Lab: networking

In this lab you will write an xv6 device driver for a network interface card (NIC).

Fetch the xv6 source for the lab and check out the `net` branch:

```
$ git fetch
$ git checkout net
$ make clean
```

Background

> Before writing code, you may find it helpful to review "Chapter 5: Interrupts and device drivers" in the xv6 book.

You'll use a network device called the E1000 to handle network communication. To xv6 (and the driver you write), the E1000 looks like a real piece of hardware connected to a real Ethernet local area network (LAN). In fact, the E1000 your driver will talk to is an emulation provided by qemu, connected to a LAN that is also emulated by qemu. On this emulated LAN, xv6 (the "guest") has an IP address of 10.0.2.15. Qemu also arranges for the computer running qemu 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 (real) computer on which you're running qemu (the "host").

You will use QEMU's "user-mode network stack". QEMU's documentation has more about the user-mode stack here. We've updated the Makefile to enable QEMU's user-mode network stack and the E1000 network card.

The Makefile 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:

```
tcpdump -XXnr packets.pcap
```

We've added some files to the xv6 repository for this lab. The file `kernel/e1000.c` contains initialization code for the E1000 as well as empty functions for transmitting and receiving packets, which you'll fill in. `kernel/e1000_dev.h` contains definitions for registers and flag bits defined by the E1000 and described in the Intel E1000 Software Developer's Manual. `kernel/net.c` and `kernel/net.h` contain a simple network stack that implements the IP, UDP, and ARP protocols. These files also contain code for a flexible data structure to hold packets, called an `mbuf`. Finally, `kernel/pci.c` contains code that searches for an E1000 card on the PCI bus when xv6 boots.

Your Job (hard)

> Your job is to complete `e1000_transmit()` and `e1000_recv()`, both in `kernel/e1000.c`, so that the driver can transmit and receive packets. You are done when `make grade` says your solution passes all the tests.

> While writing your code, you'll find yourself referring to the E1000 Software Developer's Manual. Of particular help may be the following sections:

- Section 2 is essential and gives an overview of the entire device.
- Section 3.2 gives an overview of packet receiving.
- Section 3.3 gives an overview of packet transmission, alongside section 3.4.
- Section 13 gives an overview of the registers used by the E1000.
- Section 14 may help you understand the init code that we've provided.

Browse the E1000 Software Developer's Manual. This manual covers several closely related Ethernet controllers. QEMU emulates the 82540EM. Skim Chapter 2 now to get a feel for the device. To write your driver, you'll need to be familiar with Chapters 3 and 14, as well as 4.1 (though not 4.1's subsections). You'll also need to use Chapter 13 as a reference. The other chapters mostly cover components of the E1000 that your driver won't have to interact with. Don't worry about the details at first; just get a feel for how the document is structured so you can find things later. The E1000 has many advanced features, most of which you can ignore. Only a small set of basic features is needed to complete this lab.

The `e1000_init()` function we provide you in `e1000.c` configures the E1000 to read packets to be transmitted from RAM, and to write received packets to RAM. This technique is called DMA, for direct memory access, referring to the fact that the E1000 hardware directly writes and reads packets to/from RAM.

Because bursts of packets might arrive faster than the driver can process them, `e1000_init()` provides the E1000 with multiple buffers into which the E1000 can write packets. The E1000 requires these buffers to be described by an array of "descriptors" in RAM; each descriptor contains an address in RAM where the E1000 can write a received packet. `struct rx_desc` describes the descriptor format. The array of descriptors is called the receive ring, or receive queue. It's a circular ring in the sense that when the card or driver reaches the end of the array, it wraps back to the beginning. `e1000_init()` allocates `mbuf` packet buffers for the E1000 to DMA into, using `mbufalloc()`. There is also a transmit ring into which the driver places packets it wants the E1000 to send. `e1000_init()` configures the two rings to have size `RX_RING_SIZE` and `TX_RING_SIZE`.

When the network stack in `net.c` needs to send a packet, it calls `e1000_transmit()` with an mbuf that holds the packet to be sent. Your transmit code must place a pointer to the packet data in a descriptor in the TX (transmit) ring. `struct tx_desc` describes the descriptor format. You will need to ensure that each mbuf is eventually freed, but only after the E1000 has finished transmitting the packet (the E1000 sets the `E1000_TXD_STAT_DD` bit in the descriptor to indicate this).

