Lab 6: Network Driver(default final project)

Introduction

This lab is the default final project that you can do on your own.

现在你有了一个文件系统，任何有自尊的操作系统都不应没有网络堆栈。在这个lab，你将写一个驱动用于*network inaterface card*。这个卡基于`Intel 82540EM chip`，也称为`E1000`。

Getting Started

Use Git to commit your Lab 5 source (if you haven't already), fetch the latest version of the course repository, and then create a local branch called lab6 based on our lab6 branch, `origin/lab6`:

```
git commit -m "my solution to lab5"
# nothing to commit, working tree clean
export GIT_SSL_NO_VERIFY=1
git pull
# Already up to date.
git checkout -b lab6 origin/lab6
git merge lab5
#  35 files changed, 1670 insertions(+), 86 deletions(-)
```

其实，网络卡驱动不足以使你的操作系统连接到网络。在新的lab6代码中，我们已经给你提供了一个`network stack`和一个`network server`。在之前的labs中，使用*git*抓取这个实验的代码，合并到自己的代码，查看新的`net/`目录和`kern/`中的新文件。

作为写驱动的补充，你将需要创建一个*系统调用接口*给你的驱动访问权限。你将实现缺失的网络服务器代码来在`network stack`和`driver`之间传输*packets*。你也将通过完成一个*web server*来绑定所有的东西到一起。有了新的*web server*，你将能服务你的文件系统中的文件。

许多内核设备驱动代码你将从头开始编写。和之前的labs相比，这个lab会提供更少的指导：没有骨架文件，没有系统调用接口，很多设计决定取决于你。所以，我们建议你在开始任何单独练习之前阅读整个作业。很多学生觉得这个lab比之前的lab更难。

Lab Requirements

As before, you will need to do all of the regular exercises described in the lab and at least one challenge problem. Write up brief answers to the questions posed in the lab and a description of your challenge exercise in `answers-lab6.txt`.

QEMU's virtual network

我们将使用`QEMU`的用户模式*网络栈*，因为它不需要管理权限来运行。`QEMU`的documentation有更多关于*user-net*。我们已经更新了*makefile*来打开`QEMU`的用户模式*网络栈*和虚拟的*E1000 网络卡*。

`QEMU`默认提供一个虚拟路由器运行在IP*10.0.2.2*并且给`JOS`分配了一个IP地址*10.0.2.15*。为了简化，我们将这些硬编码到了网络服务器，位于`net/ns.h`。

`net/ns.h`

```
#include <inc/ns.h>
#include <inc/lib.h>

#define IP "10.0.2.15"
#define MASK "255.255.255.0"
#define DEFAULT "10.0.2.2"

#define TIMER_INTERVAL 250

// Virtual address at which to receive page mappings containing
client requests.
#define QUEUE_SIZE  20
#define REQVA   (0x0ffff000 - QUEUE_SIZE * PGSIZE)

/* timer.c */
void timer(envid_t ns_envid, uint32_t initial_to);

/* input.c */
void input(envid_t ns_envid);

/* output.c */
```

```
void output(envid_t ns_envid);
```

尽管 `QEMU` 的虚拟网络允许 `JOS` 可以任意连接到Internet， `JOS` 的地址 *10.0.2.15* 在虚拟网络之外没有意义，虚拟网络运行在 `QEMU` 的内部（也就是说， `QEMU` 相当于一个NAT），所以我们不能直接连接到运行在 `JOS` 中的服务器，设置是来自运行 `QEMU` 的主机。为了解决这个问题，我们配置 `QEMU` 在 *host* 机器上的某个端口运行一个 *server* ，该 *server* 只需要简单地连接到 `JOS` 的某个端口，然后将数据在真正的主机和虚拟的网络之间来回运输。

你将在端口 `7(echo)` 和 `80(http)` 运行 *JOS servers* 。为了避免在共享 `Athena` 机器上的冲突， *makefile* 基于你的用户ID为这些计算机生成转发端口。运行 `make which-ports` 来弄清楚 `QEMU` 正在转发哪个端口到你的主机。为了方便， *makefile* 也提供 `make nc-7` 和 `make nc-80` ，来允许你在termianl直接和运行在这些端口上的 *servers* 直接交互。（这些目标仅是连接到一个运行的 `QEMU` 实例；你必须分别开启 `QEMU` 。）

Packet Inspection

makefile 也配置了 `QEMU` 的网络栈来记录所有的 *进入* 和 *出去* 的packets到 `qemu.pcap` ，位于lab目录。

要获取捕获数据包的十六进制/ASCII码转储，像这样使用 `tcpdump` ：

```
tcpdump -XXnr qemu.pcap
```

当然，你也可以使用 `Wireshark` 来图形化地查看 *pcap* 文件。 `Wireshark` 也知道如何去解码和查看成百上千的网络协议。如果你在 `Athena` 上，你将不得不使用 `Wireshark` 的前身， `ethereal` ，位于 *sipbnet locker* 。

Debugging the E1000

我们很幸运能使用模拟硬件。因为 `E1000` 在软件中运行，模拟的 `E1000` 可以向我们汇报，以用户可读的格式，汇报其内部状态和遇见的任何问题。通常一个使用裸机编写驱动的开发人员无法享受这种奢侈。

`E1000` 会产生很多的调试输出，所以你不得不打开指定的 *logging channels* 。下面的一些通道你可能会觉得有用：

Flag	Meaning
tx	Log packet transmit operations
txerr	Log transmit ring errors
rx	Log changes to RCTL
rxfilter	Log filtering of incoming packets
rxerr	Log receive ring errors
unknown	Log reads and writes of unknown registers
eeprom	Log reads from the EEPROM
interrupt	Log interrupts and changes to interrupt registers.

举个例子，打开"tx"和"txerr"logging，使用 `make E1000_DEBUG=tx,txerr ....`

> Note: E1000_DEBUG flags only work in the 6.828 version of QEMU.

你可以使用软件模拟硬件进一步进行调试。如果你遇到困难并且不明白为什么 `E1000` 没有按照你期望的方式响应，你可以在 `hw/net/e1000.c` 中查看 `QEMU` 的 `E1000` 实现。

The Network Server

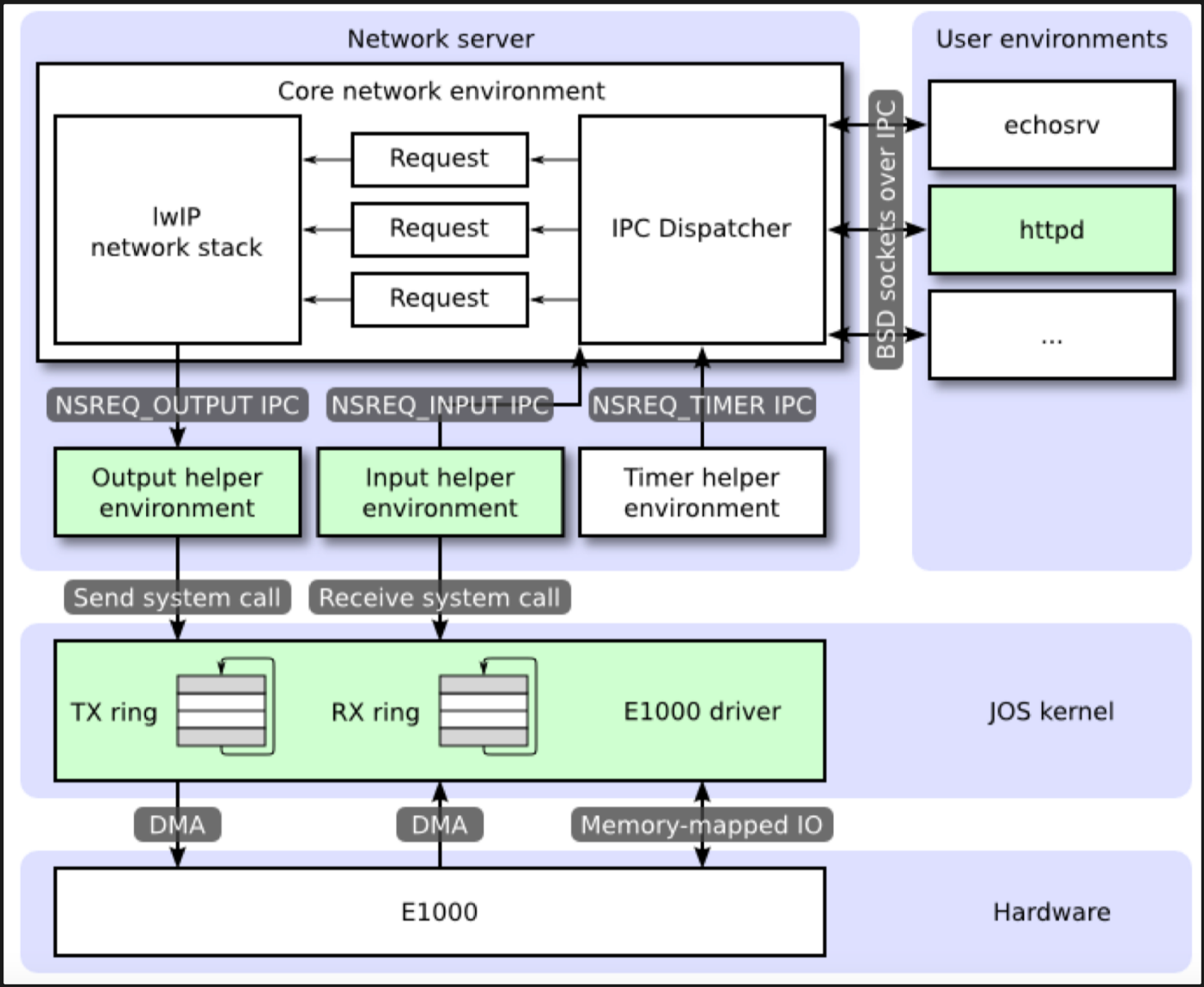
从头开始编写一个网络栈是一个艰难的工作。相反，我们将使用 `lwIP`，一个开源的轻量级的 `TCP/IP` 协议套件，其中包括网络堆栈。你可以在 `lwIP` 找到更多的信息。在这个作业，就我们而言， `lwIP` 是一个黑盒，实现了 `BSD` 套接字接口，并具有数据包输入端口和数据包输出端口。

