国科大操作系统研讨课任务书

RISC-V 版本

版本 *2025Spring*

目录

Project 5
设备驱动

1 实验说明

在之前的实验中，同学们已经实现了操作系统中的定时器中断、实现了进程间的通信，内存管理等，已经一步步构建了一个操作系统。然而，大家的操作系统仍然缺少与外部设备打交道的驱动程序。虽然我们已经有了最简单的外设——串口和 SD 卡，但这些外设的驱动程序实际上是 BIOS 提供的。本次实验中我们将学习怎么写一个设备驱动程序。

驱动程序在系统中的所占的地位也十分重要。一个操作系统可以支持多少种常见设备的驱动程序往往决定了这个操作系统的适用范围。外设有多种多样，驱动程序也五花八门，从简单到复杂。本次任务我们选择了一个代表性的“复杂”外设——网卡。在本次任务中，同学们需要完成基于 E1000 82540EM 网卡的驱动程序，以实现发送和接收数据帧的基本功能，从而使得大家的操作系统能够通过以太网与外部网络进行通信。

本次实验的各个任务如下：

任务一 实现网卡初始化与轮询发送数据帧的功能。

任务二 实现网卡初始化与轮询接收数据帧的功能。

任务三 添加网卡中断，实现基于网卡中断的收发数据帧的功能。

任务四 网卡的对传

这四个任务中，S-core 的同学只需要完成任务一和任务二。A-core 的同学需要完成任务一至任务三，C-core 的同学需要完成全部四个任务。

和之前的实验一样，我们提供了一个初始的代码框架 start code，里面会给出一些要实现的基本函数，同学们可以参考它，在它的基础上完成本实验。请将本次 Project 的 start code 增加到同学们自己实现好的 Project 4 中，进而继续增加网卡驱动的功能，`Makefile` 文件可以直接替换使用本次 project 的 `Makefile`，也可以阅读理解 `Makefile` 后，根据自己增加的 C 文件自行修改。当然，同学们也可以通过合理的设计，改变现有的代码框架，也许你能设计出更好的网卡驱动。

如果使用 start code 中提供的 `Makefile`，请在启动 QEMU 时使用 `make run-net` 和 `make debug-net`。此外，做单核的同学记得把 `Makefile` 中这两个目标对应对应的 (`QEMU_SMP_OPT`) 选项给去掉。

除了本任务书之外，同学们可以参考《8254x_GBe_SDM》[1] 手册中的内容，以使得操作系统代码的内容可以符合硬件的约定。因篇幅所限，任务书中不会对单个网卡寄

存器内部的具体布局做详细介绍，所以希望同学们能够在任务书指引的基础上，自行查阅手册第 13 章 MAC 内部寄存器的相应内容，以了解相关的寄存器布局。

从下一章开始，我们将按任务的顺序，详细介绍同学们应该完成的功能。

2 网卡简介

网卡是网络接口卡 NIC (Network Interface Card) 的简称，也称为网络适配器，它是一块被设计用来允许计算机在计算机网络上进行通讯的硬件。无论是普通电脑还是高端服务器，只要连接到局域网，就都需要安装一块网卡。如果有必要，一台电脑也可以同时安装两块或多块网卡。

网卡作为计算机与计算机或外部设备间进行通信的桥梁，主要有以下两大功能：一是将待传输的数据封装为帧，并通过网线（对无线网络来说就是电磁波）将数据发送到网络上去；二是接收网络上其它设备传过来的帧，并将帧重新组合成数据，发送到所在的计算机中。

本次实验在 QEMU 上使用的网卡是 E1000 82540EM（下文简称 E1000），同时助教团队也为大家在开发板上实现了 E1000 LITE 模块，能够支持 E1000 网卡最基本的收发数据帧（含中断）的功能。

2.1 网卡驱动和 OSI 参考模型

OSI（Open System Interconnect）参考模型，是 ISO 组织在 1985 年研究的网络互连模型，该体系结构标准定义了网络互连的七层框架，自顶向下分别为：应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。其中，自顶向下前五层都是软件实现的概念（集成在操作系统软件和第三方库中），而后两层（数据链路层和物理层）则由相应的硬件控制器实现。

网卡涵盖了 OSI 模型的数据链路层和物理层。其中，数据链路层的芯片叫做 MAC Controler (MAC)，它实现了寻址、数据帧的构建、数据差错检查、传送控制、向网络层提供标准的数据接口等功能；物理层则由 PHY 芯片进行实现，它定义了数据传送与接收所需要的电与光信号、线路状态、数据编码等，并向数据链路层提供标准的接口。

数据链路层的信息传输单位叫做数据帧 (frame)，而网络层的信息传输单位叫做数据包 (packet)。一个数据包在数据链路层可能被拆解为多个数据帧，也可能多个数据包合成为一个数据帧。在本次实验中，大家设计的驱动程序将会通过设置 MAC 寄存器来进行数据帧的收发，不需要考虑 PHY 相关寄存器的设置。

2.2 E1000 MAC 结构

如图P5-1所示，E1000 网卡的 MAC 内部集成了独有的、无法被其他模块所使用的 DMA 控制器，专门配合 MAC 做数据传输。这个小小的 DMA 控制器是本次实验的核心所在，在后续的章节中会对此进行详细的说明。

MAC 寄存器包括内部寄存器部分 (TX/RX MAC) 和可选择的 Flash、ROM 寄存器部分，后者在本实验中不予考虑。MAC 内部寄存器的起始地址需要从设备树中读取，

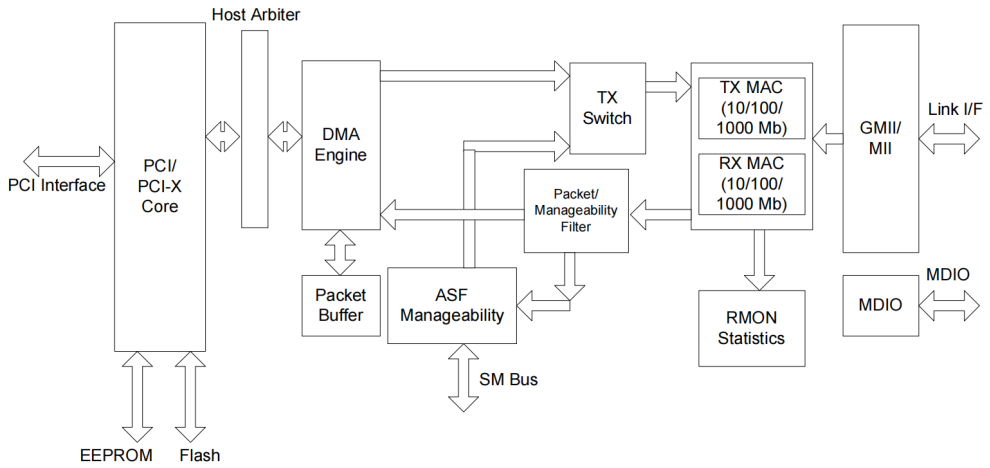

图 P5-1: E1000 网卡 MAC 结构图

为了简化实现，我们提供了相应的 BIOS 函数用于实现该功能。

```
ethernet_addr = bios_read_fdt(ETHERNET_ADDR);
```

在对某个 MAC 内部寄存器访问时，可以用**寄存器组的起始地址 + 寄存器的偏移量**的方式进行访问。但需要注意的是，`bios_read_fdt(ETHERNET_ADDR)` 读取到的都是物理地址。如果大家完成了 Project 4，开启了虚存的话，那么就无法直接访问物理地址了。因此，在这种情况下，需要将物理地址映射到内核段的虚地址上，通过虚地址进行访问。在 Linux 中，这一操作叫做 ioremap，也就是将一个物理地址映射到某个空闲的虚地址上。而在各位同学完成的小系统中，同样需要大家自己实现 ioremap 操作，以顺利通过虚地址访问对应的物理地址。相信大家在做完 Project 4 的虚存部分之后，对这样的映射操作已经并不陌生了，请参见 start code 中 ioremap 的接口定义，实现函数功能。需要提醒的就是，做完 ioremap 的操作之后需要刷一下 TLB，因为页表发生了变化。

对于 MAC 的内部寄存器，我们将在任务书接下来的几个章节中结合具体内容进行讲解，完整的内部寄存器说明详见 E1000 手册 [1] 第 13 章。

3 DMA 简介

在本次实验中，同学们需要和 MAC 上的 DMA 控制器打交道，为此有必要先简单复习一下 DMA 的概念。

DMA (Direct Memory Access) 是指外部设备不通过 CPU 而直接与系统内存交换数据的接口技术。相较于 PIO (Programming I/O) 模式而言，在 DMA 模式下，CPU 并不全程参与数据的传送工作，只需向 DMA 控制器下达指令，让 DMA 控制器来处理数据的传送，数据传送完毕再把信息反馈给 CPU，这样就很大程度上降低了 CPU 的资源占有率。

那么，如何向 DMA 控制器下达指令？以及该下达怎样的指令？这就需要大家进一步了解 DMA 描述符的概念了。

3.1 DMA 描述符

在前面的实验中，我们实践过了通过设置控制寄存器等方式来触发某些功能，比如中断等等。而在本次实验中，由于 DMA 需要提供的信息很多，所以与以往直接写寄存器的方式不同，控制 DMA 需要将必要的信息以一个约定好的数据结构放置在内存中，再将其对应的内存地址给到 DMA，并发出使能信号之后，DMA 就会自动读取其中的内容来执行操作。这一点很像虚拟内存中，我们将页表的结构建立好，CPU 会根据 `satp` 寄存器中存放的基地址自动查页表内容做地址翻译的这一过程。

上述提到的这一数据结构就叫做 DMA 描述符 (DMA Descriptor)，它是驱动程序和 DMA 硬件的交互接口。每一个 DMA 描述符都包含两个部分：第一部分主要记录数据的地址，来告知 DMA 需要从哪里获取/存放数据；第二部分主要记录描述符的状态，例如数据的大小、DMA 传输是否出错等。同时，DMA 硬件还需要知道描述符存放的具体内存位置，以获取到上述的这些信息。大家可以这样想象，DMA 就像函数，描述符就像传给函数的参数。我们通过读写相应的控制寄存器，将描述符的地址告知 DMA，DMA 就会自动读取描述符的内容，并完成相关的功能。

