Lab5：I/O 设备与网卡驱动

本次实验，将进行一个较为有趣的任务，分为两部分：第一部分，我们要为 xv6 的 E1000 网卡添加发送/接收网络包的功能；第二部分，我们要为用户程序实现接受 UDP 数据包的系统调用：receive()。同样地，在进行具体任务之前，请各位同学仔细阅读“官方实验指南”和“官方参考书”的第五个章节。
官方实验指南：https://pdos.csail.mit.edu/6.828/2024/labs/net.html

请将实验平台的代码分支切换到 lab5：

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

5.1 预备知识

xv6 目前没有网络通信功能，为此，本次实验的主要内容就是为 xv6 添加简单的网络收发包功能，让 xv6 能够和主机系统进行基本的网络通信。

首先，直觉上，为了完成这个 lab，需要为 xv6 操作系统添加一个网络设备，也即：网卡。经过之前的四次实验，相信各位同学已经基本理解了——xv6 操作系统实质上是一个运行在主机系统及其 QEMU 软件之上的虚拟机，那么，在主机系统层次，QEMU 可以利用其权限为 xv6 创建一个“模拟”或者“虚拟”的网卡。由于其普适性和代表性，这里网卡的模拟对象选择了英特尔的 E1000 系列。

对于 xv6 及其内部的进程而言，这里的 E1000 看起来就像是一个真实的硬件，并且似乎也连接到了一个真实的本地局域网之上。在这个局域网里，xv6 的 E1000 网卡被分配了一个默认为 10.0.2.15 的 IP 地址，而 QEMU 端（虚拟机软件）则拥有默认为 10.0.2.2 的 IP 地址。每当 xv6 使用 E1000，向 10.0.2.2 的 QEMU 端发送数据包时，QEMU 可以把这个数据包转发到对应的主机系统中的某个进程中（有些像一个基于软件的交换机或路由）。

上述的 E1000 设备及其 10.0.2.X 的子网，其实都是由 QEMU 用对应的代码模拟出来的。并且，10.0.2.X 的子网和主机系统的物理网卡及其对应的子网并不冲突，甚至两者可以通过 NAT 或者桥接技术进行通信。本次实验中，我们只关注 10.0.2.X 的子网内部网络通信，也即主机系统和 xv6 之间的网络通信。

QEMU 可以对于在其上运行的（多个）虚拟机进行复杂的网络配置，也比较专业，大家有兴趣可以参考下述链接，本次实验对各位同学没有强制要求：

https://wiki.qemu.org/Documentation/Networking#User_Networking_.28SLIRP.29

很幸运，本次实验的基础代码，已经给出了本次任务的基本框架（只要你成功切换到了 net 分支就有）。例如，在 kernel/e1000.c 文件里，设备 E1000 的初始化代码已经给出。但是发送包和接受包的两个对应函数体暂时为空：e1000_transmit()和 e1000_recv()，它俩需要各位的努力填充。

在 kernel/e1000_dev.h 文件夹下，包含了 E1000 网卡的控制寄存器宏定义以及一些用于控制相关设备状态信息的标志位。这些定义和标志位所对应的内存地址（经过设备与内存之间的初始化映射之后形成），都来自于 E1000 的官方文档：

https://pdos.csail.mit.edu/6.828/2024/readings/8254x_GBe_SDM.pdf（不要被这个文档的厚度给蚌埠住了，本次 lab 只需要了解其中的一小部分：Section 3.1-Section3.4）。其实，设备驱动程序对 I/O 设备的控制方法，基本都是通过“内存映射 I/O”或者“端口映射 I/O”的方式对指定的设备寄存器读写来进行的。通过对具体寄存器地址进行宏定义抽象，kernel/e1000_dev.h 将降低我们对 e1000 网卡执行控制和相关操作的难度。大家可以稍微瞄几眼这个头文件，称里面的宏定义为“设备驱动与 I/O 设备之间的控制桥梁”应该是一个比较恰当的比喻（今后，如果你有机会从事设备驱动程序开发的工作，应该可以体会到提前拥有这么一个宏定义头文件是多么美好和幸福的事情）。

