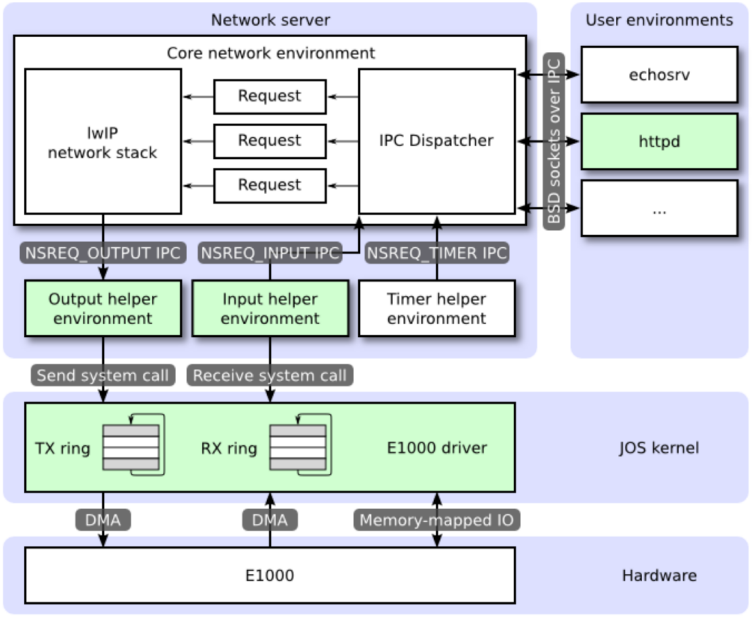
OS-Lab6 设计文档

516030910293 姚子航

一、Network Server

在本次的 lab 中,JOS 依旧采用 ExoKernel 的架构来设计网络栈，类似于文件系统，我们使用三个帮助环境来完成 Network Server 与内核的交互，而具体的网络协议并不是操作系统课程的中心，因此我们使用 LWIP 这个库来实现 TCP/IP 协议的内容，下面我们根据下方架构图来逐个分析整个网络栈的各个模块。



二、The Core Network Server Environment

核心的网络服务器环境包括了 socket call 的 dispatcher 以及 LWIP 这个库，dispatcher 类似于文件系统服务器，它提供了一些 IPC 接口来给 socket 接口使用，这些 IPC 被发到 network server 后，会交给 IWIP 来帮助我们实现一些 BSD socket 的功能，而 user environment 只需要调用 socket.c 中提供的一系列 socket API 就可以，在 JOS 中，socket 接口被设计为类似 file descriptor，用户可以像操作一个文件一样向 socket 中读写数据。

在文档中还提到了一个要点，由于 BSD socket 接口会出现 block 阻塞的情况，为了使 network server 正常运行，我们每次处理请求都会使用一个新的 thread 来处理请求，这个部分的逻辑写在 net/serv.c 中的 serve_thread 函数中，即为图片上的 IPC dispatcher，而 Output helper Environment，Input helper Environment， Timer helper Environment 都是这个 Core Network server fork 出来的环境。 我们实现的代码部分主要就是这三个帮助环境的逻辑以及更底层的 E1000 网卡驱动代码。

三、The Output Environment & Input Environment & Timer Environment

Output Environment 的功能是向网卡发送系统调用发送包，它的逻辑就是在一个循环中不断接收 LWIP 用 IPC 传过来的包，之后在调用网卡的系统调用发包。

Input Environment 的功能是不断调用系统调用收包，在收到包之后就调用 IPC 与 core network server 进行通信将收来的包传入 LWIP，再根据协议解包后就会使用 IPC 传送到目的地用户环境。

Timer Environment 的功能较为简单，就是不断地在一个间隔给 core network Environment 发送 IPC，方便 LWIP 实现 timeout 机制。

