UNIX-like operating system device driver for Network Interface Card

Name: Bingying Liang

November 21 2023

Contents

1 Abstract

This study delves into the intricacies of Network Interface Card (NIC) drivers within the xv6 operating system, a UNIX-like environment. The research focuses on evaluating the fundamental network functionalities and performance of these drivers. Key components like Ping and DNS resolution tests were successfully executed, confirming the xv6 system's core networking capabilities such as effective connectivity, latency response, and accurate network address resolution. Furthermore, the study extended to both single and multi-process environments, providing a comprehensive analysis of the NIC drivers' efficiency and stability under various network conditions. This evaluation highlighted not only the drivers' capabilities but also areas where optimization is needed, particularly in managing high network traffic and processing concurrent requests. The insights gained lay a strong foundation for future enhancements in NIC drivers, proving invaluable for both academic research and practical applications in computer networking. Overall, this research contributes significantly to the advancement of network driver technology, particularly in UNIX-like systems, guiding future developments towards more robust and efficient network solutions.

2 Introduction

2.1 Background

The pivotal role of Network Interface Cards (NIC) in linking computers to networks and the significance of their device drivers in UNIX-like systems.

Network adapter is a general term that includes all types of devices (such as NIC) and built-in modules used to connect a computer to a network, while network interface card is a form of network adapter, usually referring to a physical wired or wireless network card. Network interface drivers are software that ensure that network adapters (including Nics) are correctly recognized and used by the operating system. These three work together to enable the computer to connect to the network and exchange data. Understanding their functions and interactions is essential for solving network problems and optimizing network performance.

2.1.1 Networks Adapter

Networks are sophisticated systems, to a host, a network is just another I/O device that serves as a source and sink for data, as shown in figure1. A network adapter is a hardware device used to connect a computer or other device to a network. It can be physical hardware, such as a network card, or a chip integrated into the motherboard or mobile device. The network adapter converts the data generated by the computer into network signals and converts the received network signals into data that the computer can understand. Common network adapters include wired Ethernet adapters and wireless Wi-Fi adapters. An adapter that inserts an expansion slot on the I/O bus provides the physical interface to the network(Bryant & O'Hallaron, 2016). Data received from the network is copied from the adapter across the I/O and memory buses into memory, typically transferred by one DMA. Similarly, data can be copied from memory to the network.

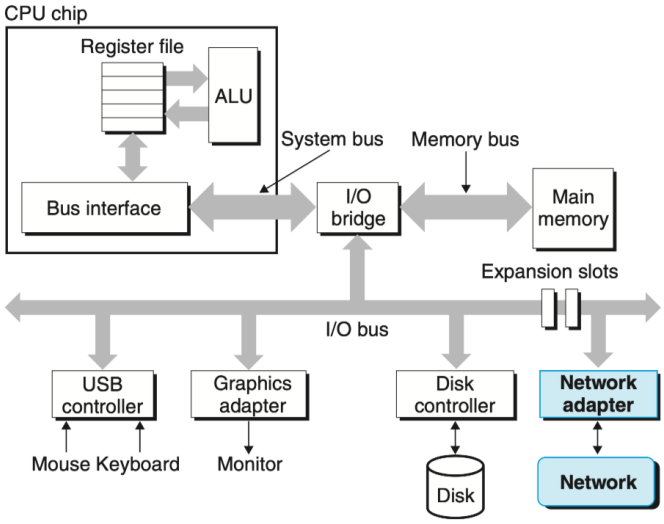

Figure 1: Hardware organization of a network host

2.1.2 Network Interface Cards

The physical layer process and some of the data link layer process run on dedicated hardware called a NIC(Tanenbaum, Feamster, & Wetherall, 2021) (Network Interface Card) show in figure2. The rest of the link layer process and the network layer process run on the main CPU as part of the operating system, with the software for the link layer process often taking the form of a device driver.

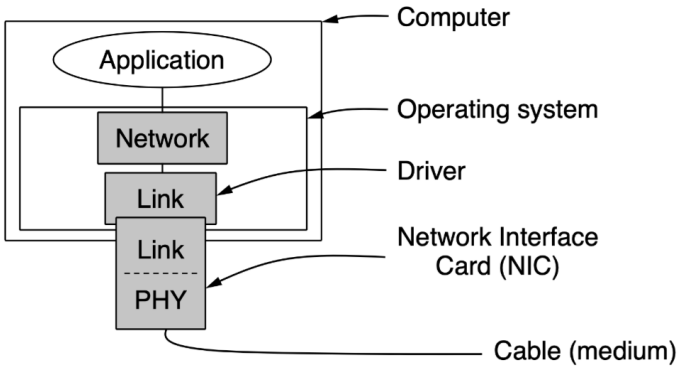

Figure 2: Implementation of the physical, data link, and network layers

2.1.3 Network Interface Driver

A driver is the code in the operating system that manages a specific device: it connotes the device hardware, tells the device to perform actions, handles the resulting interrupts, and interacts with processes that may be waiting for device I/O. Driver code can be tricky because the driver executes concurrently with the devices it manages. In addition, the driver must understand the

hardware interfaces of the device, which can be complex and poorly documented(Cox, F., Morris, & (n.d.), 2022). A network interface driver is a piece of software that manages the communication between an operating system and a network adapter. It is part of the operating system kernel and is responsible for translating the general network commands of the operating system into hardware-specific operation instructions. The driver handles interrupts from the network adapter, manages the sending and receiving of packets, and performs error checking and handling. Without the correct driver, the operating system may not recognize or use the network adapter effectively.

2.2 Objective

The paper aims to unravel the intricate workings of NIC drivers, focusing on their management of layered network protocols and their integration with the broader system architecture. It sets out to provide a clear understanding of the drivers' functionalities and their role in network communication within UNIX-like environments.