以本次网卡实验为例，实验的整体思路就是这样：同学们需要先设置一组发送/接收描述符并告知 DMA 其地址所在，然后将 DMA 需要的信息填入其中，再触发 DMA 的功能，DMA 就会自动读取数据结构中的内容，并执行发送/接收数据帧的操作，最后将执行结果填入描述符中。看起来是不是很简单？这个实验其实一点都不难嘛！然而，这其中牵扯到了很多细节上的问题：发送/接收描述符长啥样？DMA 传输需要哪些信息？多个发送/接收描述符是如何关联起来的？该如何将描述符的起始地址告知 DMA？如何触发 DMA 的发送/接收功能？我们下面来详细分解。

4 E1000 发送数据帧

E1000 发送数据帧有三个要点：其一为发送描述符的格式；其二为描述符循环数组的组织；其三为发送的使能。掌握上述三个要点，任务一则不攻自破。

这一章的内容对应于 E1000 手册 [1] 的第 3.3、3.4 和 14.5 节，同学们根据需要自行查阅。

4.1 E1000 发送描述符

E1000 的发送描述符有三种格式，分别为 Legacy、TCP/IP Data 和 TCP/IP Context 三种，后面两种是针对 TCP/IP 协议的优化措施，与本课程的主题无关。因此，本实验选用 Legacy 格式的发送描述符。

选择 Legacy 格式的发送描述符的话，需要大家置零 TDESC.CMD.DEXT 这一位，以告诉底层硬件所使用的发送描述符布局。Legacy 发送描述符的布局如图P5-2所示，可以看到单个 E1000 发送描述符大小为 16 Bytes，包含 Buffer Address、Length、CMD (command) 和 STA (status) 等字段。其中，Buffer Address 和 Length 字段必须由软件提供，其余字段可以留空。

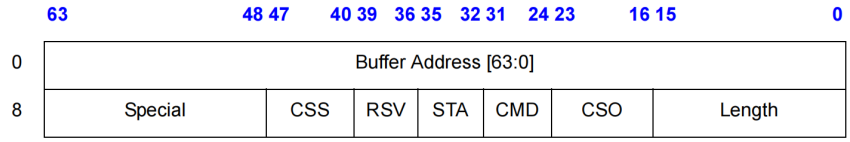

图 P5-2: E1000 发送描述符 (Legacy)

要让 DMA 控制器能够定位到数据帧的所在，就必须要将数据帧的起始地址和长度分别填入 Buffer Address 和 Length 字段。值得注意的是，填入描述符的起始地址必须是**物理地址**，这是因为 DMA 控制器的访存不会经过 MMU，使用的自然是物理地址。因此，如果大家已经正常的完成了虚拟内存实验，需要将填入描述符的地址进行转换。

对于发送描述符的 CMD 字段而言，该字段存储了一些指示 DMA 控制器的命令，其布局如图P5-3所示。其中，DEXT (Descriptor Extension) 位用来指示发送描述符的布局，该位设置成 0 的话，DMA 控制器才会按照 Legacy 的布局来解析描述符内容，否则就是另外的描述符格式了。RS (Report Status) 位置 1 的时候，DMA 控制器将会记录传输的状态于描述符的 STA 字段，反之则 STA 字段无效。EOP (End Of Packet) 位表示当前数据帧是否是数据包的最后一个帧，因为如前文所述，网络层的数据包在数据链路层可能会因为大小过大，而被拆解成多个帧进行传输。本次实验中，测试程序传输的数据包都比较小，因此可以做到一个数据包对应一个数据帧。

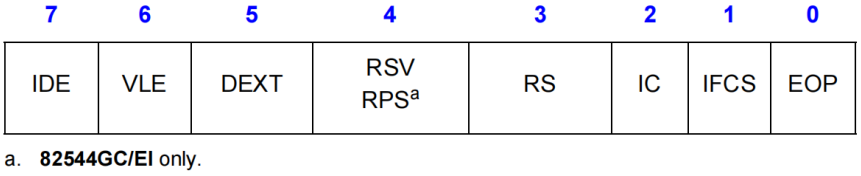

图 P5-3: E1000 发送描述符 CMD 字段

发送描述符的 STA 字段则存储了 DMA 发送数据帧的状态，例如当前帧的传输是否完成（STA.DD, Descriptor Done）。当然，为了降低 DMA 访存对内存带宽的影响，只有当 CMD.RS 为 1 时，DMA 控制器才会在传输结束之后填写 STA 字段。

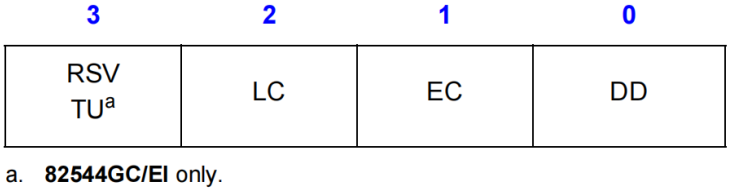

图 P5-4: E1000 发送描述符 STA 字段

除了上述提到的这些字段以外，其余发送描述符的字段在本实验中不会涉及到，就不再赘述了。

4.2 发送描述符循环数组

解决了发送描述符与发送单个数据帧所需的信息这两个问题之后，接下来我们向大家介绍一下发送描述符是怎么组织的。

如图P5-5所示，不同于 Virtio 和 Xilinx GEM 网卡的链式描述符队列，E1000 网卡的采用循环数组的方式组织描述符。这种方式虽然失去了描述符数量的可扩展性，但胜在访存性能高，因为在描述符数组按 cache block 大小对齐的情况下，一个 cache block 中可以包含多个描述符，从而顺序访问描述符数组时，cache 的命中率较高。对于 I/O 数据操作，一般来说粒度越大，效率越高。

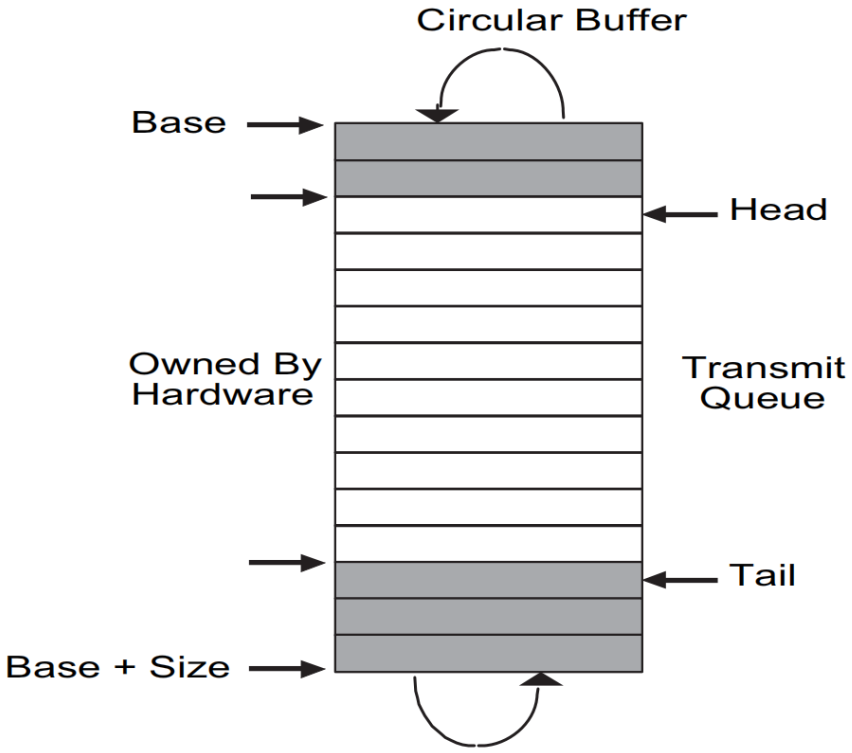

图 P5-5: E1000 发送描述符循环数组

DMA 控制器是如何得知这一循环数组所在的呢?这里就需要使用到 MAC 内部寄存器中的 TDBAL、TDBAH 和 TDLEN 三个寄存器了。TDBAL/TDBAH (Tx Descriptor Base Address Low/High) 分别存储循环数组基地址的低/高 32 位, TDLEN (Tx Descriptor Length) 则存储循环队列所占字节数，以便 DMA 控制器计算出循环数组的元素个数（因为描述符大小固定为 16 字节）。值得注意的是，这里的基地址仍然是物理地址，请各位同学务必铭记在心。

除此之外，循环数组还使用 Head 和 Tail 头尾指针来界定驱动程序和 DMA 控制器所占有的描述符。如图P5-5所示，白色部分为硬件占用的描述符，阴影部分描述符则由软件进行管理，而 Head 和 Tail 指针则分别指向硬件/软件正在处理的描述符。在发送的过程中，当 DMA 控制器将数据帧发送出去后，DMA 控制器就会将 Head 指针向后移动一格，表示数据包发送完毕，将该描述符交还给软件，以便软件填入下一个需要传

输的数据帧；而如果软件完成了待传输数据帧的填写，软件就会将 Tail 指针后移一格，表示将该描述符交给硬件进行处理。

同样，为使 DMA 控制器能够获得 Head 和 Tail 头尾指针的数值，E1000 MAC 也使用了内部寄存器来存储二者的数值：Head 指针被存入 TDH (Tx Descriptor Head) 寄存器中；Tail 指针被存入 TDT (Tx Descriptor Tail) 寄存器中。值得一提的是，TDH 和 TDT 寄存器存储的是描述符元素的索引，毕竟 MAC 有了描述符基地址，可以直接推算出来指针指向的描述符内存地址。

此外，如何判断一个数据帧是否完成了传输呢？大家可能会让软件直接读取 TDH，这也是最直观的想法。然而，根据 E1000 手册 [1] 上的描述，直接读取 TDH 的值是不准的，因为软件可能刚读完，硬件就移动了 Head 指针。最好的办法就是设置描述符的 CMD.RS 位，再通过硬件自行设置的 STA.DD 位来判断数据帧是否已完成传输。

4.3 E1000 发送使能

现在还有最后一个悬而未决的问题：如何使能 DMA 控制器发送数据帧？这个问题还请让在下分为两步作答。

在 MAC 内部寄存器中，有一个全局的发送控制寄存器 TCTL(Transmit Control)。该寄存器的 EN 位控制了发送数据帧的全局使能，即当 TCTL.EN 为 0 时，禁止一切发送数据帧的操作。