网络服务器实际上是四个环境的结合：

- 核心网络服务器环境（包括 *套接字调用转换程序* 和 `lwIP`）
- 输入环境
- 输出环境

- 计时器环境

下面的图示展示了不同的环境和它们的关系。图示展示了整个系统包括 *设备驱动器* ，稍后将对此介绍。在这个lab，你将实现绿色高亮部分。



The Core Network Server Environment

核心网络服务器环境 由套接字调用转发器和 `lwIP` 自身组成。*socket call dispatcher* 工作方式基本和文件服务器相同。用户环境使用 *stubs* (found in `lib/nsipc.c`)来发送IPC信息给 *核心网络服务器环境* 。如果你查看 `lib/nsipc.c` 你将发现网络服务器和文件服务器使用相同方式创建 *NS* 环境，使用 `NS_TYPE_NS` 。

`lib/nsipc.c`

```
// Send an IP request to the network server, and wait for a reply.
// The request body should be in nsipcbuf, and parts of the response
// may be written back to nsipcbuf.
```

```
// type: request code, passed as the simple integer IPC value.
// Returns 0 if successful, < 0 on failure.
static int
nsipc(unsigned type)
{
  static envid_t nsenv;
  if (nsenv == 0)
    nsenv = ipc_find_env(ENV_TYPE_NS);

  static_assert(sizeof(nsipcbuf) == PGSIZE);

  if (debug)
    cprintf("[%08x] nsipc %d\n", thisenv->env_id, type);

  ipc_send(nsenv, type, &nsipcbuf, PTE_P|PTE_W|PTE_U);
  return ipc_recv(NULL, NULL, NULL);
}
```

所以我们扫描 `envs`，寻找这个特殊的环境类型。对于每个用户环境IPC，网络服务器中的转发程序调用合适的*BSD 套接字接口* 函数由 `ilIP` 代表用户提供。

通常用户环境不直接使用 `nsipc_*` 调用。相反，它们使用在 `lib/sockets.c` 中的函数，提供了基于一个 *文件描述符的套接字* API。因此，用户环境通过文件描述符指向套接字，就像它们指向磁盘上的文件。

一系列的操作(connect, accept, etc.)被指定为套接字，但是read, write, close则通过 `lib/fd.c` 中的普通文件描述符调度代码。

很像文件服务器为所有打开的文件维护一个内部独一无二的ID， `lwIP` 也为所有打开的套接字生成独一无二的ID。在文件服务器和网络服务器中，我们都使用存储在 `struct Fd` 中的信息将每个环境的文件描述符映射到这些独一无二的ID空间。

尽管看起来文件服务器和网络服务器的*IPC dispatcher* 动作相同，实际上有**很大不同**。

BSD socket 调用，如 `accept` 和 `recv` 可以无限期的阻塞。如果转发器让 `lwIP` 执行其中一个阻塞调用，转发器也将会阻塞并且整个系统在同一时间**只能有一个**未处理的网络调用。因为这不可接受，网络服务器使用 *用户级线程* 来避免 *阻塞整个服务器环境* 。对于每个进入的IPC信息，转发器创建一个线程并在**新创建的线程**里处理请求。如果线程被阻塞，也只有那个线程会置于睡眠状态，而其它线程则继续运行。

作为 *核心网络环境* 的补充，这里有三个辅助环境。除了从用户应用接收信息， *核心网络环境* 的转发器也从输入环境和计时器环境接收信息。

The Output Environment

当服务用户环境套接字调用时，`lwIP`将生成数据包供网卡传输。`LwIP`将发送每个数据包，这些数据包被转运到*output helper environment*，使用`NSREQ_OUTPUT`IPC信息以及数据包所附着的IPC信息的页参数。

*输出环境*负责接收这些消息并且转发这些数据包到设备驱动，通过你即将要创建的系统调用接口。

The Input Environment

由网卡接收的数据包需要被注入`lwIP`。对于每个由设备驱动接收的数据包，输入环境拉取数据包到内核空间之外（使用你将实现的内核系统调用），使用`NSREQ_INPUT`IPC信息将数据包发送到*核心服务器环境*。

数据包输入功能和*核心服务器环境*是分开的，因为`JOS`很难做到同时接收IPC信息和拉取或等待来自设备驱动器的数据包。我们在`JOS`中没有`select`系统调用来允许环境管理多个输入资源并识别出那个输入是准备好被处理的。

如果你查看`net/input.c`和`net/output.c`，你将看到两个文件都需要被实现。这主要是因为实现依赖于你的系统调用接口。在你实现驱动和系统调用接口之后，你将为两个辅助环境写代码。

The Timer Environment

计时器环境周期地发送类型为`NSREQ_TIMER`的消息给*核心网络服务器*，通知它一个计时器已过期。来自这个线程的计时器信息被`lwIP`使用来实现不同的网络超时。

Part A: Initialization and transmitting packets

你的内核对时间没有概念，所以我们需要去添加它。目前硬件每10ms产生一个时钟中断。每个时钟中断我们可以增长一个变量来表示时间已经过了10ms。这在`kern/time.c`中实现，但是还没有完全融入到你的内核。

> Exercise 1.
>
> Add a call to `time_tick` for every clock interrupt in `kern/trap.c`. Implement `sys_time_msec` and add it to `syscall` in `kern/syscall.c` so that user space has access to the time.

```
diff --git a/kern/syscall.c b/kern/syscall.c
index 9ddf69d..4e06f0d 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -459,7 +459,8 @@ static int
sys_time_msec(void)
{
	// LAB 6: Your code here.
-	panic("sys_time_msec not implemented");
+	// panic("sys_time_msec not implemented");
+	return time_msec();
 }

@@ -504,6 +505,8 @@ syscall(uint32_t syscallno, uint32_t a1,
uint32_t a2, uint32_t a3, uint32_t a4,
		return sys_ipc_recv((void*)a1);
	case SYS_env_set_trapframe:
		return sys_env_set_trapframe(a1, (struct
Trapframe*)a2);
+	case SYS_time_msec:
+		return sys_time_msec();
	default:
		return -E_INVAL;
	}
```

```
diff --git a/kern/trap.c b/kern/trap.c
index a04c9d6..54ea93f 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -280,6 +280,9 @@ trap_dispatch(struct Trapframe *tf)
             return;
                case IRQ_OFFSET + IRQ_TIMER:
                        lapic_eoi();
+                       if(thiscpu->cpu_id == 0){
+                               time_tick();
+                       }
                        sched_yield();
                        return;
```

Use `make INIT_CFLAGS=-DTEST_NO_NS run-testtime` to test your time code. You should see the environment count down from 5 in 1 second intervals. The "`-DTEST_NO_NS`" disables starting the network server environment because it will panic at this point in the lab.