3 Literature review

3.1 UNIX-like Systems and Device Drivers

In the realm of computing, UNIX-like systems, as extensively documented in the technical reports published by Bell Labs staff(Johnson & Kernighan, 1977), are known for their distinct core characteristics. These systems are designed for multitasking and multiuser environments, enabling simultaneous use of system resources by multiple users. A fundamental aspect of UNIX-like systems is their hierarchical file system structure, adhering to the philosophy of "everything is a file." This approach treats all devices and data uniformly as files, simplifying interaction with various system components. Additionally, these systems provide a Shell environment, a command-line interface that allows users to interact directly with the operating system. Another critical feature of UNIX-like systems is their emphasis on permissions and security, with complex file permission settings that play a crucial role in ensuring the system's security and stability.

Device drivers in UNIX-like systems are integral, acting as the intermediary between hardware components and the operating system kernel. These drivers are essential for the smooth functioning of the system and are generally classified into several categories. Character Device Drivers are responsible for handling character stream devices like keyboards and mice, translating user actions into inputs that the system can process. Block Device Drivers manage block devices such as hard drives and CD-ROMs, facilitating data storage and retrieval operations. Lastly, Network Device Drivers are specialized for managing hardware that deals with network communication, including Network Interface Cards (NICs). These drivers ensure that data can be efficiently transmitted and received over network connections, playing a pivotal role in the system's networking capabilities.

3.2 Network Protocols and Packet Formats

In the context of UNIX-like systems, the network protocols and their associated packet format is crucial for developing and maintaining robust device drivers, especially for network interface cards (NICs).

Ethernet Protocol(Sommer et al., 2010) is the most widely used LAN technology. It defines wiring and signaling for the physical layer of the TCP/IP model. The packet Structure of An Ethernet packet starts with a preamble, followed by the destination MAC address, source MAC address, EtherType (indicating the protocol), payload (data), and a frame check sequence (FCS) for error checking, which shows in 3.

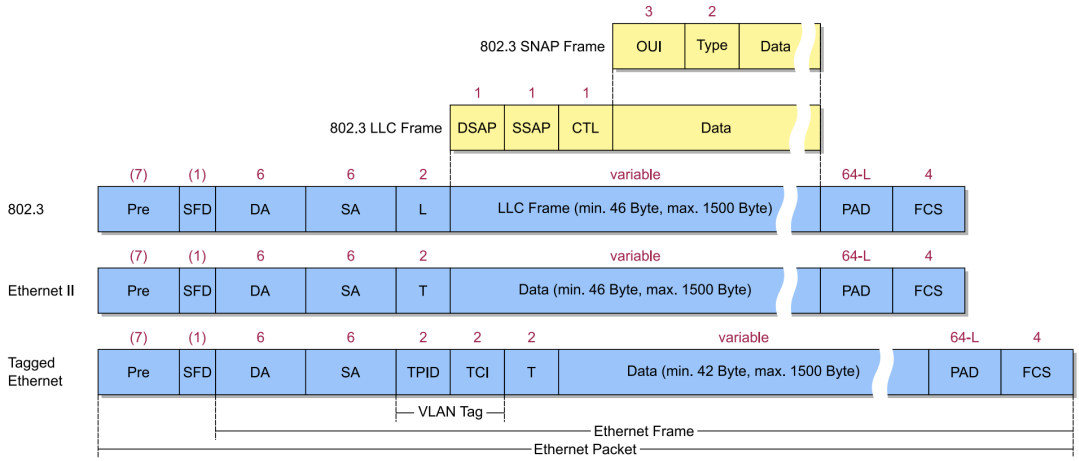

Figure 3: Ethernet IEEE 802.3 frame formats

Internet Protocol (IP) (Shiranzaei & Khan, 2015) is used for sending packets of data across networks. It facilitates the routing of these packets through different nodes to reach the destination. The structure of the IP Packet includes a header (with source and destination IP addresses, version, length, and other control information) and payload.

User Datagram Protocol (UDP) (Le et al., 2009) is a connectionless protocol that offers a direct way to send and receive datagrams over an IP network. The format of the UDP Packet consists of a source port, destination port, length, and checksum, followed by the data. It's known for its low latency and overhead but lacks reliability features like retransmission.

There are a lot of protocols in the network in this big world. These protocols and packet formats are standardized, ensuring interoperability across different systems and devices. These formats is crucial for NIC driver developers, as the driver needs to handle these packets, often modifying or inspecting headers and payloads for efficient data transmission and network communication.

3.3 Network Receive Livelock Challenges

In the context of network communications, especially within UNIX-like systems, the concept of a network receive livelock is a critical issue that needs to be addressed for efficient network operation.

A network receive livelock(Mogul & Ramakrishnan, 1997) occurs when the system spends all its resources continuously processing network packets but makes no progress in processing the actual data contained in those packets. This often occurs in high-throughput network environments, where the rate of incoming packets is so high that the system is overwhelmed with only receiving and processing these packets, leaving no resources for other tasks. In addition, most operating

systems use interface interrupts to schedule network tasks. Interrupt-driven systems can provide low overhead and good latency at low load, but degrade significantly at higher arrival rates unless care is taken to prevent several pathological phenomena. This is a form of receive liftlock in which the system spends all its time handling interrupts to the exclusion of other necessary tasks. In the extreme case, no packet is sent to the user application or the output of the system.

This has a huge impact on Unix-like systems, such as system performance: In Unix-like systems, which are commonly used in high network traffic server environments, livelocks can significantly degrade system performance and responsiveness(Cox et al., 2022). It is also a challenge at the kernel level because device drivers (including network card drivers) operate at the kernel level4, so livelocks can cause serious problems with the kernel's ability to manage network traffic efficiently.

