结题报告

基于xv6内核的网络子系统实现

姓　　名	**熊恪峥** 肖凯欣 谢哲涵 赵淇奥
学　　号	22920202204622
日　　期	2022年12月22日
学　　院	信息学院
课程名称	计算机网络与通信

目录

第一章　分工

表格 1.1: 成员及分工

成员	分工
熊恪峥	项目总体设计、PCI和网卡驱动程序、Socket API和POSIX NetDB API编写、用户库支持、更先进的内存管理和文件系统支持的实现、daytime程序编写
肖凯欣	ping/traceroute命令实现
赵淇奥	nslookup命令实现
谢哲涵	httpget命令实现

第二章　项目介绍

一、选题介绍

由6.828 Lab 6 [1]得到的灵感，我们将为xv6内核实现完整的网络子系统以及相应的API。

6.828课程中讲授xv6内核 [2]的实现，并在实验中要求学生在称之为JOS的实验性微内核操作系统中添加相应的功能。但事实上，xv6内核本身的结构和现代网络服务器中常常使用的Linux和FreeBSD更为类似。因此，我们好奇，我们是否可以对xv6内核进行修改，来添加对网络功能的支持，而不是在JOS上进行实现。这样，我们能够更好地通过实现相应的功能和适配相应的库来加深对各种网络协议以及Socket API本身的理解。

1、xv6简介

xv6是对Dennis Ritchie's and Ken Thompson的Unix Version 6 (v6)的重新实现。xv6与v6有大致相同的结构和代码风格，但是使用了ANSI C实现，并且为现代的处理器功能，如对称多核心等做出了相应的适配。它主要为MIT的操作系统教学而设计。

2、lwIP简介

lwIP [3]是一个开源的、轻量级的TCP/IP协议栈，它是一个可移植的、可裁剪的、可靠的、高效的、低成本的TCP/IP协议栈，它可以用于嵌入式系统，如微控制器、单片机、DSP等。 lwIP最初是由Swedish Institute of Computer Science的Adam Dunkels开发，现在由开源社区开发和维护。并且被多个技术公司用于商业产品中，如Intel/Altera, Analog Devices, Xilinx, TI, ST 和 Freescale。

二、项目内容

为了实现完整的网络子系统，我们需要实现以下内容：

- 网卡驱动程序
- IP协议栈
- Socket API等应用编程接口支持、DNS等辅助功能的支持
- 用户库支持、网络应用程序
- 操作系统内核其他组件的增强

因此，我们首先实现了Intel e1000网卡的驱动程序。该设备被模拟器QEMU较好地支持，因此易于测试。为了使该设备能够正常运行，需要实现PCI总线的支持。然后，我们移植了LWIP作为IP协议栈，它提供包括IP、ICMP、UDP和TCP协议的支持。为了支持该库的正常运行，我们实现了相应的时钟中断处理程序及其各组件的初始化。基于该协议栈，我们首先提供了以Socket API为代表的网络编程接口的支持，然后实现了以符合POSIX标准的`gethostbyaddr`和`gethostbyname` 为接口DNS的支持。最后，

我们实现了包括基础的TCP/UDP Echo Server、例如nslookup、ping、traceroute、httpget等常见的实用程序，以及符合RFC 867的日期时间协议的授时客户端。

此外，在开发过程中，我们严格采用了版本控制系统Git，以便于团队成员之间的协作开发。在开发流程上，我们坚持通过fork/pull request的方式进行代码的提交和合并，以保证代码的质量、防止不当更改损坏项目的其他部分。我们的工作流程和现代的软件开发工作流程高度一致、完全接轨。实现了有效的项目管理。在开发过程中，我们严格遵循了谷歌开源项目风格指南的代码规范，以保证代码的可读性和可维护性。

我们在开发结束后贯彻“从开源中来、到开源中去”的理念，将我们的项目以MIT协议开源，以供其他人参考和使用。项目地址是`https://github.com/SmartPolarBear/xv6_enhanced`

第三章　项目实现

一、总体架构

总体的项目架构如图 3.1所示。

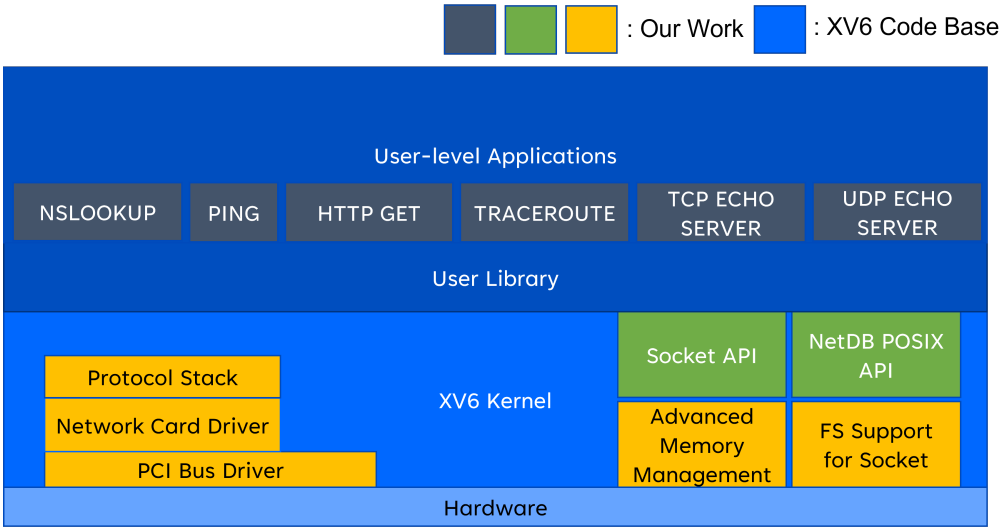


图 3.1: 项目架构

在内核态中的部分按功能分，可以分为三个部分：

一是负责处理实际数据收发、通讯的部分，自下而上分别是PCI总线的驱动、e1000网卡的驱动程序、基于LWIP的网络协议栈。二是作为向上层提供服务的部分，包括提供一套数据收发原语的Socket API、提供一套使用DNS服务的接口的POSIX NetDB API部分。三是围绕网络子系统内存分配频繁、动态内存分配要求高的需求、Socket API与类UNIX操作系统的文件系统部分结合紧密的特点，对内核进行的改进。例如对内存分配的改进、对文件系统的改进、引入了与Linux内核中实现方式类似的链表等基础数据结构。

在内核和用户之间，用户态库也需要进行相应接口的实现。我们实现了Socket API和用于查询DNS的接口，例如 `gethostbyaddr`和`gethostbyname`的用户态库接口。这些接口完全符合POSIX标准。

在用户态，我们实现了两类的应用程序：

第一类是用来调试、检验基础功能的应用程序。包括TCP、UDP的Echo Server、测试和远程主机实际通信以及DNS的 RFC 867日期时间服务客户端。第二类是常用的网络实用程序，包括nslookup、ping、traceroute、httpget等。