The Network Interface Card

编写驱动需要深入了解硬件和提供给软件的接口。lab的文档提供了一个对于如何连接*E1000*的高级概述，但是当你编写你的驱动时，你将需要额外使用英特尔手册。

Exercise 2.

Browse Intel's Software Developer's Manual for the E1000. This manual covers several closely related Ethernet controllers. QEMU emulates the 82540EM.

You should skim over chapter 2 now to get a feel for the device. To write your driver, you'll need to be familiar with chapters 3 and 14, as well as 4.1 (though not 4.1's subsections). You'll also need to use chapter 13 as reference. The other chapters mostly cover components of the E1000 that your driver won't have to interact with. Don't worry about the details right now; just get a feel for how the document is structured so you can find things later.

> While reading the manual, keep in mind that the E1000 is a sophisticated device with many advanced features. A working E1000 driver only needs a fraction of the features and interfaces that the NIC provides. Think carefully about the easiest way to interface with the card. We strongly recommend that you get a basic driver working before taking advantage of the advanced features.

PCI Interface

E1000 是一个PCI设备，意味着它会插在主板的PCI总线上。PCI总线中有地址，数据和中断线路，允许CPU和PCI设备之间进行通信，以及允许PCI设备去读写内存。

一个PIC设备在它被使用之前需要被发现和初始化。发现的过程是沿着PCI总线寻找连接设备的过程。初始化的过程是分配I/O和内存空间以及协商设备使用IRQ线。

我们已经在 `kern/pci.c` 中给你提供了PCI代码。在启动过程中执行PCI初始化，PCI代码沿着PCI总线寻找设备。当其发现设备，它读取设备的 `vendor ID` 和 `device ID` 并使用这两个值作为一个key在 `pci_attach_vendor` 数组中寻找。该数组由 `struct pci_driver` 条目组成。

```
struct pci_driver {
  uint32_t key1, key2;
  int (*attachfn) (struct pci_func *pcif);
};
```

如果被发现设备的 `vendor ID` 和 `device ID` 和数组中的条目匹配，PCI代码就调用该条目中的 `attachfn` 来执行设备初始化。（设备也可以被区分为类，这就是 `kern/pci.c` 中的另一个驱动程序表的用途。）

附加函数通过PCI函数进行初始化。尽管 *E1000* 只暴露一个函数，一个PCI卡可以暴露多个函数。这里是我们如何去表示一个PCI函数在 `JOS` ：

```
struct pci_func {
  struct pci_bus *bus;
  uint32_t dev;
  uint32_t func;

  uint32_t dev_id;
  uint32_t dev_class;

  uint32_t reg_base[6];
  uint32_t reg_size[6];
  uint8_t irq_line;

};
```

上面的结构反应了一些开发手册章节4.1中表4-1的一些条目。

Table 4-1. Mandatory PCI Registers

Byte Offset	Byte 3	Byte 2	Byte 1	Byte 0
0h	Device ID		Vendor ID	
4h	Status Register		Command Register	
8h	Class Code (020000h)			Revision ID
Ch	BIST (00h)	Header Type (00h)	Latency Timer	Cache Line Size
10h	Base Address 0[a]			
4h	Base Address 1			
18h	Base Address 2			

`struct pci_func` 的最后三个条目是我们格外关注的，因为它们记录了设备 *协商的内存*，*I/O*，和 *中断资源*。`reg_base` 和 `reg_size` 数组包含最多6个 *BARs, Base Address Registers* 的信息。`reg_base` 存储内存映射区域（或I/O端口资源的基础I/O端口）的基内存地址；`reg_size` 包含字节大小或I/O端口数量，对应于 `reg_base` 的基值；`irq_line` 包含分配给设备用于中断的IRQ线路。*E1000 BARs* 的特定意义在表4-2的第二半部分给出。

当附加函数被调用，设备被找到但还没有被启用。这意味着PCI代码还没有决定分配给设备的资源，比如 *地址空间* 和 *IRQ线*，因此 `struct pci_func` 的最后三个元素还没有被填入。附加函数应该调用 `pci_func_enable`，这将启用设备，协商资源并填入 `struct pci_func`

Exercise 3.

Implement an attach function to initialize the E1000. Add an entry to the `pci_attach_vendor` array in `kern/pci.c` to trigger your function if a matching PCI device is found (be sure to put it before the `{0, 0, 0}` entry that mark the end of the table). You can find the vendor ID and device ID of the 82540EM that QEMU emulates in section 5.2. You should also see these listed when JOS scans the PCI bus while booting.

For now, just enable the E1000 device via `pci_func_enable`. We'll add more initialization throughout the lab.

We have provided the `kern/e1000.c` and `kern/e1000.h` files for you so that you do not need to mess with the build system. They are currently blank; you need to fill them in for this exercise. You may also need to include the `e1000.h` file in other places in the kernel.

When you boot your kernel, you should see it print that the PCI function of the E1000 card was enabled. Your code should now pass the `pci attach` test of `make grade`.

```
diff --git a/kern/e1000.c b/kern/e1000.c
index 192f317..ea5b3b0 100644
--- a/kern/e1000.c
+++ b/kern/e1000.c
@@ -1,4 +1,10 @@
#include <kern/e1000.h>
#include <kern/pmap.h>
+#include <kern/pci.h>
+#include <kern/pcireg.h>

// LAB 6: Your driver code here
+int e1000_attach(struct pci_func* pcif){
+    pci_func_enable(pcif);
+    return 0;
+}

diff --git a/kern/e1000.h b/kern/e1000.h
index 8b5a513..82ba6b2 100644
```

```
--- a/kern/e1000.h
+++ b/kern/e1000.h
@@ -1,3 +1,9 @@
 #ifndef JOS_KERN_E1000_H
 #define JOS_KERN_E1000_H
+#include <kern/pci.h>
+
+#define E1000_VEN_ID            0x8086
+#define E1000_DEV_ID_82540EM    0x100E
+
+int e1000_attach(struct pci_func* pcif);
 #endif  // SOL ≥ 6

diff --git a/kern/pci.c b/kern/pci.c
index 784e072..2d39e40 100644
--- a/kern/pci.c
+++ b/kern/pci.c
@@ -31,6 +31,7 @@ struct pci_driver pci_attach_class[] = {
 // pci_attach_vendor matches the vendor ID and device ID of a PCI
device. key1
 // and key2 should be the vendor ID and device ID respectively
 struct pci_driver pci_attach_vendor[] = {
+       {E1000_VEN_ID, E1000_DEV_ID_82540EM, &e1000_attach},
        { 0, 0, 0 },
 };
```

Memory-mapped I/O

软件通过*memory-mapped I/O(MMIO)*和*E1000*进行通信。这种方式你已经在`JOS`中见过了两次：CGA控制台和LAPIC都是通过从“memory”读写进行控制和询问。

但是这些读写不会影响`DRAM`；它们直接对这些设备进行操作。

`pci_func_enable`和*E1000*协商一个MMIO区域，并在`BAR 0`(`reg_base[0]` and `reg_size[0]`)中存储他的基地址和大小。这是分配给设备的一段物理内存地址范围，意味着你需要做些什么来通过*虚拟地址*访问它。

由于MMIO区域被分配到一个非常高的物理地址（一般高于3GB），你不能使用`KADDR`来访问它，因为`JOS`256MB的限制。因此，你将不得不创建一个新的内存映射。

我们将使用 `MMIOBASE` （你的 `mmio_map_region` 将确保我们不会重写被LAPIC使用的映射）上方的区域。

因为PCI设备初始化在 `JOS` 创建用户环境之前发生，你可以创建映射位于 `kern_pgdir` ，然后它将一直可以得到。

Exercise 4.

In your attach function, create a virtual memory mapping for the E1000's BAR 0 by calling `mmio_map_region` (which you wrote in lab 4 to support memory-mapping the LAPIC).

You'll want to record the location of this mapping in a variable so you can later access the registers you just mapped. Take a look at the `lapic` variable in `kern/lapic.c` for an example of one way to do this. If you do use a pointer to the device register mapping, be sure to declare it volatile; otherwise, the compiler is allowed to cache values and reorder accesses to this memory.

To test your mapping, try printing out the device status register (section 13.4.2). This is a 4 byte register that starts at byte 8 of the register space. You should get `0x80080783` , which indicates a full duplex link is up at 1000 MB/s, among other things.

