Lab5 实验报告

姓名：李雨朔　学号：20232131090　班级：计科一班　日期：2025.1.1

一．　实验环境准备

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

```
oslab@xv6-vm:~/xv6-labs-2024$ git checkout net
branch 'net' set up to track 'origin/net'.
Switched to a new branch 'net'
oslab@xv6-vm:~/xv6-labs-2024$ git branch
  fs
* net
  pgtbl
  syscall
  util
oslab@xv6-vm:~/xv6-labs-2024$ make clean
rm -rf *.tex *.dvi *.idx *.aux *.log *.ind *.ilg *.dSYM *.zip *.pcap \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out user/usys.S user/_* \
kernel/kernel \
mkfs/mkfs fs.img .gdbinit __pycache__ xv6.out* \
ph barrier
```

二．　实现 E1000 网卡驱动

1. 使用自旋锁 e1000_lock 保护共享资源，防止多核并发访问冲突；然后通过读取 E1000_TDT 寄存器获取下一个可用的发送描述符索引 idx，检查该描述符的 E1000_TXD_STAT_DD 标志位，如果为 0 表示发送环已满，返回-1；如果该描述符之前关联了缓冲区，使用 mbuffree 释放它，然后使用 mbufalloc(0)分配新的内存缓冲区；将上层传递的数据从 addr 拷贝到新分配的 mbuf 缓冲区；设置描述符的地址、长度和命令标志；再将 mbuf 指针保存到 tx_mbufs 数组中供后续释放并更新 E1000_TDT 寄存器指向下一个描述符：

```
int
e1000_transmit(char *addr, int len)
{
  acquire(&e1000_lock);
  uint idx = regs[E1000_TDT];
  struct tx_desc *desc = &tx_ring[idx];
  // 检查发送环是否已满
  if(!(desc->status & E1000_TXD_STAT_DD)){
    release(&e1000_lock);
```

```
    return -1;
  }
  // 释放旧的缓冲区（如果有）
  if(tx_mbufs[idx]){
    mbuffree(tx_mbufs[idx]);
    tx_mbufs[idx] = 0;
  }
  // 创建一个新的 mbuf 缓冲区
  struct mbuf *m = mbufalloc(0);
  if (!m) {
    release(&e1000_lock);
    return -1;
  }
  // 拷贝数据到 mbuf
  memmove(m->head, addr, len);
  m->len = len;
  // 设置描述符
  desc->addr = (uint64)m->head;
  desc->length = m->len;
  desc->cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
    // 保存 mbuf 指针，以便后续释放
  tx_mbufs[idx] = m;
  // 更新尾指针
  regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;
  release(&e1000_lock);
  return 0;
}
```

```
int
e1000_transmit(char *addr, int len)
{
  //
  // Your code here.
  //
  // buf contains an ethernet frame; program it into
  // the TX descriptor ring so that the e1000 sends it. Stash
  // a pointer so that it can be freed after send completes.
  //
  acquire(&e1000_lock);

  uint idx = regs[E1000_TDT];
  struct tx_desc *desc = &tx_ring[idx];

  if(!(desc->status & E1000_TXD_STAT_DD)) {
    release(&e1000_lock);
    return -1;
  }

  if(tx_mbufs[idx]) {
    mbuffree(tx_mbufs[idx]);
    tx_mbufs[idx] = 0;
  }

  struct mbuf *m = mbufalloc(0);
  if(!m) {
    release(&e1000_lock);
    return -1;
  }

  memmove(m->head, addr, len);
  m->len = len;

  desc->addr = (uint64)m->head;
  desc->length = m->len;
  desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;

  tx_mbufs[idx] = m;

  regs[E1000_TDT] = (idx+1)%TX_RING_SIZE;

  release(&e1000_lock);
  return 0;
}
```

