Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Расчетно-пояснительная записка

к курсовой работе

Тема: Реализация драйвера виртуального сетевого интерфейса

Дисциплина: Операционные системы

Студент ИУ7-75Б _____ П.К. Хетагуров

(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель проекта _____ Н.Ю. Рязанова

(Подпись, дата) (И.О. Фамилия)

Москва, 2021

Содержание

Введение

Современные вычислительные системы сложно представить без поддержки интернет-сетей. На одном устройстве может быть доступ к различным подсетям по различным сетевым интерфесам и каналам передачи данных. Доступ к различным сетевым интерфейсам возможен на уровне пользователя, однако это требует от пользовательского ПО введения дополнительных критериев выбора сетевых интерфейсов. В данной работе рассматриваются вопросы разработки собственного виртуального интерфейса, который позволял бы пользователю абстрагироваться от выбора конкретного сетевого интерфейса. Совмещение нескольких сетевых интерфейсов в один облегчает доступ к нескольким подсетям со стороны некоторых прикладных приложений.

Так как сетевой интерфейс может быть определен в системе только с помощью загружаемого модуля ядра, то данная работа будет основана на анализе и разработке такого загружаемого модуля.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать и реализовать драйвер виртуального сетевого интерфейса, реализующий создание виртуального сетевого интерфейса для распределения пакетов по уже существующим сетевым интерфейсам.

Для достижения цели курсовой работы необходимо решить следующие задачи:

- проанализировать работу сетевой подсистемы Linux;
- проанализировать функции и структуры ядра, позволяющие реализовать сетевой интерфейс;
- разработать необходимые модули;
- реализовать загружаемый модуль ядра;
- протестировать реализованное ПО.

Разрабатываемое ПО должно перенаправлять пакеты, пришедшие на виртуальный интерфейс по указанным интерфейсам в зависимости от IP назначения пакета.

1.2 Сетевые интерфейсы и устройства

Для доступа к сетевым устройствам используются так называемые сетевые интерфейсы. Они являются основой сетевой подсистемы Linux. Все сетевое взаимодействие в Linux происходит через сетевые интерфейсы. Любые данные, которые компьютер отправляет в сеть или получает из сети проходят через сетевой интерфейс.

Сетевые устройства выделяют сетевые устройства как специфический тип [2][3]. На практике сетевые устройства являются символьными.

Однако интерфейсы это не файлы устройств и их нет в каталоге /dev. Интерфейсы создаются динамически и не всегда связаны с сетевыми картами. Например интерфейс ppp0 - это интерфейс VPNa, организованного по протоколу PPTP, а интерфейс lo это виртуальная сетевая карта с адресом localhost (127.0.0.1). Такие интерфейсы называются виртуальными.

Таким образом сетевые интерфейсы скрывают детали реализации конкретного сетевого устройства, прикладное программное обеспечение, обращаясь к сетевому интерфейсу не учитывает детали реализации конкретных сетевых устройств. Однако в Linux существует общепринятая схема именования сетевых интерфейсов, состоящая из префикса типа сетевого устройства и заканчивающаяся номером иакого устройства. Примеры наименования интерфейсов:

- eth0 — первый сетевой интерфейс к карте Ethernet или картам WaveLan (Radio Ethernet);
- wlan0 — сетевой интерфейс wi-fi адаптера;
- lo — сетевой интерфейс к виртуальной сетевой карте с адресом localhost (127.0.0.1);
- enp1s3 — четвертый сетевой интерфейс второй группы к карте Ethernet или картам WaveLan (Radio Ethernet).

Интерфейсы создаются автоматически для каждого обнаруженного сетевого устройства при загрузке ядра ОС.

Каждый интерфейс характеризуется определёнными параметрами, необходимыми для обеспечения его нормального функционирования, и в частности для сетевого обмена данными с помощью стека TCP/IP. Некоторые параметры интерфейса:

1. IP-адрес;

2. маска подсети;

3. аппаратный адрес сетевого устройства, соответствующего интерфейсу.

И сетевой интерфейс и драйвер сетевого устройства описываются большой структурой ядра ‘net_device‘, о которой сами разработчики, из-за смешения в ней разных уровней абстракции, отзываются как о "большой ошибке в коде ядра есть комментарий: "Actually, this whole structure is a big mistake".

1.3 struct net_device

Основной структурой, которую использует сетевая подсистема Linux является struct net_device (определена в <linux/netdevice.h> [6]). Сама структура является слишком большой для полного приведения, поэтому рассмотрим только некоторые поля. На листинге далее приведена часть структуры.

Листинг 1 – net_device

```
1  struct net_device {
2    char name[IFNAMSIZ];
3    struct netdev_name_node *name_node;
4    char * ifalias;
5    unsigned long mem_end;
6    unsigned long mem_start;
7    unsigned long base_addr;
8
9    unsigned long   state;
10
11   struct list_head  dev_list;
12   struct list_head  napi_list;
13   struct list_head  unreg_list;
14   struct list_head  close_list;
```

```
15      struct list_head  ptype_all;
16      struct list_head  ptype_specific;
17      unsigned int      flags;
18      struct net_device *next;
19      \* Продолжение структуры *\
```

Рассмотрим некоторые поля.

- char name[IFNAMSIZ] — имя устройста;
- unsigned long unsigned long mem_end , unsigned long mem_start — информация о памяти устройства. Данные поля содержат начало и конец разделяемой памяти устройства. По соглашению поля end устанавливаются, поэтому end - start = общему количеству доступной памяти на устройстве;
- unsigned long base_addr — базовый адрес ввода-вывода сетевого интерфейса. Это поле, как и предыдущие, назначаются во время обнаружения устройства. Команда ifconfig может быть использована для отображения и модификации текущего значения;
- unsigned char irq — назначеный номер прерывания;
- unsigned char if_port — показывает, какой порт используется в устройствах с несколькими портами, например устройства с поддержкой как коаксиального (IF_PORT_10BASE2) Ethernet соединения, так и Ethernet соединения с помощью витой пары (IF_PORT_10BASET). Полный список известных типов портов определен в <linux/netdevice.h>;
- unsigned long state — состояние устройства. Это поле включает несколько флагов. Драйвер обычно не использует эти флаги напрямую, но с помощью специальных функций;

- void *priv — указатель, зарезервированный для пользовательских данных;
- struct net_device *next — указатель на следующее сетевое устройство в глобальном связанном списке сетевых устройств.

Большую часть информации, связанной с сетевыми интерфесами в структуре net_device инициализируют существующие функции установки, определенные в <drivers/net/net_init.c> [3]. Прототипы таких функций:

1. void ether_setup(struct net_device *dev) — инициализирует поля для устройств Ethernet;
2. void ltalk_setup(struct net_device *dev) — инициализирует поля для устройств LocalTalk;
3. void fc_setup(struct net_device *dev) — инициализирует поля для волоконно-оптических устройств;
4. void fddi_setup(struct net_device *dev) — конфигурирует интерфейс для сети с Fiber Distributed Data Interface (распределенным интерфейсом передачи данных по волоконно-оптическим каналам, FDDI).
5. void hippi_setup(struct net_device *dev) — инициализирует поля для High-Performance Parallel Interface (высокопроизводительного параллельного интерфейса, HIPPI);
6. void tr_setup(struct net_device *dev) — выполняет настройку для сетевых интерфейсов token ring (маркерное кольцо).

