Lab6 Solution

516030910006 方俊杰

Exercise 1.

在时钟中断到来即trapno 等于 IRQ_OFFSET + IRQ_TIMER 的时候加上对 time_tick 的调用来步进计时器。
实现对系统调用 sys_time_msec 的分发和实现，其仅是对 time_msec 的封装。

Exercise 2.

浏览 Intel E1000 手册。对 E1000 网卡接收和发送数据包有一定了解

Exercise 3.

E1000 作为 PCI 设备会在初始化时扫描 PCI Bus，匹配 pci_driver 结构体以调用对应的 attach 函数。在 pci.c 的 pci_attach_vendor 数组中添加一个新的设备条目，对应的常量使用 e1000.h 中的宏。
目前仅在对应的 pci_e1000_attach 函数中 enable 设备。

```
struct pci_driver pci_attach_vendor[] = {
        {E1000_VID, E1000_DID, &pci_e1000_attach},
        { 0, 0, 0 },
};

int
pci_e1000_attach(struct pci_func *pcif)
{
        // Enable PCI function
        // Map MMIO region and save the address in 'base;
        pci_func_enable(pcif);
        return 0;
}
```

Exercise 4.

使用类似 Lab4 中对 LAPIC 的操作为 struct E1000 在内核中分配 MMIO 空间。
通过调用mmio_map_region（在lab4中编写以支持LAPIC的内存映射）为E1000的BAR 0创建虚拟内存映射。 然后将返回的地址转换为struct E1000 *，用静态变量保存。

```
int
pci_e1000_attach(struct pci_func *pcif)
{
        // Enable PCI function
        // Map MMIO region and save the address in 'base;
        pci_func_enable(pcif);
```

```
        // base: 0xfebc0000, size: 0x20000
        base = mmio_map_region(pcif->reg_base[0], pcif->reg_size[0]);
        assert(base->STATUS == 0x80080783);

        return 0;
}
```

Exercise 5.

实现手册 14.5 节对 transmit 的初始化：

1. 分配一块内存用作发送描述符队列，起始地址要16字节对齐。用基地址填充(TDBAL/TDBAH)寄存器。
2. 设置(TDLEN)寄存器，该寄存器保存发送描述符队列长度，必须128字节对齐。
3. 设置(TDH/TDT)寄存器，这两个寄存器都是发送描述符队列的下标。分别指向头部和尾部。应该初始化为0。
4. 初始化TCTL寄存器。设置TCTL.EN位为1，设置TCTL.PSP位为1。设置TCTL.CT为10h。设置TCTL.COLD为40h。
5. 设置TIPG寄存器
   很多需要用到的常量值都在 e1000.h 中有，参考 13 节 和 3.3.3 / 3.4 节

```
//struct tx_desc *tx_descs;
#define N_TXDESC (PGSIZE / sizeof(struct tx_desc))
struct tx_desc tx_descs[N_TXDESC] __attribute__((aligned(16)));
char tx_buf[N_TXDESC][TX_PKT_SIZE];

int
e1000_tx_init()
{
    // Allocate one page for descriptors

    // Initialize all descriptors
    memset(tx_descs, 0, sizeof(tx_descs));
    memset(tx_buf, 0, sizeof(tx_buf));
    for (int i=0; i<N_TXDESC; i++) {
        tx_descs[i].addr = PADDR(tx_buf[i]);
        tx_descs[i].status |= E1000_TX_STATUS_DD;
    }
    // Set hardward registers
    // Look kern/e1000.h to find useful definations
    // set TD Base Address register
    base->TDBAL = PADDR(tx_descs);
    base->TDBAH = 0;
    // set TD Length register
    base->TDLEN = sizeof(tx_descs);
    // set TD head and tail register
    base->TDH = 0;
    base->TDT = 0;
    // set Transmit Control register
    base->TCTL |= E1000_TCTL_EN;
```

```
    base->TCTL |= E1000_TCTL_PSP;
    base->TCTL |= E1000_TCTL_CT_ETHER;
    base->TCTL |= E1000_TCTL_COLD_FULL_DUPLEX;
    // set Transmit Inter Packet Gap register
    base->TIPG = E1000_TIPG_DEFAULT;
    return 0;
}

int
pci_e1000_attach(struct pci_func *pcif)
{
        ......
        // initialize packet buffer
        e1000_tx_init();

        return 0;
}
```

Exercise 6.

通过检查下一个描述符是否空闲，设置下一个描述符和更新TDT来填充函数e1000_tx以发送数据包。
检查参数，获取 TDT，检查队列是否 full，设置缓冲区，设置 描述符参数，更新 TDT。

```
int
e1000_tx(const void *buf, uint32_t len)
{
        // Send 'len' bytes in 'buf' to ethernet
        // Hint: buf is a kernel virtual address
        if(buf == NULL || len < 0 || len > TX_PKT_SIZE)
                return -E_INVAL;
        uint32_t tdt = base->TDT;
        cprintf("tx tdt:%d, tdh:%d\n", tdt, base->TDH);
        // check the next descriptor is free
        if(!(tx_descs[tdt].status & E1000_TX_STATUS_DD))
                return -E_AGAIN;// seems queue full and should retry
        // set up next descriptor
        memset(tx_buf[tdt], 0 , sizeof(tx_buf[tdt]));
        memmove(tx_buf[tdt], buf, len);
        tx_descs[tdt].length = len;
        tx_descs[tdt].status &= ~E1000_TX_STATUS_DD;
        tx_descs[tdt].cmd |= E1000_TX_CMD_EOP;
        tx_descs[tdt].cmd |= E1000_TX_CMD_RS;
        // update TDT
        base->TDT = (tdt + 1) % N_TXDESC;

        return 0;
}
```

Exercise 7.