当 TCTL.EN 被使能时，一旦 TDH 不等于 TDT，那么 DMA 控制器就会认为驱动软件已经将待发送的数据帧交给了硬件处理，是该开始工作了。此时 DMA 控制器就会读取 TDH 所指向的发送描述符的内容，传输数据帧并后移 TDH，直至 TDH 和 TDT 相等（硬件已经处理完所有待发送的数据帧）。

4.4 E1000 发送初始化

对于 E1000 的发送初始化，同学们只需要关心发送描述符和相关的 MAC 内部寄存器的初始化即可。其余涉及到 PHY 等的与课程无关的部分，各位无需多虑，我们在 BIOS 内已经帮大家提前设置好了。

在本实验中，E1000 发送初始化的大致流程如下所示，更加详细的发送初始化流程详见 E1000 手册 [1] 的 14.5 节，同学们还是根据需要自行查阅。

1. 初始化发送描述符循环数组。
2. 初始化 TDBAL、TDBAH、TDLEN、TDH、TDT 寄存器。
3. 初始化 TCTL 寄存器，设置 TCTL.EN 为 1、TCTL.PSP 为 1、TCTL.CT 为 0x10H、TCTL.COLD 为 0x40H。

4.5 实验须知

本机网络的注意事项

1. 当使用网线连接开发板和本机后，IPv4 和 IPv6 协议会向连接处发送数据包，在监测网卡时这些包会被监测到。如果不想受到这些包的干扰，请自行关闭 IPv4 和 IPv6 协议。

2. 本机的网卡连接速度和双工模式必须设置为自动检测，默认情况下就是这个设置，如果需要修改，在 Windows 下打开以太网配置中高级-连接速度和双工模式，如图P5-6所示。

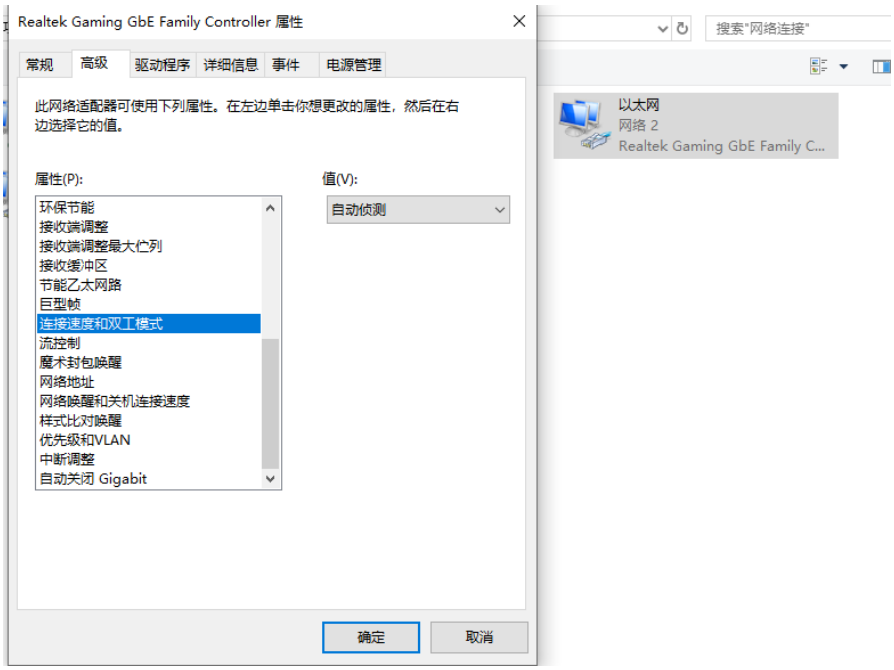

图 P5-6: windows 下 aotudetect 模式设置

捕获数据包

常用的抓包软件有 tcpdump 和 wireshark，本节将根据是否上板调试、以及本机的机型来阐述抓包工具的使用。

QEMU 抓包

QEMU 平台上为大家创建了一个虚拟的 e1000 设备，并连接到名为 tap0 的 TAP 设备上。因为创建和销毁该设备需要进行以太网设备的连接与断开，所以模拟了 e1000 网卡的 QEMU 在启动和退出时会比原来多花费一点时间，请各位同学保持耐心。

如果大家在实验课提供的虚拟机上运行 QEMU 的话，抓包工具选用 tcpdump，运行命令如下所示。当然如果大家有自己的调试需求的话，欢迎查阅 tcpdump 的资料，来修改下述命令除 `-i tap0` 外其他的命令行参数。

```
sudo tcpdump -i tap0 -XX -vvv -nn
```

如果大家在自己的本机上运行 QEMU，则需要修改 QEMU 目录/etc 下的 `qemu-ifup`，将 `IFNAME=enp0s3` 中等号后的部分替换为使用机器 `ifconfig` 之后显示的内容，即 `ifconfig` 命令打印出来的网卡名称，然后再执行上述抓包命令即可。

开发板抓包

要捕获运行同学们的操作系统从开发板上发出的数据包的话，首先需要用网线连接开发板与本机，来建立数据包的传输信道。

对于使用 Linux 和 Mac 的同学，仍然可以用 tcpdump。对于使用 Windows 系统的同学们来说，大家需要安装 wireshark 软件。安装完后，使用管理员模式打开 wireshark，选择与板子相连的以太网端口，一般情况下是本地连接。双击本地连接，如图P5-7所示:

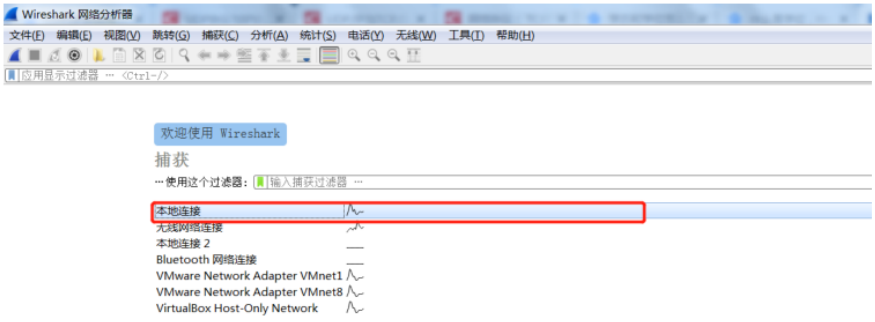

图 P5-7: wireshark 界面

之后在搜索框输入 udp，按 enter 键，则会显示捕捉到 udp 的报文。如图P5-8所示，在测试发包时查看显示的 udp 报文即可。测试收包时，也可以用同样的办法查看信道上传输的数据包。

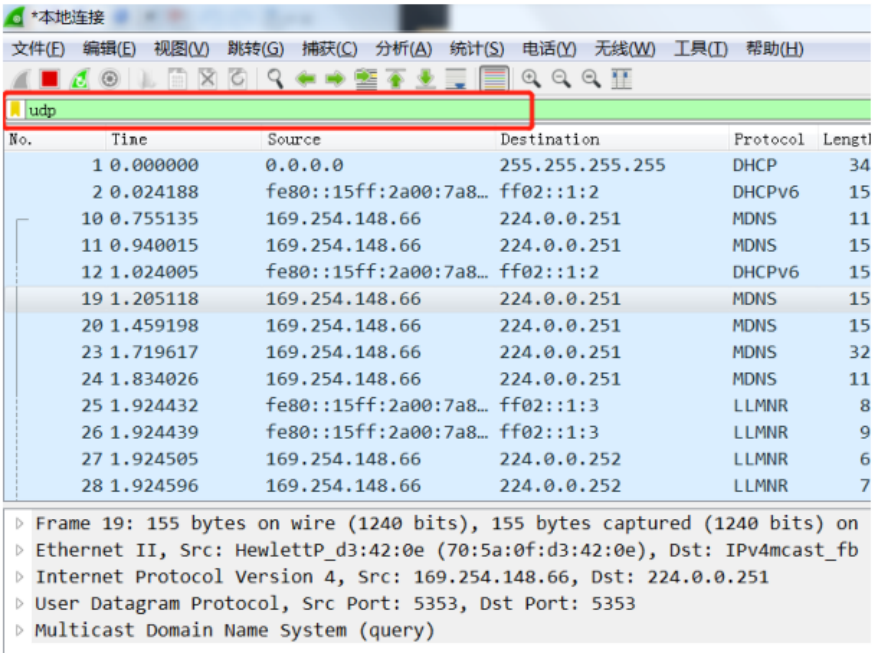

图 P5-8: wireshark 界面（续）

对于使用 mac/Linux 的同学们，大家可以使用先前提到的 tcpdump 来捕获数据包。

安装 tcpdump 后，首先获得板子与虚拟机连接的网卡名，在我们电脑中显示的网卡端口名是 `enx34298f709230`，在同学们的电脑中显示的名称可能不同，以各位电脑的显示为准。如图P5-9所示，在 terminal 中输入命令 `ifconfig`

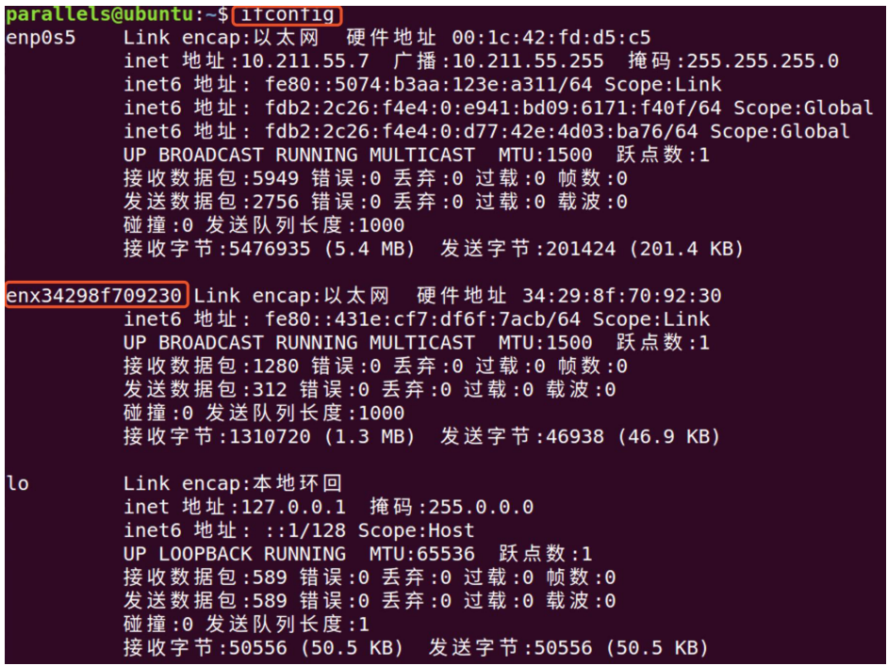

图 P5-9: 获得网卡名

如图P5-10所示，在 terminal 中输入 `sudo tcpdump -i enx34298f709230 host 224.0.0.251` 命令就可以监听目的地址为 224.0.0.251 的数据包了。如果不加 `-i enx34298f709230` 是表示抓取所有的接口收到目的地址为 224.0.0.251 的数据包。