Большинство устройств подходит для инициализации одной из этих функций. Если требуется что-то уникальное, то необходимо определить следующие поля:

1. unsigned short hard_header_len — длина аппаратного заголовка;

2. unsigned mtu - MTU (Max transfer unit);

3. unsigned long tx_queue_len — максимальная длина очереди на отправку;

4. unsigned short type — аппаратный тип интерфейса;

5. unsigned char addr_len — длина аппаратного адреса;

6. unsigned char dev_addr[MAX_ADDR_LEN] — аппаратный адрес устройства (MAC).

7. unsigned short flags — флаги интерфейса;

8. int features — специальные аппаратные возможности.

Для инициализации реализуемого сетевого интерфейса была выбрана функция инициализации void ether_setup(struct net_device *dev), как самая универсальная.

1.3.1 Функции для работы с сетевым интерфейсом. Struct net_device_ops

Функции, с помощью которых система взаимодействует с устройством определены в структуре net_device_ops, определенной в <linux/netdevice.c> [6]. Часть структуры приведена ниже:

Листинг 2 – net_device_ops

```
struct net_device_ops {
        int (*ndo_init)(struct net_device *dev);
        void (*ndo_uninit)(struct net_device *dev);
        int (*ndo_open)(struct net_device *dev);
        int (*ndo_stop)(struct net_device *dev);
        netdev_tx_t (*ndo_start_xmit) (struct sk_buff *skb, struct
    net_device *dev);
```

```
7         void (*ndo_change_rx_flags)(struct net_device *dev, int
    flags);
8         void (*ndo_set_rx_mode)(struct net_device *dev);
9         void (*ndo_set_multicast_list)(struct net_device *dev);
10        int (*ndo_set_mac_address)(struct net_device *dev, void *
    addr);
11        int (*ndo_validate_addr)(struct net_device *dev);
12        int (*ndo_set_config)(struct net_device *dev, struct ifmap
     *map);
13        int (*ndo_change_mtu)(struct net_device *dev, int new_mtu)
    ;
14        void (*ndo_tx_timeout) (struct net_device *dev);
15        struct net_device_stats* (*ndo_get_stats)(struct
    net_device *dev);
16        /* Several lines omitted */
17 };
```

Поставленную задачу можно решить с помощью следующих функций:

1. ndo_open — вызывается при открытии интерфейса;
2. ndo_close — вызывается при закрытии интерфейса;
3. ndo_start_xmit — вызывается при передачи пакета через интерфейс.

1.4 Виртуальные интерфейсы tun/tap

TUN и TAP — виртуальные сетевые драйверы ядра системы. Они представляют собой программные сетевые устройства, которые отличаются от обычных аппаратных сетевых карт.

TAP эмулирует Ethernet устройство и работает на канальном уровне модели OSI, оперируя кадрами Ethernet. TUN (сетевой туннель) работает на сетевом уровне модели OSI, оперируя IP пакетами. TAP используется для создания сетевого моста, тогда как TUN для маршрутизации [5].

Пакет, посылаемый операционной системой через TUN/TAP устройство обрабатывается программой, которая контролирует это устройство. Получение данных происходит через специальный файловый дескриптор, таким образом программа просто считывает данные с файлового дескриптора. Сама программа также может отправлять пакеты через TUN/TAP устройство выполняя запись в тот же файловый дескриптор. В таком случае TUN/TAP устройство доставляет (или «внедряет») такой пакет в сетевой стек операционной системы, эмулируя тем самым доставку пакета с внешнего устройства.

Не смотря на внешнюю схожесть TAP интерфейса с планируемым решением, нельзя просто строить решение на его основе из-за различной внутренней логики виртуальных интерфейсов.

1.5 Обработка пакетов на уровне сетевого интерфейса

В сетевых интерфейсах существует очередь обрабатываемых пакетов. На программном уровне следующий пакет из буфера пакетов, требующих обработки представлен структурой sk_buff.

На рисунке 1 проиллюстрирован процесс получения интернет-пакетов. Сетевые интерфейсы работают с пакетами 2-го и 3-го уровня модели OSI. Для применения к поставленной задачи необходимо как отправлять на несколько интерфейсов, так и считывать пакеты, отправленные в ответ. Для этого, как видно из схемы, необходимо вклинится в процесс обработки пакета. Linux позволяет добавить обработчик входящих пакетов с помощью функции netdev_rx_handler_register.

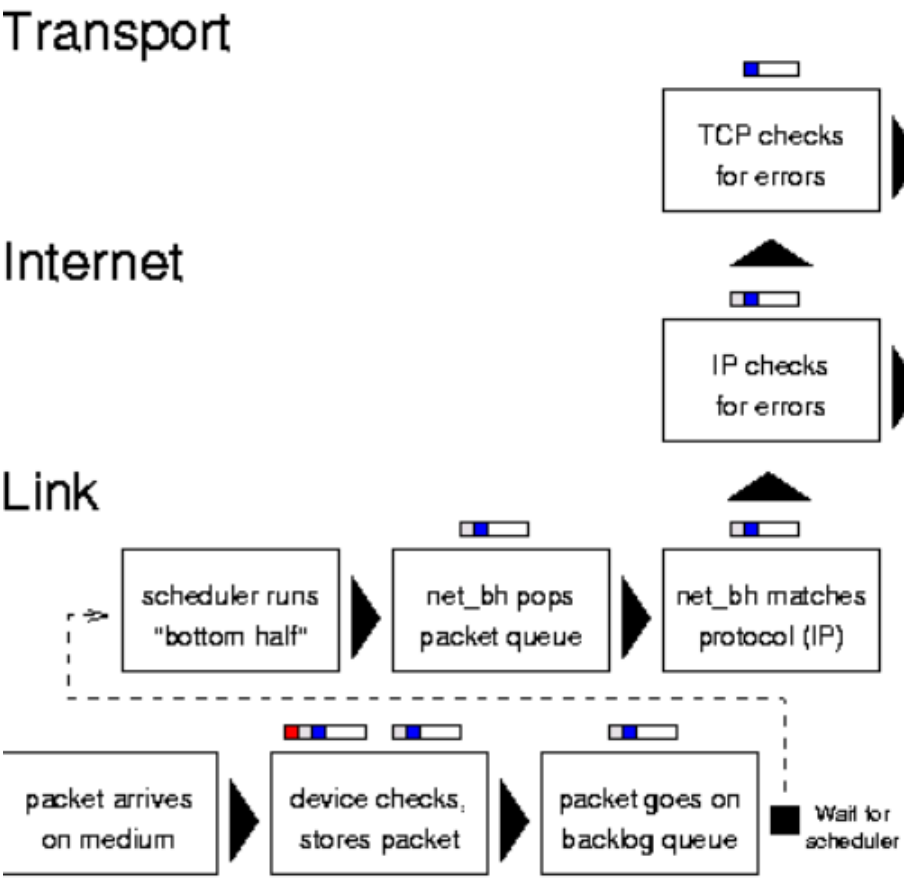

Рисунок 1 – Обработка входящего пакета

Исходящий же пакет можно перенаправить на обработку в другой интерфейс с помощью подмены поля dev в структуре sk_buff.