四、本次 Lab & challenge

由图中所示，本次 lab 我们实现的部分被绿色所标注，即网卡驱动的 kernel 函数，output helper 和 input helper 的主体逻辑及一点点 httpd 的功能补充，总体来讲难度主要集中在没有任何骨架，整体的 transmit，receive 驱动函数需要我们透彻的了解硬件机制，而 Output 和 Input 函数需要我们透彻的理解上述架构图的运行流程。

本次 challenge 我选择的是实现从硬件 EEPROM 读取 mac 地址，在翻阅开发文档后，我根据硬件的设置做了如下调整：

1.更改 E1000 结构体的定义，将 EERD 寄存器暴露出来：

```
struct E1000 {
    volatile uint32_t CTRL;          /* 0x00000  Device Control - RW */
    volatile uint32_t CTRL_DUP;      /* 0x00004  Device Control Duplicate (Shadow) - RW */
    volatile const uint32_t STATUS;  /* 0x00008  Device Status - RO */
    uint32_t reserved1;
    volatile uint32_t EECD;          /* 0x00010 EEPROM/Flash Control - RW */
    volatile uint32_t EERD;          /* 0x00014 EEPROM Read - RW */
    uint32_t reserved[46];
    ...
}
```

2.之后根据功能需要，我添加了三个接口：

```
uint16_t read_word_from_EEPROM(uint8_t addr);
uint32_t read_mac_low_address();
uint32_t read_mac_high_address();
```

这三个接口的功能很显然：从 EEPROM 读取一个 word、读取 mac 的高位低位地址

从 EEPROM 读取数据的流程我使用了 EERD(EEPROM Read register),具体实现如下：

1).在 EERD 的 addr 部分写入读取地址（mac 地址的三个字节分别在 0x00,0x01,0x02）

2).将 EERD.START 位置 1

3).不断轮询 EERD.DONE 位直到被置为 1

4).从 EERD.DATA 部分读取数据，清空 EERD 寄存器

```
uint16_t
read_word_from_EEPROM(uint8_t addr){
    //fill address in EERD
    base->EERD &= (~0xfffc);
    base->EERD |= (addr << 8);
    //start read
    base->EERD |= E1000_EERD_START;
    //poll until read is done
    while((base->EERD & E1000_EERD_DONE) == 0) ;
    base->EERD &= (~E1000_EERD_DONE);
    uint16_t data = (base->EERD >> 16) & 0xffff;
    base->EERD &= 0xffff;
    return data;
```

```
}

uint32_t
read_mac_low_address(){
	uint32_t mac_word0 = read_word_from_EEPROM(0x00);
	uint32_t mac_word1 = read_word_from_EEPROM(0x01);
	return (mac_word0 | (mac_word1 << 16));
}

uint32_t
read_mac_high_address(){
	uint32_t mac_word2 = read_word_from_EEPROM(0x02);
	return mac_word2;
}
```

3.之后，我将上述接口做成 syscall 系统调用暴露给 user，在 net/lwip/jos/jif/jif.c 的 low_level_init 函数中使用 syscall：

```
uint32_t low_addr = sys_read_mac_low_address();
uint32_t high_addr = sys_read_mac_high_address();
// netif->hwaddr[0] = 0x52;
// netif->hwaddr[1] = 0x54;
// netif->hwaddr[2] = 0x00;
// netif->hwaddr[3] = 0x12;
// netif->hwaddr[4] = 0x34;
// netif->hwaddr[5] = 0x56;
netif->hwaddr[0] = (low_addr&0xff);
netif->hwaddr[1] = ((low_addr>>8)&0xff);
netif->hwaddr[2] = ((low_addr>>16)&0xff);
netif->hwaddr[3] = ((low_addr>>24)&0xff);
netif->hwaddr[4] = (high_addr&0xff);
netif->hwaddr[5] = ((high_addr>>8)&0xff);
```

更改 QEMU 设置的 mac 地址，运行 make grade，分数未发生改动，证明我的实现是正确的。