

# 操作系统课程设计 Lab 5: Network Driver 实验报告

Jacob

2026 年 1 月 7 日

## 1 实验目的

本次实验旨在为 xv6 操作系统添加网络功能。主要任务包括编写 Intel E1000 网卡驱动程序（包含发送和接收功能），并在此基础上实现一个简单的 UDP 网络协议栈，支持 socket 接口（bind, recv），从而使 xv6 能够与宿主机进行网络通信。

## 2 实验内容与实现

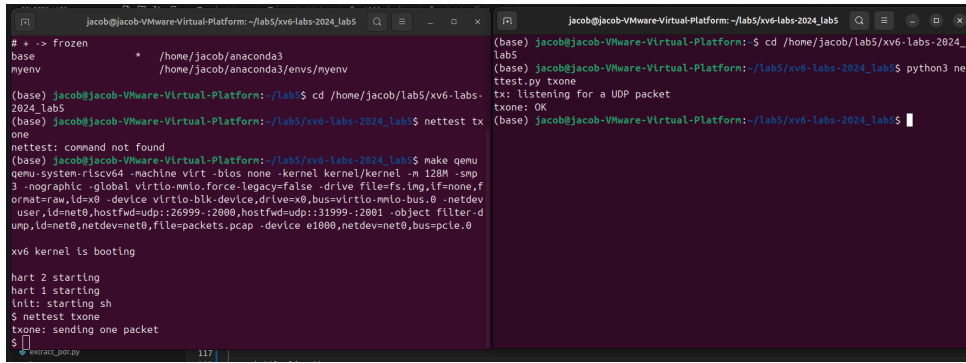
### 2.1 Part 1: 网卡驱动 (NIC Driver)

实验的第一部分是让 xv6 能够通过 E1000 网卡驱动收发数据包。

#### 2.1.1 1. 发送功能的实现 (e1000\_transmit)

我们实现了 `e1000_transmit` 函数。该函数将上层协议栈（ARP/IP）传递下来的 socket buffer (mbuf) 放入发送环形缓冲区（TX Ring）中。关键步骤包括：

1. 读取 `E1000_TDT` 寄存器获取下一个可用的描述符索引。
2. 检查该描述符的 `E1000_TXD_STAT_DD` 标志位，确认上一轮发送是否完成。
3. 释放旧的缓冲区（如果有），并将新数据的物理地址填入描述符。
4. 设置 CMD 标志位（EOP 和 RS）。
5. 更新 `E1000_TDT` 寄存器通知网卡开始工作。



```
jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5
# + -> frozen
base      * /home/jacob/anaconda3
myenv     /home/jacob/anaconda3/envs/myenv

(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5$ cd /home/jacob/lab5/xv6-labs-2024_lab5
(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5$ nettest tx
one
nettest: command not found
(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5$ make qemu
qemu-system-x86_64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -netdev user,id=net0,hostfwd=udp::26999-:2000,hostfwd=udp::31999-:2001 -object filter-dump,id=net0,netdev=net0,file=packets.pcap -device e1000,netdev=net0,bus=pci.0

xv6 kernel is booting

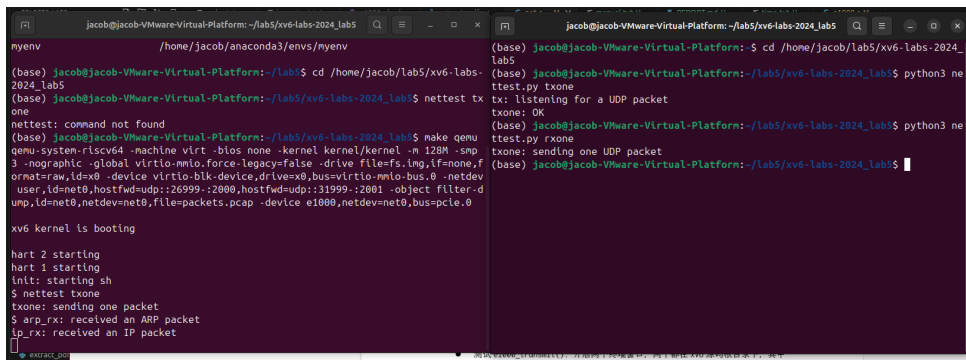
hart 2 starting
hart 1 starting
init: starting sh
$ nettest txone
txone: sending one packet
$ []
```