1.6 Вывод

В результате проведенного анализа было определено:

- основной структурой сетевой подсистемы Linux является struct net_device;
- функции для работы со struct net_device определены в struct net_device_ops, на которую есть ссылка в struct net_device;
- была выбрана функция инициализации void ether_setup(struct net_device *dev);

- для реализации задачи были выбраны три функции работы с сетевыми интерфейсами (ndo_open, ndo_close, ndo_start_xmit).

Были проанализированы особенности работы сетевой подсистемы Linux. Проанализирована структура struct net_device, позволяющая создавать виртуальное сетевое устройство. Была проанализирована структура struct net_device_ops, хранящая функции работы с struct net_device.

2 Конструкторский раздел

2.1 IDEF0

В системе можно выделить два основных модуля. Модуль получения и модуль отправки пакета. Далее приведены схемы IDEF0 выделенных модулей.

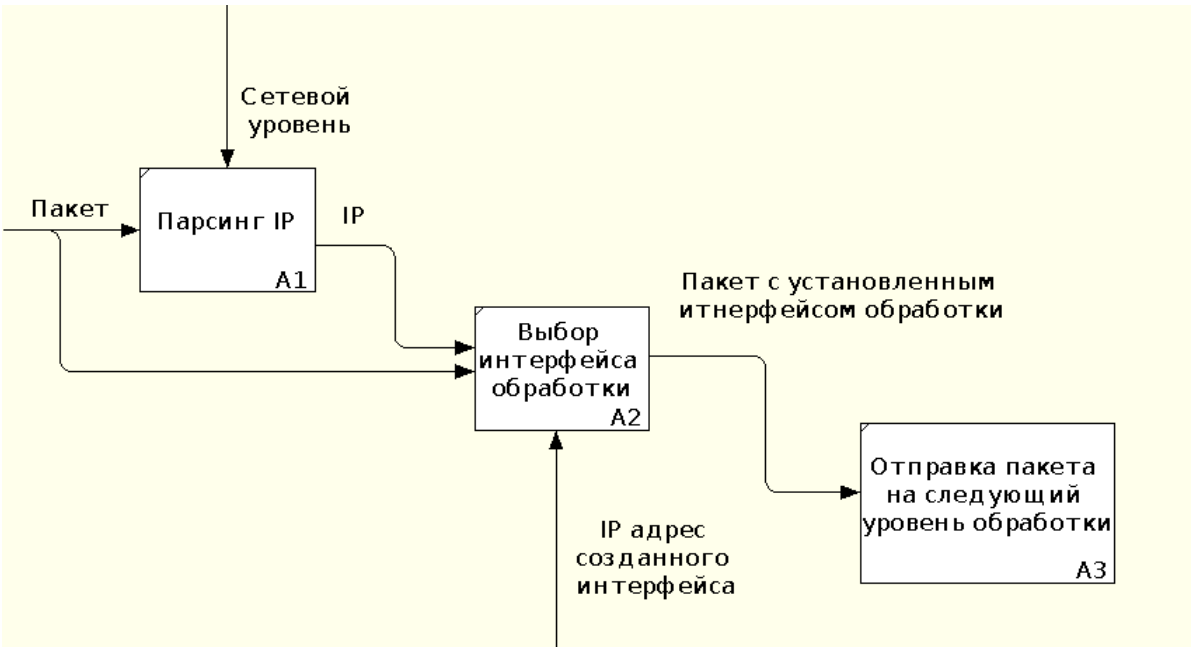

Рисунок 2 – IDEF0 получения пакета

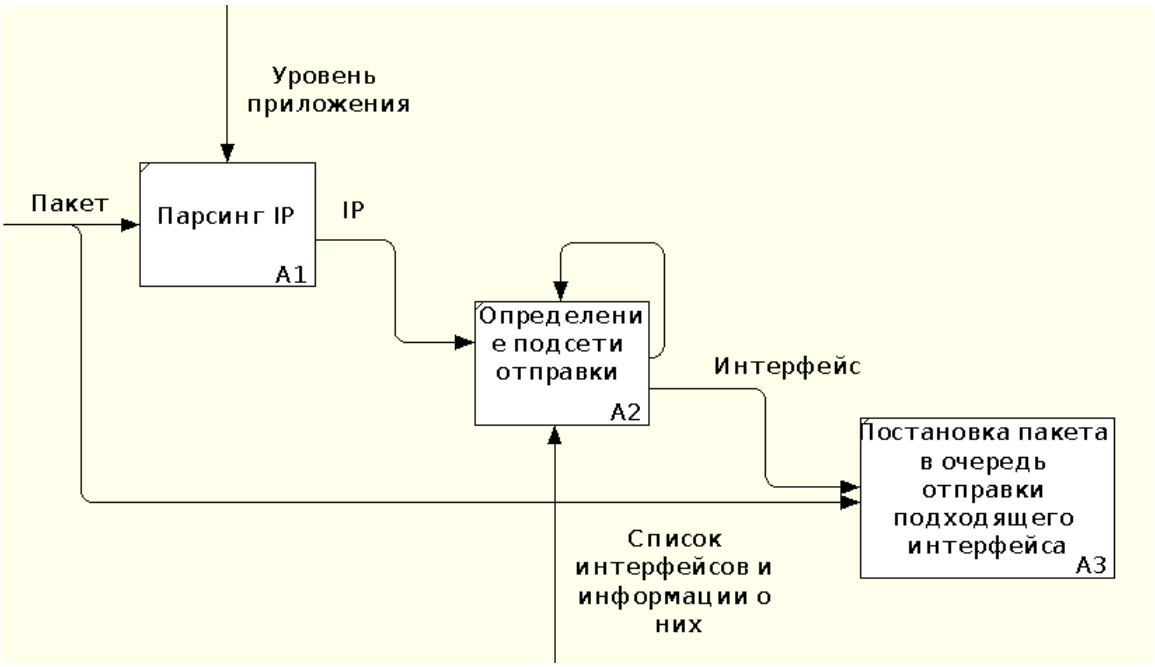

Рисунок 3 – IDEF0 отправки пакета

2.2 Алгоритм получения пакета

Схема алгоритма обработки принимаемого пакета на одном из интерфейсов, скрываемых виртуальным представлена на рисунке 4.