`kern/e1000.c`

```
volatile void* e1000;
#define E1000_REG(offset)    (*(volatile uint32_t *)(e1000 +
offset))

// LAB 6: Your driver code here
int e1000_attach(struct pci_func* pcif){
 pci_func_enable(pcif);
 e1000 = mmio_map_region(pcif->reg_base[0], pcif->reg_size[0]);
 cprintf("e1000: status 0x%08x\n", E1000_REG(E1000_STATUS));
 // cprintf("device status:[%08x]\n", *(uint32_t *)((uint8_t
*)e1000 + E1000_STATUS));
 return 0;
}
```

kern/e1000.h

```
#define E1000_STATUS   0x00008  /* Device Status - RO */
```

Hint: You'll need a lot of constants, like the *locations of registers* and *values of bit masks*. Trying to copy these out of the developer's manual is error-prone and mistakes can lead to painful debugging sessions. We recommend instead using QEMU's e1000_hw.h header as a guideline. We don't recommend copying it in verbatim, because it defines far more than you actually need and may not define things in the way you need, but it's a good starting point.

DMA

你可以想象通过读写 *E1000* 的寄存器来转发和接收数据包，但是这将很慢并需要 *E1000* 内部缓存数据包数据。相反，*E1000* 使用 *DMA, Direct Memory Access* 来从内存直接读写数据包数据，而不需要CPU的参与。

驱动负责给转发和接收序列分配内存；设置DMA描述符；配置 *E1000* 中这些序列的位置，但这之后的所有事情都是异步的。

为了转发数据包，驱动

- 复制数据包到转发队列的下一个DMA描述符
- 通知 *E1000*，另一个数据包已经就绪
- 当有时间发送数据包时，*E1000* 从描述符中将数据复制出来

同样的，当 *E1000* 接收一个数据包时，它将数据包复制到接收队列的下一个DMA描述符，驱动可以在下次有机会的时候读取该描述符。

接收和转发队列在高级层次上是相似的。都是由一系列描述符组成。

尽管这些描述符的具体结构不尽相同，每个描述符包含一些标志和包含数据包数据的缓冲区的物理地址。（这些数据包数据不是用于发送，就是由操作系统分配给卡用于将接受的数据包写入）

队列被实现为循环数组，意味着当卡或驱动到达数组的终点时，它会回到起点。两个队列都有 *头指针* 和 *尾指针*，并且两个指针之间的内容都是描述符。硬件总是从头部消耗描述符然后移动头指针，而驱动总是在尾部加入描述符然后移动尾指针。

- 描述符在转发队列中

  表示等待被发送的数据包，因此在稳定状态下，转发队列为空
- 描述符在接收队列中

该队列中的描述符为空闲描述符，卡可以接收数据包到该描述符。因此在稳定状态下，接收队列由所有可用的接收描述符组成）

正确更新尾寄存器，而不混淆 *E1000* 是很棘手的；小心！

指向这些数组和描述符中数据包缓冲区地址的指针必须都是物理地址，因为硬件需要直接执行DMA访问物理RAM而不是通过MMU。

Transmitting Packets

E1000 的转发和接收函数都是互相基本独立的，所以我们可以同一时间在一个上面工作。我们将首先进军转发数据包，因为我们不能测试接收，而不首先转发一个"I'm here"数据包。

首先，你将不得不初始化 *卡* 来转发，遵循 *章节14.5* 描述的步骤（不用担心子章节）。转发初始化首先要**设置转发队列**，精确的队列结构在 *章节3.4* 描述，描述符的结构在 *章节3.3.3* 描述。

我们将不会使用 *E1000* 的TCP卸载功能，所以你可以专注于“传统转发描述符格式”。你应该现在读取这些章节并熟悉这些结构。

C Structures

你会发现使用C语言结构题来描述 *E1000* 结构很方便。就像你之前看到过的 `struct Trapframe`，C语言结构体让你能在内存精确地布局数据。C语言可以在字段之间插入padding，但是 *E1000* 结构的布局不会出现这个问题。如果你遇到了字段对齐的问题，查看GCC的 `packed` 特性。

举个例子，考虑表3-8给出的 *传统转发描述符* 。

```
63            48 47   40 39   32 31   24 23   16 15            0
+----------------------------------------------------------------+
|                         Buffer address                         |
+---------------+-------+-------+-------+-------+---------------+
|    Special    |  CSS  | Status|  Cmd  |  CSO  |    Length     |
+---------------+-------+-------+-------+-------+---------------+
```

The first byte of the structure starts at the top right, so to convert this into a C struct, read from right to left, top to bottom. If you squint at it right, you'll see that all of the fields even fit nicely into a standard-size types:

```
struct tx_desc
{
  uint64_t addr;
  uint16_t length;
  uint8_t cso;
  uint8_t cmd;
  uint8_t status;
  uint8_t css;
  uint16_t special;
};
```

你的驱动将必须为 *转发描述符数组* 和 *数据包缓冲区* 保留内存， *数据包缓冲区* 由 *转发描述符* 指向。

有很多的方法来解决这个问题，从动态分配页到简单地在全局变量中声明它们。无论你选择哪种方法，记住 *E1000* **直接访问物理内存**，意味着它访问的任何缓冲区都必须是连续的物理内存。

同样，也有很多方式来处理 *数据包缓存* 。最简单的，也是我们推荐的，在驱动初始化过程中，为每个描述符保留数据包缓冲区空间并且简单地将数据包数据复制进出 *预先分配的缓冲区* 。

以太网数据包的最大大小为**1518字节**，限制了这些缓冲区需要的大小。

更多精心设计的驱动可以动态地分配 *数据包缓冲区* （比如，为了降低当网络使用率低时的内存开销）或者甚至通过用户空间直接传递缓冲区（也被称为 `zero copy` ），但是从简单的开始是一个不错的选择。

Exercise 5.

Perform the initialization steps described in *section 14.5* (but not its subsections). Use *section 13* as a reference for the registers the initialization process refers to and sections 3.3.3 and 3.4 for reference to the transmit descriptors and transmit descriptor array.

Be mindful of the alignment requirements on the transmit descriptor array and the restrictions on length of this array. Since TDLEN must be 128-byte aligned and each transmit descriptor is 16 bytes, your transmit descriptor array will need some multiple of 8 transmit descriptors. However, don't use more than 64 descriptors or our tests won't be able to test transmit ring overflow.

For the TCTL.COLD, you can assume full-duplex operation. For TIPG, refer to the default values described in table 13-77 of section 13.4.34 for the IEEE 802.3 standard IPG (don't use the values in the table in section 14.5).

e1000.h

```
#define E1000_STATUS   0x00008  /* Device Status - RO */

#define E1000_TCTL     0x00400  /* TX Control - RW */
#define E1000_TIPG     0x00410  /* TX Inter-packet gap -RW */
#define E1000_TDBAL    0x03800  /* TX Descriptor Base Address Low
- RW */
#define E1000_TDBAH    0x03804  /* TX Descriptor Base Address High
- RW */
#define E1000_TDLEN    0x03808  /* TX Descriptor Length - RW */
#define E1000_TDH      0x03810  /* TX Descriptor Head - RW */
#define E1000_TDT      0x03818  /* TX Descripotr Tail - RW */

/* Transmit Control */
#define E1000_TCTL_EN   0x00000002  /* enable tx */
#define E1000_TCTL_PSP  0x00000008  /* pad short packets */
#define E1000_TCTL_CT   0x00000ff0  /* collision threshold */
#define E1000_TCTL_COLD 0x003ff000  /* collision distance */

/* Collision related configuration parameters */
#define E1000_COLLISION_THRESHOLD   0x10
#define E1000_CT_SHIFT              4

/* Collision distance is a 0-based value that applies to half-
duplex-capable hardware only. */
#define E1000_COLLISION_DISTANCE    0x40
#define E1000_COLD_SHIFT            12

/* Default values for the transmit IPG register */
#define E1000_DEFAULT_TIPG_IPGT     10
#define E1000_DEFAULT_TIPG_IPGR1    4
#define E1000_DEFAULT_TIPG_IPGR2    6
#define E1000_TIPG_IPGT_MASK        0x000003FF
```

```
#define E1000_TIPG_IPGR1_MASK       0x000FFC00
#define E1000_TIPG_IPGR2_MASK       0x3FF00000
#define E1000_TIPG_IPGR1_SHIFT      10
#define E1000_TIPG_IPGR2_SHIFT      20

struct e1000_tx_desc{
 uint64_t addr;
 uint16_t length;
 uint8_t cso;
 uint8_t cmd;
 uint8_t status;
 uint8_t css;
 uint16_t special;
}__attribute__((packed));
```

