Lab: Network Drivers

In this lab you will be responsible for completing the DMA based network driver for QEMU's emulation of E1000 NIC. On this emulated LAN, xv6 (the "guest") has an IP address of 10.0.2.15. Qemu arranges for the computer running qemu (the "host") to appear on the LAN with IP address 10.0.2.2. When xv6 uses the E1000 to send a packet to 10.0.2.2, qemu delivers the packet to the appropriate application on the host. You will be using qemu's 'user mode' network stack.

There already exists a partially completed network stack that handles IP, UDP and ARP. The stack for transmitting packets from userspace to the network is complete. However, the code to receive packets and deliver them to userspace is missing.

Part 1: Completing the NIC driver

Your task is to complete the driver for the `e1000` network card. The initialization and the interrupt handler stubs are already written for you. You are tasked with interfacing with the network stack and sending and receiving packets via the DMA buffers.

Relevant documentation/references:

- QEMU docs on user mode networking stack
- Interrupts and Device Drivers - xv6
- Intel E1000 SDM
  - Section 2
  - Section 3.2
  - Section 3.3 + 3.4
  - Section 13
  - Section 14

Part 2: Completing the network stack

You will modify the network stack to receive UDP packets, queue them and allow user processes to read them.

You need to provide the implementations of the following system calls:

- `send(short sport, int dst, short dport, char *buf, int len)`: This system call sends a UDP packet to the host with IP address `dst`, and (on that host) the process listening to port `dport`. The packet's source port number will be `sport` (this port number is reported to the receiving process, so that it can reply to the sender). The content ("payload") of the UDP packet will the `len` bytes at address `buf`. The return value is 0 on success, and −1 on failure.
- `recv(short dport, int *src, short *sport, char *buf, int maxlen)`: This system call returns the payload of a UDP packet that arrives with destination port `dport`. If one or more packets arrived before the call to `recv()`, it should return right away with the earliest waiting packet. If no packets are waiting, `recv()` should wait until a packet for `dport` arrives. `recv()` should see arriving packets for a given port in arrival order. `recv()` copies the packet's 32-bit source IP address to `*src`, copies the packet's 16-bit UDP source port number to `*sport`, copies at most maxlen bytes of the packet's UDP payload to buf, and removes the packet from the queue. The system call returns the number of bytes of the UDP payload copied, or −1 if there was an error.
- `bind(short port)`: A process should call `bind(port)` before it calls `recv(port, ...)`. If a UDP packet arrives with a destination `port` that hasn't been passed to `bind()`, network stack should discard that packet. The reason for this system call is to initialize any structures the network stack needs in order to store arriving packets for a subsequent `recv()` call.

All the addresses and port numbers passed as arguments to these system calls, and returned by them, must be in host byte order.

If there are errors or omissions in your E1000 code, they may only start to cause problems during the ping tests. For example, the ping tests send and receive enough packets that the descriptor ring indices will wrap around.

Testing

For this lab, **testing must be done outside the docker container**. Install `qemu-system-riscv64` on your linux/mac machines (use WSL2 if you're on windows).

In one window, run `make qemu` to boot the xv6 kernel, and run `nettest grade` inside xv6. Then, in another window, run `./nettest.py grade`.

The output in xv6 should look like this:

```
$ nettest grade
txone: sending one packet
arp_rx: received an ARP packet
ip_rx: received an IP packet
bp is 0!
ping0: starting
ping0: OK
ping1: starting
ping1: OK
ping2: starting
ping2: OK
ping3: starting
bp is 0!
bp is 0!
bp is 0!
bp is 0!
bp is 0!
bp is 0!
bp is 0!
bp is 0!
bp is 0!
ping3: OK
dns: starting
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
dns: OK
free: OK
```

For individual tests, run `nettest` to see the options. The Makefile also configures QEMU to record all incoming and outgoing packets to the file `packets.pcap` in your lab directory. It may be helpful to review these recordings to confirm that xv6 is transmitting and receiving the packets you expect. To display the recorded packets, run `tcpdump -XXnr packets.pcap`.

Starting the lab

For this lab alone, you will have to run the kernel outside of docker. Building can still be done in docker, but the qemu configured in docker does not have support for networking emulation. Install qemu on your systems and then run `make qemu` on your machine (after building the kernel inside docker).

- Clone the repo: `git clone git://g.csail.mit.edu/xv6-labs-2024`
- `cd xv6-labs-2024`
- Switch to the net branch: `git switch net`

- Start writing code

Please start the lab well ahead of time, since this is going to be more challenging than the other labs.

Submission

Regular submission guidelines apply. Make a tar.gz of the repo with your solutions, the report explaining your approach and the code, and screenshots of the test execution.

On use of Agentic LLMs for code

Please read this and this before you plan to start using any AI agent or LLMs to write your code or do your thinking for you.