二、设备驱动程序的实现

1、PCI总线的驱动

通用串行总线（PCI）是一种用于确保高性能、低开销的传输的本地总线。PCI总线的组件和卡片接口时处理器无关的，因此该总线结构可以被多种处理器架构利用。

为了初始化PCI总线，需要驱动程序访问PCI配置空间，从而获取PCI设备的基本信息，例如设备的ID、设备的基地址等。访问PCI配置空间有两种方式，但只有访问I/O端口的方式是符合标准的。这种方式使用两个32位的I/O端口，一个用于指定配置地址，一个用于读写配置数据。配置地址的格式如图 3.2所示。据此，我们实现的PCI驱动程序在代码`kernel/pci.c`中，它会访问PCI配置空间，获取PCI设

图 3.2: PCI配置地址的格式

Bit 31	Bits 30-24	Bits 23-16	Bits 15-11	Bits 10-8	Bits 7-0
Enable Bit	Reserved	Bus Number	Device Number	Function Number	Register Offset[1]

备的基本信息，然后加入设备表中。为了保证该驱动程序的可扩展性，我们的实现将不同驱动程序实现的相应初始化程序注册到数组中，如代码 3.1。

代码 3.1: PCI驱动程序的初始化程序注册

```
1 struct pci_driver pci_attach_vendor[] = {
2         {0x8086, 0x100e, &e1000_nic_attach},
3         {0x1af4, 0x1000, &virtio_nic_attach},
4         {0, 0, NULL},
5 };
```

当驱动程序初始化的时候，每当注册新设备，都会在其中查找相应的初始化程序，然后调用该初始化程序。查找的方式是读取设备类和设备ID，分别位于配置空间的偏移0x0b和0x0a中。如果找到相应的初始化程序，就调用该初始化程序。

当PCI驱动程序完成后，操作系统启动时，会调用`pci_init`函数，该函数会遍历所有的PCI设备，完成上述功能，并输出设备的基本信息。输出的信息如图 3.3所示。

图 3.3: PCI设备的基本信息

2、e1000网卡的驱动程序

有了上述PCI驱动程序和相应扩展架构的实现。我们就可以实现e1000网卡的驱动程序了。e1000网卡的驱动程序的重要组成部分是网卡的初始化程序，该程序会初始化网卡的寄存器，设置相应信息，使网卡进入可用状态。

网卡的初始化

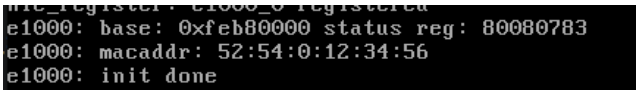
有了完善的PCI驱动程序，e1000网卡的初始化在操作上就是对相应寄存器地址的读写。方法是调用PCI驱动程序的`pciw`和`pcir`函数，分别用于写和读。初始化大致分为以下几个步骤：

1. 从PCI配置空间中读取网卡的基地址寄存器，得到网卡的基地址。
2. 从e1000的eeprom中读取MAC地址。
3. 设置多播表为全0。
4. 设置接受和发送的缓冲区等相关设置。

由于本项目并没有要求大批数据的收发，只需要基本功能，因此e1000网卡工作在轮询模式下，不使用中断。 e1000网卡的收、发分别使用64、16个物理内存页作为缓冲区，其中每两个Receive Descriptor和Transmission Descriptor 共用一个缓冲区，因此每个缓冲区的大小是2KB。这样划分可以较好地保证缓冲区的对齐。

驱动程序完成后，会在启动时输出网卡相关的硬件信息，如图 3.4所示。52:54:00:12:34:56是QEMU默

图 3.4: 网卡的硬件信息



认的虚拟网卡MAC地址，可见实现是正确的。

网卡的收发

从网卡收发数据是通过写相应的Read/Transmission Descriptor而实现的。默认情况下，这些Descriptor的Status字段为0，表示准备好接收或发送数据。当网卡收到数据时，会将数据写入相应的缓冲区，并将Status字段置为0，命令字段设置RS和EOP，表示数据已经正确写入，要求网卡回报状态。然后再在相应的寄存器中写入使用的Descriptor 的编号。

之后，驱动程序轮询网卡的状态寄存器，当网卡的状态寄存器中的RD和TD字段都为1时，表示数据已经正确接收或发送。其中发送部分的代码如代码 3.2所示。

代码 3.2: 数据发送

```
1  int e1000_net_send(void *state, const void *data, int len)
2  {
3          netdev_t *card = (netdev_t *)state;
4          e1000_t *e1000 = (e1000_t *)card->priv;
5          uint32 tail = pcir(e1000, E1000_TDT);
6          struct TD *next_desc = &e1000->tx_descs[tail];
7
8          if (!(next_desc->status & E1000_TXD_STAT_DD) && (next_desc->cmd & E1000_TXD_CMD_RS))
9          {
10                 return -1;
11         }
12
13         if (len > PKTSIZE)
14         {
15                 len = PKTSIZE;
16         }
17
18         memmove(e1000->tbuf[tail]->buf, data, len);
19         next_desc->length = len;
20         next_desc->status = 0;
21         next_desc->cmd = (E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP);
```

```
22
23          pciw(e1000, E1000_TDT, (tail + 1) % E1000_TXDESC_LEN);
24          while (!(next_desc->status & 0x0f))
25          {
26                  microdelay(1);
27          }
28          return len;
29  }
```

三、基于LWIP的网络协议栈

由于网络协议栈的实现工作量巨大、相当复杂。我们采用了常用的LWIP作为协议栈。该协议栈实现了常见用的TCP/IP协议，如IP、TCP、UDP、ICMP、ARP、DHCP、DNS等。一方面，该协议栈较为轻量级，一方面，配置该库只需要在一头文件中定义相应的宏。因此我们选择了该实现。

1、LWIP的配置

LWIP的配置主要是在`lwipopts.h`中进行的。该文件中定义了各种宏，用于控制LWIP的功能。

由于协议栈运行在内核态。因此在LWIP库的视角下，LWIP的代码工作在没有操作系统的模式下，因此，我们配置文件中主要的定义如代码 3.3所示。

代码 3.3: LWIP配置

```
1   #define NO_SYS 1
2
3   #define LWIP_NETCONN 0
4   #define LWIP_SOCKET 0
5
6   #define LWIP_ARP 1
7   #define LWIP_DHCP 1
8   #define LWIP_DNS 1
9   #define LWIP_ETHERNET 1
10  #define LWIP_ICMP 1
11  #define LWIP_RAW 1
```

首先，我们关闭了LWIP的socket和netconn接口，因为它们不能工作在无操作系统模式（NO_SYS）下，因此，我们需要禁用这些功能，仅保留基础的功能，然后用基础的API实现我们的Socket等接口。