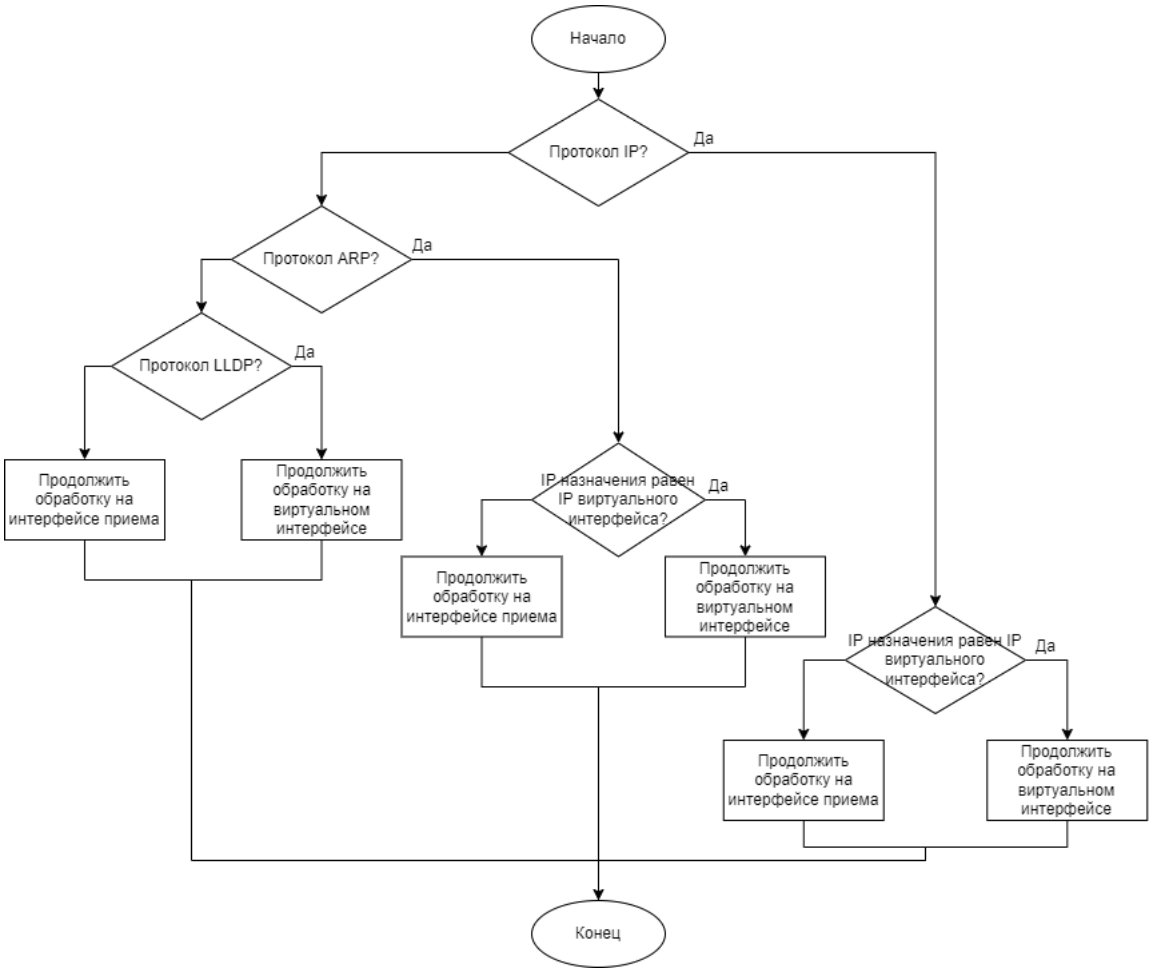

Рисунок 4 – Схема алгоритма обработки принимаемого пакета

2.3 Алгоритм отправки пакета

Так как необходимо хранить информацию о нескольких интерфейсах, было принято решение использовать связанный список, элементами которого будет информация о интерфейсе.

Схема алгоритма обработки отправляемого пакета, поступившего на виртуальный интерфейс представлена на рисунке 5.

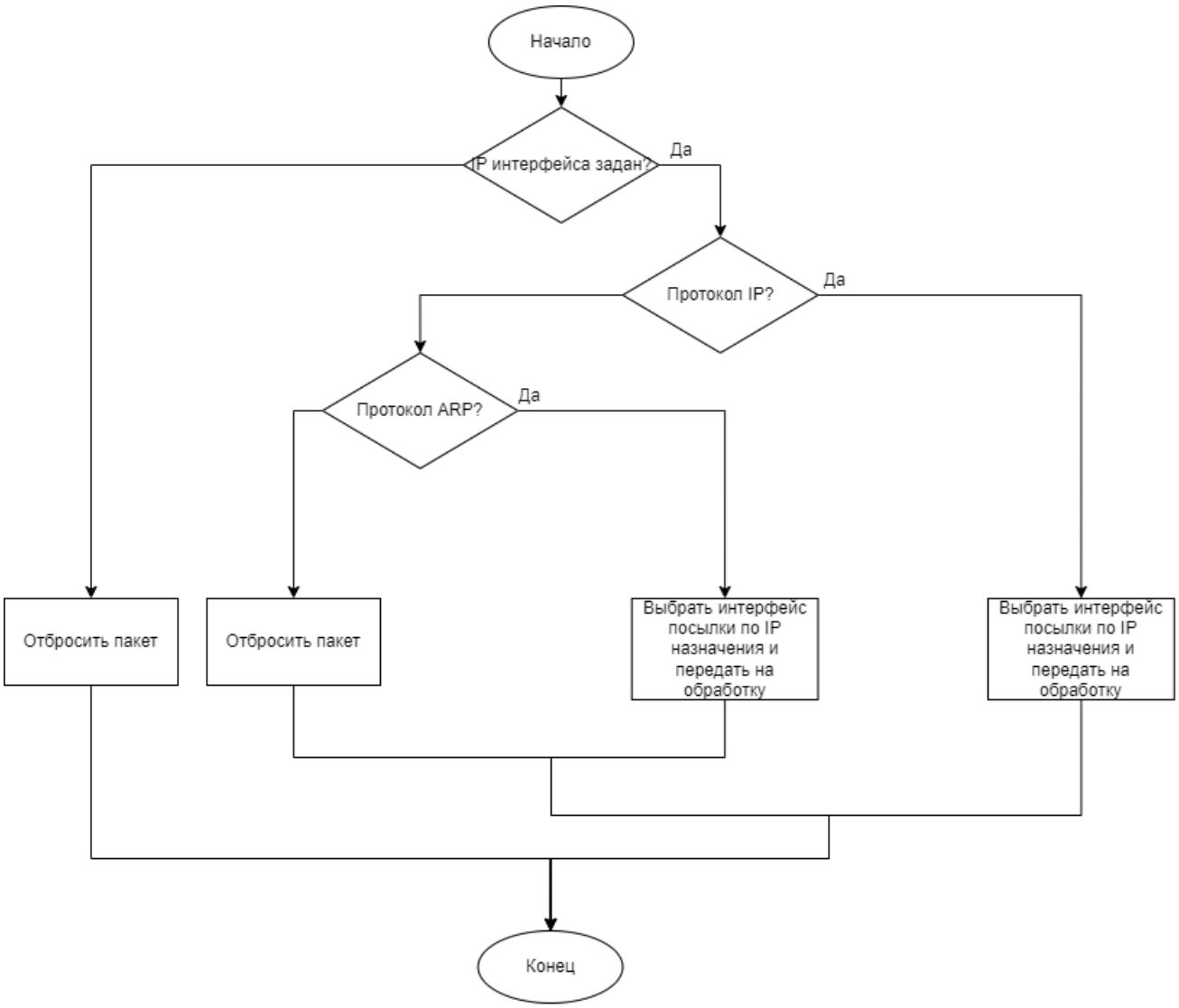

Рисунок 5 – Схема алгоритма обработки отправляемого пакета

2.4 Алгоритм инициализации

Схема алгоритма инициализации структур для работы 6.