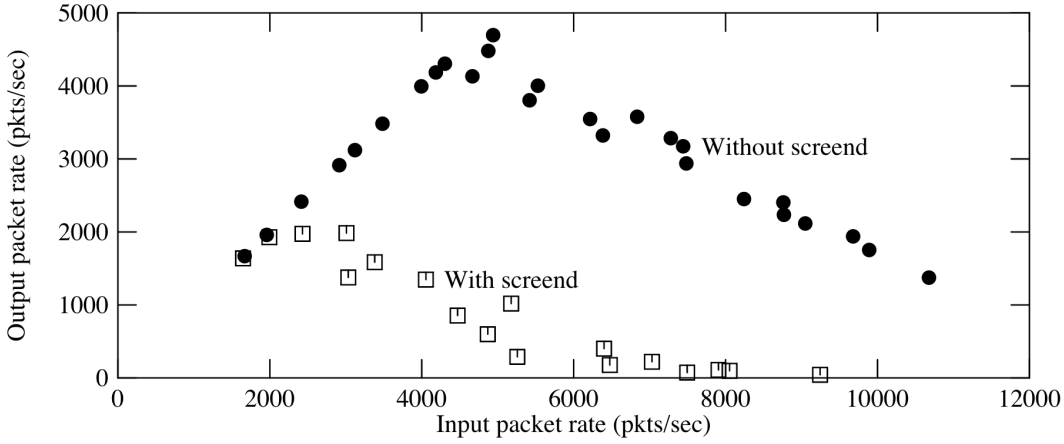

Figure 4: Forwarding performance of unmodified kernel

Modern Unix-like systems have adopted multiple strategies that use an interrupt-driven system proposed by Mogul and Ramakrishnan et al.(Mogul & Ramakrishnan, 1997) to alleviate livelock and make them more robust in handling network traffic. Network receive livelocks pose a significant challenge in network communication in high throughput environments. Understanding and solving this problem is essential for the development of efficient and reliable network device drivers in UniX-like systems, ensuring stable and high-performance network operations.

4 Methodology

This section will focus on analyzing the behavior of Network Interface Card (NIC) drivers within a UNIX-like environment. I use the xv6 operating system(Cox et al., 2022) to analyze this part.

4.1 Experimental Setup

Used the QEMU Emulator(*Documentation/Networking—QEMU*, n.d.), the purpose is to create a controlled, simulated UNIX-like environment for testing. Set up QEMU to emulate specific

network conditions and hardware interfaces, including the emulation of NICs. Setting up QEMU to emulate specific network conditions and hardware interfaces, including the emulation of NICs. The E1000(“8254x Family of Gigabit Ethernet Controllers Software Developer’s Manual”, n.d.) NIC simulation is also used because it is widely used and well documented, making it a suitable choice for simulation. The emulator is configured to simulate the E1000 NIC, allowing realistic interaction between the NIC driver and the emulated hardware.

4.2 Analysis Techniques

In xv6 kernel, kernel/E1000.c contains initialization code for the E1000, which has some functions for transmitting and receiving packets. And in kernerl/e1000_dev.h contains definitions for registers and flag bits defined by the E1000 and described in the Intel E1000(“8254x Family of Gigabit Ethernet Controllers Software Developer’s Manual”, n.d.). 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. And kernel/pci.c contains code that searches for an E1000 card on the PCI bus when xv6 boots(*Linux/drivers/net/ethernet/intel/e1000*, n.d.).

4.2.1 E1000 interaction methods

The E1000 uses direct memory access (DMA) technology to write received packets directly into the computer’s memory, which is very useful when data is large and can be used as a cache. It is also possible to write the descriptor (see 5) (“8254x Family of Gigabit Ethernet Controllers Software Developer’s Manual”, n.d.) to a specific location in memory, so that the E1000 itself finds the data to send and sends it. Packets, whether received or sent, are described by an array of descriptors. In the receive and send sections below, we’ll look at the format of the receive and send descriptors, respectively.

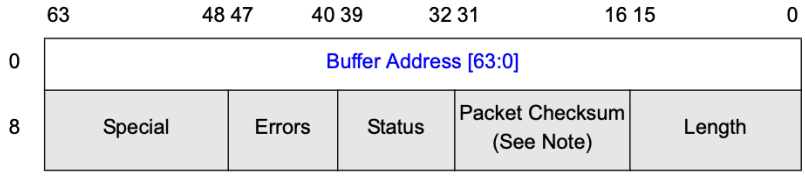

Figure 5: Receive Descriptor (RDESC) Layout

In the xv6 kernel, kernel/e1000_dev.h shows the definitions [E1000 3.3.3] in the following:

```
1 /* Receive Descriptor bit definitions [E1000 3.2.3.1] */
2 #define E1000_RXD_STAT_DD       0x01    /* Descriptor Done */
3 #define E1000_RXD_STAT_EOP      0x02    /* End of Packet */
4
5 // [E1000 3.2.3]
6 struct rx_desc
```

```
{
  uint64 addr;       /* Address of the descriptor's data buffer */
  uint16 length;     /* Length of data DMAed into data buffer */
  uint16 csum;       /* Packet checksum */
  uint8 status;      /* Descriptor status */
  uint8 errors;      /* Descriptor Errors */
  uint16 special;
};
```

Putting an array descriptor in memory, and that array is interpreted as a circular queue. If a new packet is received by the network card, the descriptor at the head position of the ring queue is checked. The data is then written to the buffer of the head descriptor, which is the address of the addr record. Also important here are the status and length attributes. The network card sets these properties when it writes. Here, length represents the length of the packet written to addr. status can represent the following states figure6:

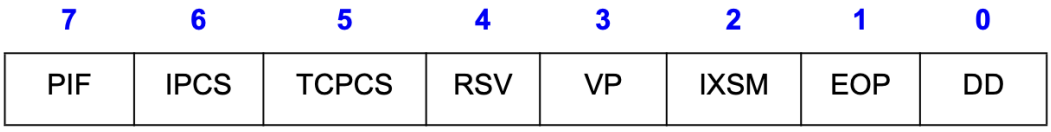

Figure 6: Receive Status (RDESC.STATUS) Layout

DD (Descriptor Done) is the flag. It indicates that the network card has received the packet

4.2.2 Receiving

If the network card receives the data, an interrupt is generated and the corresponding interrupt handler is called to process the newly arrived data. When the network card receives new data, it writes to the buffer at the head descriptor of the ring queue; let's discuss how the card and driver manage this buffer in detail. The structure of the receiving descriptor ring queue show in figure7

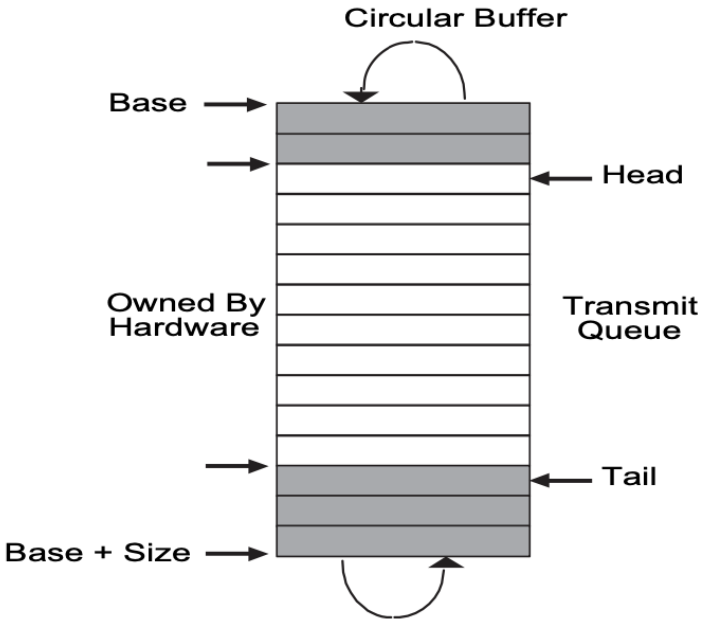

Figure 7: Receive Descriptor Ring Structure

When initialized, head is 0 and tail is the queue buffer minus one. The light-colored region from head to tail is free. All the packets in this area have already been processed by the software, so if a new packet arrives, the network card will write the data to the beginning of the area, which is the head, and overwrite the old data. The network card increments the value of head when it overwrites the old data. The software processes the dark areas in order. When the ring queue is read, the descriptor buffer at tail + 1 is read, which position has the longest waiting time of all unprocessed data), and tail is incremented by one when the buffer is processed.

To facilitate the processing of network data, xv6 also defines a struct, struct mbuf, as follows:

4.2.3 Sending

The format of the send descriptor is as follows figure9:

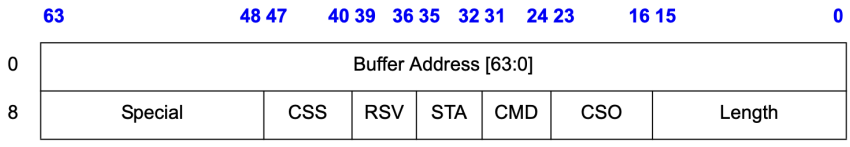

Figure 8: Transmit Descriptor (TDESC) Layout – Legacy Mode

In the xv6 kernel, kernel/e1000_dev.h shows the definitions [E1000 3.3.3] in the following:

```
1 /* Transmit Descriptor command definitions [E1000 3.3.3.1] */
2 #define E1000_TXD_CMD_EOP    0x01 /* End of Packet */
```

```
#define E1000_TXD_CMD_RS     0x08 /* Report Status */

/* Transmit Descriptor status definitions [E1000 3.3.3.2] */
#define E1000_TXD_STAT_DD    0x00000001 /* Descriptor Done */

// [E1000 3.3.3]
struct tx_desc
{
  uint64 addr;
  uint16 length;
  uint8 cso;         // checksum offset
  uint8 cmd;         // command field
  uint8 status;      //
  uint8 CSS;         // checksum start field
  uint16 special;    //
};
```

The addr and length properties do the same thing as the receive descriptor, except that the cmd and status properties are used. As with the receive flag, the DD flag is used to indicate whether the data pointed to by the status flag has been sent. cmd describes some Settings, or commands to the network card, when transmitting this packet. The command shows in the following:

7	6	5	4	3	2	1	0
IDE	VLE	DEXT	RSV RPS[a]	RS	IC	IFCS	EOP

Figure 9: Transmit Command (TDESC.CMD) Layout

For example, RPS (Report Packet Sent). When set, the network card reports the status of the packet sent. For example, the DD bit of the descriptor is set by the network card after the data pointed to by the descriptor has been sent; EOP (End of Packet) indicates that this descriptor is the end of the packet. If the packet to be sent is particularly large then may use the cache space of many descriptors to store a single packet. Then can set the EOP command to the last descriptor of this packet. Only then can we add additional functionality to this descriptor, such as IC, which stands for join and check.

Slightly different from the ring queue figure10 that receives descriptors, the head-to-tail area of the sending descriptor (the light area on the way) represents data that would like to be sent, but that the network card has not yet been sent.