实现系统调用 sys_net_send，即对 e1000_tx 进行封装

Exercise 8.

实现 net/output.c：
在无限循环中进行两个操作

1. ipc_recv，读取到来自 CNS 的请求，把页面放到 Nsipc
2. sys_net_send，尝试通过syscall发送页面，使用 union Nsipc 的 pkt 字段，可能由于队列满会失败，需要重试

```
void
output(envid_t ns_envid)
{
        binaryname = "ns_output";
        uint32_t req, whom;
        int perm, r;
        while (true) {
                // recv req from core-ns env
                req = ipc_recv((int32_t *) &whom, &nsipcbuf, &perm);
                if (req != NSREQ_OUTPUT) {
                        cprintf("output env: ipc recv not
NSREQ_OUTPUT\n");
                        continue;
                }
                // syscall send pkt, if queue full, yield or loop
                while ((r = sys_net_send(nsipcbuf.pkt.jp_data,
nsipcbuf.pkt.jp_len)) < 0) {
                        if (r != -E_AGAIN)
                                panic("output env: send fail but not
full\n");
                        sys_yield();
                }
        }
}
```

Question 1.

How did you structure your transmit implementation? In particular, what do you do if the transmit ring is full?

transmit 在 output 环境中实现，使用 ipc 获取到核心网络环境发送的数据包，使用系统调用 sys_net_send 尝试发送，syscall 最终会调用 e1000_tx，在其实现中，获取到 TDT，即发送队列的尾部，通过查看 DD 位状态可以知道是否可用，如果不可用则代表队列满，此时返回 E_AGAIN，告诉 output 环境进行重试，故output 环境的实现中，对于 sys_net_send 的调用也是写在循环中

Exercise 9.

阅读第3.2节。 忽略有关中断和校验和卸载的任何内容，并且不必关心阈值的详细信息以及卡的内部缓存如何工作。

有关包地址过滤、接收描述符、接受描述符状态域、捕捉接收描述符、接收描述符队列结构及地址过滤器等内容。

Exercise 10.

类似 E5 参考 14.4节配置 receive 的初始化：

1. 为接收队列初始化空间
2. 设置地址接收寄存器为QEMU的MAC地址
3. 设置接收循环队列的基地址寄存器RDBAH/RDBAL及队列长度寄存器RDLEN
4. 设置队列头尾指针，注意与 transmit 不同
5. 设置接收控制寄存器

Exercise 11.

类似传输，实现 e1000_rx 和 系统调用 sys_net_recv。
系统调用 sys_net_recv 的参数 len 似乎没用，我的实现把从网卡获取到的 data 的 len 作为返回值

Exercise 12.

实现 input 环境，与 output 对称。
注意 hint，ipc_send 给核心网络服务环境的 nsipcbuf 不能立刻被覆写。可以使用每次分配新的内存给从网卡接收到的数据包，也可以在 ipc_send 之后让出 CPU 以使得核心网络环境完成读取，但是不稳定。

Question 2.

How did you structure your receive implementation? In particular, what do you do if the receive queue is empty and a user environment requests the next incoming packet?

receive 中我的实现就是轮询,类似传输，每次调用 sys_net_recv 尝试收包，然后使用 ipc_send 发送给核心网络环境。特别的，如果队列是空的即网卡没有收到包，那么通过读取到的RDT状态可以判断，然后返回 E_AGAIN，input 环境就会放弃CPU等待下次被调度到重试收包过程

Exercise 13.

实现 user/httpd.c 中的 send_file和send_data完成Web服务器。根据要求了解 socket、fd 的用法实现上述两个函数

```
static int
send_data(struct http_request *req, int fd)
{
        // LAB 6: Your code here.
        //panic("send_data not implemented");
        struct Stat stat;
        int r;
        if(fstat(fd, &stat) < 0){
                return 0;
        }
        char *buf = malloc(stat.st_size);
```

```
        if((r = readn(fd, buf, stat.st_size)) < 0){
                return 0;
        }
        if((r = write(req->sock, buf, stat.st_size)) != stat.st_size){
                panic("send data fail, %d sent\n", r);
        }
        return 0;
}

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");
        char path[MAXPATHLEN];
        struct Stat stat;
        memmove(path, req->url, strlen(req->url));
        if((fd = open(path, O_RDONLY)) < 0){
                send_error(req, 404);
                goto end;
        }
        if((r = fstat(fd,&stat)) < 0){
                goto end;
        }
        if(stat.st_isdir){
                send_error(req, 404);
                goto end;
        }
        file_size = stat.st_size;

        ......

        r = send_data(req, fd);

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

Question

3. What does the web page served by JOS's web server say?