我们启用了ARP、DHCP、DNS、Ethernet、ICMP和RAW协议。这些协议对于Socket API和POSIX NetDB API的实现是不可或缺的。

2、LWIP依赖函数的实现

NO_SYS下的LWIP只依赖极少数的函数，这些函数主要是用于时间管理、和随机数生成。这些函数实现在`kernel/lib/lwip/os/`中的文件中。分别是sys_now和lwip_rand。由于LWIP仅利用时间戳进行时间差的计算，不用于任何时间戳的标注。因此可以直接反馈系统时钟中断中记录的Ticks。而随机数的生成，我们采用了的是简单的线性同余法。这种方法能使用位运算快速实现，有利于内核态代码的高效工作。随机数生成的代码如代码 3.4所示。

代码 3.4: LWIP随机数生成

```
1   unsigned long lwip_rand(void)
2   {
3           uint32 time = sys_now();
4           static unsigned int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
```

```
5         unsigned int b;
6         b  = ((z1 << 6) ^ z1) >> 13;
7         z1 = ((z1 & time) << 18) ^ b;
8         b  = ((z2 << 2) ^ z2) >> 27;
9         z2 = ((z2 & time) << 2) ^ b;
10        b  = ((z3 << 13) ^ z3) >> 21;
11        z3 = ((z3 & time) << 7) ^ b;
12        b  = ((z4 << 3) ^ z4) >> 12;
13        z4 = ((z4 & time) << 13) ^ b;
14        return (z1 ^ z2 ^ z3 ^ z4);
15 }
```

3、LWIP的初始化和工作循环

LWIP的主要工作都在kernel/net/net.c中完成。首先，在NO_SYS模式下，要使用LWIP，需要用netif->input 函数将数据传递到LWIP中。需要注意的是，LWIP不原生支持并发，而XV6支持对称多处理器。因此，为了防止LWIP内部的数据结构出现不一致的情况，我们对任何使用LWIP的上下文都使用同一自旋锁保护。这意味着只有一个进程能同时访问网络功能。这虽然一定程度上降低了效率，但是是不可或缺的。

其次，为了驱动LWIP不断收取、分发、处理数据，需要不断调用sys_check_timeouts函数。因此，在xv6的IRQ0中断，即时钟中断中，我们调用该函数。这样，LWIP就能初步正常工作了。

为了实现正常的收发数据，我们需要做好网卡驱动程序和LWIP的对接。在初始化时，我们为每一个网卡都创建一个netif结构体。然后在其中设置网卡的MAC地址、IP地址、子网掩码、网关地址、MTU等信息。最后，我们调用netif_add函数将该网卡加入LWIP 的网络接口列表中。这样，LWIP就能够使用该网卡了。

但在收发数据前，首先，我们需要获取合适的IP地址，而非要求用户手动输入，这符合现代操作系统的实际。因此我们使用了DHCP协议。首先，调用LWIP的dhcp_start函数，然后若未获取IP地址，就反复试图从网卡中读入数据并交给LWIP处理。这样，LWIP 最终就能正确地设置网卡的IP地址了。这时，正在启动的操作系统内核会输出获取的IP地址等配置信息。这部分代码如代码??所示。

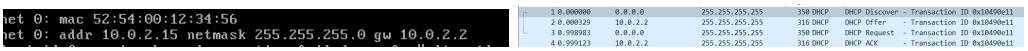
代码 3.5: 网卡初始化

```
1  void
2  netadd(int n)
3  {
4         struct netif *new = &netif[n];
5         netdev_t *card = find_card_by_id(n);
6
7         int i;
8         char addr[IPADDR_STRLEN_MAX], netmask[IPADDR_STRLEN_MAX], gw[IPADDR_STRLEN_MAX];
9
10        if (!netif_add_noaddr(new, (void *)card, linkinit, netif_input))
11        {
12                panic("netadd");
13        }
14
15        new->name[0] = 'e';
16        new->name[1] = 'n';
17
18        netif_set_link_up(new);
19        netif_set_up(new);
20
```

```
21          cprintf("net %d: mac ", n);
22          for (i = 0; i < ETH_HWADDR_LEN; ++i)
23          {
24                  if (i)
25                  {
26                          cprintf(":");
27                  }
28                  if (new->hwaddr[i] < 0x10)
29                  {
30                          cprintf("0");
31                  }
32                  cprintf("%x", new->hwaddr[i]);
33          }
34          cprintf("\n");
35
36          if (!card->opts->open || card->opts->open(card))
37          {
38                  panic("netadd: cannot open device\n");
39          }
40
41          dhcp_start(new);
42          /* wait until DHCP succeeds */
43          while (!dhcp_supplied_address(new))
44          {
45                  linkinput(new);
46                  sys_check_timeouts();
47          }
48
49          ipaddr_ntoa_r(netif_ip_addr4(new), addr, sizeof(addr));
50          ipaddr_ntoa_r(netif_ip_netmask4(new), netmask, sizeof(netmask));
51          ipaddr_ntoa_r(netif_ip_gw4(new), gw, sizeof(gw));
52          cprintf("net %d: addr %s netmask %s gw %s\n", n, addr, netmask, gw);
53
54  }
```

在实际运行中，QEMU虚拟机下的xv6输出如图3.5a所示。使用QEMU参数保存相应的数据包，如图3.5b所示。可见，一个正确的、典型的DHCP过程正常完成了。首先，没有配置好IP地址的网卡发送了一个DHCP Discover数据包，源地址是0.0.0.0。然后位于10.0.2.2的DHCP服务器以广播的方式返回了一个DHCP Offer数据包，这时，网卡继续以0.0.0.0作为源地址发送了一个DHCP Request数据包，请求DHCP服务器分配IP地址。最后，DHCP服务器返回了一个DHCP ACK数据包，这时，网卡就能正确地获取到IP地址、子网掩码和网关地址等重要信息。在此例中，分别为 10.0.2.15、255.255.255.0和10.0.2.2。

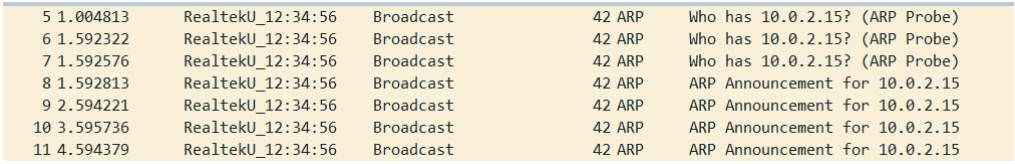
图 3.5: DHCP初始化



(a) DHCP获取的内容　　(b) 收发的数据包

由于时钟中断不断驱动LWIP进行工作，当DHCP初始化完后，网卡立即发送、接受ARP数据包，并且能够正常地进行网络通信。所捕捉的数据包如图3.6所示。 此时，所有驱动程序和网络协议栈都可以正常地协同工作。为实现重要的用户编程接口打下了良好的基础。