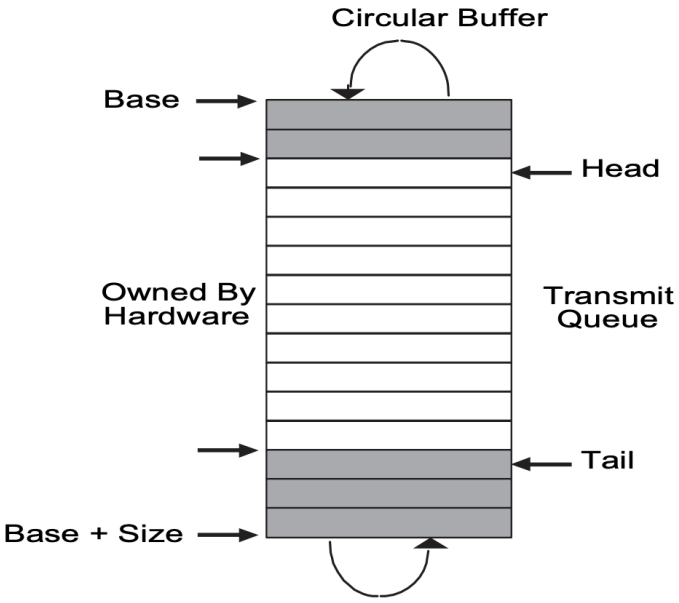

Figure 10: Transmit Descriptor Ring Structure

Where the head points to the data to be sent that has been waiting the longest and from which the network card starts to send. And when we do that, we'll increment the tail by one, and if want to add a new descriptor, from the direction of the tail, increment the tail by one.

4.2.4 Xv6 description of network data

To facilitate the processing of network data, xv6 also defines a struct, struct mbuf in kernel/net.h as follows:

```
1  //
2  // packet buffer management
3  //
4
5  #define MBUF_SIZE              2048
6  #define MBUF_DEFAULT_HEADROOM  128
7
8  struct mbuf {
9    struct mbuf  *next; // the next mbuf in the chain
10   char         *head; // the current start position of the buffer
11   unsigned int len;   // the length of the buffer
12   char         buf[MBUF_SIZE]; // the backing store
13 };
14
15 char *mbufpull(struct mbuf *m, unsigned int len);
16 char *mbufpush(struct mbuf *m, unsigned int len);
```

```
char *mbufput(struct mbuf *m, unsigned int len);
char *mbuftrim(struct mbuf *m, unsigned int len);

// The above functions manipulate the size and position of the buffer:
//            <- push            <- trim
//             -> pull            -> put
// [-headroom-][------buffer------][-tailroom-]
// |----------------MBUF_SIZE-----------------|
//
// These marcos automatically typecast and determine the size of header structs.
// In most situations you should use these instead of the raw ops above.
#define mbufpullhdr(mbuf, hdr) (typeof(hdr)*)mbufpull(mbuf, sizeof(hdr))
#define mbufpushhdr(mbuf, hdr) (typeof(hdr)*)mbufpush(mbuf, sizeof(hdr))
#define mbufputhdr(mbuf, hdr) (typeof(hdr)*)mbufput(mbuf, sizeof(hdr))
#define mbuftrimhdr(mbuf, hdr) (typeof(hdr)*)mbuftrim(mbuf, sizeof(hdr))
```

In kernel/e1000.c, there is a function called e1000_transmit() that receives a mbuf network data, writes it to the DMA memory address, and tells the network card to send the data.

The headroom can be pushed into it to store network protocol headers. After receiving network data, we can also pull the middle buffer into it to convert it into the following header kernel/net.h:

```
// an Ethernet packet header (start of the packet).
struct eth {
  uint8  dhost[ETHADDR_LEN];
  uint8  shost[ETHADDR_LEN];
  uint16 type;
} __attribute__((packed));
```

and the conversion part can be used with the net_rx() function in kernel/net.c

```
// called by e1000 driver's interrupt handler to deliver a packet to the
// networking stack
void net_rx(struct mbuf *m)
{
  struct eth *ethhdr;
  uint16 type;

  ethhdr = mbufpullhdr(m, *ethhdr);
  if (!ethhdr) {
    mbuffree(m);
    return;
  }

  type = ntohs(ethhdr->type);
```

```
  if (type == ETHTYPE_IP)
    net_rx_ip(m);
  else if (type == ETHTYPE_ARP)
    net_rx_arp(m);
  else
    mbuffree(m);
}
```

While the buffer part is the body of the data, the rest of the tailroom is char buf[MBUF_SIZE] This cache is the rest of the first two parts removed. In struct mbuf, len is the length of the body and head is the end of headroom. There are a number of mbuf-related functions in net.c, the main ones being mbufalloc() and mbuffree() for mbuf allocation and release, respectively.

4.2.5 Register

The control registers of the E1000 are accessible through specific memory mappings. Specifically, by adding some offset to the regs global variable in kernel/e1000.c. These offsets are defined in kernel/e1000_dev.h.

```
/* Registers */
#define E1000_CTL      (0x00000/4)  /* Device Control Register - RW */
#define E1000_ICR      (0x000C0/4)  /* Interrupt Cause Read - R */
#define E1000_IMS      (0x000D0/4)  /* Interrupt Mask Set - RW */
#define E1000_RCTL     (0x00100/4)  /* RX Control - RW */
#define E1000_TCTL     (0x00400/4)  /* TX Control - RW */
#define E1000_TIPG     (0x00410/4)  /* TX Inter-packet gap -RW */
#define E1000_RDBAL    (0x02800/4)  /* RX Descriptor Base Address Low - RW */
#define E1000_RDTR     (0x02820/4)  /* RX Delay Timer */
#define E1000_RADV     (0x0282C/4)  /* RX Interrupt Absolute Delay Timer */
#define E1000_RDH      (0x02810/4)  /* RX Descriptor Head - RW */
#define E1000_RDT      (0x02818/4)  /* RX Descriptor Tail - RW */
#define E1000_RDLEN    (0x02808/4)  /* RX Descriptor Length - RW */
#define E1000_RSRPD    (0x02C00/4)  /* RX Small Packet Detect Interrupt */
#define E1000_TDBAL    (0x03800/4)  /* TX Descriptor Base Address Low - RW */
#define E1000_TDLEN    (0x03808/4)  /* TX Descriptor Length - RW */
#define E1000_TDH      (0x03810/4)  /* TX Descriptor Head - RW */
#define E1000_TDT      (0x03818/4)  /* TX Descripotr Tail - RW */
#define E1000_MTA      (0x05200/4)  /* Multicast Table Array - RW Array */
#define E1000_RA       (0x05400/4)  /* Receive Address - RW Array */
```