Рисунок 6 – Схема алгоритма инициализации структур для работы

3 Технологический раздел

3.1 Выбор языка программирования и среды программирования

В качестве языка программирования для реализации поставленной задачи был выбран язык C. Он является языком реализации модулей ядра и самого ядра ОС Linux. В качестве компилятора был использован компилятор gcc. Средой разработки был выбран текстовый редактор Visual Studio Code.

3.2 Описание основных структур

Для реализации хранения нескольких интерфейсов для распределения пакетов было принято решение хранить в связанном списке ссылки на структуры net_device и дополнительную информацию об этих интерфейсах. Голова связанного списка хранится в приватной зоне памяти создаваемого виртуального интерфейса. На листинге 3 Приведена структура узла вышеупомянутого связанного списка.

Листинг 3 – struct interfaces

```
1  struct interfaces {
2    u32 address; // адрес подсети
3    u32 mask;    // маска подсети
4    struct net_device *device; // ссылка на интерфейс
5    struct interfaces *next;   // следующий узел
6  };
```

В приватной области памяти создаваемого виртуального сетевого интерфейса создается структура priv, хранящая статистику интерфейса и начало связанного списка интерфейсов для распределения пакетов. На листинге 4 представлена структура priv.

Листинг 4 – Структура приватной области интерфейса

```
struct priv
{
  struct net_device_stats stats;
  struct interfaces *next;
};
```

3.3 Реализация алгоритма получения пакета

В соответствии с алгоритмом из конструкторской части была спроектирована функция получения пакета.

Для избежания блокировки используемых интерфейсов необходимо передавать в созданный интерфейс только пакеты, предназначенные ему. Для этого в функции получения пакета производится проверка на соответствие IP. На листинге 5 представлена функция получения пакета.

Листинг 5 – Функция получения

```
tatic rx_handler_result_t handle_frame(struct sk_buff **pskb)
{
  struct sk_buff *skb = *pskb;
  struct in_device *in_dev = child->ip_ptr;
  struct in_ifaddr *ifa = in_dev->ifa_list;
  if (!ifa)
  {
    return RX_HANDLER_PASS;
  }
  u32 child_ip = ifa->ifa_address;
  if (skb->protocol == htons(ETH_P_IP))
  {
    struct iphdr *ip = ip_hdr(skb);
    LOG("INCOME: IP to IP=%s", strIP(ip->daddr));
    if (!ifa || ip->daddr != child_ip)
    {
      return RX_HANDLER_PASS;
    }
```

```
19      }
20      else if (skb->protocol == htons(ETH_P_ARP))
21      {
22        struct arphdr *arp = arp_hdr(skb);
23        struct arp_eth_body *body = (void *)arp + sizeof(struct
    arphdr);
24        int i, ip = child_ip;
25        LOG("INCOME: ARP for IP=%s", strAR_IP(body->ar_tip));
26        for (i = 0; i < sizeof(body->ar_tip); i++)
27        {
28          if ((ip & 0xFF) != body->ar_tip[i])
29          break;
30          ip = ip >> 8;
31        }
32        if (i < sizeof(body->ar_tip))
33        return RX_HANDLER_PASS;
34      }
35      else if (skb->protocol != htons(0xCC88))
36      {
37        return RX_HANDLER_PASS;
38      }
39
40      LOG("INCOME: PASS");
41      struct priv *priv = netdev_priv(child);
42      priv->stats.rx_packets++;
43      priv->stats.rx_bytes += skb->len;
44      skb->dev = child;
45      return RX_HANDLER_ANOTHER;
46    }
```

3.4 Реализация алгоритма отправки пакета

В соответствии с алгоритмом из конструкторской части была спроектирована функция отправки пакета.

Функция обработки входящего пакета в создаваемый сетевой интерфейс представлена на листинге 6. Видно, что при нахождении подходящего интерфейса меняется поле dev структуры sk_buff, а затем буфер отправляется на дальнейшую обработку функцией dev_queue_xmit.

Листинг 6 – start_xmit

```
1  static netdev_tx_t start_xmit(struct sk_buff *skb, struct
    net_device *dev)
2  {
3    struct in_device *in_dev = child->ip_ptr;
4    struct in_ifaddr *ifa = in_dev->ifa_list;
5    if (ifa)
6    {
7      struct priv *priv = netdev_priv(dev);
8      priv->stats.tx_packets++;
9      priv->stats.tx_bytes += skb->len;
10     LOG("GET IP %d, %s", get_ip(skb), strIP(get_ip(skb)));
11     struct net_device *device = find_device_sub(priv->next,
    get_ip(skb));
12     if (device)
13     {
14       skb->dev = device;
15       skb->priority = 1;
16       dev_queue_xmit(skb);
17       LOG("OUTPUT: injecting frame from %s to %s. Tarhet IP: %s"
    , dev->name, skb->dev->name, strIP(get_ip(skb)));
18       return NETDEV_TX_OK;
19     }
20   }
21   return NETDEV_TX_OK;
22 }
```

3.5 Реализация алгоритма инициализации

В соответствии с алгоритмом из конструкторской части была спроектирована функция инициализации необходимых структур.

Функция init, вызываемая при загрузке модуля и производящая первичную инициализацию, представлена на листинге ??. В функции загрузки модуля происходит вызов функций инициализации и добавление элементов в связный список интерфейсов. Также происходит регистрация виртуального интерфейса и установка функции обработчика пакетов в связанные интерфейсы.