kern/e1000.c

```
#define TX_BUF_SIZE 1536  // 16-byte aligned for performance
#define NTXDESC     64

static struct e1000_tx_desc e1000_tx_queue[NTXDESC]
__attribute__((aligned(16)));
static uint8_t e1000_tx_buf[NTXDESC][TX_BUF_SIZE];

static void
e1000_tx_init()
{
 // initialize tx queue
 int i;
 memset(e1000_tx_queue, 0, sizeof(e1000_tx_queue));
 for (i = 0; i < NTXDESC; i++) {
     e1000_tx_queue[i].addr = PADDR(e1000_tx_buf[i]);
   // 注意这里要对描述符中的`status`和`cmd`字段进行初始化
     e1000_tx_queue[i].status = E1000_TXD_STAT_DD;
     e1000_tx_queue[i].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
 }

 // initialize transmit descriptor registers
 E1000_REG(E1000_TDBAL) = PADDR(e1000_tx_queue);
 E1000_REG(E1000_TDBAH) = 0;
```

```
    E1000_REG(E1000_TDLEN) = sizeof(e1000_tx_queue);
    E1000_REG(E1000_TDH) = 0;
    E1000_REG(E1000_TDT) = 0;

    // initialize transmit control registers
    E1000_REG(E1000_TCTL) &= ~(E1000_TCTL_CT | E1000_TCTL_COLD);
    E1000_REG(E1000_TCTL) |= E1000_TCTL_EN | E1000_TCTL_PSP |
                            (E1000_COLLISION_THRESHOLD <<
   E1000_CT_SHIFT) |
                            (E1000_COLLISION_DISTANCE <<
   E1000_COLD_SHIFT);
    E1000_REG(E1000_TIPG) &= ~(E1000_TIPG_IPGT_MASK |
   E1000_TIPG_IPGR1_MASK | E1000_TIPG_IPGR2_MASK);
    E1000_REG(E1000_TIPG) |= E1000_DEFAULT_TIPG_IPGT |
                            (E1000_DEFAULT_TIPG_IPGR1 <<
   E1000_TIPG_IPGR1_SHIFT) |
                            (E1000_DEFAULT_TIPG_IPGR2 <<
   E1000_TIPG_IPGR2_SHIFT);
   }
```

Try running `make E1000_DEBUG=TXERR,TX qemu`. If you are using the course qemu, you should see an *"e1000: tx disabled"* message when you set the TDT register (since this happens before you set TCTL.EN) and no further "e1000" messages.

现在转发已经被初始化了，你将写代码来转发一个数据包并使用户空间能通过系统调用来访问。

为了转发一个数据包，你必须将其加入转发队列的尾部，也就是将数据包数据复制到下一个数据包缓冲区，然后更新 `TDT` (transmit descriptor tail)寄存器来通知 *卡* 转发队列中有另一个数据包。（注意，`TDT` 是 *转发描述符数组* 的下标，不是一个字节的位移；文档对于这个不是特别清晰。）

然而，转发队列就那么大。如果 *卡* 落后在转发数据包后面，转发队列满了会发生什么呢?

为了检测该条件，你将需要一些 *E1000* 的反馈。不幸的是，你不能就使用 `TDH` (transmit descriptor head)寄存器；文档明确指出从软件读取该寄存器是不可靠的。**但是**，如果你在转发描述符的command字段设置了 `RS` 位，然后，当 *卡* 转发那个描述符中的数据包， *卡* 将设置描述符中status字段的 `DD` 位。如果描述符中的 `DD` 位被设置，你就知道 *回收* 那个描述符并使用它转发另一个数据包是安全的。

那如果用户调用了你的 *转发系统调用* ，但是下一个描述符 `DD` 位没有设置，表明那个转发队列是满的？你将不得不决定这种情况下应该做什么。

你应该简单地丢掉数据包。网络协议对这种情况是有弹性的，但如果你丢了太多数据包，协议可能不会恢复。反而你应该告诉用户环境它需要重新尝试，就像你在 `sys_ipc_try_send` 中做的。这样做的优点是可以推迟生成数据的环境。

Exercise 6.

Write a function to transmit a packet by checking that the next descriptor is free, copying the packet data into the next descriptor, and updating `TDT`. Make sure you handle the transmit queue being full.

`e1000_transmit()` in `kern/e1000.c`

```
int e1000_transmit(const void* buf, size_t size){
int tail = E1000_REG(E1000_TDT);

 if(size > 1518){
     return -1;
 }
struct e1000_tx_desc* tail_desc = &(e1000_tx_queue[tail]);
 if(!(tail_desc->status & E1000_TXD_STAT_DD)){
     // state is not DD
     return -1;
 }
 memcpy(e1000_tx_buf[tail], buf, size);
 tail_desc->length = (uint16_t)size;
 // clear DD
 tail_desc->status &= ~E1000_TXD_STAT_DD;
 // tail_desc->cmd |= E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
 E1000_REG(E1000_TDT) = (tail + 1) % NTXDESC;
 return 0;

}
```

Now would be a good time to test your packet transmit code. Try transmitting just a few packets by directly calling your transmit function from the kernel. You don't have to create packets that conform to any particular network protocol in order to test this. Run `make E1000_DEBUG=TXERR,TX qemu` to run your test. You should see something like

```
e1000: index 0: 0x271f00 : 9000002a 0
...
```

as you transmit packets. Each line gives the index in the transmit array, `the buffer address` of that transmit descriptor, the `cmd/CSO/length fields`, and the `special/CSS/status fields`. If QEMU doesn't print the values you expected from your transmit descriptor, check that you're filling in the right descriptor and that you configured `TDBAL` and `TDBAH` correctly. If you get *"e1000: TDH wraparound @0, TDT x, TDLEN y"* messages, that means the E1000 ran all the way through the transmit queue without stopping (if QEMU didn't check this, it would enter an infinite loop), which probably means you aren't manipulating TDT correctly. If you get lots of *"e1000: tx disabled"* messages, then you didn't set the transmit control register right.

一旦QEMU运行，你可以运行 `tcpdump -XXnr qemu.pcap` 来查看你转发的数据包数据。

如果你看到来自QEMU你所期望的 *"e1000:index"* 信息，但是你的数据包捕获是空的，请再次检查你是否填入了每个必须的字段和你转发描述符中的位（*E1000* 可能进入了你的转发描述符，但是它不认为自己应该发送些什么）。

Exercise 7.

Add a system call that lets you transmit packets from user space. The exact interface is up to you. Don't forget to check any pointers passed to the kernel from user space.

```
diff --git a/inc/lib.h b/inc/lib.h
index 66740e8..5763e77 100644
--- a/inc/lib.h
+++ b/inc/lib.h
@@ -60,6 +60,7 @@ int   sys_page_unmap(envid_t env, void *pg);
int    sys_ipc_try_send(envid_t to_env, uint32_t value, void *pg,
int perm);
int    sys_ipc_recv(void *rcv_pg);
unsigned int sys_time_msec(void);
+int sys_net_transmit(const void* buf, size_t size);

diff --git a/inc/syscall.h b/inc/syscall.h
index 36f26de..66c6d22 100644
--- a/inc/syscall.h
+++ b/inc/syscall.h
@@ -18,6 +18,7 @@ enum {
     SYS_ipc_try_send,
     SYS_ipc_recv,
     SYS_time_msec,
+       SYS_net_transmit,
        NSYSCALLS
 };

diff --git a/kern/syscall.c b/kern/syscall.c
index 4e06f0d..32f410a 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -13,6 +13,8 @@
 #include <kern/sched.h>
 #include <kern/time.h>

+#include <kern/e1000.h>
+
 // Print a string to the system console.
 // The string is exactly 'len' characters long.
 // Destroys the environment on memory errors.
@@ -463,6 +465,13 @@ sys_time_msec(void)
        return time_msec();
 }
```

```
+// Transmit a packet from user space
+static int
+sys_net_transmit(const void* buf, size_t size){
+       // segfault when address of buf is invalid
+       user_mem_assert(curenv, buf, size, PTE_U);
+       return e1000_transmit(buf, size);
+}
 // Dispatches to the correct kernel function, passing the
arguments.
 int32_t
 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t
a3, uint32_t a4, uint32_t a5)
@@ -507,6 +516,8 @@ syscall(uint32_t syscallno, uint32_t a1,
uint32_t a2, uint32_t a3, uint32_t a4,
                return sys_env_set_trapframe(a1, (struct
Trapframe*)a2);
        case SYS_time_msec:
                return sys_time_msec();
+       case SYS_net_transmit:
+               return sys_net_transmit((const void*)a1,
(size_t)a2);
        default:
                return -E_INVAL;
        }

diff --git a/lib/syscall.c b/lib/syscall.c
index 9e1a1d9..9445b9b 100644
--- a/lib/syscall.c
+++ b/lib/syscall.c
@@ -122,3 +122,9 @@ sys_time_msec(void)
 {
        return (unsigned int) syscall(SYS_time_msec, 0, 0, 0, 0,
0, 0);
 }
+
+int
+sys_net_transmit(const void* buf, size_t size)
+{
```

```
+       return (int) syscall(SYS_net_transmit, 0, (uint32_t)buf,
size, 0, 0, 0);
+}
```

