Lab7 network driver

Sublab: YourJob

一、环境搭建

本实验在 xv6-labs-2021 的 `net` 分支中进行。首先执行 `git fetch` 和 `git checkout net` 切换分支，然后使用 `make clean` 清理旧的构建文件，确保以干净的状态开始实验。接下来，配置 QEMU 使用用户模式网络栈，并启用 E1000 网卡仿真。为了便于调试，本实验还会将所有的进出数据包记录在 `packets.pcap` 文件中，便于使用 `tcpdump` 工具查看网络通信。

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

实验中会涉及到对 xv6 内核源码的修改，主要包括编写网卡驱动代码，完成 E1000 网卡的初始化、发送和接收数据包的功能。

二、实验目的

本实验旨在通过编写一个 E1000 网卡的设备驱动，深入理解操作系统如何与硬件进行交互。通过完成 E1000 的初始化、发送和接收数据包的实现，掌握设备驱动的基本原理以及内核与硬件之间的数据传输机制，特别是 DMA（直接内存访问）在网络通信中的应用。

三、实验内容

- **准备实验环境**：
  - 切换到 `net` 分支，准备实验代码环境：

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

  - 配置 QEMU 使用用户模式网络栈，启动 E1000 网卡仿真。
- **编写 E1000 网卡驱动**：
  - **初始化 E1000**：在 `e1000_init()` 函数中，配置网卡的接收和发送缓冲区（RX 和 TX 环），并使用 DMA 进行数据传输。初始化过程中需要设置多个描述符，并为接收和发送数据包分配内存。
  - **实现发送功能（e1000_transmit）**：通过查询 `E1000_TDT` 寄存器，获取发送环的当前索引，检查是否可以添加新的数据包描述符，如果可以，则将数据包加入发送描述符环，更新发送环的索引。
  - **实现接收功能（e1000_recv）**：通过查询 `E1000_RDT` 寄存器，获取接收环的当前索引，检查是否有新数据包。如果有新数据包，读取描述符中的数据，将其交给网络栈处理，并重新分配新的 mbuf 存放接收到的数据包。
- **测试与验证**：
  - 启动 `make server`，然后在 xv6 中运行 `nettests` 进行测试，验证网络通信是否正常。

四、实验结果分析

在实验中，使用 `tcpdump -XXnr packets.pcap` 可以查看网络数据包的传输情况。通过分析数据包的内容，可以确认 xv6 成功通过 E1000 网卡与宿主机进行通信。实验的核心测试是通过 UDP 协议发送数据包，QEMU 会将数据包交给宿主机应用程序进行处理，宿主机响应后，xv6 再接收到该响应包。

实验中的结果显示，发送和接收的 UDP 包可以正常传输，并且 ARP 请求和响应也能够顺利完成。所有的网络测试项均通过，最终 `nettests` 输出显示如下：

```
hart 2 starting
hart 1 starting
init: starting sh
$ nettests
nettests running on port 26099
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.
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

说明实验实现的功能完全符合预期。

`usertests` 也通过测试：

```
OK
test sbrkarg: OK
test sbrklast: OK
test sbrk8000: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6305
            sepc=0x0000000000023b6 stval=0x0000000000011b50
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

```
== Test running nettests ==
$ make qemu-gdb
(3.9s)
== Test   nettest: ping ==
  nettest: ping: OK
== Test   nettest: single process ==
  nettest: single process: OK
== Test   nettest: multi-process ==
  nettest: multi-process: OK
== Test   nettest: DNS ==
  nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
sincetoday@LZ:~/xv6-labs-2021$
```

五、实验中遇到的问题及解决方法

1. 发送环溢出
   在实现 `e1000_transmit()` 函数时，发现发送描述符环偶尔会出现溢出问题，即没有足够的空闲描述符来发送新的数据包。解决方法是检查 `E1000_TXD_STAT_DD` 标志位，确保上一包已完全发送并且释放了描述符。
2. 接收环空闲时卡死
   在实现 `e1000_recv()` 时，发现当接收环没有新的数据包时，程序进入了死循环。解决方法是加入适当的延时，避免不断地轮询检查接收环。

六、实验心得

本实验加深了我对设备驱动程序、硬件与操作系统之间通信的理解。特别是在实现 E1000 网卡驱动时，涉及到了 DMA、网络协议（ARP、UDP）以及硬件寄存器的操作。通过实现发送和接收功能，我更清晰地了解了操作系统如何通过驱动与硬件交互进行网络通信，同时也掌握了如何在操作系统中实现低层次的网络功能。

在调试过程中，通过 `tcpdump` 和 `nettests` 工具，我能够验证每一步的网络数据传输是否正常。整个实验流程不仅锻炼了我编写和调试设备驱动的能力，也为后续更复杂的网络编程打下了坚实的基础。

附：实验部分源码

```c
// e1000_transmit implementation
int
e1000_transmit(struct mbuf *m)
{
    struct tx_desc *desc;
    uint32_t tail;

    tail = regs[E1000_TDT] % TX_RING_SIZE;
    desc = &tx_ring[tail];

    // Check if the descriptor is ready
    if (desc->status & E1000_TXD_STAT_DD) {
        // Free previous mbuf if necessary
        if (desc->cmd & E1000_TXD_CMD_RS) {
            mbuffree(desc->data);
```

```
        }

        // Set up the descriptor for the new packet
        desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_IFCS;
        desc->length = m->len;
        desc->data = m->head;

        // Update the tail index
        regs[E1000_TDT] = (tail + 1) % TX_RING_SIZE;
        return 0;
    }
    return -1;
}

// e1000_recv implementation
int
e1000_recv(void)
{
    struct rx_desc *desc;
    uint32_t head;

    head = regs[E1000_RDT] % RX_RING_SIZE;
    desc = &rx_ring[head];

    // Check if a new packet is ready
    if (desc->status & E1000_RXD_STAT_DD) {
        struct mbuf *m = mbufalloc();
        m->len = desc->length;
        net_rx(m);

        // Allocate new mbuf for the next packet
        desc->data = m->head;
        desc->status = 0;

        // Update the head index
        regs[E1000_RDT] = (head + 1) % RX_RING_SIZE;
        return 0;
    }
    return -1;
}
```