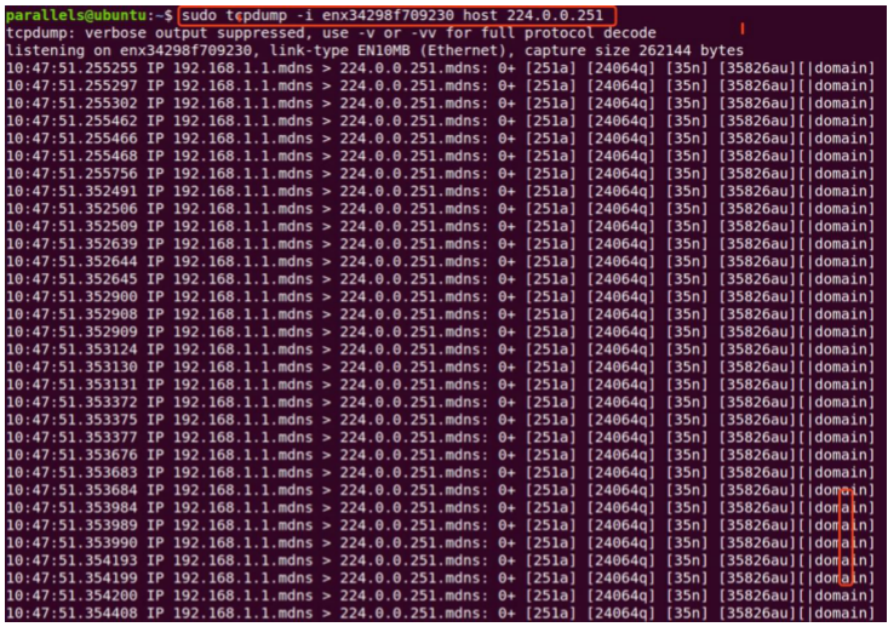

图 P5-10: tcpdump

4.6 收发数据包小程序

本次实验中，我们为同学们提供了一个简单的收发数据包的小程序 pktRxTx。

由于小程序的使用说明篇幅较长，在任务书中长篇大论地说明难免有喧宾夺主的嫌疑，因此我们在任务书中就不赘述了。具体使用说明详见本实验的附件资源《pktRxTx使用说明》。

5 任务一：实现网卡初始化与轮询发送数据帧的功能

5.1 实验要求

本次实验需要通过系统调用，实现网络层发送数据包函数 `sys_net_send`，并且能在 shell 中正确执行测试程序 `send` 和 `fly`。实现 `sys_net_send` 时，需要调用 `e1000_transmit` 将网络层数据包转化为数据链路层传输的数据帧进行发送。

本次实验需要使用 wireshark 或 tcpdump 软件进行抓包，查看网卡有没有把数据包发送出来，如图P5-11所示，为发包成功的参考图。

另外，大家的驱动程序需要支持发送的数据帧数目大于描述符的数量的情况，这会作为一个检查点，请大家注意。显然，这样大家会面临当用户进程调用 `sys_net_send` 的时候，可用的发送描述符和帧缓冲区已经不足以本次发送的这种情况。要解决这个问题，一种简单的方法是陷入内核之后一直轮询是否有数据帧已经传输完毕归还描述符，有则继续传输；但这样会导致一个进程一直占用 CPU（别忘了，CPU 陷入内核之后默认是关中断的），与操作系统的设计理念不符，所以比较合理的方法应该是阻塞用户进程直到有描述符和缓冲区可用，但这需要利用网卡中断，且听下回分解。在任务一中，大家只支持前一种方法也可以。不过，有一个易实现的更好的策略是，在描述符和缓冲区不足时阻塞用户进程，每次定时器中断触发时都将其唤醒，但唤醒后重新检查可用描述符数量直到足够，否则继续阻塞。

5.2 实验步骤

1. 实现 `ioremap` 函数，并在 `main.c` 中调用该函数以建立 e1000 寄存器的虚实地址映射。

2. 实现 `e1000_configure_tx` 函数，该函数用于 E1000 发送初始化。

3. 实现 `e1000_transmit` 函数，该函数用于数据帧的发送。

4. 实现 `do_net_send` 函数，该函数用于网络层数据包的发送，并将其封装为系统调用 `sys_net_send`。

5.3 注意事项

对于前面的 Project 只完成了 S-core 的同学来说，这里可能会遇到之前没有实现系统调用的问题，那么这里可以直接只用 `do_net_send` 函数。

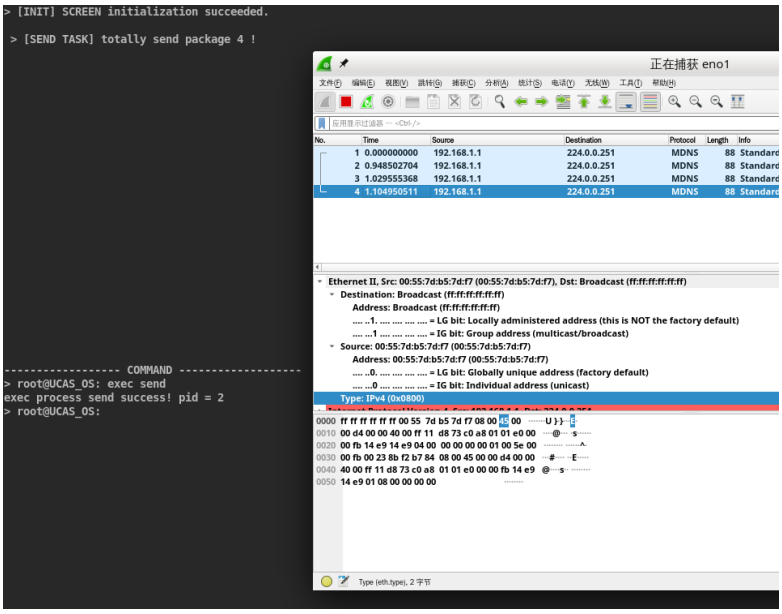

图 P5-11: 发包效果示意图

想详细了解网卡驱动相关寄存器或者在实验中遇到了问题可以详细阅读《8254x_GBe_SDM》[1] 手册中的内容。这里有一个调试的小建议，大家可以手动打印出 E1000 各寄存器的数值来查看寄存器是否设置正确（可以使用 `printl`），也可以结合 wireshark 或 tcpdump 来查看数据包发送的状况。

需要注意的是，网卡是不认识虚地址的。所以，请注意所有给网卡看的地址都是**物理地址**。另外，由于 DMA 和 RISC-V 处理器都需要读写描述符以及发送缓冲区，所以必须保证读写的 cache 一致性，以确保 DMA 设备和 RISC-V 处理器都能看到双方的修改。这需要借助于 RISC-V 的 `fence` 指令。在 start code 中给了一段写好的嵌入式汇编的 `fence` 指令的示例：

```
#define RISCV_FENCE(p, s) \
        __asm__ __volatile__ ("fence " #p "," #s : : : "memory")

/* These barriers need to enforce ordering on both devices or memory. */
#define mb()            RISCV_FENCE(iorw,iorw)
#define rmb()           RISCV_FENCE(ir,ir)
#define wmb()           RISCV_FENCE(ow,ow)
```

在 start code 中也为大家封装好了 `flush_dcache` 函数，请大家在读描述符/缓冲区之前、写描述符/缓冲区之后，调用该函数来刷 dcache。

除此之外，本次实验当中的 ioremap 映射的地址是内核地址，也就是说只有内核才能够访问，所以需要像建立内核页表一样将页表项的 `A/D` 位都拉高。同时，在本实验当中的 `do_net_send` 和下文将会提到的 `do_net_recv` 函数直接在内核中访问用户态地址。为了避免在内核当中发生例外，因此需要将页表项的 `A/D` 位都置上。总的来说，就是大家建立页表的时候，需要将所有页表项的 `A/D` 位都置为 1.

6 E1000 接收数据帧

接下来，我们来讲一讲 E1000 是如何接收数据帧的。与发送数据帧类似，接收数据帧同样需要描述符的帮助，同样需要关注描述符的格式、组织、接收使能三个核心点。相信大家有了处理发送描述符的经验之后，处理接收描述符应该也是轻而易举了。

这一章节的内容对应 E1000 手册 [1] 第 3.2、14.4 两小节，感兴趣的同学可以自行查阅。

6.1 E1000 接收描述符

接收描述符仅有一种布局，其布局如P5-12图所示。可以看到，单个 E1000 接收描述符大小同样为 16 Bytes，且包含了 Buffer Address、Length 和 Status 等字段。其中，白色的 Buffer Address 为软件提供，表示 DMA 控制器收到数据帧之后，需要把数据帧存放到的内存地址（当然，这里也是物理地址）；阴影部分则由硬件在收到数据帧后自动填写，软件在把接收描述符转交给硬件之前，需要把这个描述符的阴影部分清零。这也是十分自然的分工方式，试想一下，在数据帧还没收到的情况下，软件怎么可能知道诸如 Length 字段的信息呢?