值得各位注意，在本次实验中，其实不需要关注太多具体的网络协议细节，因为这些针对特定的协议进行处理的代码已经给出：在 kernel/net.c 和 kernel/net.h 文件中，xv6 已经实现了三种网络协议，分别是：UDP、IP 和 ARP。我们实现的 E1000 驱动程序需要在接收到一个以太网帧的时候，将以太网帧向网络协议栈的上层传递，而上层的处理逻辑会识别其是否为 IP 或者 ARP 当中的一个协议（通过检查以太网帧中的 type 字段），再将其传递给对应协议的处理函数。在本次实验的第二部分中，我们需要实现 IP 协议的处理函数，识别 UDP 数据包，并做一些相应的处理。当然，本次实验的核心要求不包括实现 TCP 协议，但我们鼓励有余力的同学将其作为一个额外的挑战来完成。

kernel/pci.c 中已经给出了 xv6 启动时，在其 PCI 总线上（总线也是虚拟的）搜索 E1000 网卡设备及其初始化的流程，这些细节应该也是 QEMU 为其虚拟机模拟 PCI 设备的核心代码，值得学习和借鉴，本次实验不强制要求各位贵宾深入了解。

5.2 第一部分：NIC

5.2.1 发送

发送网络数据包方向：从“内存”到“网卡”。

因为进程发送数据的速度可能会大于网卡处理这些数据的速度（硬件单元的处理数据的频率不一致），所以，一般情况下，数据在两个不同的硬件单元进行传递时，通常需要借助于“I/O 缓冲”来屏蔽这两者之间的速度差异。

I/O 缓冲的相关知识我们在上个学期操作系统理论课的第五章有过接触。在 xv6-riscv 发送网络数据包的这个具体场景下，I/O 缓冲建立在 xv6 的内存之中（内核层），并且用“循环队列”的方式来对其进行管理，称其为“**发送环**”。发送环包含了若干个元素，每一个元素的类型都为结构体 struct tx_desc（好吧，发送环其实就是一个类型为 struct tx_desc 的数组），称其为“**发送描述符**”（下文简称“**描述符**”）。它的定义位于 kernel/e1000_dev.h 文件中的末尾处。

图 5.1 描述了发送环的具体结构：Head 和 Tail 指针的值均来自 E1000 网卡上的寄存器，Head 和 Tail 指针之间的描述符元素为硬件所拥有（即正在被网卡发送的数据包），我们的驱动程序不能操作这些描述符，其余的均为“空闲”的描述符。E1000 的初始化代码 e1000_init()已经将这个“发送环”传递给网卡，并将 E1000 中的 Head 和 Tail 指针的值均设置为 0（此时所有描述符就都是空闲的了）。

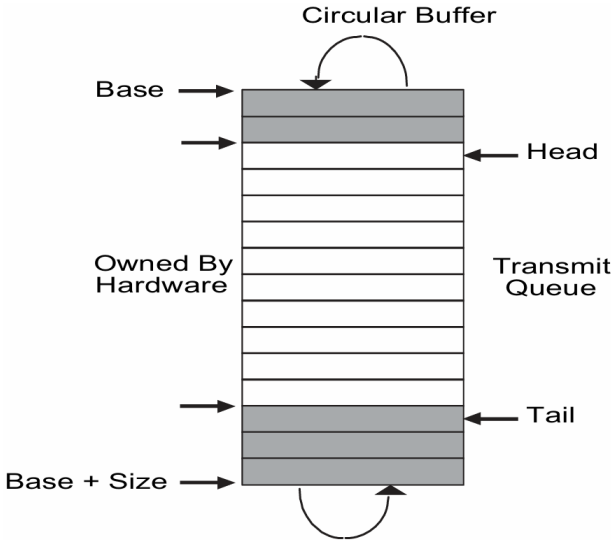

图 5.1 发送环的结构

说到这里，kernel/net.c 的 e1000_transmit()任务的核心逻辑就已经基本浮出水面了：将上层协议栈传递过来的数据缓冲区指针“嵌入”至发送环中某个空闲的描述符中，并设置好 Tail 寄存器使得网卡可以开始发送数据。**kernel/net.c 中的 e1000_init()的函数已经给出了初始化发送环和接收环的初始代码，大家在填写 e1000_transmit()和 e1000_recv()两个函数之前，最好仔细读个几遍。**

