同济大学操作系统课程设计 - Xv6 - Lab: networking

【实验目的】

通过本次实验，深入了解网络驱动程序的实现细节，掌握如何使用 DMA 技术高效地处理数据包的发送与接收。实验过程中熟悉环形缓冲区的使用，并理解如何通过描述符与网卡进行通信。在测试阶段，通过对 `xv6` 网络栈与 `e1000` 驱动程序的调试，巩固对网络协议栈与设备驱动程序的理解。

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

要启动实验室，需要先切换到 net分支：

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

使用一个名为 E1000 的网络设备来处理网络通信。对于 xv6（以及你编写的驱动程序）而言，E1000 看起来就像是一个连接到真实以太网局域网（LAN）的真实硬件设备。实际上，驱动程序将与 qemu 提供的 E1000 仿真进行交互，并连接到同样由 qemu 仿真的 LAN。在这个仿真的局域网中，xv6（"客户机"）的 IP 地址为 10.0.2.15。qemu 还会安排运行 qemu 的计算机在局域网上以 10.0.2.2 的 IP 地址出现。当 xv6 使用 E1000 发送数据包到 10.0.2.2 时，qemu 会将数据包传送到运行 qemu 的计算机（"主机"）上的相应应用程序。

`e1000.c` 中提供的 `e1000_init()` 函数在设置 E1000 网络设备以通过直接内存访问（DMA）处理网络通信方面起着至关重要的作用。此设置允许 E1000 直接在 RAM 中读取和写入数据包，从而提高了网络操作的效率。

关键概念：

1. **直接内存访问（DMA）**：
    - E1000 使用 DMA 在网络设备和 RAM 之间直接传输数据包。这样可以消除 CPU 进行数据传输的需要，从而提高了传输效率。
2. **接收环（Receive Ring）**：
    - 由于数据包传输速度可能快于驱动程序的处理速度，`e1000_init()` 为 E1000 提供了多个缓冲区，E1000 可以将接收到的数据包写入这些缓冲区。E1000 需要这些缓冲区由 RAM 中的 "描述符" 数组进行描述；每个描述符包含一个 RAM 中的地址，E1000 可以在该地址写入接收到的数据包。这个描述符数组称为接收环或接收队列。接收环是一个循环队列，当网卡或驱动程序到达数组末尾时，会回到数组的开头。`e1000_init()` 使用 `mbufalloc()` 为 E1000 分配用于 DMA 的 mbuf 数据包缓冲区。
3. **发送环（Transmit Ring）**：

- 发送环是一个描述符数组，驱动程序将要发送的数据包放置在其中，然后 E1000 从中读取并发送数据包。`e1000_init()` 将接收环和发送环配置为 `RX_RING_SIZE` 和 `TX_RING_SIZE` 大小。

4. **发送数据包（e1000_transmit()）**：
   - 当 `net.c` 中的网络栈需要发送数据包时，它会调用 `e1000_transmit()`，传入一个包含要发送的数据包的 mbuf。你的发送代码必须将指向数据包数据的指针放入发送环（TX ring）中的描述符中。描述符的格式由 `struct tx_desc` 描述。在 E1000 完成数据包传输之后，必须确保每个 mbuf 被释放。
5. **接收数据包（e1000_recv()）**：
   - 每当 E1000 接收到以太网数据包时，它首先通过 DMA 将数据包写入接收环（RX ring）描述符所指向的 mbuf 中，然后生成一个中断。你的 `e1000_recv()` 代码必须扫描接收环并将每个新数据包的 mbuf 通过调用 `net_rx()` 交给网络栈（在 `net.c` 中）。之后，你需要分配一个新的 mbuf 并将其放入描述符中，以便当 E1000 再次到达接收环中的该点时，能够找到一个新的缓冲区来 DMA 新的数据包。
6. **与 E1000 控制寄存器交互**：
   - 除了读写 RAM 中的描述符环之外，驱动程序还需要通过内存映射的控制寄存器与 E1000 交互，以检测接收到的数据包何时可用，并通知 E1000 驱动程序已填充了要发送的数据包的 TX 描述符。全局变量 `regs` 保存了指向 E1000 第一个控制寄存器的指针；你的驱动程序可以通过将 `regs` 作为数组来访问其他寄存器。你特别需要使用索引 `E1000_RDT` 和 `E1000_TDT`。

为了测试驱动程序，可以在一个窗口中运行 `make server`，然后在另一个窗口中运行 `make qemu`，接着在 xv6 中运行 `nettests`。`nettests` 的第一个测试尝试发送一个 UDP 数据包到主操作系统，该数据包将被发送到 `make server` 运行的程序。如果还没有完成实验，E1000 驱动程序将不会真正发送数据包，因此不会发生任何特别的事情。