Transmitting Packets: Network Server

现在你已经有了访问你设备驱动转发边的系统调用接口，是时候发送数据包了。*output helper environment* 的目标就是循环做下面的事：

- 从 *核心网络服务器* 接收 `NSREQ_OUTPUT` IPC信息
- 发送数据包，伴随着IPC信息到 *网络设备驱动* ，使用你在上面添加的系统调用

`NSREQ_OUTPUT` IPC信息使用 `net/lwip/jos/jif/jif.c` 中的 `low_level_output` 发送，它将 `lwIP` 堆栈粘合到 `JOS` 网络系统。

每个IPC都包括一个页，页由一个 `union Nsipc` 组成，数据包位于它的 `struct jif_pkt pkt` 字段(see `inc/ns.h`)。

```
struct jif_pkt {
    int jp_len;
    char jp_data[0];
};
```

`jp_len` 表示数据包的长度。所有IPC页面后续的字节都是数据包内容。

在结构的尾部使用像 `jp_data` 这样的0长度数组是一个普通的C语言技巧（有些人可能会觉得厌恶）来表示没有预先确定长度的缓冲区。因为C语言不检查数据边界，只要你确定结构之后有足够的未使用空间，你就可以使用 `jp_data` 作为一个任意大小的数组。

当设备驱动队列中没有更多空间时，要清楚此时 *设备驱动* ， *输出环境* ， *核心网络服务器* 之间的交互。

核心网络服务器 使用IPC向 *输出环境* 发送数据包。如果 *输出环境* 被挂起由于一个发送数据包的系统调用，而驱动又没有多余的缓冲区空间给新的数据包，此时 *核心网络服务器* 将会阻塞，并等待 *输出服务器* 接收IPC调用。

> Exercise 8.
>
> Implement `net/output.c` .

`output()` in `net/output.c`

```
extern union Nsipc nsipcbuf;

void
output(envid_t ns_envid)
{
  binaryname = "ns_output";

  // LAB 6: Your code here:
  //   - read a packet from the network server
  //   - send the packet to the device driver
int r;
int whom;
int32_t req;
int perm;
  while(1){
    if((req = ipc_recv(&whom, &nsipcbuf, &perm) == NSREQ_OUTPUT)){
      // sys_net_transmit(nsipcbuf.pkt.jp_data,
nsipcbuf.pkt.jp_len);
      // return;
      while((r = sys_net_transmit(nsipcbuf.pkt.jp_data,
nsipcbuf.pkt.jp_len)) < 0){
        sys_yield();
      }
    }
  }
}
```

You can use `net/testoutput.c` to test your output code without involving the whole network server. Try running make E1000_DEBUG=TXERR,TX run-net_testoutput. You should see something like

```
Transmitting packet 0
e1000: index 0: 0x271f00 : 9000009 0
Transmitting packet 1
e1000: index 1: 0x2724ee : 9000009 0
...
```

and `tcpdump -XXnr qemu.pcap` should output

```
reading from file qemu.pcap, link-type EN10MB (Ethernet)
-5:00:00.600186 [|ether]
  0x0000:  5061 636b 6574 2030 30                 Packet.00
-5:00:00.610080 [|ether]
  0x0000:  5061 636b 6574 2030 31                 Packet.01
...
```

To test with a larger packet count, try `make E1000_DEBUG=TXERR,TX NET_CFLAGS=-DTESTOUTPUT_COUNT=100 run-net_testoutput`. If this overflows your transmit ring, double check that you're handling the DD status bit correctly and that you've told the hardware to set the DD status bit (using the RS command bit).

Your code should pass the `testoutput` tests of `make grade`.

Part B: Receiving packets and the web server

Receiving Packets

就像你为转发数据包所做的那样，你将配置 *E1000* 来接收数据包并提供一个 *接收描述符队列* 和 *接收描述符* 。 *章节3.2* 描述了数据包接收如何工作，包括 *接收队列结构* 和 *接收描述符* ，详细的初始化过程在 *章节14.4* 给出。

> **Exercise 9.**
>
> Read `section 3.2`. You can ignore anything about interrupts and checksum offloading (you can return to these sections if you decide to use these features later), and you don't have to be concerned with the details of thresholds and how the card's internal caches work.

接收队列和转发队列非常相似，除了接收队列由空的数据包缓冲区组成，这些空缓冲区等待被到来的数据包填充。

因此，当网络空闲的时候，转发队列是空的（因为所有的数据包都被发送了），但是接收队列是满的（都是空的数据包缓冲区）。

当 *E1000* 接收数据包时，它首先检查其是否匹配 *卡* 的过滤条件（比如，检查数据包是否发送到 *E1000* 的MAC地址）并且忽略掉不满足任何过滤条件的数据包。

否则，*E1000* 尝试取出下一个接收描述符从接收队列的 *头部* 。如果 `head(RDH)` 已将赶上了 `tail(RDT)` ，表明接收队列中没有空闲的描述符，所以 *卡* 会抛弃数据包。

如果队列中有空闲的接收描述符，它将数据包数据复制到由描述符指向的缓冲区，设置描述符的 `DD(Descriptor Donw)` 和 `EOP(End of Packet)` 状态位，并且增加 `RDH` 。

如果 *E1000* 收到一个数据包比接收描述符中的数据缓冲区更大，它将从接收描述符队列中取出所需数量的描述符来存储数据包的全部内容。为了表明这种情况发生，它将在 *所有* 这些描述符设置 `DD` 状态位，但是只在 *最后* 的描述中设置 `EOP` 状态位。

你可以在你的驱动中处理这种情况，也可以配置 *卡* 不接受“长数据包”（也称为 *jumbo frames* ，巨型帧），并且确保你的接收缓冲区足够大来存储可能的最大的标准以太网数据包（1518字节）。

Exercise 10.

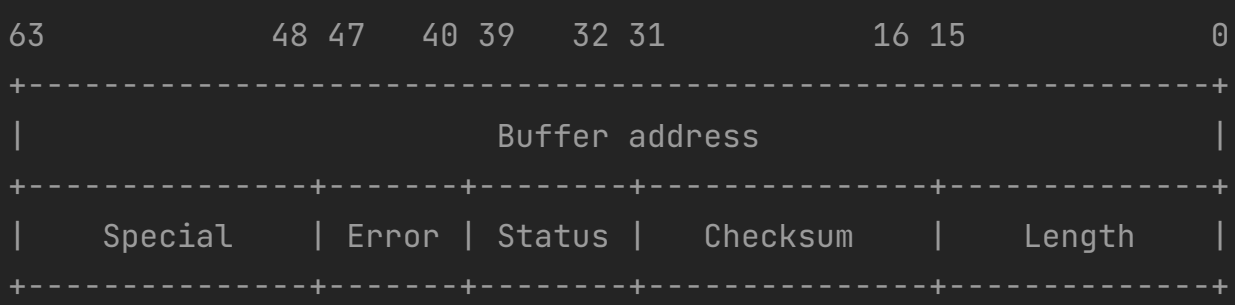
Set up the receive queue and configure the *E1000* by following the process in *section 14.4* . You don't have to support "long packets" or multicast. For now, don't configure the card to use interrupts; you can change that later if you decide to use receive interrupts. Also, configure the E1000 to strip the Ethernet CRC, since the grade script expects it to be stripped.

By default, the card will filter out *all* packets. You have to configure the Receive Address Registers (RAL and RAH) with the card's own MAC address in order to accept packets addressed to that card. You can simply hard-code QEMU's default MAC address of 52:54:00:12:34:56 (we already hard-code this in lwIP, so doing it here too doesn't make things any worse). Be very careful with the byte order; MAC addresses are written from lowest-order byte to highest-order byte, so 52:54:00:12 are the low-order 32 bits of the MAC address and 34:56 are the high-order 16 bits.