技术细节上，我们可以仔细阅读“官方实验文档”的提示：

- First ask the E1000 for the TX ring index at which it's expecting the next packet, by reading the E1000_TDT control register.
- Then check if the the ring is overflowing. If E1000_TXD_STAT_DD is not set in the descriptor indexed by E1000_TDT, the E1000 hasn't finished the corresponding previous transmission request, so return an error.
- Otherwise, use kfree() to free the last buffer that was transmitted from that descriptor (if there was one).
- Then fill in the descriptor. Set the necessary cmd flags (look at Section 3.3 in the E1000 manual) and stash away a pointer to the buffer for later freeing.
- Finally, update the ring position by adding one to E1000_TDT modulo TX_RING_SIZE.
- If e1000_transmit() added the packet successfully to the ring, return 0. On failure (e.g., there is no descriptor available), return -1 so that the

经过我的翻译、整理和补充，变成大家好理解的话就是：

- 第一步，通过读取控制寄存器 E1000_TDT 的值（用 regs[E1000_TDT]的方式来读取），来获取 Tail 指针，即下一个可以存放描述符的位置，记为 idx；
- 第二步，通过判断 idx 所对应的描述符中的 E1000_TXD_STAT_DD 标志位是否为零，来判断“发送环”是否存在溢出的情况——如果为零，则发送环中没有空闲的描述符，e1000_transmit()应该返回错误；
- 第三步，如果 idx 所对应的描述符中的 E1000_TXD_STAT_DD 标志位为 1，说明网卡已经完成了对该描述符中数据的处理，你应该释放这个描述符所对应的缓冲区（使用 kfree 函数）。
- 第四步，然后，将上层传入的数据缓冲区 buf“封装”到该描述符中：需要将 buf 和 len 赋予至 idx 所对应的描述符中的合适字段，并且，需要为该描述符

的 cmd 字段设置合适的标志位，来指明该描述符所对应的 buf 需要在将来被 DMA 传输至网卡后发送（参考 E1000 手册的 3.3 节）；

- 第五步，记得把这个 buf 的指针保存起来，用于之后的释放（与第三步的操作对应）；
- 第六步，更新“发送环”的 E1000_TDT。不要忘记取模（E1000_TDT modulo TX_RING_SIZE），以免越界；
- 最后，如果 e1000_transmit()成功地向发送环添加了一个描述符的话，则返回 0；如果失败，则返回-1，将此信息告诉 e1000_transmit()的调用者。

5.2.2 接收

接收网络数据包方向：从“网卡”到“内存”。

有了 xv6 发送网络包的研发经验之后，kernel/net.c 的 e1000_recv()接收网络包的核心逻辑其实也比较清晰了：将“接收环”（rx_ring）中的某个接收描述符（下文简称“描述符”）所包含的数据缓冲区“提炼”出来，然后再将其向网络协议栈的上层进行传递（通过调用函数 net_rx()实现）。

技术细节上，我们也可以借鉴“官方实验文档”的提示：

- First ask the E1000 for the ring index at which the next waiting received packet (if any) is located, by fetching the E1000_RDT control register and adding one modulo RX_RING_SIZE.
- Then check if a new packet is available by checking for the E1000_RXD_STAT_DD bit in the status portion of the descriptor. If not, stop.
- Deliver the packet buffer to the network stack by calling net_rx().
- Then allocate a new buffer using kalloc() to replace the one just given to net_rx(). Clear the descriptor's status bits to zero.
- Finally, update the E1000_RDT register to be the index of the last ring descriptor processed.
- e1000_init() initializes the RX ring with buffers, and you'll want to look at how it does that and perhaps borrow code.
- At some point the total number of packets that have ever arrived will exceed the ring size (16); make sure your code can handle that.
- The e1000 can deliver more than one packet per interrupt; your e1000_recv should handle that situation.