2. 使用自旋锁保护接收环的并发访问，然后获取待处理描述符索引，循环处理所有就绪描述符；从 rx_mbufs 数组中获取与该描述符关联的 mbuf，设置数据包长度；调用 net_rx()函数将数据包向上传递给网络协议栈；为处理完的描述符分配新的 mbuf 缓冲区；更新描述符和寄存器；继续处理下一个：

```
static void
e1000_recv(void)
```

```
{
    acquire(&e1000_lock);
    uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    while(1){
        struct rx_desc *desc = &rx_ring[idx];
        // 检查描述符是否就绪
        if(!(desc->status & E1000_RXD_STAT_DD)){
            break;
        }
        struct mbuf *m = rx_mbufs[idx];
        if(!m){
            panic("e1000_recv: no buffer");
        }
        m->len = desc->length;
         // 将数据包传递给上层协议栈
        net_rx((char *)m->head, m->len);
        // 重新分配一个缓冲区
        rx_mbufs[idx] = mbufalloc(0);
        if(!rx_mbufs[idx]){
            panic("e1000_recv: mbufalloc failed");
        }
        // 更新描述符
        desc->addr = (uint64)rx_mbufs[idx]->head;
        desc->status = 0;
        // 更新 RDT 寄存器
        regs[E1000_RDT] = idx;
        // 处理下一个描述符
        idx = (idx + 1) % RX_RING_SIZE;
    }
    release(&e1000_lock);
}
```

```
static void
e1000_recv(void)
{
  //
  // Your code here.
  //
  // Check for packets that have arrived from the e1000
  // Create and deliver a buf for each packet (using net_rx()).
  //

  acquire(&e1000_lock);
  uint32 idx = (regs[E1000_RDT]+1)%RX_RING_SIZE;

  while(1) {
    struct rx_desc *desc = &rx_ring[idx];

    //uint idx = (regs[E1000_RDT]+1)%RX_RING_SIZE;
    //struct rx_desc *desc = &rx_ring[idx];
    if(!(desc->status & E1000_RXD_STAT_DD)) {
      return;
    }

    struct mbuf *m = rx_mbufs[idx];
    if(!m) {
      panic("e1000_recv: no buffer");
    }

    m->len = desc->length;

    //rx_mbufs[idx]->len = desc->length;
    //net_rx(rx_mbufs[idx]);
    net_rx((char *)m->head, m->len);
    rx_mbufs[idx] = mbufalloc(0);

    desc->addr = (uint64)rx_mbufs[idx]->head;
    desc->status = 0;

    regs[E1000_RDT] = idx;
    idx = (idx+1)%RX_RING_SIZE;
  }
  release(&e1000_lock);
}
```

3. 编译:

```
oslab@xv6-vm:~/xv6-labs-2024$ python3 nettest.py txone &
[1] 3668
oslab@xv6-vm:~/xv6-labs-2024$ tx: listening for a UDP packet
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.img,if=none,f
ormat=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-d
ump,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 txone
```

三． 实现 UDP 接收系统调用

1. 实现端口绑定和数据包队列管理的相关数据结构:

```
#define NPORT 100
#define MAX_QUEUE_SIZE 10

struct recv_queue {
    struct mbuf *packets[MAX_QUEUE_SIZE];
    int head;          // 队列头索引
    int tail;          // 队列尾索引
    int count;         // 队列中数据包数量
};

struct port_binding {
    int port;                // 绑定的端口号
    int pid;                 // 绑定进程的 PID
    struct recv_queue queue;   // 接收队列
    struct spinlock lock;      // 保护队列的锁
};

struct port_binding ports[NPORT];
```