4.3 Implementation e1000_transmit and e1000_recv

Synthesizing the above analysis, the overall framework is as follows

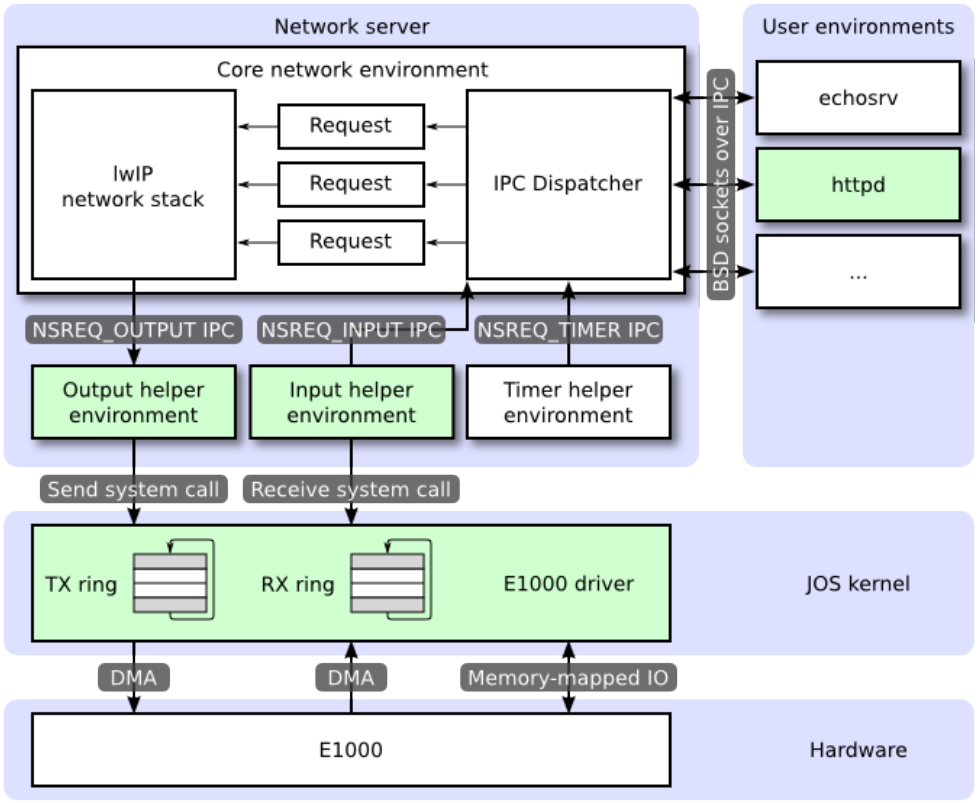

Figure 11: The entire system including the device driver

The main logic is in the following:

1. Send call chain:
   $sys\_write() \rightarrow filewrite() \rightarrow sockwrite() \rightarrow net\_tx\_udp() \rightarrow net\_tx\_ip() \rightarrow net\_tx\_eth() \rightarrow e1000\_transmit()$

2. Receive call chain:
   $e1000\_intr() \rightarrow e1000\_recv() \rightarrow net\_rx() \rightarrow net\_rx\_ip() \rightarrow net\_rx\_udp() \rightarrow sockrecvudp() \rightarrow mbufq\_pushtail()$

The implementation of kernel/e1000.c e1000_transmit and e1000_recv function is

```
1  int
2  e1000_transmit(struct mbuf *m)
3  {
4    //
5    //
6    // the mbuf contains an ethernet frame; program it into
7    // the TX descriptor ring so that the e1000 sends it. Stash
8    // a pointer so that it can be freed after sending.
9    //
```

```
10
11      acquire(&e1000_lock);
12      int idx = regs[E1000_TDT];
13
14      if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0) {
15          release(&e1000_lock);
16          return -1;
17      }
18
19      if (tx_mbufs[idx])
20          mbuffree(tx_mbufs[idx]);
21
22      tx_mbufs[idx] = m;
23      tx_ring[idx].length = m->len;
24      tx_ring[idx].addr = (uint64) m->head;
25      tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
26      regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;
27
28      release(&e1000_lock);
29
30    return 0;
31  }
32
33  static void
34  e1000_recv(void)
35  {
36    //
37    // Code here.
38    //
39    // Check for packets that have arrived from the e1000
40    // Create and deliver an mbuf for each packet (using net_rx()).
41    //
42      while (1) {
43          int idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
44          if ((rx_ring[idx].status & E1000_RXD_STAT_DD) == 0) {
45              return;
46          }
47          rx_mbufs[idx]->len = rx_ring[idx].length;
48          net_rx(rx_mbufs[idx]);
49          rx_mbufs[idx] = mbufalloc(0);
50          rx_ring[idx].status = 0;
51          rx_ring[idx].addr = (uint64)rx_mbufs[idx]->head;
52          regs[E1000_RDT] = idx;
53      }
```

```
}
```