Листинг 7 – Функции загрузки и выгрузки модуля; label

```
1  int __init init(void)
2  {
3    int err = 0;
4    struct priv *priv;
5    char ifstr[40];
6    sprintf(ifstr, "%s%s", ifname, "%d");
7
8    child = alloc_netdev(sizeof(struct priv), ifstr,
   NET_NAME_UNKNOWN, setup);
9    if (child == NULL)
10   {
11     ERR("%s: allocate error", THIS_MODULE->name);
12     return -ENOMEM;
13   }
14   priv = netdev_priv(child);
15   struct net_device *device = __dev_get_by_name(&init_net, link)
   ; // parent interface
16   if (!device)
17   {
18     ERR("%s: no such net: %s", THIS_MODULE->name, link);
19     err = -ENODEV;
20     free_netdev(child);
```

```
21        return err;
22      } else if (device->type != ARPHRD_ETHER && device->type !=
    ARPHRD_LOOPBACK)
23      {
24        ERR("%s: illegal net type", THIS_MODULE->name);
25        err = -EINVAL;
26        free_netdev(child);
27        return err;
28      }
29
30      struct interfaces *second = kmalloc(sizeof(struct interfaces),
     GFP_KERNEL);
31      second->address = charToIP(0, 0, (char)0, (char)0);
32      second->mask = charToIP(0, 0, (char)0, (char)0);
33      second->device = device;
34      second->next = NULL;
35
36      struct interfaces *first = kmalloc(sizeof(struct interfaces),
    GFP_KERNEL);
37      first->address = charToIP(192, 168, (char)1, (char)0);
38      first->mask = charToIP(255, 255, (char)255, (char)0);
39      first->device = device;
40      first->next = second;
41
42      priv->next = first;
43      memcpy(child->dev_addr, device->dev_addr, ETH_ALEN);
44      memcpy(child->broadcast, device->broadcast, ETH_ALEN);
45      if ((err = dev_alloc_name(child, child->name)))
46      {
47        ERR("%s: allocate name, error %i", THIS_MODULE->name, err);
48        err = -EIO;
49        free_netdev(child);
50        return err;
51      }
52      register_netdev(child);
```

```
53      rtnl_lock();
54      netdev_rx_handler_register(device, &handle_frame, NULL);
55      rtnl_unlock();
56      LOG("module %s loaded", THIS_MODULE->name);
57      LOG("%s: create link %s", THIS_MODULE->name, child->name);
58      LOG("%s: registered rx handler for %s", THIS_MODULE->name,
        priv->next->device->name);
59      return 0;
60  }
```

3.6 Дополнительные функции

Для поиска подходящего интерфейса по связанному списку была написана функция find_device_sub. В ней осуществляется проверка принадлежности адреса назначения к подсети по этому интерфейсу. Она представлена на листинге 8.

Листинг 8 – find_device_sub

```
1  static struct net_device *find_device_sub(struct interfaces *subs,
       u32 addr)
2  {
3      struct net_device *device = NULL;
4      while (subs && !device)
5      {
6          u32 res = apply_mask(subs->mask, addr);
7          if (res == subs->address)
8          {
9              device = subs->device;
10         }
11         else
12         {
13             subs = subs->next;
14         }
15     }
```

```
16
17      return device;
18  }
```

На листинге 9 показана инициализация структуры net_device_ops и определение оставшихся необходимых функций.

Листинг 9 – net_device_ops

```
1  static int open( struct net_device *dev ) {
2     netif_start_queue( dev );
3     LOG( "%s: device opened", dev->name );
4     return 0;
5  }
6
7  static int stop(struct net_device *dev)
8  {
9      netif_stop_queue(dev);
10     LOG("%s: device closed", dev->name);
11     return 0;
12 }
13
14 static struct net_device_stats *get_stats(struct net_device *dev)
15 {
16     return &((struct priv *)netdev_priv(dev))->stats;
17 }
18
19 static struct net_device_ops crypto_net_device_ops = {
20     .ndo_open = open,
21     .ndo_stop = stop,
22     .ndo_get_stats = get_stats,
23     .ndo_start_xmit = start_xmit,
24 };
```

Полный текст программы можно посмотреть в Приложении А.

3.7 Makefile

В листинге 10 приведен файл Makefile, используемый для сборки модуля ядра.

Листинг 10 – Makefile

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
MAKE = make

TARGET1 = cvirt
obj-m := $(TARGET1).o

all: default clean

default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	@rm -f *.o .*.cmd .*.flags *.mod.c *.order
	@rm -f .*.*.cmd *.symvers *~ *.*~ TODO.*
	@rm -fR .tmp*
	@rm -rf .tmp_versions

disclean: clean
	@rm -f *.ko
```

4 Исследовательская часть

На рисунке 7 показана загрузка модуля в ядро.

```
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ sudo insmod cvirt.ko
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 08:00:27:61:1b:e8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.43.181/24 brd 192.168.43.255 scope global dynamic noprefixroute
 enp0s3
       valid_lft 3215sec preferred_lft 3215sec
    inet6 fe80::5568:d07b:16b2:234f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
4: virt0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOW
N group default qlen 1000
    link/ether 08:00:27:61:1b:e8 brd ff:ff:ff:ff:ff:ff
```

Рисунок 7 – Загрузка модуля в ядро

На рисунке 8 и 9 показана настройка интерфейса для приема и передачи пакетов.

```
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ sudo ip addr add 192.168.43.182/2
4 dev virt0
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ ip a | grep virt0
4: virt0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOW
N group default qlen 1000
    inet 192.168.43.182/24 scope global virt0
pashok@pashok-VirtualBox:~/Desktop/virt-proto$
```

Рисунок 8 – Присвоение адреса интерфейсу

```
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
192.168.43.0    0.0.0.0         255.255.255.0   U     0      0        0 virt0
192.168.43.0    0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ sudo ip route add 0.0.0.0/0 via 1
92.168.43.71 dev virt0
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.43.71   0.0.0.0         UG    0      0        0 virt0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
192.168.43.0    0.0.0.0         255.255.255.0   U     0      0        0 virt0
192.168.43.0    0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
pashok@pashok-VirtualBox:~/Desktop/virt-proto$
```

Рисунок 9 – Добавление пути в таблицу путей Linux

На рисунке 10 показано, что через виртуальный интерфейс теперь можно получить доступ к машинам с внешним IP. А на рисунке 11 пока-

заны логи ядра, показывающие, что действительно происходит распределение по интерфейсам.

```
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ ping 192.168.43.71 -I virt0
PING 192.168.43.71 (192.168.43.71) from 192.168.43.182 virt0: 56(84) bytes of da
ta.
64 bytes from 192.168.43.71: icmp_seq=1 ttl=64 time=27.4 ms
64 bytes from 192.168.43.71: icmp_seq=2 ttl=64 time=19.7 ms
64 bytes from 192.168.43.71: icmp_seq=3 ttl=64 time=25.0 ms
^C
--- 192.168.43.71 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 19.714/24.079/27.460/3.237 ms
pashok@pashok-VirtualBox:~/Desktop/virt-proto$ ping 1.1.1.1 -I virt0
PING 1.1.1.1 (1.1.1.1) from 192.168.43.182 virt0: 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=54 time=77.6 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=54 time=55.5 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=54 time=42.5 ms
^C
--- 1.1.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 42.591/58.604/77.652/14.473 ms
pashok@pashok-VirtualBox:~/Desktop/virt-proto$
```

Рисунок 10 – Ping