2. ip_rx()函数负责处理接收到的 IP 数据包，识别 UDP 包并将其放入对应端口的接收队列：

```
void
ip_rx(struct mbuf *m)
{
    struct ip *ip_hdr = (struct ip *)(buf+sizeof(struct eth));
     // 检查是否为 UDP 包
    if (ip_hdr->ip_p != IPPROTO_UDP)
        return;
    // 获取 UDP 头部
    struct udp *udp_hdr = (struct udp *)(ip_hdr + 1);
    uint16 dport = ntohs(udp_hdr->dport);
    acquire(&ports[port].lock);
    // 检查是否有进程绑定到这个端口
    if (ports[port].pid == 0) {
        release(&ports[port].lock);
       return;
    }
    struct mbuf *m = mbufalloc(0);
    if(!m) {
      release(&ports[dport].lock);
      return;
```

```
    }
    memmove(m->head, buf, len);
    m->len = len;
    m->next = ports[dport].queue;
    ports[dport].queue = m;
    // 唤醒等待该端口的进程
    wakeup(&ports[port]);
    release(&ports[port].lock);
    return;
  }
```

```
void
ip_rx(char *buf, int len)
{
  // don't delete this printf; make grade depends on it.
  static int seen_ip = 0;
  if(seen_ip == 0)
    printf("ip_rx: received an IP packet\n");
  seen_ip = 1;

  //
  // Your code here.
  //
  //struct eth *eth_hdr = (struct eth *)buf;
  struct ip *ip_hdr = (struct ip *)(buf+sizeof(struct eth));

  if(ip_hdr->ip_p != IPPROTO_UDP) {
    return;
  }

  struct udp *udp_hdr = (struct udp *)(ip_hdr+1);
  uint16 dport = ntohs(udp_hdr->dport);

  acquire(&ports[dport].lock);
  if(ports[dport].proc == 0) {
    release(&ports[dport].lock);
    return;
  }

  struct mbuf *m = mbufalloc(0);
  if(!m) {
    release(&ports[dport].lock);
    return;
  }

  memmove(m->head, buf, len);
  m->len = len;

  m->next = ports[dport].queue;
  ports[dport].queue = m;

  wakeup(&ports[dport]);
  release(&ports[dport].lock);
}
```

3. sys_bind()函数允许进程绑定到指定端口：

```
  uint64
  sys_bind(void)
```

```
{
    int port;
    argint(0, &port);
    if (port < 0 || port >= NPORT) {
        return -1;
    }
    acquire(&ports[port].lock);
    // 检查端口是否已被绑定
    if (ports[port].pid != 0) {
        release(&ports[port].lock);
        return -1;
    }
    // 绑定端口到当前进程
    ports[port].proc = myproc();
    release(&ports[port].lock);
    return 0;
}
```

```
uint64
sys_bind(void)
{
  //
  // Optional: Your code here.
  //

  int port;
  argint(0, &port);

  if(port<0 || port>=NPORT) {
    return -1;
  }

  acquire(&ports[port].lock);
  if(ports[port].proc != 0) {
    release(&ports[port].lock);
    return -1;
  }

  ports[port].proc = myproc();
  release(&ports[port].lock);

  return 0;
}
```