经过我的翻译、整理和补充，变成人话就是：

- 第一步，获取“接收环”中下一个要被“接收处理”的描述符 rx_desc 的位置 idx；获取的方法是查询其控制寄存器 E1000_RDT 的值，先加 1 再取模（modulo RX_RING_SIZE），然后再赋值给 idx；
- 第二步，与发送过程类似，此时 idx 的值指向了一个描述符。需要通过查询其 E1000_RXD_STAT_DD 状态位来判断其是否为一个真正待“接收处理”的描述符；如果不是，则表明接收环中没有需要待接收处理的数据，e1000_recv()处理逻辑终止；
- 第三步，如果 idx 指向的是一个待处理的描述符，则开启一个循环，用来处理“接收环”中所有尚未处理的描述符。这是因为接收环中可能存在多个尚未处理的描述符，而 idx 此时指向的只是这些数据中最早到达接收环的那一个；
- 第四步（第三步的循环内操作），“接收处理”一个描述符的大致操作是：调用 net_rx()，参数为 idx 所指向的描述符中的 addr 和 length 成员。将此数据包向上层传递；
- 第五步（第三步的循环内操作），因为我们已经将此数据包向上层传递，也就是已经将 idx 指向的描述符中的数据缓冲区“归为己有”，所以需要使用 kalloc 重新为这个描述符分配一个新的缓冲区，以便网卡可以“满血复活”；随后更新描述符的 addr 和 length 成员，使其指向你新分配的缓冲区，并将描述符的 status 设置为 0。
- 第六步（第三步的循环内操作），更新 E1000_RDT 寄存器的值为 idx；递增 idx（别忘了取模），让 idx 指向下一个未处理的 rx_desc 结构体，直至全部处理结束。

> 注意使用锁来保护对 E1000 网卡共享资源的访问，防止多个 CPU 上的进程同时调用 e1000_transmit 等函数。同时也防止在内核正在操作网卡时，一个突如其来的中断打断内核的执行，从而引发竞争条件。

5.2.3 测试代码

当前分支的 xv6 代码目录下有一个 python 脚本 nettest.py，用于本次实验的各项功能测试。 在此场景下，xv6 扮演客户端，nettest.py 扮演服务端。客户端则会向服务端发送 UDP 数据包。后者接收后，则会向客户端发送响应信息。

- 测试 e1000_transmit()：开启两个终端窗口，两个都在 xv6 源码根目录下，其中一个敲：“python3 nettest.py txone”； 另外一个敲“make qemu“，进入 xv6 的内部终端后，敲“nettest txone”。顺利的话，nettest.py 应该输出：“txone: OK”。
- 测试 e1000_recv()：同样地，开启两个终端窗口，其中一个敲“make qemu”， 进入 xv6 的内部终端后，在另一个终端中敲“python3 nettest.py rxone“。顺利的话，你应该能在 xv6 中看到以下两行输出：

```
arp_rx: received an ARP packet
ip_rx: received an IP packet
```

5.3 第二部分：UDP Receive

5.3.1 实现进程之间的 UDP 通信

现在，我们的 xv6 已经具备了发送和接收网络数据包的能力，但还缺少一个关键环节——如何将接收到的网络数据包传递给对应的用户进程？由于 xv6 可能同时运行多个用户进程，我们需要一种机制来判断每个数据包应该交给谁。其实，答案正如本节标题所示：我们可以借助 UDP 协议，将收到的 IP 数据包“多路复用”到具体的用户进程。

具体来说，实现用户进程接收和发送 UDP 数据包需要以下 4 个系统调用 API：

- `send(short sport, int dst, short dport, char *buf, int len)`：将用户地址空间下的长度为 len 的数据缓冲区 buf 封装到 UDP 包中，并发送到 IP 地址为 dst、端口号为 dport 的主机进程上。xv6 已经实现了这个系统调用，我们**不需要**实现。