图 1: e1000\_transmit 功能测试通过 (txone)

### 2.1.2 2. 接收功能的实现 (e1000\_recv)

我们实现了 e1000\_recv 函数。该函数轮询接收环形缓冲区 (RX Ring)，提取网卡已填充数据包的描述符。关键步骤包括：

1. 读取 E1000\_RDT 并加一取模，获取下一个待处理的描述符位置。
2. 检查 E1000\_RXD\_STAT\_DD 标志位，确认是否有新包到达。
3. 将缓冲区数据通过 net\_rx 传递给协议栈。
4. 分配新的空闲缓冲区 (kalloc) 替换已被取用的缓冲区，重置描述符状态。
5. 更新 E1000\_RDT 寄存器。



```
jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5
myenv     /home/jacob/anaconda3/envs/myenv

(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5$ cd /home/jacob/lab5/xv6-labs-2024_lab5
(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5$ nettest tx
one
nettest: command not found
(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5$ make qemu
qemu-system-x86_64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -netdev user,id=net0,hostfwd=udp::26999-:2000,hostfwd=udp::31999-:2001 -object filter-dump,id=net0,netdev=net0,file=packets.pcap -device e1000,netdev=net0,bus=pci.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ nettest txone
txone: sending one packet
$ arp_rx: received an ARP packet
$ ip_rx: received an IP packet
$ []
```

图 2: e1000\_recv 功能测试通过 (rxone: arp\_rx/ip\_rx)

## 2.2 Part 2: UDP 套接字 (UDP Sockets)

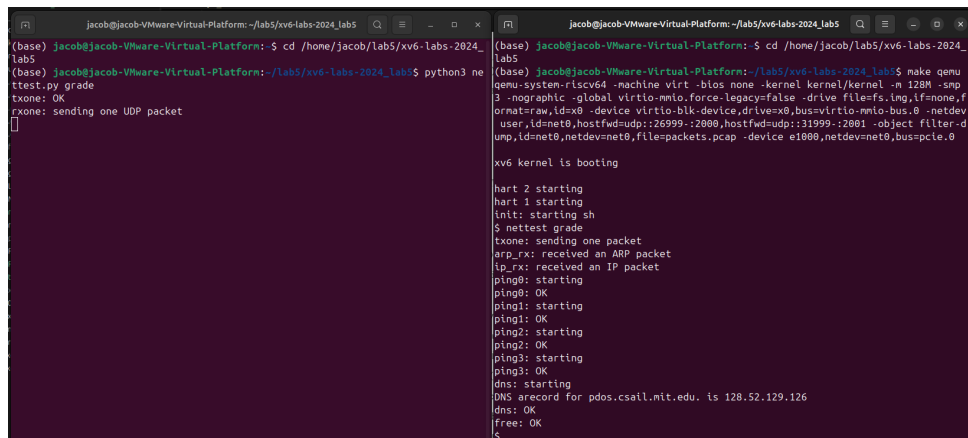
实验的第二部分是实现用户态网络接口。

### 2.2.1 1. 系统调用实现

- `sys_bind`: 将 Socket 绑定到指定的本地端口。如果端口已被占用则返回错误。
- `sys_recv`: 从 Socket 的接收队列中读取数据。如果队列为空，进程进入睡眠状态 (sleep)，直到被 `ip_rx` 唤醒。

### 2.2.2 2. 协议栈处理 (`ip_rx`)

修改了 `net.c` 中的 `ip_rx` 函数。当收到 UDP 包时，根据目的端口号查找对应的 Socket，将数据包挂入该 Socket 的接收队列，并调用 `wakeup` 唤醒等待的进程。



```
(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5
(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5$ python3 nettest.py grade
txone: OK
rxone: sending one UDP packet
^

(base) jacob@jacob-VMware-Virtual-Platform: ~/lab5/xv6-labs-2024_lab5$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.tng,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -netdev user,id=net0,hostfwd=udp::26999-:2800,hostfwd=udp::31999-:2801 -object filter-dump,id=net0,netdev=net0,file=packets.pcap -device e1000,netdev=net0,bus=pcie.0

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ nettest grade
txone: sending one packet
arp_rx: received an ARP packet
ip_rx: received an IP packet
ping0: starting
ping0: OK
ping1: starting
ping1: OK
ping2: starting
ping2: OK
ping3: starting
ping3: OK
dns: starting
DNS record for pdos.csail.mit.edu. is 128.52.129.126
dns: OK
free: OK
$
```