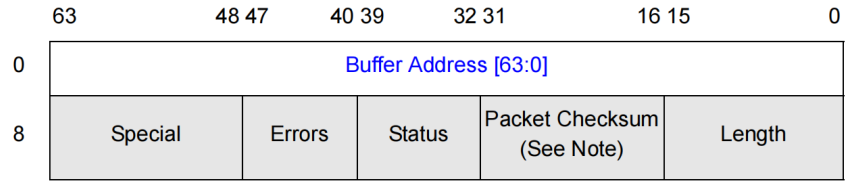

图 P5-12: E1000 接收描述符

对于接收描述符的 STA (status) 字段而言，其布局如图P5-13所示。该字段包含了当前接收描述符的状态信息，例如：填入描述符的数据帧是否是数据包的最后一个帧 (STA.EOP, End Of Packet)；硬件是否完成了对当前接收描述符的处理 (STA.DD, Descriptor Done)。值得注意的是，对于一个数据包拆分成多个数据帧的情况，如果接收描述符的 STA.EOP 没有设置的话，那么描述符内只有 Buffer Address、Length 和 STA.DD 有效。也就是说，DMA 控制器只会把接收数据帧的状态信息存放在接收到的最后一个帧的描述符内，这样可以避免存储冗余信息，一定程度上节约内存带宽。

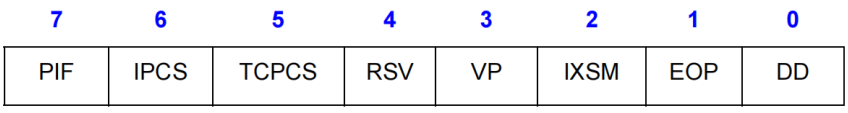

图 P5-13: E1000 接收描述符 STA 字段

除了上述提到的这些字段以外，接收描述符的其他字段在本实验中不会涉及到，就不再赘述了。

6.2 接收描述符循环数组

与发送描述符类似，E1000 网卡也采用如图P5-14的循环数组形式组织接收描述符，发送/接收循环数组的管理方式也大同小异。大家可以与前面的4.2一节对照着看，以加深对描述符数组的理解。

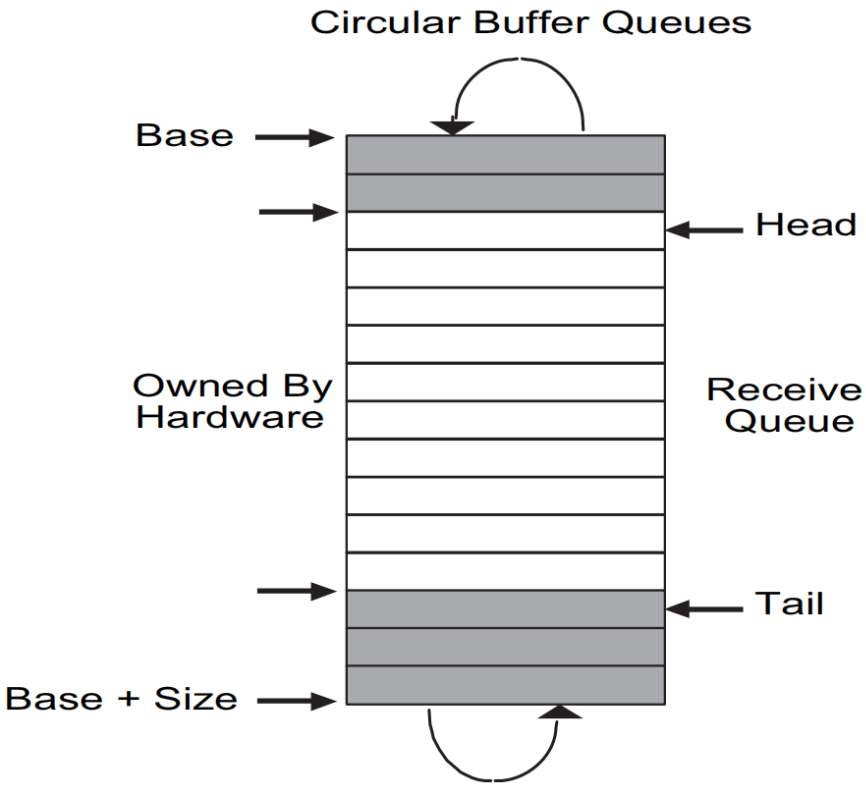

图 P5-14: E1000 接收描述符循环队列

MAC 内部寄存器中有 RDBAL、RDBAH 和 RDLEN 三个寄存器，用来告知 DMA 控制器接收描述符循环数组的位置和长度。RDBAL/RDBAH (Rx Descriptor Base Address Low/High) 分别存储循环数组基地址的低/高 32 位，RDLEN (Rx Descriptor Length) 则存储循环队列所占字节数，以便 DMA 控制器计算出循环数组的元素个数（接收描述符的大小也是 16 字节）。再次提醒大家，填入 RDBAL/RDBAH 的地址仍然是物理地址。

接收描述符循环数组也使用了 Head 和 Tail 头尾指针来划分循环数组。如图P5-14所示，白色部分为硬件占用的描述符，阴影部分描述符则由软件进行管理，而 Head 和 Tail 指针则分别指向硬件/软件正在处理的描述符。当 DMA 控制器接收到新的数据帧的时候，如果硬件拥有接收描述符，那么 DMA 控制器就会将数据帧和接收的信息填入描述符和其关联的接收缓冲区中，并后移 Head 指针，将该接收描述符交给软件处理；而如果所有接收描述符都被软件占用的话，DMA 控制器就只好丢弃这个数据帧了。如果软件完成了描述符上数据的接收，那么软件就会将该描述符的状态部分清空，并将 Tail 指针后移一格，转交给硬件进行控制。

类似的，MAC 内部寄存器中也有 RDH (Rx Descriptor Head) 和 RDT (Rx Descrip-

tor Tail) 两个寄存器来分别存储 Head 和 Tail 指针的数值。同样，RDH 和 RDT 寄存器存储的是接收描述符元素的索引。

此外，与 TDH 类似，E1000 手册 [1] 也建议大家不要直接读取 RDH 的值，因为软件可能刚读完，硬件就移动了 Head 指针。因此，要判断数据帧是否接收成功，最好的办法就是查看描述符的 STA.DD 位是否为 1。

6.3 E1000 接收使能

在 MAC 内部寄存器中，有一个全局的接收控制寄存器 RCTL (Receive Control)。该寄存器的 EN 位控制了接收数据帧的全局使能，即当 RCTL.EN 为 0 时，禁止 DMA 控制器接收新的数据帧。

当 RCTL.EN 被使能时，DMA 控制器也不会盲目接收数据链路层上所有的数据帧，而是只接收广播数据帧、以及发给自己的数据帧。那么，如何区分数据帧是否满足接收条件呢？大家以前可能或多或少听过 MAC 地址这个名词。MAC 地址 (Media Access Control Address) 也称为以太网地址或物理地址，用于在网络中唯一标示一个网卡，由网络设备制造商在生产网卡时烧录在其 EEPROM 中。MAC 地址的长度为 48 位，通常表示为 12 个 16 进制数，如：`00-16-EA-AE-3C-40` 就是一个 MAC 地址，`FF-FF-FF-FF-FF-FF` 是一个广播 MAC 地址。只要不更改自己的 MAC 地址，MAC 地址在世界上就是唯一的。形象地说，MAC 地址就如同身份证上的身份证号码，具有唯一性。

E1000 网卡的 MAC 地址存放在 EEPROM 中，需要从 EEPROM 中将其读取出来，再写入 MAC 内部寄存器的 RAL0/RAH0 (Recieve Address Low/High 0) 中，并设置 RAH0.AV (Address Valid) 为 1。DMA 控制器在接收数据帧的时候，会判断数据帧以太网报头的 MAC 地址是否与 RAx（16 组 RAL、RAH 寄存器）中记录的 MAC 地址相匹配。为了简化实现，start code 里面为大家提供了固定好的 MAC 地址，大家只需要把这个地址写入到 RAL0 和 RAH0 就行了，不用去读取 EEPROM。除此之外，为了支持广播数据帧的接收，请大家注意设置 RCTL.BAM (Broadcast Accept Mode) 为 1。

匹配了 MAC 地址之后，DMA 控制器还要找地方存放接收到的数据帧。正所谓有朋自远方来，不亦说乎，但得找地方给人家住。如果 RDH 不等于 RDT，即 DMA 控制器拥有可以存放数据帧的接收描述符，那么当数据链路层有新的数据帧到来时，DMA 控制器就会把数据帧的数据部分自动填入描述符对应的接收缓冲区，然后填写描述符的接收状态之后，后移 RDH，以将该描述符交给软件处理。也就是说，和发送数据帧的使能相比，触发 DMA 控制器接收的条件是数据链路层有新的数据帧到来，这是一个被动的条件。当然，如果 RDH 等于 RDT 的话，那么即使有新的数据帧到来，DMA 控制器也没地方安置这位远道而来的客人，只能说：“拜拜了您嘞！”。

此外，为了防止接收数据帧时接收缓冲区越界，导致 DMA 攻击，E1000 网卡在 RCTL 里面规定了接收缓冲区的大小。在本实验中，大家需要设置 RCTL.BSEX (Buffer Size Extension) 为 0，RCTL.BSIZE (Buffer Size) 为 0，以表明接收缓冲区大小为 2048 字节。

6.4 E1000 接收初始化

对于 E1000 的接收初始化，同学们只需要关心接收描述符和相关的 MAC 内部寄存器的初始化即可。其余涉及到 PHY 等的与课程无关的部分，各位无需多虑，我们在 BIOS 内已经帮大家提前设置好了。

在本实验中，E1000 接收初始化的大致流程如下所示，更加详细的接收初始化流程详见 E1000 手册 [1] 的 14.3 节，感兴趣的同学可以自行查阅。

1. 将给定的 MAC 地址填入 RAL0 和 RAH0 寄存器。
2. 初始化接收描述符循环队列。
3. 初始化 RDBAL、RDBAH、RDLEN、RDH、RDT 寄存器。
4. 初始化 RCTL 寄存器，设置 RCTL.EN 为 1、RCTL.BAM 为 1、RCTL.BSEX 为 0、RCTL.BSIZE 为 0。