一旦完成实验，E1000 驱动程序将发送数据包，qemu 会将其传送到主机计算机，`make server` 会看到数据包，并发送响应数据包，然后 E1000 驱动程序和 `nettests` 会看到响应数据包。然而，在主机发送回复之前，它会发送一个 "ARP" 请求数据包给 xv6 以获取其 48 位以太网地址，并期望 xv6 返回一个 ARP 回复。当你完成 E1000 驱动程序的工作后，`kernel/net.c` 将处理这些事务。如果一切顺利，`nettests` 将打印 `testing ping: OK`，而 `make server` 将打印 `a message from xv6!`。

实验步骤

1. 实现 `e1000_transmit` 函数

   `e1000_transmit` 函数负责将网络数据包从内存中传输到网络设备，以便通过网络发送出去。这个函数的实现需要将数据包的指针放入发送环（Transmit Ring）的描述符中，并通知 E1000 网卡开始发送数据包。

   代码如下：

```
e1000_transmit(struct mbuf *m)
{
  acquire(&e1000_lock);
  // 查询ring里下一个packet的下标
  int idx = regs[E1000_TDT];

  if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0) {
    release(&e1000_lock);
    return -1;
  }
  if (tx_mbufs[idx])
    mbuffree(tx_mbufs[idx]);

  tx_mbufs[idx] = m;
  tx_ring[idx].length = m->len;
  tx_ring[idx].addr = (uint64) m->head;
  tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
  regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;

  release(&e1000_lock);

  return 0;
}
```

2. 实现 `e1000_recv` 函数

   `e1000_recv` 函数负责从网络设备中接收数据包，并将其交给网络栈处理。该函数需要检查接收环（Receive Ring）中的描述符，找到已填充的缓冲区，并将数据包传递给网络栈。

   代码如下：

```
  while (1) {
    int idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    if ((rx_ring[idx].status & E1000_RXD_STAT_DD) == 0) {
      return;
    }
    rx_mbufs[idx]->len = rx_ring[idx].length;
    net_rx(rx_mbufs[idx]);
    rx_mbufs[idx] = mbufalloc(0);
    rx_ring[idx].status = 0;
    rx_ring[idx].addr = (uint64)rx_mbufs[idx]->head;
    regs[E1000_RDT] = idx;
  }

}
```

3. 启动服务器以测试驱动程序

   打开一个终端窗口，执行以下命令启动服务器程序，用于接收从 `xv6` 发送过来的测试数据包：

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab7$ make server
python3 server.py 26099
listening on localhost port 26099
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
```

4. 在 xv6 中测试数据包的发送与接收

   在另一个终端窗口中运行 `make qemu`，启动 `xv6`。然后在 `xv6` 命令行中运行以下命令，测试 `e1000` 驱动程序的数据包发送与接收功能：

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab7$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-de
vice,drive=x0,bus=virtio-mmio-bus.0 -netdev user,id=net0,hostfwd=udp::26999-:26
00 -object filter-dump,id=net0,netdev=net0,file=packets.pcap -device e1000,netd
ev=net0,bus=pcie.0

xv6 kernel is booting

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.
$
```

5. 结果验证

   - 当驱动程序正确实现时，`nettests` 将输出 `testing ping: OK`，表示数据包的发送与接收功能正常。
   - 在服务器窗口中，将看到从 `xv6` 发送过来的消息，确认网络通信已成功完成。

【分析讨论】

首先，通过实现 `e1000_transmit` 函数，我学会了如何将数据包从内存发送到网络设备。这部分工作让我体会到，驱动程序不仅仅是简单地操作硬件设备，更重要的是如何合理地管理系统资源，如内存和处理器时间，以保证网络通信的高效性和可靠性。描述符环（TX Ring）的使用让我明白了如何高效地管理数据传输过程中的缓冲区，避免资源浪费。

其次，在实现 `e1000_recv` 函数时，我进一步理解了接收数据包的复杂性。当数据包到达网卡时，驱动程序必须及时处理，以免丢包。而通过 DMA 技术，数据可以直接传输到内存中的指定位置，这大大提高了数据传输的效率。同时，环形缓冲区的使用也使得驱动程序可以连续处理多个数据包，有效防止缓冲区溢出。

最后，在调试与测试阶段，我深刻体会到编写驱动程序的挑战性。尤其是在测试驱动程序与操作系统网络栈的协作时，必须确保每个数据包都能正确地发送和接收，任何一个小错误都可能导致整个通信过程的失败。通过测试 `nettests`，我得以验证驱动程序的正确性，当看到 `testing ping: OK` 的提示时，成就感油然而生。

总的来说，这次实验不仅让我掌握了网络驱动程序的基本原理和实现方法，也让我在实践中得到了深入的理解和提升。这些经验对于我今后在操作系统和网络通信领域的进一步学习和工作都有着重要的帮助。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
== Test running nettests ==
$ make qemu-gdb
(4.5s)
== 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
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab7$
```