When the E1000 receives each packet from the ethernet, it first DMAs the packet to the mbuf pointed to by the next RX (receive) ring descriptor, and then generates an interrupt. Your `e1000_recv()` code must scan the RX ring and deliver each new packet's mbuf to the network stack (in `net.c`) by calling `net_rx()`. You will then need to allocate a new mbuf and place it into the descriptor, so that when the E1000 reaches that point in the RX ring again it finds a fresh buffer into which to DMA a new packet.

In addition to reading and writing the descriptor rings in RAM, your driver will need to interact with the E1000 through its memory-mapped control registers, to detect when received packets are available and to inform the E1000 that the driver has filled in some TX descriptors with packets to send. The global variable `regs` holds a pointer to the E1000's first control register; your driver can get at the other registers by indexing `regs` as an array. You'll need to use indices `E1000_RDT` and `E1000_TDT` in particular.

To test your driver, run `make server` in one window, and in another window run `make qemu` and then run `nettests` in xv6. The first test in `nettests` tries to send a UDP packet to the host operating system, addressed to the program that `make server` runs. If you haven't completed the lab, the E1000 driver won't actually send the packet, and nothing much will happen.

After you've completed the lab, the E1000 driver will send the packet, qemu will deliver it to your host computer, `make server` will see it, it will send a response packet, and the E1000 driver and then `nettests` will see the response packet. Before the host sends the reply, however, it sends an "ARP" request packet to xv6 to find out its 48-bit Ethernet address, and expects xv6 to respond with an ARP reply. `kernel/net.c` will take care of this once you have finished your work on the E1000 driver. If all goes well, `nettests` will print `testing ping: OK`, and `make server` will print `a message from xv6!`.

`tcpdump -XXnr packets.pcap` should produce output that starts like this:

```
reading from file packets.pcap, link-type EN10MB (Ethernet)
15:27:40.861988 IP 10.0.2.15.2000 > 10.0.2.2.25603: UDP, length 19
        0x0000:  ffff ffff ffff 5254 0012 3456 0800 4500  ......RT..4V..E.
        0x0010:  002f 0000 0000 6411 3eae 0a00 020f 0a00  ./....d.>.......
        0x0020:  0202 07d0 6403 001b 0000 6120 6d65 7373  ....d.....a.mess
        0x0030:  6167 6520 6672 6f6d 2078 7636 21         age.from.xv6!
15:27:40.862370 ARP, Request who-has 10.0.2.15 tell 10.0.2.2, length 28
        0x0000:  ffff ffff ffff 5255 0a00 0202 0806 0001  ......RU........
        0x0010:  0800 0604 0001 5255 0a00 0202 0a00 0202  ......RU........
        0x0020:  0000 0000 0000 0a00 020f                 ..........
15:27:40.862844 ARP, Reply 10.0.2.15 is-at 52:54:00:12:34:56, length 28
        0x0000:  ffff ffff ffff 5254 0012 3456 0806 0001  ......RT..4V....
        0x0010:  0800 0604 0002 5254 0012 3456 0a00 020f  ......RT..4V....
        0x0020:  5255 0a00 0202 0a00 0202                 RU........
15:27:40.863036 IP 10.0.2.2.25603 > 10.0.2.15.2000: UDP, length 17
        0x0000:  5254 0012 3456 5255 0a00 0202 0800 4500  RT..4VRU......E.
        0x0010:  002d 0000 0000 4011 62b0 0a00 0202 0a00  .-....@.b.......
        0x0020:  020f 6403 07d0 0019 3406 7468 6973 2069  ..d.....4.this.i
        0x0030:  7320 7468 6520 686f 7374 21              s.the.host!
```

Your output will look somewhat different, but it should contain the strings "ARP, Request", "ARP, Reply", "UDP", "a.message.from.xv6" and "this.is.the.host".

`nettests` performs some other tests, culminating in a DNS request sent over the (real) Internet to one of Google's name server. You should ensure that your code passes all these tests, after which you should see this output:

```
$ nettests
nettests running on port 25603
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
```

You should ensure that `make grade` agrees that your solution passes.