5 Evaluation

MIT xv6(“xv6-public”, 2023) provides some test scripts to verify the basic network functions of the network interface Card (NIC) driver. The test script code is as follows:

```
#!/usr/bin/env python3

import re
import subprocess
from gradelib import *

r = Runner(save("xv6.out"))

@test(0, "running nettests")
def test_nettest():
    server = subprocess.Popen(["make", "server"],
        stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    r.run_qemu(shell_script([
        'nettests'
    ]), timeout=30)
    server.terminate()
    server.communicate()

@test(40, "nettest: ping", parent=test_nettest)
def test_nettest_():
    r.match('^testing ping: OK$')

@test(20, "nettest: single process", parent=test_nettest)
def test_nettest_():
    r.match('^testing single-process pings: OK$')

@test(20, "nettest: multi-process", parent=test_nettest)
def test_nettest_fork_test():
    r.match('^testing multi-process pings: OK$')

@test(19, "nettest: DNS", parent=test_nettest)
def test_nettest_dns_test():
    r.match('^DNS OK$')

#@test(10, "answers-net.txt")
#def test_answers():
```

```
#     # just a simple sanity check, will be graded manually
#     check_answers("answers-net.txt")

@test(1, "time")
def test_time():
    check_time()

run_tests()
```

For the test result is in the following figure12 figure13:

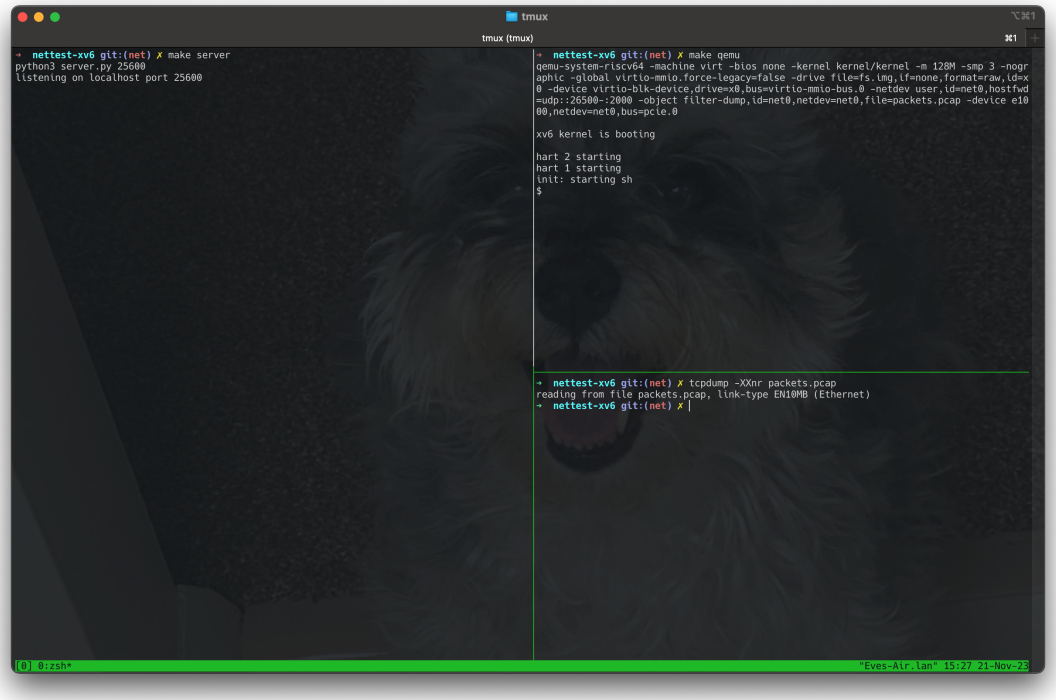

Figure 12: Initization

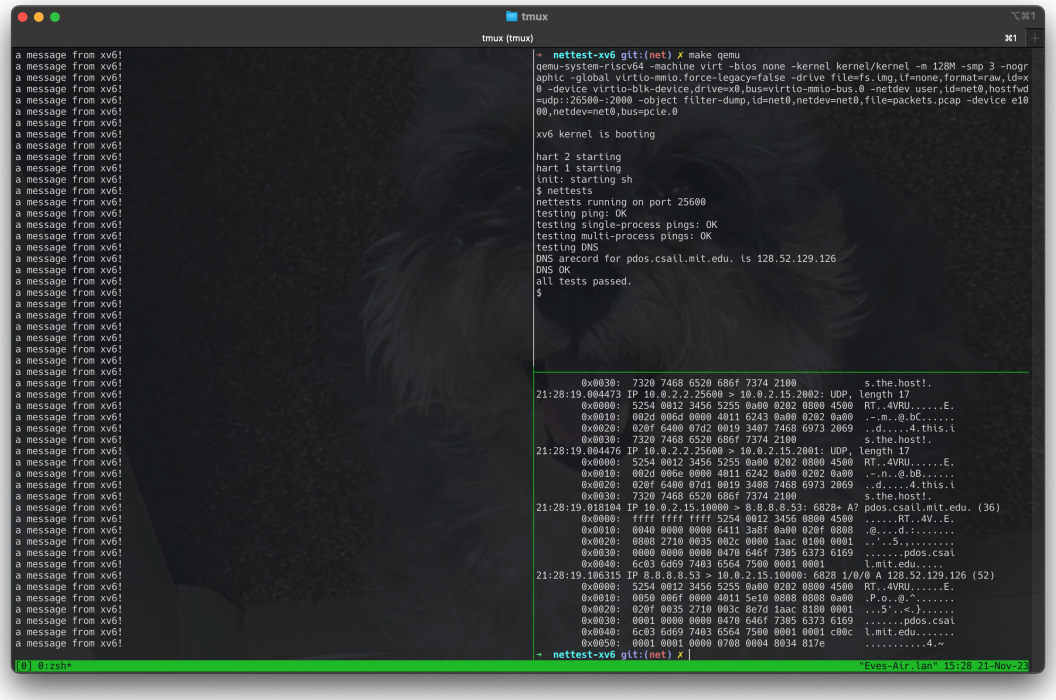

Figure 13: Test Result

The initial phase involved conducting Ping tests using the test_nettest_ function to verify basic network connectivity and latency response. The key aspect of this test was to confirm the effectiveness and reliability of the system in executing fundamental network communication tasks. Proper functioning of the Ping feature was verified by matching the script's output with "testing ping: OK." This step is crucial for ensuring the basic health and responsiveness of the system's network stack.