图 3.6: ARP数据包



5	1.004813	RealtekU_12:34:56	Broadcast	42 ARP	Who has 10.0.2.15? (ARP Probe)
6	1.592322	RealtekU_12:34:56	Broadcast	42 ARP	Who has 10.0.2.15? (ARP Probe)
7	1.592576	RealtekU_12:34:56	Broadcast	42 ARP	Who has 10.0.2.15? (ARP Probe)
8	1.592813	RealtekU_12:34:56	Broadcast	42 ARP	ARP Announcement for 10.0.2.15
9	2.594221	RealtekU_12:34:56	Broadcast	42 ARP	ARP Announcement for 10.0.2.15
10	3.595736	RealtekU_12:34:56	Broadcast	42 ARP	ARP Announcement for 10.0.2.15
11	4.594379	RealtekU_12:34:56	Broadcast	42 ARP	ARP Announcement for 10.0.2.15

四、Socket API

Socket API是一组用于网络编程的接口，它是POSIX标准的一部分。它提供了一组函数，来支持主要的用户程序间的通信功能，例如：

- 设置和建立与其它网络中用户的连接；
- 向其它网络中用户发送数据、从其它网络中用户接收数据；
- 关闭连接、释放资源。

此外，它还能够提供一些高级的功能，例如：

- 询问网络系统中的主机和状态；
- 按照其他用户的的要求，执行系统功能和控制功能；

在我们的项目中，Socket API处于核心位置，是耗费了最长时间、完成了最多代码的部分。

Internet Socket主要有三类：数据报套接字、流套接字和原始套接字。数据报套接字提供了无连接的Socket服务，主要用于UDP协议；流套接字提供了面向连接的Socket服务，常见的协议有TCP、SCTP等；原始套接字提供了对IP协议的完全访问，它允许用户构造任何类型的IP数据包，包括IP头部和数据部分。前两种对典型的网络应用程序来说十分重要，而原始套接字则是一种高级的功能，对我们在章节八中实现的网络应用程序nslookup、 ping和traceroute来说也相当重要，因此我们需要恰当第实现这三类套接字。

LWIP也提供了三种不同的API，分别为RAW IP API、RAW TCP API、RAW UDP API。它们提供了一层较为“薄”的封装，提供了构造这三种类型数据包的基础接口。我们将借助它们实现Socket API。

注意到Socket类型和使用的协议不一定是一一对应的，例如，对于流套接字，用户可以使用TCP协议也可以使用STCP。这样一来，我们并不应该在实现中将利用不同协议的Socket实现耦合在一起，而应当进行逻辑上和实现上的分离，这样才能保证这一部分的可扩展性和可维护性。

1、架构设计

首先，我们将Socket的基本操作定义在结构体sockopt中，如代码 ??所示。

代码 3.6: sockopts结构体

```
1 typedef struct sockopts
2 {
3         struct
4         {
5                 int type, protocol;
6         } meta;
7
8         int (*alloc)(struct socket *s);
```

```
9         int (*connect)(struct socket *s, struct sockaddr *addr, int addrlen);
10        int (*bind)(struct socket *s, struct sockaddr *addr, int addrlen);
11        int (*listen)(struct socket *s, int backlog);
12        int (*accept)(struct socket *s, struct socket *newsock, struct sockaddr *addr, int *addrlen)
              ;
13        int (*send)(struct socket *s, void *buf, int len, int flags);
14        int (*recv)(struct socket *s, void *buf, int len, int flags);
15        int (*sendto)(struct socket *s, void *buf, int len, int flags, struct sockaddr *addr, int
              addrlen);
16        int (*recvfrom)(struct socket *s, void *buf, int len, int flags, struct sockaddr *addr, int
              *addrlen);
17        int (*close)(struct socket *s);
18
19        int (*getsockopt)(struct socket *s, int level, int optname, void *optval, int *optlen);
20        int (*setsockopt)(struct socket *s, int level, int optname, void *optval, int optlen);
21    } sockopts_t;
```

该结构体存储了一种针对特定“协议-Socket类型”二元组的Socket实现，它包含相应的元数据，以及一组函数指针。它们将指向具体的Socket实现，例如，对于TCP流套接字，它们将指向TCP流套接字的实现。

每支持一种新的“协议-Socket类型”二元组，我们只需要实现相应的Socket实现，并将其注册到全局数组sockopts中，它定义在文件kernel/net/socket.c中，如代码??所示。

代码 3.7: sockopts数组

```
1   sockopts_t *sockopts[] = {
2           &tcp_opts,
3           &udp_opts,
4           &raw_opts,
5   };
```

它显示了我们支持的三种“协议-Socket类型”二元组，分别是TCP流套接字、UDP数据报套接字和原始套接字。

每一个用户创建的Socket在内核中都会用一个socket结构体记录相应的参数，其中就有一个指针指向 sockopts数组中的某一项，这个指针在创建Socket时通过遍历上述数组来确定，如果无法找到，就可以向用户汇报错误。kernel/net/socket.c中任何其他函数实现的主要部分都是根据该指针来调用相应的Socket实现。并不实际处理通信的过程，而是将这些过程交给具体的Socket实现来处理，它们位于kernel/net/tcpsock.c、 kernel/net/udpsock.c、kernel/net/rawsock.c等具体的实现文件中。

这样，就完美实现了具有高度可扩展性的Socket实现。实现了不同协议、不同类型Socket实现在逻辑上和代码构成上的分离。使得调试、维护、扩展都变得非常方便。

上述sockopt结构体中包含了几乎全部的Socket API的定义，但不是所有具体的Socket实现都能够实现全部的API。对于那些不支持的操作，具体的实现只需要提供一个简单的、返回错误代码的函数实现即可。

下面，我们以UDP协议的数据报套接字为例，来介绍一种我们已经支持的协议和类型的Socket API的具体实现。

2、Socket API实现（以UDP为例）

其它的Socket实现也是类似的，可参见源代码中的kernel/net/tcpsock.c、 kernel/net/udpsock.c、kernel/net/rawsock.c 等具体的实现文件，这里就不再赘述了。

五、POSIX NetDB API和DNS解析

为了实现网络应用程序的方便，仅有Socket API是不够的。因为在现实场景中，用户极少会指定IP进行操作，而是使用域名。因此，需要实现一个域名解析的功能，即DNS解析。

现代类UNIX系统中，经常使用的接口是getaddrinfo和getnameinfo，然而这组API的实现比较复杂，功能比较多，不利于我们形成可以正常运行的系统。因此，我们选择实现较为传统的gethostbyname和gethostbyaddr接口。这两个接口虽然已经被POSIX标准弃用，但是实现简单、功能更为正交，因此更适合我们的实现。