```
[ 1215.587524] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 192.168.43.71
[ 1215.612500] ! INCOME: IP to IP=192.168.43.182
[ 1215.612591] ! INCOME: PASS
[ 1218.681929] ! INCOME: ARP for IP=192.168.43.182
[ 1218.681970] ! INCOME: PASS
[ 1218.682052] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 192.168.43.71
[ 1221.721818] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 35.224.170.84
[ 1221.966561] ! INCOME: IP to IP=192.168.43.182
[ 1221.966605] ! INCOME: PASS
[ 1221.966708] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 35.224.170.84
[ 1221.967158] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 35.224.170.84
[ 1222.360296] ! INCOME: IP to IP=192.168.43.182
[ 1222.360355] ! INCOME: PASS
[ 1222.360368] ! INCOME: IP to IP=192.168.43.182
[ 1222.360379] ! INCOME: PASS
[ 1222.360391] ! INCOME: IP to IP=192.168.43.182
[ 1222.360400] ! INCOME: PASS
[ 1222.360483] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 35.224.170.84
[ 1222.360801] ! OUTPUT: injecting frame from virt0 to enp0s3. Target IP: 35.224.170.84
```

Рисунок 11 – Логи ядра

Заключение

В процессе выполнения курсовой работы по курсу операционные системы был реализован загружаемый модуль ядра для распределения пакетов по существующим интерфейсам.

Проанализирована сетевая подсистема Linux, изучены структуры, позволяющие работать с виртуальными сетевыми интерфейсам.

Сетевой интерфейс был разработан и реализован, была показана его работоспособность.

Список литературы

1. Andrew S. Tanenbaum, Herbert BOS // Modern Operating Systems FOURTH EDITION с. 479-480.

2. Типы устройств ОС Linux [Электронный ресурс] URL: http://dmilvdv.narod.ru/Translate/LDD3/ldd_classes_devices_modules.html (дата обращения: 22.12.2021).

3. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman // Linux Drivers Development, Third Edition с.497.

4. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman // Linux Drivers Development, Third Edition с.503.

5. TUN/TAP интерфейсы [Электронный ресурс] URL: https://www.kernel.org/doc/html/latest/networking/tuntap.html (дата обращения: 22.12.2021).

6. netdevice.h. Исходный код ядра v5.4 [Электронный ресурс] URL: https://elixir.bootlin.com/linux/v5.4/source/include/linux/netdevice.h (дата обращения: 22.12.2021). 22.12.2021).

Приложение А

Листинг 11 – Полный код разработанного модуля ядра

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/inetdevice.h> //if_addr
#include <net/ip.h>

static char *link = "enp0s3"; // имя родительского интерфейса
module_param(link, charp, 0);

static char *ifname = "virt"; // имя создаваемого интерфейса
module_param(ifname, charp, 0);

static struct net_device *child = NULL;
struct interfaces {
  u32 address; // адрес подсети
  u32 mask;    // маска подсети
  struct net_device *device; // ссылка на интерфейс
  struct interfaces *next;   // следующий узел
};

struct priv
{
  struct net_device_stats stats;
  struct interfaces *next;
};

struct arp_eth_body {
  unsigned char  ar_sha[ ETH_ALEN ];     // sender hardware
 address
  unsigned char  ar_sip[ 4 ];            // sender IP address
  unsigned char  ar_tha[ ETH_ALEN ];     // target hardware
 address
```

```
30      unsigned char  ar_tip[ 4 ];            // target IP address
31    };
32
33    #define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
34    #define LOG(...) printk(KERN_INFO "! "__VA_ARGS__)
35
36    static u32 apply_mask(u32 addr, u32 mask)
37    {
38      return (addr & mask);
39    }
40    static char *strIP(u32 addr);
41    static struct net_device *find_device_sub(struct interfaces *
       subs, u32 addr)
42    {
43      struct net_device *device = NULL;
44      while (subs && !device)
45      {
46        u32 res = apply_mask(subs->mask, addr);
47        if (res == subs->address)
48        {
49          device = subs->device;
50        }
51        else
52        {
53          subs = subs->next;
54        }
55      }
56
57      return device;
58    }
59
60
61
62    static char *strIP(u32 addr)
63    { // диагностика IP в точечной нотации
```

```
64      static char saddr[MAX_ADDR_LEN];
65      sprintf(saddr, "%d.%d.%d.%d",
66      (addr)&0xFF, (addr >> 8) & 0xFF,
67      (addr >> 16) & 0xFF, (addr >> 24) & 0xFF);
68      return saddr;
69    }
70
71    static char* strAR_IP( unsigned char addr[ 4 ] ) {
72      static char saddr[ MAX_ADDR_LEN ];
73      sprintf( saddr, "%d.%0d.%d.%d",
74      addr[ 0 ], addr[ 1 ], addr[ 2 ], addr[ 3 ] );
75      return saddr;
76    }
77
78    static void print_ip(struct sk_buff *skb)
79    {
80      if (skb->protocol == htons(ETH_P_IP))
81      {
82        struct iphdr *ip = ip_hdr(skb);
83        char daddr[MAX_ADDR_LEN], saddr[MAX_ADDR_LEN];
84        strcpy(daddr, strIP(ip->daddr));
85        strcpy(saddr, strIP(ip->saddr));
86        LOG("re: from IP=%s to IP=%s with length: %u", saddr, daddr,
      skb->len);
87      }
88      else if (skb->protocol == htons(ETH_P_ARP))
89      {
90        struct arphdr *arp = arp_hdr(skb);
91        struct arp_eth_body *body = (void *)arp + sizeof(struct
     arphdr);
92        LOG("re: ARP for %s", strAR_IP(body->ar_tip));
93      }
94      return 0;
95    }
96
```

```
97  static u32 charToIP( unsigned char fir, unsigned char sec,
     unsigned char thd, unsigned char frth ) {
98    u32 fourth = frth;
99    u32 third = thd;
100   u32 second = sec;
101   u32 first = fir;
102   LOG("%d %d", (fourth << 24) | (third << 16), (second << 8) |
     first);
103   return  (fourth << 24)  | (third << 16) | (second << 8) | (
     first);
104 }
105
106 static u32 get_ip(struct sk_buff *skb)
107 {
108   if (skb->protocol == htons(ETH_P_IP))
109   {
110     struct iphdr *ip = ip_hdr(skb);
111     return (ip->daddr);//&0xFF | (ip->daddr >> 8) & 0xFF |
112     //(ip->daddr >> 16) & 0xFF | (ip->daddr >> 24) & 0xFF;
113   }
114   else if (skb->protocol == htons(ETH_P_ARP))
115   {
116     struct arphdr *arp = arp_hdr(skb);
117     struct arp_eth_body *body = (void *)arp + sizeof(struct
     arphdr);
118
119     return (body->ar_tip[0]) | (body->ar_tip[1] << 8) | (body->
     ar_tip[2] << 16) | (body->ar_tip[3] << 24);
120   }
121 }
122