```
<html>
        <head>
                <title>jhttpd on JOS</title>
        </head>
        <body>
                <center>
                        <h2>This file came from JOS.</h2>
                        <marquee>Cheesy web page!</marquee>
                </center>
        </body>
</html>
```

显示的是 “This file came from JOS.” 然后 “Cheesy web page!” 在下面滚动

4. How long approximately did it take you to do this lab?
   大概 25 个小时。
   遇到了一些问题耗费了较多时间：比如 QEMU 版本问题耗费了较长时间配置环境，接收数据包调试了很久等。

Challenge

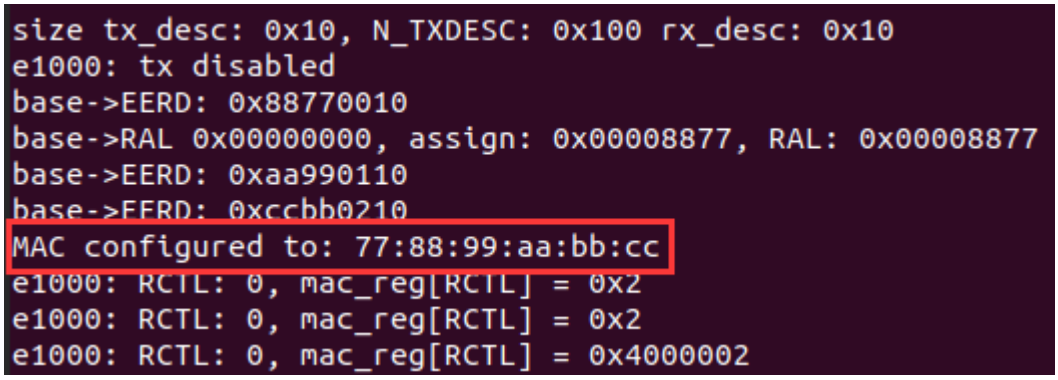
使用 EEPROM 加载 E1000 网卡的 MAX 地址，配置其他 MAC 地址。
首先查阅了手册了解 EEPROM 的使用方法，其格式见手册 13.4.4 节，有 START、DONE、ADDR、DATA 四个字段。当读取一个 word 时，软件在 ADDR 写入需要读取数据的地址，同步在 START字段写入 1b，然后 EEPROM 会读取数据放到 DATA 字段，并设置 DONE 为 1b。去读时可以轮询该字段直到 DONE 被设置，此时就可以使用 DATA 字段值。
另外需要一个新的 syscall 来获取 E1000 网卡的 MAC 值供 lwip 使用。
实现:
在 inc/lib.h、inc/syscall.h、lib/syscall.c、kernel/syscall.c 里实现并分发一个新的 syscall 叫做 sys_net_get_mac 用于调用新定义在 e1000.h 中的 e1000_read_mac 函数，该函数会返回 E1000 网卡设置的 MAC 地址。
在 e1000.h 中使用了条件编译，声明了一个 DY_MAC 变量，在其被定义时，会宏定义一系列用于 EEPROM 读取的常量，在 e1000.c 定义两个静态变量 RAL、RAH 保存设置的 MAC 地址，在 e1000_rx_init 函数中会使用 EEPROM 进行三次读取操作并保存得到的值。

```
        // Challenge: read MAC address from EEPROM
        base->EERD = 0x0 << E1000_EEPROM_RW_ADDR_SHIFT;
        base->EERD |= E1000_EEPROM_START;
        while (!(base->EERD & E1000_EEPROM_DONE));
        base->RAL = (base->EERD >> 16);
        RAL = (base->EERD >> 16);

        base->EERD = 0x1 << E1000_EEPROM_RW_ADDR_SHIFT;
        base->EERD |= E1000_EEPROM_START;
        while (!(base->EERD & E1000_EEPROM_DONE));
        base->RAL |= base->EERD & 0xffff0000;
        RAL |= base->EERD & 0xffff0000;
```

```
        base->EERD = 0x2 << E1000_EEPROM_RW_ADDR_SHIFT;
        base->EERD |= E1000_EEPROM_START;
        while (!(base->EERD & E1000_EEPROM_DONE));
        cprintf("base->EERD: 0x%08x\n", base->EERD);
        base->RAH = base->EERD >> 16;
        RAH = base->EERD >> 16;
        cprintf("MAC configured to: %02x:%02x:%02x:%02x:%02x:%02x\n",
                RAL & 0xff, (RAL>>8) & 0xff, (RAL>>16) & 0xff,
(RAL>>24) & 0xff,
                RAH & 0xff, (RAH>>8) & 0xff);
```

然后修改 GNUmakefile 163 行 QEMUOPTS -net 加上 macaddr='77:88:99:AA:BB:CC' 选项
运行测试命令 make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-net_testinput
得到输出



但是网卡没有正常工作，查看原因发现 E1000 网卡的数据中 RAL 和 RHL 在赋值后仍旧是无效的，无法解决。