Subsequently, an in-depth test of the network communication capabilities in a single-process environment was conducted. Through the test_nettest_ function, the stability and efficiency of the driver in handling network requests without involving inter-process communication were evaluated. This test aimed to verify the performance of the driver in a single-task environment, serving as an important check on its simplicity and effectiveness.

Furthermore, to comprehensively assess the driver's performance, multi-process network communication tests were also conducted. Utilizing the test_nettest_fork_test function allowed for testing and evaluating the driver's performance in handling multiple concurrent network requests. This test is significant as it simulates the complexity and diversity of real-world network traffic, providing a thorough examination of the driver's capabilities.

Finally, tests on the DNS resolution functionality were performed using the test_nettest_dns_test function to ensure the system's accuracy in processing and resolving domain names in network requests. DNS resolution is a basic yet critical function in network communication, and this test ensured the system's precision and efficiency in resolving network addresses.

Through this series of tests, the capability of the NIC drivers in the xv6 operating system to handle basic network functionalities was verified, and in-depth insights into their performance

under various network conditions were gained. These test results provide essential baseline data and insights for future development and optimization of the system, constituting a key step in ensuring the stability and efficiency of network communications.

6 Contribution

This series of tests on the Network Interface Card (NIC) drivers in the xv6 operating system has made substantial contributions to understanding network functionalities in UNIX-like systems. Key achievements include verifying fundamental network capabilities like connectivity and latency response through successful Ping and DNS resolution tests. The evaluation of NIC driver performance in both single and multi-process environments revealed its efficiency and stability across various network conditions, highlighting areas for potential optimization, such as handling high network traffic and concurrent requests. These insights provide a solid foundation for future improvements in NIC driver technology, enhancing both academic research and practical applications in computer networking. Overall, this study marks a significant step forward in advancing network driver technology for more robust and efficient network solutions in UNIX-like environments.

7 Acknowledgement

The study of network card driver in xv6 operating system has been strongly supported by various individuals and institutions. Heartfelt thanks to the academic tutors, whose expertise was invaluable, and to the xv6 development community for maintaining an educational and open platform for this study. Important support and feedback from technical staff and peers are gratefully acknowledged. In addition, contributions from the wider open source community, especially those involved in tools such as QEMU, have played a crucial role in facilitating this research. Finally, the importance of academic research in computer science and network technology is emphasized by thanking funding agencies and institutions for providing the necessary resources. The combined efforts of all these contributors are integral to advancing knowledge in the field of network driver technology.

8 Summary and conclusions

This study on the xv6 operating system's Network Interface Card (NIC) drivers highlighted their effective basic network functionality, as demonstrated by successful Ping and DNS resolution tests. These tests confirmed key capabilities such as connectivity and accurate address resolution. The evaluation in different processing environments revealed the drivers' efficiency and stability, indicating their suitability for real-world applications. However, it also identified potential areas for improvement, particularly in handling high network traffic and concurrent requests. These findings provide a foundation for future enhancements in NIC driver technology for UNIX-like systems. Overall, the study contributes significantly to understanding NIC driver interactions with network protocols and paves the way for advancements in network communication efficiency and reliability.

References

8254x family of gigabit ethernet controllers software developer's manual [Computer software manual]. (n.d.). Retrieved from the official Intel website. Retrieved from https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/ethernet/8254x-family-gigabit-ethernet-controllers-software-developers-manual.html

Bryant, R. E., & O'Hallaron, D. R. (2016). *Computer systems: A programmer's perspective (third edition)*. Pearson.

Cox, K., R., F., Morris, ., & (n.d.), R. (2022). *xv6: A simple, unix-like teaching operating system.*

Documentation/networking—qemu. (n.d.). Retrieved 20 November 2023. Retrieved from https://wiki.qemu.org/Documentation/Networking#User_Networking_.28SLIRP.29

Johnson, S. C., & Kernighan, B. W. (1977, Jan). *The programming language b* (Computer Science Tech. Rep. No. 8). Murray Hill, NJ, USA: Bell Laboratories. Retrieved from http://web.archive.org/web/20180831015050/https://www.bell-labs.com/usr/dmr/www/bintro.html

Le, T., Kuthethoor, G., Hansupichon, C., Sesha, P., Strohm, J., Hadynski, G., ... Parker, D. (2009). Reliable user datagram protocol for airborne network. In *Milcom 2009 - 2009 ieee military communications conference* (p. 1-6). doi: 10.1109/MILCOM.2009.5380007

Linux/drivers/net/ethernet/intel/e1000. (n.d.). Retrieved 20 November 2023. Retrieved from https://github.com/torvalds/linux/tree/master/drivers/net/ethernet/intel/e1000

Mogul, J. C., & Ramakrishnan, K. K. (1997). Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, *15*(3), 217–252. doi: 10.1145/263326.263335

Shiranzaei, A., & Khan, R. Z. (2015). Internet protocol versions — a review. In *2015 2nd international conference on computing for sustainable global development (indiacom)* (p. 397-401).

Sommer, J., Gunreben, S., Feller, F., Kohn, M., Mifdaoui, A., Sass, D., & Scharf, J. (2010). Ethernet – a survey on its fields of application. *IEEE Communications Surveys & Tutorials*, *12*(2), 263–284. Retrieved from https://doi.org/10.1109/SURV.2010.021110.00086 doi: 10.1109/SURV.2010.021110.00086

Tanenbaum, A. S., Feamster, N., & Wetherall, D. (2021). *Computer networks (sixth edition, global edition)*. Pearson.

xv6-public [Computer software manual]. (2023). [C]. Original work published 2015. Retrieved from GitHub: https://github.com/mit-pdos/xv6-public.