlated 16B  jited 35B  memlock 4096B
```

The program that we've created is attached here



This flow diagram is taken from another site, This diagram is partially relevent with our program, but good to create a basic understanding

Now, some points that I found important

- So we need high-performance programmable access to networking packets before they enter the networking stack.
- Some eBPF helpers are accessible only by GPL-licensed programs, so we need to add the license at the end of program, like this

```
char _license[] SEC("license") = "GPL";
```

- we're not loading XDP programs on the default interface. Instead, we use the eth1 interface for testing. if we use the default interface then we may lose internet access because our XDP program is dropping the packets.
- I tried to load the XDP program via xdp-loader but Ubuntu doesn't have this module available.

I can also test this program on CentOS and Fedora, But currently, I don't have the .ova image for both of them to create a VM on XEN orchestra, As ubuntu provides .ova images openly but CentOS and Fedora Don't.

https://webthesis.biblio.polito.it/15948/1/tesi.pdf

https://blog.aquasec.com/libbpf-ebpf-programs

https://github.com/libbpf/libbpf

Basic definitions and concepts are covered in other pdf that I created earlier as my learning resource