1、架构设计

近年来，传统DNS解析的问题已经引发了广泛的关注。因此，新的DNS标准也正在普及，例如DNS over TLS、 DNS over HTTPS等。这些新的标准都是为了解决DNS的安全性问题，已经被Windows、Linux等主流操作系统所支持。因此，考虑到未来的发展，我们的DNS实现的架构也应当具有可扩展性。

另一个需要考虑的问题是，gethostbyname和gethostbyaddr的返回值是连续的内存块，其中变长内容由相对起始的偏移量表示，需要进行计算。因此，不把查询逻辑和构造返回值的逻辑分离开来，就会导致代码难以维护、难以扩展。

因此，我们将DNS的基本操作抽象为一个netdb_proto结构体，该结构体包含了DNS的基本操作，如代码 3.8所示。

代码 3.8: DNS操作结构体

```
typedef struct netdb_proto
{
        void (*proto_init)();
        int (*proto_query)(char *name, netdb_query_type_t type, netdb_answer_t **answer);
        int (*proto_free_answer)(netdb_answer_t *answer);
        int (*proto_dump_answer)(const netdb_answer_t *answer);
        void (*proto_shutdown)();
} netdb_proto_t;
```

其中，proto_init用于初始化DNS操作，proto_query用于查询，proto_free_answer用于释放查询结果， proto_dump_answer用于打印查询结果，proto_shutdown用于操作系统内核关闭时的清理。

为了分离构造返回值的逻辑，我们底层的DNS实现将DNS的查询结果构造成一个netdb_answer结构体组成的链表，该结构体包含了DNS的查询结果，而系统调用的实现会进一步将该链表转换为合理的hostent返回值。

这样，架构设计就能实现高度的可扩展性，同时达到了模块中高内聚、模块间低耦合的目的。

2、基本DNS查询的实现和测试

普通DNS使用明文传输、基于UDP协议。我们利用LWIP的UDP RAW API实现了DNS数据的传输。

而构造DNS报文的过程则比较复杂，因为DNS将字符串部分进行了压缩，为了构造和解析DNS报文，需要实现相应的解析逻辑。RFC 1035中对DNS报文的格式进行了详细的描述：

> In order to reduce the size of messages, the domain system utilizes a compression scheme which eliminates the repetition of domain names in a message. In this scheme, an entire domain name or a list of labels at the end of a domain name is replaced with a pointer to a prior occurance of the same name.

简而言之，DNS报文中的字符串部分会被压缩，压缩的方式是将字符串部分的每个子串都用一个指针指向之前出现过的子串，高位的两位用于标识该指针是否指向字符串的开始，如果是，则该指针指向的是字符串的开始。代码 3.9给出了字符串转化为正常字符串的实现。

代码 3.9: 字符串压缩解压缩

```
static inline char *parse_dns_name(char *dst, char *name, char *ans)
{
        char *p = name;
        char *d = dst;
        char *s = ans;
        int len = 0;
        int i = 0;
        int offset = 0;
        int first = 1;

        while (*p != 0)
        {
                if ((*p & 0xc0) == 0xc0)
                {
                        offset = (*p & 0x3f) << 8;
                        p++;
                        offset |= *p;
                        p = s + offset;
                        continue;
                }

                len = *p;
                p++;

                if (first)
                {
                        first = 0;
                }
                else
                {
                        *d = '.';
                        d++;
                }

                for (i = 0; i < len; i++)
                {
                        *d = *p;
                        d++;
                        p++;
                }
        }

        *d = 0;

        return p + 1;
}
```

然后，我们需要实现DNS报文的构造。该代码较长，因此我们将其放在了附录中，见附录：代码清单 中的代码 5.1。接着，在LWIP UDP接收数据的回调函数中，我们实现了DNS回报的解析。这些部分大体

就是对协议内容的简单实现。

proto_query的查询既要支持普通的DNS查询，也要支持从IP地址到域名的反向查询。但该功能不需要实现额外的功能，RFC 1035中定义了一种特殊的域，即in-addr.arpa，该域中的域名用于IP到域名的反向查询。因此，我们只需要将IP地址转化为in-addr.arpa域中的域名，然后进行正常的DNS查询，并取结果中的PTR记录即可。RFC中的相关描述如下：

> The Internet uses a special domain to support gateway location and Internet address to host mapping. Other classes may employ a similar strategy in other domains. The intent of this domain is to provide a guaranteed method to perform host address to host name mapping, and to facilitate queries to locate all gateways on a particular network in the Internet.

由于本函数实现过长，故不在报告中完整呈现，请参见源代码中kernel/net/baredns.c 中的相关内容。

基本DNS功能的测试依赖于用户应用程序nslookup的实现，具体数据和内容见应用程序章节的 nslookup小节。该API连通此命令一同测试，能够正常工作。

六、内核改进

1、内存分配改进

原始的xv6内核中，内存分配只有kalloc和kfree两个函数，分别用于分配和释放内核内存。该函数的作用是按页分配内存，每次分配的内存大小为一页，即4KB。这种分配方式存在两个问题：

1. 由于每次分配的内存大小为一页，因此分配的内存大小不可控，如果需要分配的内存大小不是一页的整数倍，那么就会造成内存浪费。
2. 当分配超过一页时，难以保证分配的内存是连续的。

然而，在网络驱动程序、网络协议栈、Socket API等部分，需要频繁地进行内存分配，而且分配的内存大小不固定、分配的大小常常远小于 4KB。为了避免耗尽内存，提高内存分配的效率。需要改进内存分配的方式。

常见的类UNIX操作系统，例如Linux有更加复杂的内存分配机制。例如Buddy/Slab分配器，可以分配任意大小的内存块。其中Buddy分配器分配2幂次大小的内存块，而Slab分配任意指定大小的内存块。这种分配方式的优点是可以分配任意大小的内存块，然而完整的实现相当复杂，不利于我们按期完成项目。因此，我们采用了一种简单的内存分配方式，即利用xv6现有的分配器分配页大小的内存。然后使用Slab分配器分配任意大小的内存块。具体的实现在kernel/slab.c中。

Slab分配器的原理是将内存分为多个大小相同的内存块，每个内存块称为一个Slab。Slab分配器维护一个Slab链表，每个Slab中都有一个空闲链表，用于记录空闲的内存块。当需要分配内存时，Slab分配器从空闲链表中取出一个内存块，当需要释放内存时，Slab分配器将内存块插入空闲链表。当Slab中的内存块全部被分配时，Slab分配器从Slab链表中取出一个Slab，当Slab中的内存块全部被释放时，Slab分配器将Slab插入Slab链表。它们操作的内存空间均由xv6的分配器分配。