The E1000 only supports a specific set of receive buffer sizes (given in the description of RCTL.BSIZE in 13.4.22). If you make your receive packet buffers large enough and disable long packets, you won't have to worry about packets spanning multiple receive buffers. Also, remember that, just like for transmit, the receive queue and the packet buffers must be contiguous in physical memory.

You should use at least 128 receive descriptors

Receive Descriptor(RDESC)




```
@@ -54,6 +83,15 @@ struct e1000_tx_desc{
   uint16_t special;
}__attribute__((packed));

+struct e1000_re_desc{
 +    uint64_t addr;
 +    uint16_t length;
 +    uint16_t checksum;
 +    uint8_t status;
 +    uint8_t err;
 +    uint16_t special;
 +}__attribute__((packed));
```

```
@@ -44,6 +44,35 @@
 #define E1000_TXD_CMD_RS    0x08  /* Report Status */
 #define E1000_TXD_STAT_DD   0x01  /* Descriptor Done */

+
+#define E1000_RDBAL    0x02800  /* RX Descriptor Base Address Low
- RW */
+#define E1000_RDBAH    0x02804  /* RX Descriptor Base Address
High - RW */
+#define E1000_RDLEN    0x02808  /* RX Descriptor Length - RW */
+#define E1000_RDH      0x02810  /* RX Descriptor Head - RW */
+#define E1000_RDT      0x02818  /* RX Descriptor Tail - RW */
+
+#define E1000_RAL       0x05400  /* Receive Address - RW Array */
+#define E1000_RAH       0x05404
```

```
+#define E1000_RAH_AV  0x80000000        /* Receive descriptor
valid */
+
+/* Receive Control */
+#define E1000_RCTL     0x00100  /* RX Control - RW */
+#define E1000_RCTL_EN       0x00000002  /* enable */
+#define E1000_RCTL_LBM      0x000000c0  /* loopback mode */
+#define E1000_RCTL_RDMTS    0x00000300  /* rx desc min threshold
size */
+#define E1000_RCTL_SZ       0x00030000  /* rx buffer size */
+#define E1000_RCTL_SECRC    0x04000000  /* strip ethernet CRC */
+#define E1000_RCTL_BSEX     0x02000000  /* Buffer size extension
*/
+
+#define E1000_RCTL_LBM_NO       0x00000000  /* no loopback mode
*/
+#define E1000_RCTL_LBM_SHIFT    6
+
+#define E1000_RCTL_RDMTS_HALF   0x00000000
+#define E1000_RCTL_RDMTS_SHIFT  8
+
+#define E1000_RCTL_SZ_2048      0x00000000  /* rx buffer size
2048 */
+#define E1000_RCTL_SZ_SHIFT     16
+


diff --git a/kern/e1000.c b/kern/e1000.c
index 9b2d035..cffcc35 100644
--- a/kern/e1000.c
+++ b/kern/e1000.c
@@ -15,6 +15,15 @@ volatile void* e1000;
 static struct e1000_tx_desc e1000_tx_queue[NTXDESC]
__attribute__((aligned(16)));
 static uint8_t e1000_tx_buf[NTXDESC][TX_BUF_SIZE];

+#define RE_BUF_SIZE 1518        // 后面可能需要更改这里的大小
+#define NREDESC 128
+
```

```
+static struct e1000_re_desc e1000_re_queue[NREDESC]
__attribute__((aligned(16)));
+static uint8_t e1000_re_buf[NREDESC][RE_BUF_SIZE];
+
+#define JOS_DEFAULT_MAC_LOW     0x12005452
+#define JOS_DEFAULT_MAC_HIGH    0x00005634
+
```

`e1000_re_init()` in `kern/e1000.c`

```
static void
e1000_re_init()
{
    // initialize re queue
    int i;
    memset(e1000_re_queue, 0, sizeof(e1000_re_queue));
    for(i = 0;i < NREDESC; ++i){
        e1000_re_queue[i].addr = PADDR(e1000_re_buf[i]);
    }

    // initialize receive address registers
    // by default, it comes from EEPROM
    E1000_REG(E1000_RAL) = JOS_DEFAULT_MAC_LOW;
    E1000_REG(E1000_RAH) = JOS_DEFAULT_MAC_HIGH;
    E1000_REG(E1000_RAH) |= E1000_RAH_AV;

    // initialize receive descriptor registers
    E1000_REG(E1000_RDBAL) = PADDR(e1000_re_queue);
    E1000_REG(E1000_RDBAH) = 0;
    E1000_REG(E1000_RDLEN) = sizeof(e1000_re_queue);
    E1000_REG(E1000_RDH) = 0;
    E1000_REG(E1000_RDT) = NREDESC - 1;

    // initialize transmit control registers
    E1000_REG(E1000_RCTL) &= ~(E1000_RCTL_LBM | E1000_RCTL_RDMTS |
E1000_RCTL_SZ | E1000_RCTL_BSEX);
    E1000_REG(E1000_RCTL) |= E1000_RCTL_EN | E1000_RCTL_SECRC;
}
```

```
@@ -76,5 +113,6 @@ int e1000_attach(struct pci_func* pcif){
     e1000_tx_init();
     // char *str = "hello";
     // int r = e1000_transmit(str, 6);
+    e1000_re_init();
     return 0;
 }
```

You can do a basic test of receive functionality now, even without writing the code to receive packets. Run `make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-net_testinput. ``testinput` will transmit an *ARP (Address Resolution Protocol)* announcement packet (using your packet transmitting system call), which QEMU will automatically reply to. Even though your driver can't receive this reply yet, you should see a "e1000: unicast match[0]: 52:54:00:12:34:56" message, indicating that a packet was received by the E1000 and matched the configured receive filter. If you see a "e1000: unicast mismatch: 52:54:00:12:34:56" message instead, the E1000 filtered out the packet, which means you probably didn't configure RAL and RAH correctly. Make sure you got the byte ordering right and didn't forget to set the "Address Valid" bit in RAH. If you don't get any "e1000" messages, you probably didn't enable receive correctly.

现在你已经准备好去实现接收数据包了。

为了接收一个数据包，你的驱动将必须跟踪保存下一个收到的数据包的描述符（暗示：取决于你的设计，*E1000* 可能已有跟踪这个的寄存器）。和转发相似，文档声明了 `RDH` 寄存器不能从软件可靠地读取。所以为了判断一个数据包是否被传送到描述符的数据包缓冲区，你将必须读取描述符中的 `DD` 状态位。

- 如果 `DD` 状态位已经被设置

  你可以将数据包数据复制出描述符的数据缓冲区，然后告诉 *卡* 该描述符是空闲的，通过更新队列的尾下表， `RDT` 。

- 如果 `DD` 状态位没有被设置

  表明没有数据包被接收，就相当于转发队列都是满的。这种情况有两种方法来处理：

  1. 简单地返回一个 `try again` 错误，要求调用者重新发送

     尽管由于那是暂时的情况，这个方法对于满的转发队列都工作地很好，但对于 *空的队列* 不太合理，因为接收队列可能会长时间保持空的状态。

2. 将调用环境挂起，直到接收队列中有数据包处理

   这个策略和 `sys_ipc_recv` 很相似。

   就像IPC的例子，因为每个CPU只有一个内核栈，一旦我们离开内核，栈上的状态就会丢失。我们需要设置一个标志来表明一个环境由于接收队列下溢被挂起，并且记录系统调用参数。

   这个方法的缺点就是复杂：*E1000* 必须能够生成 *接收中断*，驱动必须处理它们，来重新开始正在等待数据包的阻塞环境。

Exercise 11.

Write a function to receive a packet from the E1000 and expose it to user space by adding a system call. Make sure you handle the receive queue being empty.