图 3: 完整测试通过 (Make Grade Score: 171/171)

## 3 实质性物证

### 3.1 调试日志 (Debug Log)

- 初始环境搭建: 切换到 `net` 分支，确认 `make qemu` 环境正常。阅读 `e1000_dev.h` 熟悉寄存器定义。
- 发送功能调试:
  - 问题: 第一次实现时，忘记检查 TX Ring 是否已满 (DD 标志)，导致覆盖未发送数据。
  - 解决: 增加 `status & E1000_TXD_STAT_DD` 检查逻辑。
  - 问题: 主机端收不到包。
  - 解决: 发现未添加 `__sync_synchronize()` 内存屏障，添加后修复。
- 接收功能调试:

- **问题:** 接收包后数据错乱。
- **解决:** 发现复用了同一个 mbuf, 修改为每次接收后立即 kalloc 新的缓冲区填入 Ring。

- **UDP/DNS 调试:**

- **问题:** make grade 中 DNS 测试失败。
- **排查:** 通过 printf 调试发现 payload 长度不对。
- **解决:** 修正了 sys\_recv 和 ip\_rx 中关于 IP 头长度和 UDP 包长度的计算公式。

## 3.2 用户手册 (User Manual)

### 3.2.1 功能特性

本系统扩展了 xv6 内核, 支持 E1000 网卡驱动及 UDP/IP 协议栈。

### 3.2.2 API 接口

```
// 1. 绑定端口
// 成功返回 0, 失败返回 -1
int bind(short port);

// 2. 发送数据
// dst 为目的 IP, dport 为目的端口
int send(short sport, int dst, short dport, char *buf, int len);

// 3. 接收数据
// 阻塞直到收到数据。src/sport 将填入发送方信息。
int recv(short dport, int *src, short *sport, char *buf, int maxlen);
```

### 3.2.3 使用方法

```
$ make qemu
$ nettest txone # 测试发送
$ nettest rxone # 测试接收 (需配合 server)
```

## 4 实验心得与 AI 使用感想

本次实验是我在操作系统课程中最具挑战性的一次。通过亲手实现网卡驱动，我深刻理解了操作系统内核如何通过 DMA descriptor ring 与外设硬件进行交互，以及软硬件协同工作的奥妙。这也让我明白了为什么网络 I/O 通常是异步的，以及操作系统如何通过中断和轮询的结合来高效处理数据流。

在本次实验中，我借助了 **GitHub Copilot** 等大模型工具辅助开发，其作用主要体现在以下几个方面：

1. **文档阅读与概念解析**: E1000 的硬件手册非常厚重，寄存器繁多。通过询问大模型，我能够快速定位到 TDT, RDT 等关键寄存器的作用，以及 TX/RX Descriptor 中各个标志位的含义（如 CMD\_EOP, STAT\_DD），这极大地缩短了我的“冷启动”时间。
2. **代码辅助与纠错**: 在编写驱动代码时，AI 能够根据上下文补全冗长的结构体定义和宏定义代码。更重要的是，它像一位结对编程的导师，在我忘记编写内存屏障（Memory Barrier）或忘记并在临界区加锁时，给出了风险提示，帮助我规避了潜在的并发 Bug。
3. **调试思路启发**: 在遇到 DNS 测试不通过的问题时，我将现象描述给 AI，它提示我关注“数据包长度计算”和“字节序转换”问题。这一提示直接帮助我发现了代码中未正确减去 UDP 头部长度的逻辑错误。

总的来说，AI 并没有替代我的思考，而是成为了放大我能力的工具，让我从繁琐的语法和文档检索中解放出来，能够更专注于操作系统核心逻辑（如并发控制、内存管理）的思考与实现。