7 任务二：实现网卡初始化与轮询接收数据帧的功能

7.1 实验要求

本次实验需要通过系统调用，实现网络层数据包接收函数 `sys_net_recv`，并且能在 shell 中正确执行测试程序 recv 和 fly。实现 `sys_net_recv` 时，需要调用 `e1000_poll` 将数据链路层的数据帧转化为网络层的数据包，交付给网络层。

本次实验需要使用 pktRxTx 小程序对 QEMU/开发板发送测试数据包，启动交互界面的命令为 `sudo ./pktRxTx -m 1`，然后在界面中输入 `send <` *发送数据包的数目* `>` 即可。

当然，大家会遇到当用户进程调用 `sys_net_recv` 时，尚无数据帧到达的情况。所以要想让用户进程能收到数据的话，一种简单方法是陷入内核之后一直轮询是否有数据帧已经接收到，有则取走数据并返回用户态；但这同样会有陷入内核后进程始终占用 CPU 的问题，所以也需要阻塞直到有一定量的数据帧到达。这也要利用网卡中断，且听下回分解。在本任务中使用前一种方法也是可以的。不过，相对好一些的方法仍然是与任务一类似，阻塞进程并在定时器中断时唤醒，唤醒后重新检查是否有数据帧到达。

相似的，大家的驱动程序需要支持接收的数据帧数目大于描述符的数量这种情况，这也会作为一个检查点，请大家注意。

此外，由于测试程序 recv 比较简单，接收包的个数以及打印格式都是固定的，不便于查看接收情况，所以这里又为大家准备了增强版的测试程序 recv2，适用于 P3 完成了至少 A-core（支持字符串命令行参数）的同学。直接使用 `exec recv2` 启动它时，它与 recv 没什么两样，但它支持可选的命令行参数，格式为：`recv2 -n[NPKT] -l[LLIM] [-c]`。其中 `[NPKT]` 是一次接收包的个数，`[LLIM]` 是每个包的最大打印长度（避免覆盖整个屏幕），`-c` 选项使得打印格式是字符而非十六进制。根据以往经验，大家把命令行参数取为 `recv2 -n72 -l80 -c`，然后用 pktRxTx 发包小程序发送 72 个以上的包，看到下面P5-15上半部分的打印效果即可。

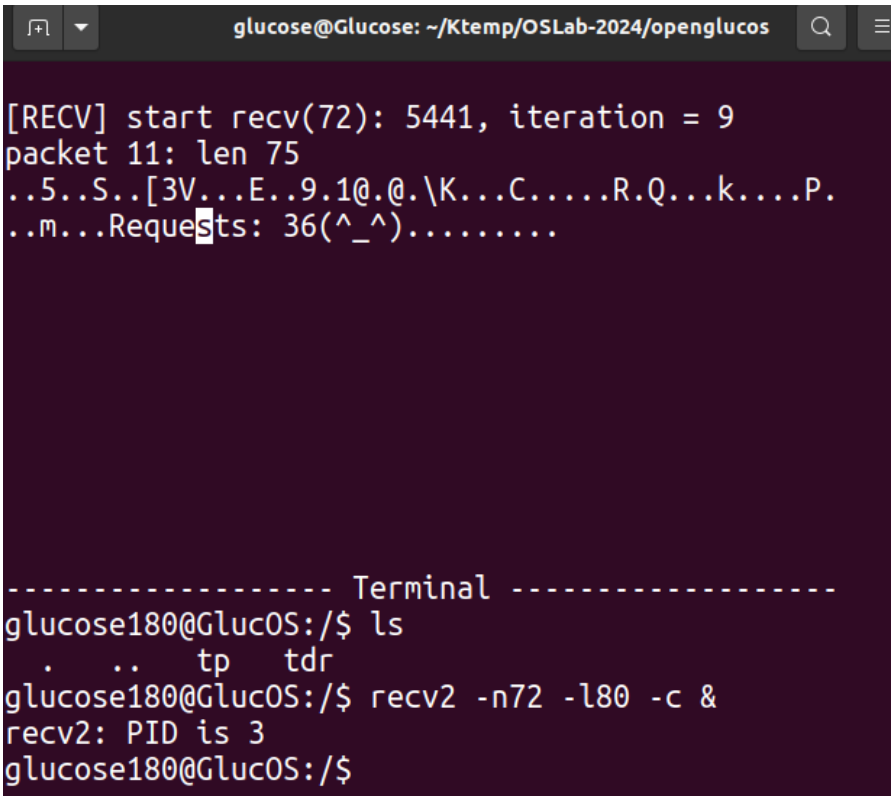

图 P5-15: 测试程序 recv2 的效果

7.2 实验步骤

1. 实现 `e1000_configure_rx` 函数，该函数用于 E1000 接收初始化。
2. 实现 `e1000_poll` 函数，该函数用于数据帧的接收。
3. 实现 `do_net_recv` 函数，该函数用于网络层数据包的接收，并将其封装为系统调用 `sys_net_recv`。

7.3 注意事项

对于前面的 Project 只完成了 S-core 的同学来说，这里可能会遇到之前没有实现系统调用的问题，那么这里可以直接只用 `do_net_recv` 函数。

需要再次强调的是，所有给网卡看的地址都是**物理地址**。另外，为了保证描述符和接收缓冲区的 cache 一致性，请大家在读描述符/缓冲区之前、写描述符/缓冲区之后，调用 `flush_dcache` 函数来刷 dcache。具体的细节请参考任务书5.3节。

8 网卡四级中断流程

在 Project 2 中，大家已经接触过定时器中断 (`IRQ_S_TIMER`) 的例外处理流程了，在 Project 3 中也对核间中断 (`IRQ_S_SOFT`) 有了一定的了解。在 RISC-V 的中断定义中，定时器中断和核间中断都属于局部中断 (Local Interrupt)，连接在核级中断控制器 (CLINT, Core-Level INTerrupt controller) 上；而大家在本次任务中所要接触的网卡中

断，则是连接在系统级中断控制器 (PLIC, Platform-Level Interrupt Controller) 上的。E1000 网卡产生中断信号之后，首先要发送给 PLIC，再由 PLIC 向处理器发送外部中断 (`IRQ_S_EXT`) 信号。

如图P5-16所示，CLINT 和 PLIC 最大的区别在于，CLINT 没有仲裁，包括 software 和 Timer，一有中断马上响应（software 中断怎么产生的：用软件直接写一个寄存器当作软件中断）；而 PLIC 需要一个仲裁决定谁先中断，存在一个优先级的问题 [2]。因此，CLINT 与 PLIC 的不同之处，也决定了大家处理 E1000 网卡中断的流程势必不会与定时器中断完全一样。那么下面就让我们来看看网卡中断的处理流程吧。

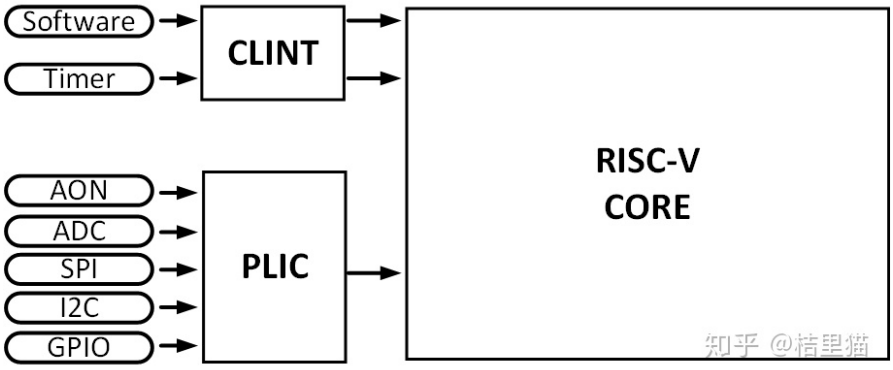

图 P5-16: RISC-V 中断控制器架构图 [2]

第一级：各种例外的总入口，在我们的实验课中就是 stvec。每当 CPU 发现一个应在 S 态下处理的例外时，硬件就自动跳转到该入口地址上。

第二级：各种例外的入口，在我们的实验课中需要通过 `scause` 进行判断。例如对应于定时器中断，`scause` 就是 `IRQ_S_TIMER`，操作系统就根据 `scause` 调用定时器中断的处理函数；而对于外设中断，`scause` 就是 `IRQ_S_EXT`，操作系统就调用外设中断的处理函数。在这一级中，如果是定时器中断的话，操作系统已经能够分析出例外的源头了；而对于外设中断而言，这还远远不够，因为还不知道具体是哪个外设导致的哪种中断，因此需要继续往下走。

第三级： PLIC 中断处理入口。由于所有外设的中断信号都连接在 PLIC 上，因此这一级中我们需要与 PLIC 打交道，通过 `plic_claim`（下一节会细讲）来获取是哪一个外设发出了中断。如果是 E1000 网卡的话，`plic_claim` 就会查询到 E1000 网卡外设的 ID 号，并跳转到 E1000 中断处理入口。

第四级： E1000 中断处理入口。外部设备通常不止有一种中断类型，E1000 网卡也一样。E1000 网卡包含了 4 种接收中断和 5 种发送中断。在这一级中，大家将会读取 E1000 的中断原因寄存器 ICR (Interrupt Cause Read)，判断具体是哪一种 E1000 网卡中断。

8.1 PLIC 简介

PLIC 的整体架构图如图P5-17所示。可以看出来，PLIC 接入各个外设的中断信号之后，会使用多路选择器来选择出当前优先级最高、且中断被使能的外设中断，并向中