`e1000_receive()` in `kern/e1000.c`

```
int e1000_receive(void* buf, size_t size){

int tail = E1000_REG(E1000_RDT);
int next = (tail + 1) % NREDESC;          // ⚠️ 这里所有的描述符都是尾描述符的下一个描述符
int length;
// struct e1000_re_desc* tail_desc = &(e1000_re_queue[tail]);
struct e1000_re_desc* tail_next_desc = &(e1000_re_queue[next]);
 if(!(tail_next_desc->status & E1000_RXD_STAT_DD)){
     // state is not DD
     return -1;
 }
 // if(((length = tail_desc->length) > size)){
 if((length = tail_next_desc->length) > size){
     // the length of received packet is larger than RE_BUF_SIZE
     return -2;
 }
 // memcpy(buf, e1000_re_buf[tail], size);
 memcpy(buf, e1000_re_buf[next], length);
 // clear DD
 tail_next_desc->status &= ~E1000_RXD_STAT_DD;
 E1000_REG(E1000_RDT) = next;
```

```
 return length;
}

To add a system call.

diff --git a/inc/lib.h b/inc/lib.h
index 5763e77..0dab8aa 100644
--- a/inc/lib.h
+++ b/inc/lib.h
@@ -61,6 +61,7 @@ int   sys_ipc_try_send(envid_t to_env, uint32_t
value, void *pg, int perm);
int    sys_ipc_recv(void *rcv_pg);
unsigned int sys_time_msec(void);
int sys_net_transmit(const void* buf, size_t size);
+int sys_net_recv(void* buf, size_t size);

+++ b/inc/syscall.h
@@ -19,6 +19,7 @@ enum {
     SYS_ipc_recv,
     SYS_time_msec,
     SYS_net_transmit,
+       SYS_net_recv,
        NSYSCALLS
 };

diff --git a/kern/syscall.c b/kern/syscall.c
index 32f410a..5ca3718 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -472,6 +472,16 @@ sys_net_transmit(const void* buf, size_t
size){
        user_mem_assert(curenv, buf, size, PTE_U);
        return e1000_transmit(buf, size);
 }
+
+// Receive a packet from network in user space
+//
+// Return 0 on success, < 0 on error.
+static int
```

```
+sys_net_recv(void* buf, size_t size){
+       user_mem_assert(curenv, buf, size, PTE_U);
+       return e1000_receive(buf, size);
+}
+
 // Dispatches to the correct kernel function, passing the
arguments.
 int32_t
 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t
a3, uint32_t a4, uint32_t a5)
@@ -518,6 +528,8 @@ syscall(uint32_t syscallno, uint32_t a1,
uint32_t a2, uint32_t a3, uint32_t a4,
                return sys_time_msec();
        case SYS_net_transmit:
                return sys_net_transmit((const void*)a1,
(size_t)a2);
+       case SYS_net_recv:
+               return sys_net_recv((void*)a1, (size_t)a2);
        default:
                return -E_INVAL;
        }

diff --git a/lib/syscall.c b/lib/syscall.c
index 9445b9b..d950969 100644
--- a/lib/syscall.c
+++ b/lib/syscall.c
@@ -128,3 +128,9 @@ sys_net_transmit(const void* buf, size_t size)
 {
        return (int) syscall(SYS_net_transmit, 0, (uint32_t)buf,
size, 0, 0, 0);
 }
+
+int
+sys_net_recv(void* buf, size_t size)
+{
+       return (int) syscall(SYS_net_recv, 0, (uint32_t)buf, size,
0, 0, 0);
+}
```

Receiving Packets: Network Server

在网络服务器 *输入环境* 中，你将需要使用你的新系统调用来接收数据包并将它们传递给 *核心网络服务器环境* ，使用 `NSREQ_INPUT` IPC信息。

这些IPC输入信息应该附加一个页，页中有一个 `union Nsipc` ，在其 `struct jif_pkt pkt` 字段中填入从网络接收的数据包。

Exercise 12.

Implement `net/input.c` .

`input()` in `ner/input.c`

```
extern union Nsipc nsipcbuf;

void
input(envid_t ns_envid)
{
  binaryname = "ns_input";

  // LAB 6: Your code here:
  //   - read a packet from the device driver
  //   - send it to the network server
  // Hint: When you IPC a page to the network server, it will be
  // reading from it for a while, so don't immediately receive
  // another packet in to the same physical page.
uint8_t inputbuf[INPUT_BUFSIZE];
int r, i;
  while(1){
    memset(inputbuf, 0, INPUT_BUFSIZE);

    while((r = sys_net_recv(inputbuf, sizeof(inputbuf))) == -1){
      sys_yield();
    }
    if(r < 0){
```

```
        panic("%s: inputbuf too small\n", binaryname);
      }
      nsipcbuf.pkt.jp_len = r;
      memcpy(nsipcbuf.pkt.jp_data, inputbuf, r);
      ipc_send(ns_envid, NSREQ_INPUT, &nsipcbuf, PTE_P | PTE_U);
      // send it to the network server
      sys_yield();

    }
  }
```

Run `testinput` again with `make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-net_testinput`. You should see

```
Sending ARP announcement...
Waiting for packets...
e1000: index 0: 0x26dea0 : 900002a 0
e1000: unicast match[0]: 52:54:00:12:34:56
input: 0000   5254 0012 3456 5255  0a00 0202 0806 0001
input: 0010   0800 0604 0002 5255  0a00 0202 0a00 0202
input: 0020   5254 0012 3456 0a00  020f 0000 0000 0000
input: 0030   0000 0000 0000 0000  0000 0000 0000 0000
```

The lines beginning with "input:" are a hexdump of QEMU's ARP reply.

Your code should pass the `testinput` tests of make grade. Note that there's no way to test packet receiving without sending at least one ARP packet to inform QEMU of JOS' IP address, so bugs in your transmitting code can cause this test to fail.

To more thoroughly test your networking code, we have provided a daemon called `echosrv` that sets up an echo server running on port 7 that will echo back anything sent over a TCP connection. Use make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-echosrv to start the echo server in one terminal and make nc-7 in another to connect to it. Every line you type should be echoed back by the server. Every time the emulated E1000 receives a packet, QEMU should print something like the following to the console:

```
e1000: unicast match[0]: 52:54:00:12:34:56
e1000: index 2: 0x26ea7c : 9000036 0
e1000: index 3: 0x26f06a : 9000039 0
e1000: unicast match[0]: 52:54:00:12:34:56
```

At this point, you should also be able to pass the `echosrv` test.

The Web Server

一个网络服务器最简单的功能就是发送一个文件的内容给正在请求的客户端。

我们已经提供了一个非常简单的网络服务器的*skeleton code*，位于`user/httpd.c`。*skeleton code*处理到来的*连接*并解析*headers*。

Exercise 13.

The web server is missing the code that deals with sending the contents of a file back to the client. Finish the web server by implementing `send_file` and `send_data`.

`send_file` in `user/httpd.c`

```
static int
send_file(struct http_request *req)
{
  int r;
  off_t file_size = -1;
  int fd;

  // open the requested url for reading
  // if the file does not exist, send a 404 error using send_error
  // if the file is a directory, send a 404 error using send_error
  // set file_size to the size of the file

  // LAB 6: Your code here.
  // panic("send_file not implemented");
  if((fd = open(req->url, O_RDONLY)) < 0){
    if((r = send_error(req, 404)) < 0){
      goto end;
```

```
    }
  }
struct Stat stat;
  if((r = fstat(fd, &stat)) < 0){
    goto end;
  }
  if(stat.st_isdir){
    close(fd);
    return send_error(req, 404);
  }

  if ((r = send_header(req, 200)) < 0)
    goto end;

  if ((r = send_size(req, file_size)) < 0)
    goto end;

  if ((r = send_content_type(req)) < 0)
    goto end;

  if ((r = send_header_fin(req)) < 0)
    goto end;

  r = send_data(req, fd);

end:
  close(fd);
  return r;
}
```

`send_data` in `user/httpd.c`

```
static int
send_data(struct http_request *req, int fd)
{
  // LAB 6: Your code here.
  // panic("send_data not implemented");
int r;
char buf[BUFFSIZE];
  while ((r = read(fd, buf, BUFFSIZE)) > 0) {
```

```
        if (write(req->sock, buf, r) != r) {
            die("Failed to send bytes to client");
        }
    }
    return 0;
}
```

Once you've finished the web server, start the webserver (`make run-httpd-nox`) and point your favorite browser at http://host:port/index.html, where host is the name of the computer running QEMU (If you're running QEMU on athena use hostname.mit.edu (hostname is the output of the hostname command on athena, or localhost if you're running the web browser and QEMU on the same computer) and port is the port number reported for the web server by make which-ports . You should see a web page served by the HTTP server running inside JOS.

At this point, you should score 105/105 on make grade.

This completes the lab. As usual, don't forget to run `make grade` and to write up your answers and a description of your challenge exercise solution. Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab6.txt` . When you're ready, commit your changes with `git commit -am 'my solutions to lab 6` ', then `make handin` and follow the directions.