123 static rx_handler_result_t handle_frame(struct sk_buff **pskb)
124 {
125   struct sk_buff *skb = *pskb;
126   struct in_device *in_dev = child->ip_ptr;
```

```
127     struct in_ifaddr *ifa = in_dev->ifa_list;
128     if (!ifa)
129     {
130       return RX_HANDLER_PASS;
131     }
132     u32 child_ip = ifa->ifa_address;
133     if (skb->protocol == htons(ETH_P_IP))
134     {
135       struct iphdr *ip = ip_hdr(skb);
136       LOG("INCOME: IP to IP=%s", strIP(ip->daddr));
137       if (!ifa || ip->daddr != child_ip)
138       {
139         return RX_HANDLER_PASS;
140       }
141     }
142     else if (skb->protocol == htons(ETH_P_ARP))
143     {
144       struct arphdr *arp = arp_hdr(skb);
145       struct arp_eth_body *body = (void *)arp + sizeof(struct
    arphdr);
146       int i, ip = child_ip;
147       LOG("INCOME: ARP for IP=%s", strAR_IP(body->ar_tip));
148       for (i = 0; i < sizeof(body->ar_tip); i++)
149       {
150         if ((ip & 0xFF) != body->ar_tip[i])
151         break;
152         ip = ip >> 8;
153       }
154       if (i < sizeof(body->ar_tip))
155       return RX_HANDLER_PASS;
156     }
157     else if (skb->protocol != htons(0xCC88))
158     {
159       return RX_HANDLER_PASS;
160     }
```

```
161
162     LOG("INCOME: PASS");
163     struct priv *priv = netdev_priv(child);
164     priv->stats.rx_packets++;
165     priv->stats.rx_bytes += skb->len;
166     skb->dev = child;
167     return RX_HANDLER_ANOTHER;
168   }
169
170   static int open( struct net_device *dev ) {
171     netif_start_queue( dev );
172     LOG( "%s: device opened", dev->name );
173     return 0;
174   }
175
176   static int stop(struct net_device *dev)
177   {
178     netif_stop_queue(dev);
179     LOG("%s: device closed", dev->name);
180     return 0;
181   }
182
183   static netdev_tx_t start_xmit(struct sk_buff *skb, struct
      net_device *dev)
184   {
185     struct in_device *in_dev = child->ip_ptr;
186     struct in_ifaddr *ifa = in_dev->ifa_list;
187     if (ifa)
188     {
189       struct priv *priv = netdev_priv(dev);
190       priv->stats.tx_packets++;
191       priv->stats.tx_bytes += skb->len;
192       struct net_device *device = find_device_sub(priv->next,
      get_ip(skb));
193       if (device)
```

```
194       {
195         skb->dev = device;
196         skb->priority = 1;
197         dev_queue_xmit(skb);
198         LOG("OUTPUT: injecting frame from %s to %s. Target IP: %s"
    , dev->name, skb->dev->name, strIP(get_ip(skb)));
199         return NETDEV_TX_OK;
200       }
201     }
202     return NETDEV_TX_OK;
203   }
204
205   static struct net_device_stats *get_stats(struct net_device *dev
     )
206   {
207     return &((struct priv *)netdev_priv(dev))->stats;
208   }
209
210   static struct net_device_ops crypto_net_device_ops = {
211     .ndo_open = open,
212     .ndo_stop = stop,
213     .ndo_get_stats = get_stats,
214     .ndo_start_xmit = start_xmit,
215   };
216
217   static void setup(struct net_device *dev)
218   {
219     int j;
220     ether_setup(dev);
221     memset(netdev_priv(dev), 0, sizeof(struct priv));
222     dev->netdev_ops = &crypto_net_device_ops;
223     for (j = 0; j < ETH_ALEN; ++j) // Заполнить MAC адрес
224     {
225       dev->dev_addr[j] = (char)j;
226     }
```

```
227  }
228
229  int __init init(void)
230  {
231    int err = 0;
232    struct priv *priv;
233    char ifstr[40];
234    sprintf(ifstr, "%s%s", ifname, "%d");
235
236    child = alloc_netdev(sizeof(struct priv), ifstr,
     NET_NAME_UNKNOWN, setup);
237    if (child == NULL)
238    {
239      ERR("%s: allocate error", THIS_MODULE->name);
240      return -ENOMEM;
241    }
242    priv = netdev_priv(child);
243    struct net_device *device = __dev_get_by_name(&init_net, link)
     ; // parent interface
244    if (!device)
245    {
246      ERR("%s: no such net: %s", THIS_MODULE->name, link);
247      err = -ENODEV;
248      free_netdev(child);
249      return err;
250    } else if (device->type != ARPHRD_ETHER && device->type !=
     ARPHRD_LOOPBACK)
251    {
252      ERR("%s: illegal net type", THIS_MODULE->name);
253      err = -EINVAL;
254      free_netdev(child);
255      return err;
256    }
257
```

```
258     struct interfaces *second = kmalloc(sizeof(struct interfaces),
        GFP_KERNEL);
259     second->address = charToIP(0, 0, (char)0, (char)0);
260     second->mask = charToIP(0, 0, (char)0, (char)0);
261     second->device = device;
262     second->next = NULL;
263
264     struct interfaces *first = kmalloc(sizeof(struct interfaces),
       GFP_KERNEL);
265     first->address = charToIP(192, 168, (char)43, (char)71);
266     first->mask = charToIP(255, 255, (char)255, (char)0);
267     first->device = device;
268     first->next = second;
269
270     priv->next = first;
271     memcpy(child->dev_addr, device->dev_addr, ETH_ALEN);
272     memcpy(child->broadcast, device->broadcast, ETH_ALEN);
273     if ((err = dev_alloc_name(child, child->name)))
274     {
275       ERR("%s: allocate name, error %i", THIS_MODULE->name, err);
276       err = -EIO;
277       free_netdev(child);
278       return err;
279     }
280     register_netdev(child);
281     rtnl_lock();
282     netdev_rx_handler_register(device, &handle_frame, NULL);
283     rtnl_unlock();
284     LOG("module %s loaded", THIS_MODULE->name);
285     LOG("%s: create link %s", THIS_MODULE->name, child->name);
286     LOG("%s: registered rx handler for %s", THIS_MODULE->name,
      priv->next->device->name);
287     return 0;
288   }
289
```

```
290  void __exit exit(void)
291  {
292    struct priv *priv = netdev_priv(child);
293    struct interfaces *next = priv->next;
294    while (next)
295    {
296      rtnl_lock();
297      netdev_rx_handler_unregister(next->device);
298      rtnl_unlock();
299      LOG("unregister rx handler for %s\n", next->device->name);
300      next = next->next;
301    }
302    unregister_netdev(child);
303    free_netdev(child);
304    LOG("module %s unloaded", THIS_MODULE->name);
305  }
306
307  module_init(init);
308  module_exit(exit);
309
310  MODULE_AUTHOR("Pavel Khetagurov");
311  MODULE_LICENSE("GPL v2");
312  MODULE_VERSION("0.1");
```