4. sys_recv()函数允许进程从绑定的端口接收 UDP 数据包：

```
uint64
```

```
sys_recv(void)
{
  int dport;
  uint64 src_addr, sport_addr, buf_addr;
  int maxlen;
  // 获取参数
  argint(0, &dport);
  argaddr(1, &src_addr);
  argaddr(2, &sport_addr);
  argaddr(3, &buf_addr);
  argint(4, &maxlen);
  // 参数验证
  if (dport < 0 || dport >= NPORT || maxlen < 0) {
    return -1;
  }
  acquire(&ports[dport].lock);
  // 等待队列中有数据包
  while (ports[dport].queue.count == 0) {
    sleep(&ports[dport], &ports[dport].lock);
  }
  // 从队列中取出数据包
  struct mbuf *m = ports[dport].queue;
  ports[dport].queue = m->next;
  if(ports[dport].queue == 0) {
    ports[dport].queue_tail = 0;
  }
  release(&ports[dport].lock);
  char *buf = m->head;
  // 解析 IP 和 UDP 头部
  struct ip *ip_hdr = (struct ip *)(buf+sizeof(struct eth));
  int ip_hlen = (ip_hdr->ip_vhl & 0x0F) * 4;
  struct udp *udp_hdr = (struct udp *)((char *)ip_hdr + ip_hlen);
  // 获取源 IP 和端口
  uint32 src_ip = ntohl(ip_hdr->ip_src);
  uint16 sport = ntohs(udp_hdr->sport);
  // 计算 UDP 数据长度
  int udp_len = ntohs(udp_hdr->ulen) - sizeof(struct udp);
  int copy_len = udp_len < maxlen ? udp_len : maxlen;
  char *payload = (char *)udp_hdr + sizeof(struct udp);
  // 将数据复制到用户空间
  if (copyout(myproc()->pagetable, src_addr, (char *)&src_ip, sizeof(src_ip)) < 0 ||
    copyout(myproc()->pagetable, sport_addr, (char *)&sport, sizeof(sport)) < 0 ||
    copyout(myproc()->pagetable, buf_addr, payload, copy_len) < 0) {
    mbuffree(m);
```

```
        return -1;
    }
    // 释放 mbuf
    mbuffree(m);
    return copy_len;
}
```

```
uint64
sys_recv(void)
{
  //
  // Your code here.
  //

  int dport;
  uint64 src_addr, sport_addr, buf_addr;
  int maxlen;

  argint(0, &dport);

  argaddr(1, &src_addr);

  argaddr(2, &sport_addr);

  argaddr(3, &buf_addr);

  argint(4, &maxlen);

  if(dport<0 || dport>=NPORT)  return -1;

  if(maxlen < 0)  return -1;

  acquire(&ports[dport].lock);

  while(ports[dport].queue == 0) {
    sleep(&ports[dport], &ports[dport].lock);
  }

  struct mbuf *m = ports[dport].queue;
  ports[dport].queue = m->next;
  if(ports[dport].queue == 0) {
    ports[dport].queue_tail = 0;
  }

  release(&ports[dport].lock);

  char *buf = m->head;
  //struct eth *eth_hdr = (struct eth *)buf;
  struct ip *ip_hdr = (struct ip *)(buf+sizeof(struct eth));
  int ip_hlen = (ip_hdr->ip_vhl & 0x0F)*4;
  struct udp *udp_hdr = (struct udp *)((char *)ip_hdr+ip_hlen);
```

```
  uint32 src = ntohl(ip_hdr->ip_src);
  uint16 sport = ntohs(udp_hdr->sport);

  if(copyout(myproc()->pagetable, src_addr, (char*)&src, sizeof(src))<0) {
    mbuffree(m);
    return -1;
  }

  if(copyout(myproc()->pagetable, sport_addr, (char*)&sport, sizeof(sport))<0) {
    mbuffree(m);
    return -1;
  }

  //int ip_len = ntohs(ip_hdr->len);
  //int ip_hlen = ip_hdr->ihl*4;
  int udp_len = ntohs(udp_hdr->ulen);
  int payload_len = udp_len-sizeof(struct udp);
  char *payload = (char *)udp_hdr+sizeof(struct udp);

  int copy_len = payload_len;
  if(copy_len>maxlen)  copy_len = maxlen;
  if(copy_len<0) copy_len = 0;

  //char *payload = (char*)(udp_hdr+1);
  if(copyout(myproc()->pagetable, buf_addr,payload, copy_len) < 0) {
    mbuffree(m);
    return -1;
  }

  mbuffree(m);
  return copy_len;
}
```

5. 测试：

```
oslab@xv6-vm:~/xv6-labs-2024$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kerne
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=vir
ump,id=net0,netdev=net0,file=packets.pcap -device e1000,ne

xv6 kernel is booting

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

```
oslab@xv6-vm:~/xv6-labs-2024$ python3 nettest.py grade



txone: OK
rxone: sending one UDP packet
```

四． 实验主观心得

本次实验是操作系统课程中极具挑战性的一次实践，通过实现 E1000 网卡驱动和 UDP

接收系统调用，我对操作系统的设备驱动和网络子系统有了更深入的理解：

1. 学习了如何通过内存映射 I/O 方式与硬件设备交互，理解了发送环和接收环的工作原理。
2. 掌握了数据包从网卡到应用程序的完整流程，包括以太网帧处理、IP 协议解析和 UDP 数据包处理。
3. 深入理解了 xv6 系统调用的实现机制，特别是参数传递和用户空间/内核空间数据拷贝。
4. 学会了使用自旋锁保护共享数据结构，防止多进程访问冲突。
5. 发送环和接收环的循环队列结构需要仔细理解，特别是头尾指针的管理。
6. 需要正确管理 mbuf 缓冲区的分配和释放，避免内存泄漏。
7. 需要考虑各种错误情况，如发送环满、内存分配失败等。