我们的实现接口大致于Linux相应组件的接口相同。kernel/slab.h中定义了Slab的存储结构，如代码 3.10所示。

代码 3.10: Slab的存储结构

```
1 typedef struct slab
2 {
3         kmem_cache_t *cache;
4         size_t in_use;
```

```
5         size_t next_free;
6         void *objects;
7         kmem_bufctl *bufctl;
8
9         spinlock_t lock;
10
11        list_head_t link;
12  } slab_t;
```

而这些Slab由称之为Cache的结构管理。Cache的存储结构如代码 3.11所示，在kernel/include/slab.h中。

代码 3.11: Cache的存储结构

```
1   typedef struct kmem_cache
2   {
3         list_head_t full, partial, free;
4
5         list_head_t link;
6
7         size_t obj_size, obj_count;
8         uint flags;
9         spinlock_t lock;
10
11        char name[KMEM_CACHE_NAME_MAXLEN];
12  } kmem_cache_t;
```

有了这些结构，我们就可以实现kmem_cache_create、kmem_cache_alloc、kmem_cache_free等接口。

在这些接口之上，我们也实现了kmalloc，来方便不创建对应Cache，只指定大小的内存分配。这些接口在驱动程序、Socket API和DNS相关的实现中发挥了关键的作用。

2、文件系统改进

xv6没有常见的VFS结构，但是其文件系统实现包含了对不同类型的文件节点的初步支持。由于Socket API在分配、关闭Socket时需要创建相应类型的文件节点来返回File Descriptor，因此我们在xv6的文件系统实现上做了一些扩展。

kernel/fs.h中定义了文件节点的存储结构，如代码 3.12所示。

代码 3.12: 文件节点的存储结构

```
1
2   typedef enum file_type
3   {
4         FD_NONE = 1,
5         FD_PIPE,
6         FD_DEVICE,
7         FD_SOCKET,
8   } file_type_t;
9
10  typedef struct file
11  {
12        file_type_t type;
13        int ref; // reference count
14        char readable;
15        char writable;
```

```
16
17          union
18          {
19                  struct pipe *pipe;
20                  struct inode *ip;
21          };
22          uint off;
23  }file_t;
```

我们在类型中加入了FD_SOCKET，并在union中加入了socket成员。这样就存储了Socket的相关信息。

负责处理文件节点的函数包括file_alloc、file_close、file_dup等。我们在相应函数中加入了对Socket的处理，调用Socket相关的具体实现，这里举一例说明，如代码 3.13所示。

代码 3.13: file_alloc函数的修改

```
1   void
2   fileclose(struct file *f)
3   {
4           struct file ff;
5
6           acquire(&ftable.lock);
7           if (f->ref < 1)
8           {
9                   panic("fileclose");
10          }
11          if (--f->ref > 0)
12          {
13                  release(&ftable.lock);
14                  return;
15          }
16          ff = *f;
17          f->ref = 0;
18          f->type = FD_NONE;
19          release(&ftable.lock);
20
21          if (ff.type == FD_PIPE)
22          {
23                  pipeclose(ff.pipe, ff.writable);
24          }
25          else if (ff.type == FD_INODE)
26          {
27                  begin_op();
28                  iput(ff.ip);
29                  end_op();
30          }
31          else if (ff.type == FD_SOCKET)
32          {
33                  socketclose(ff.socket);
34          }
35  }
```

在这段代码中，遇到Socket类型的文件节点时，调用了socketclose函数实现Socket专属的关闭逻辑。

3、常用数据结构实现

设备驱动程序、DNS的查询过程中，我们需要一些能够动态地插入和删除元素的数据结构。因此，我们实现了Linux类似的侵入式链表。

侵入式（Intrusive）链表是一种链表的实现方式，它将链表的节点单独定义为一个结构体，并将链表节点的指针作为结构体的成员，这是链表中相应的指针都处于数据的内部，所以称之为侵入式链表。

这种链表实现的好处在于可以用同一套链表操作函数来操作不同的具体数据。当需要从链表节点获取数据时，可以通过结构体中内存偏移的计算来获取数据的指针，从而实现链表节点和数据的互相获取。这种计算的方法在Linux内核中被称为container_of。代码 3.14给出了这种计算的实现。

代码 3.14: container_of宏的实现

```
1 // offset of a struct member
2 #define offsetof(type, member)  __builtin_offsetof (type, member)
3
4 // get struct pointer from member
5 #define container_of(ptr, type, member) ({                      \
6         const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
7         (type *)( (char *)__mptr - offsetof(type,member) );})
```

而其余的链表操作函数实现定义在include/list.h中。包括插入、删除、遍历等操作。这些操作函数的实现都是基于list_head结构体的。

七、用户态库

用户态库是用户态和系统调用实现间的“胶水”。它对用户提供了POSIX兼容的编程接口。将相关参数按照系统调用的约定存入内存，然后调用INT 0x80中断，将控制权交给内核。用户态库的实现在lib/ulib目录下。

由于在32位环境下，传参数不会用到寄存器，因此已经正确地将参数存入内存中，只需要使用int指令调用相应终端即可，这部分代码实现在lib/ulib/usys.S中。如代码 3.15所示。

代码 3.15: usys.S中的系统调用实现

```
1 #define SYSCALL(name) \
2   .globl name; \
3   name: \
4     movl $SYS_ ## name, %eax; \
5     int $T_SYSCALL; \
6     ret
```

这定义了一个宏，用于定义系统调用的实现。宏中的SYS_name是系统调用的编号，T_SYSCALL是中断号。

这样就可以简单地用系统调用的名称迅速定义大量的系统调用库函数接口了。

八、应用程序

1、nslookup

程序介绍

nslookup是一个用来查询因特网域名服务器的程序。程序允许使用者向域名服务器查询一个域名或地址的信息，通常被用来诊断域名解析系统的基本结构信息，查询域名解析是否正常工作，在网络故障时排查网络问题。

功能分析

一个完整的nslookup程序拥有两种使用模式：交互式与　交互式。交互式的nslookup允许用户交互式查询多个主机/域名信息，并且具有多种交互命令用于处理复杂的需求、提升用户体验。而　交互式的nslookup只允许用户一次性查询单个主机/域名信息。　交互式的nslookup也　持通过命令　指定参数。本项　的　标是实现一个简易的nslookup程序，因此此程序只实现了nslookup的核　功能：　交互式地对域名或地址进　查询，并向用户显示查询结果。

程序实现

在本项目开始时，我尝试通过如下方法实现程序的查询功能：

1. 通过socket编程与域名服务器通信
2. 构造DNS QUERY报文，向域名服务器发起域名查询请求
3. 解析DNS RESPONSE报文，解析由域名服务器返回的主机信息

但是该方法　对的最大的障碍是，由于DNS复杂的压缩规则，通过编程解析DNS REPLY报文实现难度较　。

因此我最后通过调用内核中已经实现的函数：gethostbyname与gethostbyaddr，获得主机信息。内核中，netdb.h中实现了一组用来获取主机信息的函数，其中包括gethostbyname与 gethostbyaddr。这组函数会返回一个保存有主机信息的hostent结构体。其定义如下代码 3.16所示。