断目标 (interrupt target，即处理器核) 发送外设中断信息，设置处理器核的 xIP.xEIP 位。

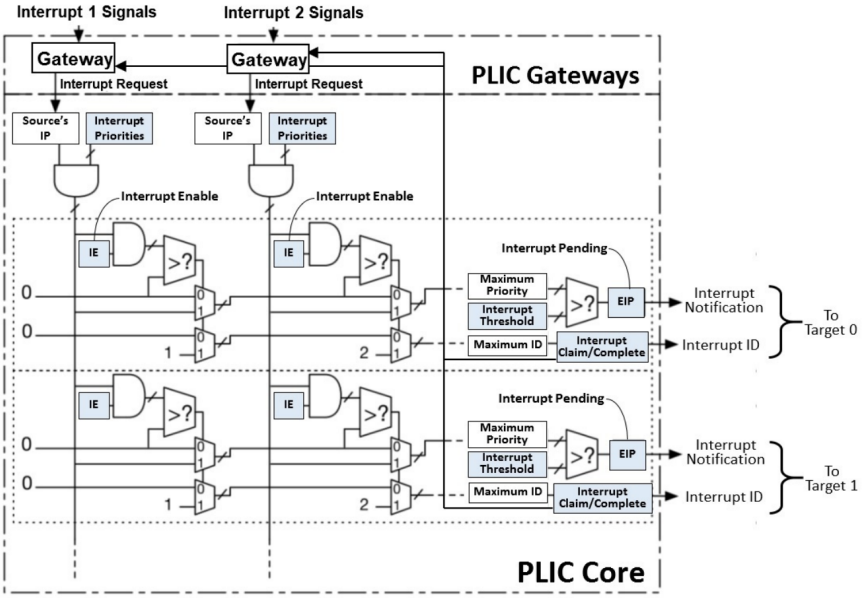

图 P5-17: PLIC 整体架构图

8.2 PLIC 初始化

正如上文所言，一个外设中断想要被 PLIC 选中，就必须满足开启使能、优先级大于 threshold 这两个条件。这也是 PLIC 初始化所需要做的事情——设置中断优先级 threshold、对需要用到的外设中断使能并设置优先级。

本次实验中，PLIC 的初始化代码已经在 start code 中为大家提供好。感兴趣的同学可以自行查阅 PLIC 手册 [3]，了解完整的初始化流程。

8.3 PLIC 中断处理

PLIC 的中断处理流程如图P5-18所示,本次实验课中,大家需要完成其中的"Handler Running" 部分。

中断源 (Interrupt Source) 对应于各个 I/O 外设，各个中断源的中断信号都统一发送到 PLIC Gateway 上。Gateway 作为 PLIC 的中继，负责将中断源的各类中断信号转换为 PLIC core 的通用格式，同时控制向 PLIC core 发起 request 的整个流程。

PLIC core 负责所有中断请求的仲裁和分发。任何时候，最多只能有一个 pending interrupt request 存在于 PLIC core 中，由对应 IP 位保存。PLIC core 对每个 interrupt source 都会被赋予一个独立的 priority 和 ID 标识,也包含一个 matrix of interrupt enable bits IE, 用以控制 interrupt 的使能。此外, PLIC core 对每个中断目标 (Interrupt Target) 都赋予一个独立的平台相关的 priority threshold 寄存器，用以控制 interrupt 发生的门限。

当某一中断满足发送条件（使能、门限、优先级）时，PLIC Core 就会向中断目标发送外部中断信号 EIP。EIP 发送给中断目标的过程被称为 interrupt notification。如

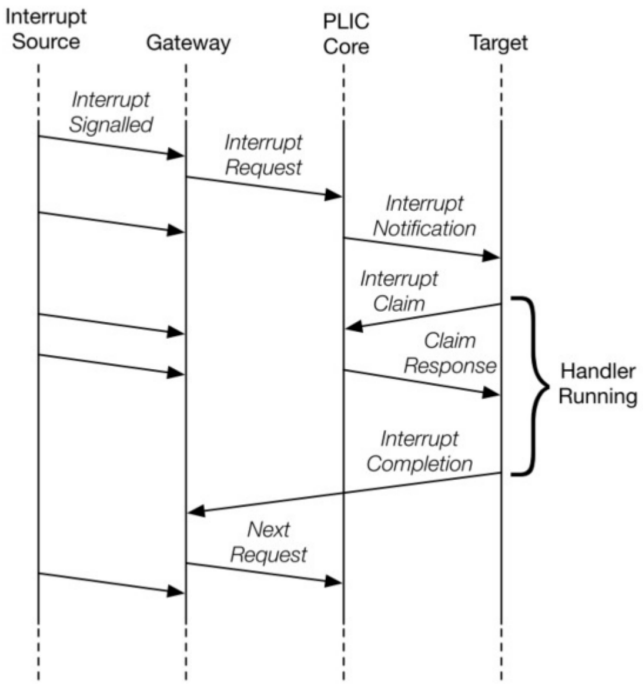

图 P5-18: PLIC 中断处理流程

果中断目标是 RISC-V 核，notification 会赋予对应特权态的 xeip bit。一个简单的实现是将 PLIC 的 EIP 直接硬链接到中断目标的 xeip 上；复杂些的设计可以通过 message 实现。

当中断目标接收到 notification 时（对于 RISC-V 处理器，则是引发了一个外部中断例外），中断目标需要进行 interrupt claim，即读取 PLIC 的 claim 寄存器，来表示接受 PLIC 的中断信号。PLIC Core 收到 claim 之后，会选出最高优先级的 source ID，并将其对应的 IP 位清除。这个 ID 会发送给中断目标，即 claim response。如果 ID 为 0，表示没有需要处理的中断。

中断目标通过读取 claim 寄存器，接收到 PLIC 传来的 ID 号之后，就能知道具体是哪一个外设发生了中断。对于本次网卡实验，E1000 网卡在 QEMU 上的 ID 为 33，在开发板上的 ID 为 3，这主要是因为二者所使用的设备树不同。假设是 E1000 网卡发生了中断，匹配了 ID 之后，大家就可以读取 E1000 ICR 寄存器，来查询 E1000 网卡具体的中断信息并进行处理了。

当 E1000 网卡中断处理完毕之后，中断目标需要向 PLIC 的 complete 寄存器写入先前读取 claim 寄存器获得的 ID 号，以表示当前中断处理完毕。值得注意的是，如果大家忽略了这一步，那么 PLIC 就会认为当前的中断还没处理完，就不会处理下一个中断信号了。

8.4 E1000 网卡中断

E1000 网卡中断包含了 4 种接收中断和 5 种发送中断。而在本次实验课中，大家只需要实现 RXDMT0 接收中断和 TXQE 发送中断就可以了，板卡上实现的 E1000 LITE 模块也仅支持这两种中断类型。

E1000 中断寄存器

本次实验中，大家将会接触到 E1000 MAC 内部寄存器中的四个中断寄存器：ICR、ICS、IMS 和 IMC。以下分别对它们进行介绍：

ICR (Interrupt Cause Read) 寄存器

ICR 寄存器记录了各个网卡中断触发的情况。当某个网卡中断的条件满足时，该中断在 ICR 寄存器中对应的比特将会被硬件自动置一。同时，当 ICR 寄存器中有任何一位为 1，且其在 IMS 中对应位也为 1 时，PLIC 上 E1000 网卡对应的 pending bit 就会被自动置 1，从而向处理器核发送一个外部中断信号。

ICR 寄存器在被读取的时候，其内所有的寄存器位都会被自动清除。因此，读取这个寄存器会隐式地承认 (acknowledge) 任何挂起的中断事件。此外，在 ICR 寄存器中对某一位写入 1 的话，也会清除该位，但写入 0 的话对该位没有任何影响。

ICS (Interrupt Cause Set) 寄存器

ICS 寄存器用于让软件来设置一个中断条件，其内任何位被写入 1 之后，都会在 ICR 寄存器中置一相应的中断位，从而在 IMS 对应位也为 1 的情况下，产生一个外部中断。与 ICR 寄存器类似，向 ICS 寄存器任一位写入 0 的话，不会对 ICR 寄存器产生任何影响。

IMS (Interrupt Mask Set) 寄存器

IMS 寄存器用于软件使能各个 E1000 网卡中断：如果其对应的掩码位设置为 1，则启用该中断；如果其对应的掩码位设置为 0，则禁用该中断，即使 ICR 相应位置一，也不会产生外设中断。

软件可以通过写入 1 至该寄存器中相应的掩码位来启用特定的中断，但需要注意的是，任何用 0 写的位都没有变化。因此，如果软件希望禁用先前已启用的特定中断条件，那么它必须向 IMC 寄存器相应位写入 1，而不是在该寄存器中写入一个 0。此外，读取此寄存器将返回具有中断掩码集的位，不会像 ICR 那样清零该寄存器本身。

IMC (Interrupt Mask Clear) 寄存器

IMC 寄存器用于软件禁用网卡中断。当软件向该寄存器的某一位写入 1 时，IMS 寄存器的相应位将会被硬件自动清零；而软件向该寄存器某一位写入 0 时，则不会对 IMS 寄存器产生任何影响。

E1000 TXQE 发送中断

TXQE (Transmit Queue Empty) 发送中断顾名思义，就是当硬件占有的发送描述符数量为 0 时触发，即 TDH == TDT，表明当前轮次的数据帧发送已经结束，需要软件给硬件分配更多的发送描述符进行新一轮的传输。

E1000 RXDMT0 接收中断

RXDMT0 (Receive Descriptor Minimum Threshold) 接收中断指的是，当硬件所持有的接收描述符占全部的比例小于某一特定值时，则触发该中断，提醒软件应当及时取走已接收的数据帧，并为硬件分配更多的接收描述符。

上述提到的特定值在 RCTL.RXDMT 位可以进行设置。在本次实验课中，同学们需要设置 RCTL.RXDMT 为 00，从而使得每当硬件所持有的接收描述符占比小于总数的 1/2 时，自动触发 RXDMT0 中断。

9 任务三：有网卡中断的收发包

本实验为 A-core 同学需要完成的任务。

9.1 实验要求

本实验与任务一、二的不同点就是：当进程不满足收发数据包的条件，例如网络上没有新的数据帧时，则进程应当阻塞并主动让出 CPU 控制权给其他进程，等到满足条件后再由中断唤醒之。

具体而言，对于接收进程来说，当没有数据帧到达网卡或到达的数据帧小于给定的数目时，则接收进程将会被阻塞；当有数据帧到达网卡时，接收进程被唤醒，开始打印接收的数据。因此需要注册网卡中断函数，在网卡中断函数中判断是否唤醒接收或发送进程。

发送进程也要做类似的处理：在网卡的发送进程中，当发送描述符循环数组已满时，则发送进程将会被阻塞；当数据帧发送完成时唤醒发送进程。

9.2 实验步骤

1. 完成 E1000 TXQE 和 RXDMT0 的网卡中断处理函数 `e1000_handle_txqe` 和 `e1000_handle_rxdmt0`。

2. 在 `irq.c` 中注册外部中断 `IRQ_S_EXT` 的处理函数 `handle_irq_ext`，该函数需要结合 PLIC claim 和 completion 操作，完成对网卡中断的处理。