- `recv(short dport, int *src, short *sport, char *buf, int maxlen)`：接收来自 IP 地址为 src、端口号为 dport 的主机进程发送的 UDP 数据包，并将 UDP 数据包的所携带的载荷（payload）拷贝到 buf 中，最多拷贝 maxlen 个字节。
- `bind(short port)`：绑定端口号 port 到调用进程。进程在调用 recv 之前，应该先调用 bind 系统调用，表示调用进程对端口号为 port 的 UDP 包"感兴趣"；当内核接收到端口号为 port 的 UDP 包时，应该将此包传递给该进程。同时，内核应该丢弃未被任何进程绑定的 UDP 包。
- `unbind(short port)`：与 bind 的作用相反：解除端口号 port 与调用进程的绑定。我们**不需要**实现这个系统调用。

在本次实验中，我们需要实现 recv 和 bind 系统调用。此外，我们还需要实现 IP 协议的处理函数 ip_rx()。ip_rx()和上诉 4 个系统调用都已经在 kernel/net.c 中定义好，我们要做的就是补全这些函数。**建议大家在开始实验之前先仔细阅读 net.c 中已经实现好的部分函数代码，分析并理解处理网络数据包的方法。**我先列出官方实验指南中给的一些提示，然后再做补充。

> - *Create a struct to keep track of bound ports and the packets in their queues.*
> - *Refer to the sleep(void *chan, struct spinlock *lk) and wakeup(void *chan) functions in kernel/proc.c to implement the waiting logic for recv().*
> - *The destination addresses that sys_recv() copies the packets to are virtual addresses; you will have to copy from the kernel to the current user process.*
> - *Make sure to free packets that have been copied over or have been dropped.*

根据提示，我们可以一步步来：

- 首先，我们需要定义一个数据结构 M，将端口号和对应该端口号所接收到的数据包队列建立起映射。队列的长度必须是有限长的，以防止因短时间内过多的数据包到达导致系统内存被消耗殆尽；
- 实现 net_rx()：net_rx()需要确定上层传下来的数据包是否是一个 UDP 包，如果是，则从 UDP 首部中取出端口号，根据端口号将数据包存放在对应的队列里面（并调用 wakeup 函数唤醒等待该队列的进程)。如果不是一个 UDP 包或者对应的队列满了，则丢弃该包；
- 实现 sys_bind()：sys_bind()的行为其实就是往（1）中的数据结构 M 插入一项映射，用伪代码来描述就是：M.insert(port, *an empty queue*)；

- 实现 sys_recv()：sys_recv()从指定的队列中取出一个数据包。如果队列为空，则调用 sleep 函数使进程等待；将数据包的 UDP payload 拷贝到用户提供的地址，并将 IP 首部的源地址和 UDP 首部的端口号也拷贝到用户空间。
- 注意在 sys_recv()的参数中用户提供的地址都是用户空间下的虚拟地址，所以我们需要使用 copyout 函数将数据从内核拷贝到用户地址，拷贝端口号和 IP 地址的时候注意网络和主机之间字节序不一致的问题；
- 注意调用 kfree()释放任何不再需要的数据包；
- 阅读 kernel/proc.c 中 sleep 和 wakeup 函数的代码，理解它们是如何通过一个“通道”（channel）来让一个进程等待特定资源的，以及另一个进程是如何唤醒所有正在等待该资源的进程的。

5.3.2 测试代码

开启两个终端窗口，一个在 xv6 源码根目录下，敲“python3 nettest.py grade”；

另外一个在 xv6 的源码根目录下，敲“make qemu”，顺利进入 xv6 的内部终端后，敲“nettest grade”，nettest 会执行一系列的测试项目，当屏幕出现“free: OK”后，说明你已经通过了所有的测试，完成了本次实验。

5.4 提交实验文档

请将上述实验过程写进你的实验报告，提交对应的 PDF 格式文档，并需要在其末尾附上一个详细真诚的实验心得或感想，包括你在本次实验中，是如何借助大模型的。