代码 3.16: hostent结构体定义

```
1        struct  hostent {
2             char    *h_name;        /* official name of host */
3             char    **h_aliases;    /* alias list */
4             int     h_addrtype;     /* host address type */
5             int     h_length;       /* length of address */
6             char    **h_addr_list;  /* list of addresses from name
7 server */
8     };
```

程序具体实现如下：

1. 首先通过命令行接受用户输入参数，并对用户输入作初步的输入检查
2. 对于用户的输入，使用inet_pton判断输入类型，对地址和域名分别处理
3. 对于输入的域名，通过调用gethostbyname获取描述该主机的hostent结构体
4. 对于输入的地址，通过调用gethostbyaddr获取描述该主机的hostent结构体
5. 对于获取的hostent结构体，使用用户友好的形式输出显示。

2、ping和traceroute

ping的实现

利用原始套接字发送ICMP报文，将ICMP报文头的消息类型设置为8（ICMP request）。ICMP_HDR报文头如代码 3.17所示。

代码 3.17: ICMP_HDR结构体定义

```
struct icmp_hdr {
        unsigned char type;             /* message type */
        unsigned char code;             /* type sub-code */
        unsigned short checksum;
        unsigned short id;
        unsigned short sequence;
        unsigned int timestamp;         /* not part of ICMP, but we need it */
};
```

然后，通过原始套接字实现的ICMP报文的发送，如代码 3.18所示。

代码 3.18: ICMP报文发送函数

```
struct sockaddr_in dest;
dest.sin_family = AF_INET;
dest.sin_port = htons(0);
dest.sin_addr.s_addr = ip;
char buff[sizeof(ICMP_HDR) + 32];
ICMP_HDR *pIcmp = (ICMP_HDR *) buff;
pIcmp->icmp_type = 8;
pIcmp->icmp_code = 0;
pIcmp->icmp_id = (uint16_t) pID;
pIcmp->icmp_checksum = 0;
pIcmp->icmp_sequence = 0;
memset(&buff[sizeof(ICMP_HDR)], 'E', 32);
```

然后解析接收到的ICMP报文，判断是否为ICMP reply，如果是，则计算往返时间，否则丢弃并报错。如代码 3.19所示。

代码 3.19: ICMP报文接收函数

```
ICMP_HDR *pRecvIcmp = (ICMP_HDR *) (recvBuf + 20); // (ICMP_HDR*)(recvBuf + sizeof(IPHeader));
IPHeader *ipHeader = (IPHeader *) recvBuf;
```

其中IPHeader结构体定义如代码 3.20所示。

代码 3.20: IPHeader结构体定义

```
        typedef struct IPHeader {
    uint8_t iphVerLen;
    uint8_t ipTOS;
    uint16_t ipLength;
    uint16_t ipID;
    uint16_t ipFlags;
    uint8_t ipTTL;
    uint8_t ipProtocol;
    uint16_t ipChecksum;
    uint32_t ipSource;
    uint32_t ipDestination;
} IPHeader;
```

由此可以显示相应的信息，如代码 3.21所示。

代码 3.21: ping显示信息

```
printf(" %d bytes from %s:", (int) nRet, inet_ntoa(from.sin_addr));
```

```
2 printf(" icmp_seq = %d. ", pRecvIcmp->icmp_sequence);
3 printf(" ttl = %d. ", ipHeader->ipTTL);
4 printf(" time: %d ms", (int) nTick - (int) pRecvIcmp->icmp_timestamp);
```

traceroute的实现

与ping类似，traceroute从1开始设置ttl，当报文每经过一个路由节点时，ttl减小1，当ttl为0时，节点丢弃报文，并向主机发送一个超时报文，报文的ip数据报中就有该节点的ip地址；随后增大ttl，获取下一个节点，就可以得知主机和目的地址之间的所有路由节点的ip地址

但是，因为部分节点不会响应icmp报文，发送超时报文，所以要为套接字设置超时。其中设置超时的代码如代码 3.22所示。

代码 3.22: 设置超时

```
1 struct timeval timeout = {5,0};
2 setsockopt(sRaw,SOL_SOCKET,SO_RCVTIMEO,(char *)&timeout,sizeof(struct timeval));
```

而设置ttl的代码如代码 3.23所示。

代码 3.23: 设置TTL

```
1         setsockopt(sRaw, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl))
```

3、httpget

4、RFC 867授时服务客户端daytime

Daytime Protocol是1983年由RFC 867定义的，用于授时服务，客户端通过TCP连接或者到服务器，服务器使用的端口号是13，服务器返回一个ASCII字符串，表示当前的时间，格式为：$DayMonddhh:mm:ssyyyy$

为了测试与远程服务器的连接，我们可以使用这个协议，客户端的代码如代码 3.24所示。该实现使用Socket编程，使用了TCP连接，连接到服务器后，发送一个空的数据包，服务器会返回一个32位的整数，表示从1900年1月1日到现在的秒数，客户端可以通过这个时间戳来输出一个表示时间的用户可读的字符串。

代码 3.24: daytime客户端

```
1  #define SERVER_PORT 13
2  #define SERVER_HOSTIP 0x2c8c8a80
3  char buf[512];
4
5  int main(int argc, char **argv)
6  {
7  //      struct hostent *hp;
8          int sockfd, r;
9          struct sockaddr_in addr = {
10                 .sin_family = PF_INET, .sin_port =  hton16(SERVER_PORT),
11         };
12
13         addr.sin_addr.s_addr = hton32(SERVER_HOSTIP);
14
15         sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
16
```

```
17          r = connect(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
18          if (r < 0)
19          {
20                  printf(1, "daytime: connect failed: %d\n", r);
21                  exit(-1);
22          }
23
24          ssize_t n;
25
26          n = recv(sockfd, buf, sizeof(buf));
27          if (n <= 0)
28          {
29                  goto end;
30          }
31
32          write(1, buf, n); // stdout
33
34  end:
35          close(sockfd);
36          return 0;
37  }
```

5、TCP和UDP的Echo Server

在实现Socket API的过程中，我们需要简单易行的方式来测试我们的实现是否正确，这里我们使用一个简单的Echo Server来测试我们的实现。

我们实现的TCP和UDP的Echo Server中使用TCP的代码如代码 3.25所示。是一个单进程的简易Echo Server，在用于测试我们的Socket API的实现是否正确的用途中发挥了重要的作用。