Hints

Start by adding print statements to `e1000_transmit()` and `e1000_recv()`, and running `make server` and (in xv6) `nettests`. You should see from your print statements that `nettests` generates a call to `e1000_transmit`.

Some hints for implementing `e1000_transmit`:

- First ask the E1000 for the TX ring index at which it's expecting the next packet, by reading the `E1000_TDT` control register.
- Then check if the the ring is overflowing. If `E1000_TXD_STAT_DD` is not set in the descriptor indexed by `E1000_TDT`, the E1000 hasn't finished the corresponding previous transmission request, so return an error.
- Otherwise, use `mbuffree()` to free the last mbuf that was transmitted from that descriptor (if there was one).

- Then fill in the descriptor. `m->head` points to the packet's content in memory, and `m->len` is the packet length. Set the necessary cmd flags (look at Section 3.3 in the E1000 manual) and stash away a pointer to the mbuf for later freeing.
- Finally, update the ring position by adding one to `E1000_TDT` modulo `TX_RING_SIZE`.
- If `e1000_transmit()` added the mbuf successfully to the ring, return 0. On failure (e.g., there is no descriptor available to transmit the mbuf), return -1 so that the caller knows to free the mbuf.

Some hints for implementing `e1000_recv`:

- First ask the E1000 for the ring index at which the next waiting received packet (if any) is located, by fetching the `E1000_RDT` control register and adding one modulo `RX_RING_SIZE`.
- Then check if a new packet is available by checking for the `E1000_RXD_STAT_DD` bit in the `status` portion of the descriptor. If not, stop.
- Otherwise, update the mbuf's `m->len` to the length reported in the descriptor. Deliver the mbuf to the network stack using `net_rx()`.
- Then allocate a new mbuf using `mbufalloc()` to replace the one just given to `net_rx()`. Program its data pointer (`m->head`) into the descriptor. Clear the descriptor's status bits to zero.
- Finally, update the `E1000_RDT` register to be the index of the last ring descriptor processed.
- `e1000_init()` initializes the RX ring with mbufs, and you'll want to look at how it does that and perhaps borrow code.
- At some point the total number of packets that have ever arrived will exceed the ring size (16); make sure your code can handle that.

You'll need locks to cope with the possibility that xv6 might use the E1000 from more than one process, or might be using the E1000 in a kernel thread when an interrupt arrives.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Submit

You will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type `make handin` to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

`make handin` will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let `make handin` generate it again (*myapi.key* must not include newline characters).

If you run `make handin` and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with `??`. You can cause `git` to track a new file that you create using `git add filename`.

If `make handin` does not work properly, try fixing the problem with the curl or Git commands. Or you can run `make tarball`. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2020/handin.py/

Optional Challenges:

Some of the benefits of the challenge exercises below are only measurable/testable on real, high-performance hardware, which means x86-based computers.

- In this lab, the networking stack uses interrupts to handle ingress packet processing, but not egress packet processing. A more sophisticated strategy would be to queue egress packets in software and only provide a limited number to the NIC at any one time. You can then rely on TX interrupts to refill the transmit ring. Using this technique, it becomes possible to prioritize different types of egress traffic. (easy)
- The provided networking code only partially supports ARP. Implement a full ARP cache and wire it in to `net_tx_eth()`. (moderate)
- The E1000 supports multiple RX and TX rings. Configure the E1000 to provide a ring pair for each core and modify your networking stack to support multiple rings. Doing so has the potential to increase the throughput that your networking stack can support as well as reduce lock contention. (moderate), but difficult to test/measure
- `sockrecvudp()` uses a singly-linked list to find the destination socket, which is inefficient. Try using a hash table and RCU instead to increase performance. (easy), but a serious implementation would difficult to test/measure
- ICMP can provide notifications of failed networking flows. Detect these notifications and propagate them as errors through the socket system call interface.
- The E1000 supports several stateless hardware offloads, including checksum calculation, RSC, and GRO. Use one or more of these offloads to increase the throughput of your networking stack. (moderate), but hard to test/measure
- The networking stack in this lab is susceptible to receive livelock. Using the material in lecture and the reading assignment, devise and implement a solution to fix it. (moderate), but hard to test.
- Implement a UDP server for xv6. (moderate)
- Implement a minimal TCP stack and download a web page. (hard)

If you pursue a challenge problem, whether it is related to networking or not, please let the course staff know!