3. 在 `e1000.c` 或者 `net.c` 中的合适位置，使能 TXQE 与 RXDMT0 中断。

10 任务四：双端口监听

本任务依然是 C-core 的同学需要完成的任务。同学们需要自行实现测试程序实现如下功能：

该测试程序会启动两个线程，分别接收两个端口，即在现有的接收接口上增加一个端口参数。内核在网卡中断的时候根据包的内容判断该包发给哪个接口，然后唤醒对应的监听线程。包的格式为 **以太网报头（14 字节）+IP 报头（20 字节）+TCP 报头（20 字节）+ 数据 payload**，端口部分是 TCP 报头的 dport 字段（TCP 报头开始的第 3、4 字节，uint16_t 类型），设置了两个定死的值 50001 和 58688。这两个端口值没有特殊的含义，只是随机的生成了这两个数字。我们这里实现设备驱动并不需要关心网络包的具体含义，仅满足网络包的格式即可，TCP/IP 协议的相关知识请大家自行查阅计算机网络的知识。在使用 pktRxTx -m 1 命令之后，pktRxTx 程序就会随机发出这两

种端口号的数据包。要求两个线程都能接收到数据，不能出现某个线程一段时间饿死的情况。

11 注意事项

以太网默认最大传输单元 MTU 为 1500，所以**数据 payload** 部分长度不能超过 1500 字节。

实测板卡环境中会有杂包干扰传输，一个解决方案是将**数据 payload** 的第一个字节设为一个固定的 magic code，比如 `0x42`。通过检查收到的包中是否包含 magic code 来过滤杂包。

同学们在使用 qemu 调试此任务时可以同时运行两个 qemu 进程，在这两个 qemu 进程间完成对传。以下是 Makefile 中 make run-net 和 make debug-net 的配置：

```
QEMU            = $(DIR_QEMU)/riscv64-softmmu/qemu-system-riscv64
QEMU_OPTS       = -nographic -machine virt -m 256M -kernel $(UBOOT) -bios none \
                  -drive if=none,format=raw,id=image,file=${ELF_IMAGE} \
                  -device virtio-blk-device,drive=image \
                  -monitor telnet::45454,server,nowait -serial mon:stdio \
                  -D $(QEMU_LOG_FILE) -d oslab
QEMU_DEBUG_OPT  = -s -S
QEMU_SMP_OPT    = -smp 2
QEMU_NET_OPT    = -netdev tap,id=mytap,ifname=tap0,script=${DIR_QEMU}/etc/qemu-ifup,downscript=${DIR_QEMU}/etc/qemu-ifdown \
                  -device e1000,netdev=mytap
run-net:
    -@sudo kill `sudo lsof | grep tun | awk '{print $$2}'`
    sudo $(QEMU) $(QEMU_OPTS) $(QEMU_NET_OPT) $(QEMU_SMP_OPT)
debug-net:
    -@sudo kill `sudo lsof | grep tun | awk '{print $$2}'`
    sudo $(QEMU) $(QEMU_OPTS) $(QEMU_DEBUG_OPT) $(QEMU_NET_OPT) $(QEMU_SMP_OPT)
gdb:
    $(GDB) $(ELF_MAIN) -ex "target remote:1234"
```

启动两个 qemu 进程并配置网络需要对 Makefile 和 qemu 的文件进行修改。以下是修改提示：

1. 命令 `-@sudo kill `sudo lsof | grep tun | awk '{print $$2}'`` 目的是终止与 tun 相关的进程，第一个 qemu 进程启动后，第二个 qemu 进程启动前请不要执行该命令。

2. `QEMU_OPTS` 的参数 `-drive if=none,format=raw,id=image,file=${ELF_IMAGE}` 中的 `${ELF_IMAGE}` 是同学们的操作系统镜像文件，两个 qemu 进程无法共用一个镜像文件，所以请复制一份镜像文件供第二个 qemu 进程使用。

3. `QEMU_OPTS` 的参数 `-D $(QEMU_LOG_FILE)` 是日志文件，两个 qemu 进程无法共用一个日志文件，所以请指定另一个日志文件供第二个 qemu 进程使用。

4. `QEMU_DEBUG_OPT` 的参数 `-s -S` 等价于 `-gdb tcp:1234 -S`，与 make gdb 的指定端口对应，如果想要同时 gdb 调试两个 qemu 进程，请给第二个 qemu 进程指定不同的端口，并启动两个 gdb 进程，分别指定对应的端口。

5. `QEMU_NET_OPT` 的参数 `ifname=tap0` 指定了宿主机上的虚拟网络接口名称，请给第二个 qemu 进程指定一个不一样的名称，例如 `tap1`。

6. `QEMU_NET_OPT` 的参数 `script=${DIR_QEMU}/etc/qemu-ifup, downscript =${DIR_QEMU}/etc/qemu-ifdown` 表示在启动 qemu 前会运行 script 配置网络环境，在 qemu 退出后会运行 downscript 清理网络环境。qemu-ifup 的代码如下：

```
#! /bin/bash

IFNAME=enp0s3

set -x
if [ -n "$1" ];then
#create bridge, add physical interface to bridge
    ip link set $IFNAME down
    ip link add br0 type bridge
    ip link set br0 up
    ip link set $IFNAME master br0
    ip link set $IFNAME up

#add tap device to bridge
        ip link set $1 up
        sleep 0.5s
        ip link set $1 master br0

#config ip fro bridge
        pkill -9 dhclient
    sleep 5
        dhclient -v br0

        exit 0
else
        echo "ERROR: no interface specified"
        exit 1
fi
```

简单来说 qemu-ifup 创建了一个新的网络桥接 br0，将宿主机的物理网络接口 enp0s3 加入到桥接 br0 中，将指定的 tap 设备添加到桥接 br0 中，强制停止现有的 DHCP 客户端进程，并为桥接接口 br0 请求一个新的 IP 地址。第一个 qemu 进程已经通过 qemu-ifup 配置好了网络环境，所以对于第二个 qemu 进程来说，只需要将自己的 tap 设备添加进桥接 br0 中即可，请为第二个 qemu 进程写一个新的 qemu-ifup，例

如 qemu-ifup2，并指定 qemu-ifup2 的路径为 script 的值，下面是一个 qemu-ifup2 的例子：

```
#! /bin/bash

IFNAME=enp0s3

set -x
if [ -n "$1" ];then
#add tap device to bridge
        ip link set $1 up
        sleep 0.5s
        ip link set $1 master br0

        exit 0
else
        echo "ERROR: no interface specified"
        exit 1
fi
```

同学们如果找不到搭档可以选择一个人使用两块板卡进行对传，下面给出一台电脑连接两块板卡的一种可行方案。

1. 启动虚拟机后连接两块板卡，打开板卡电源，会在虚拟机**设备-USB** 选项中看到如P5-19两个“Xilinx TUL [0700]”。通过点击将两个“Xilinx TUL [0700]”都打上勾，中途可能出现如P5-20的报错，目前不知道怎么解决，但是实测板卡可以连上。在虚拟机终端输入命令 `ls /dev | grep ttyUSB` 来检查板卡是否连接到虚拟机上，如果连接成功应得到如下结果。

```
stu@stu:~$ ls /dev | grep ttyUSB
ttyUSB0
ttyUSB1
ttyUSB2
ttyUSB3
```

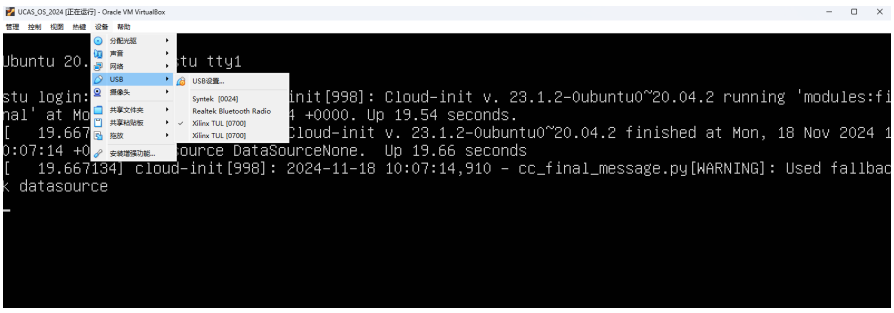

图 P5-19: 连接两块板卡后虚拟机**设备-USB** 选项

2. Makefile 中 make minicom 配置如下

```
TTYUSB1      = /dev/ttyUSB1
```

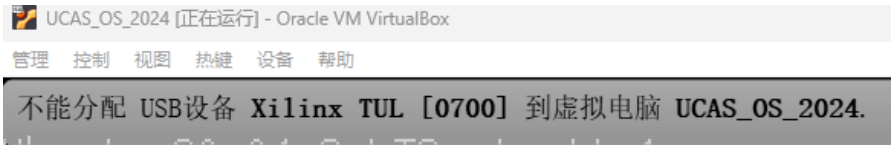

图 P5-20: 连接第两块板卡后虚拟机报错

```
MINICOM          = minicom
minicom:
        sudo $(MINICOM) -D $(TTYUSB1) -X ~/OSLab-RISC-V/oslab-log.txt
```

调试第二块板卡只需要仿照该配置对 `/dev/ttyUSB3` 进行串口调试即可，注意指定一个新的日志文件比如 `/OSLab-RISC-V/oslab-log1.txt`。

参考文献

[1] Intel, "Pci/pci-x family of gigabit ethernet controllers software developer's manual." https://pdos.csail.mit.edu/6.S081/2020/readings/8254x_GBe_SDM.pdf, 2009. [Online; accessed 17-November-2022].

[2] 桔里猫, "Riscv ai soc 实战（五，中断管理）." https://zhuanlan.zhihu.com/p/162888552, 2020. [Online; accessed 18-November-2022].

[3] A. W. etc., "Risc-v platform-level interrupt controller specification." https://github.com/riscv/riscv-plic-spec/releases/download/1.0.0_rc5/riscv-plic-1.0.0_rc5.pdf, 2022. [Online; accessed 17-November-2022].