代码 3.25: TCP Echo Server

```
1   #include "types.h"
2   #include "user.h"
3   #include "socket.h"
4   #include "inet.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9           int soc, acc, peerlen, ret;
10          struct sockaddr_in self, peer;
11          unsigned char *addr;
12          char buf[2048];
13
14          printf(1, "Starting TCP Echo Server\n");
15          soc = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
16          if (soc == 1)
17          {
18                  printf(1, "socket: failure\n");
19                  exit(soc);
20          }
21          printf(1, "socket: success, soc=%d\n", soc);
22          self.sin_family = AF_INET;
23          self.sin_addr.s_addr = INADDR_ANY;
24          self.sin_port = hton16(7);
```

```
25          if (bind(soc, (struct sockaddr *)&self, sizeof(self)) == -1)
26          {
27                  printf(1, "bind: failure\n");
28                  close(soc);
29                  exit(soc);
30          }
31          addr = (unsigned char *)&self.sin_addr;
32          printf(1, "bind: success, self=%d.%d.%d.%d:%d\n", addr[3], addr[2], addr[1], addr[0], ntoh16
                (self.sin_port));
33          listen(soc, 100);
34          printf(1, "waiting for connection...\n");
35          peerlen = sizeof(peer);
36          acc = accept(soc, (struct sockaddr *)&peer, &peerlen);
37          if (acc == -1)
38          {
39                  printf(1, "accept: failure\n");
40                  close(soc);
41                  exit(soc);
42          }
43          addr = (unsigned char *)&peer.sin_addr;
44          printf(1, "accept: success, peer=%d.%d.%d.%d:%d\n", addr[3], addr[2], addr[1], addr[0],
                ntoh16(peer.sin_port));
45          while (1)
46          {
47                  memset(buf, 0, sizeof(buf));
48                  ret = recv(acc, buf, sizeof(buf));
49                  if (ret <= 0)
50                  {
51                          printf(1, "EOF\n");
52                          break;
53                  }
54                  printf(1, "recv: %d bytes data received\n", ret);
55                  hexdump(buf, ret);
56
57                  if (send(acc, buf, ret) != ret)
58                  {
59                          printf(1, "send: failure\n");
60                          break;
61                  }
62          }
63          close(soc);
64          return 0;
65  }
```

第四章　项目管理

第五章　项目感想

一、熊恪峥

在这次计算机网络课程项目中，我负责了项目的设计、分工、全部内核态部分的实现和少数用户态程序的实现。在这个过程中，我一方面增强了对xv6内核代码的理解，另一方面也学习了现有成熟的网络API是怎么一步一步从0开始构建起来。从驱动程序如何将数据从网卡中读出、从用户给定的位置送给网卡，到内核如何处理不同的网络协议，我自己编码一步步将它们实现出来，这是一次非常有意义的经历。也让我对我们习以为常的“开箱即用” 的API接口的底层做什么、怎么做的问题有了更深的理解。

此外，由于我们项目中的代码量级极大，为了有效地管理，我们使用了git进行版本控制，为了处理协议、Socket数据报类型等复杂的组合、灵活地复用代码，我不仅调查和研究了现有操作系统内核相应的处理方式，也加入了自己在实践中积累经验得出的设计，这也让我有很大的收获。

二、赵淇奥

在这次计算机网络课程设计项目中，我负责编写了nslookup程序。这个程序的功能是向dns服务器查询给定的地址或域名。在之前计网课程的实验中我多次接触过linux内置的该命令，而通过本次项目，亲自实现该命令，让我更加了解计算机网络编程、熟悉计算机网络内核协议栈，并且对DNS的工作原理有了更深刻的认识。

三、肖凯欣

在项目中我负责编写ping程序和traceroute程序。在计算机网络的理论课上，这两个程序都作为网络层协议应用的示例。在程序实现中需要我根据程序需要生成、发送相应的ICMP报文，并对收到的ICMP报文进行解析和处理。在过程中我对于计算机网络编程，网络层协议以及ICMP协议的工作原理有了更加深刻的认识。

四、谢哲涵

在项目中我负责编写httpget程序。在计算机网络的实验课中，我们对使用HTTP连接获取互联网上的内容有了初步的理解和认识，而在实现这个命令过程中我对于计算机网络编程，应用层协议有了更加深刻的认识。

参考文献

[1] Frans Kaashoek. 6.828 operating system engineering, fall 2006. 2006.

[2] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011.

[3] Adam Dunkels. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science*, 2(77), 2001.

附录：代码清单

代码 5.1: DNS报文构造

```
static inline struct netdb_answer *make_query(char *name, int type)
{
        assert_holding(&netdb_lock);

        char *p = netdb_qbuf + 2048;

        dns_header_t *header = (dns_header_t *)p;
        header->xid = byteswap16(0x1204); // should be random though
        header->flags = byteswap16(0x0100); // Q, RD
        header->qdcount = byteswap16(1); // 1 question

        p += sizeof(dns_header_t);
        dns_question_t *question = (dns_question_t *)p;
        question->dnstype = byteswap16(type); // A or PTR
        question->dnsclass = byteswap16(1); // IN

        p += sizeof(dns_question_t);
        question->name = p;

        const uint8 hostlen = (uint8)strlen(name);
        memmove(question->name + 1, name, hostlen + 1);

        uint8 *prev = (uint8 *)question->name;
        uint8 count = 0;

        for (uint8 i = 0; i < hostlen; i++)
        {
                if (name[i] == '.')
                {
                        *prev = count;
                        prev = (uint8 *)(question->name + i + 1);
                        count = 0;
                }
                else
                {
                        count++;
                }
        }
        *prev = count;

        p = netdb_qbuf;
        size_t packet_len = sizeof(dns_header_t) + sizeof(dns_question_t) - sizeof(char *) + hostlen
             + 2;

        memmove(p, header, sizeof(dns_header_t));
        p += sizeof(dns_header_t);

        memmove(p, question->name, hostlen + 1);
        p += hostlen + 2;

        memmove(p, &question->dnstype, sizeof(question->dnstype));
        p += sizeof(question->dnstype);
```

```
52          memmove(p, &question->dnsclass, sizeof(question->dnsclass));
53
54          netbegin_op();
55          struct pbuf *pbuf = pbuf_alloc(PBUF_TRANSPORT, packet_len, PBUF_RAM);
56          if (!pbuf)
57          {
58                  netend_op();
59                  return NULL;
60          }
61          memmove(pbuf->payload, netdb_qbuf, packet_len);
62          query_ans = NULL;
63          udp_recv(dns_pcb, netdb_dns_recv, NULL);
64          err_t err = udp_send(dns_pcb, pbuf);
65
66          if (err != ERR_OK)
67          {
68                  netend_op();
69                  pbuf_free(pbuf);
70                  return NULL;
71          }
72          netend_op();
73
74          if (!query_ans)
75          {
76                  if (sleepddl(&dns_pcb, &netdb_lock, DNS_TIMEOUT) == 0)
77                  {
78                          return NULL;
79                  }
80          }
81          return query_ans;
82  }
```