Основы работы сетевого стека ядра Linux

В данном разделе разбирается устройство сетевого стека ядра Linux от сетевой карты до сокетов.

1. Драйвер сетевой карты
    1. Основные технологии
        1. DMA
        2. DCA
        3. Кольцевые очереди
        4. SoftIRQ
        5. NAPI
    2. Установка и удаление
    3. Получение сетевых пакетов
    4. Отправка сетевых пакетов
2. Сокеты ядра Linux
3. Полезные материалы
4. Источники

Драйвер сетевой карты

В ядре «Linux» существует только три типа устройств [1]:

- символьные устройства;
- блочные устройства;
- сетевые интерфейсы.

За работу сетевого интерфейса отвечает устройство NIC (Network interface controller) или сетевая карта — это аппаратный компонент, который устанавливается в компьютер или сервер для подключения к локальной сети (LAN) [2]. В эталонной модели OSI сетевая карта отвечает не только за работу физического уровня, определяя способ передачи сетевых пакетов, но и за канальный уровень, управляя получаемыми и отправляемыми кадрами. Для этого устройство выполняет следующие функции:

1. формирование кадров и проверка хеш-сумм;
2. запись кадров в оперативную память;
3. фильтрация трафика по адресам;
4. генерация прерываний;
5. аппаратное ускорение обработки пакетов.

Для управления сетевой картой и обеспечением доступа к передаваемым пакетам используется драйвер сетевой карты. Его реализация зависит от используемой операционной системы, но его выполняемые функции остаются общими:

1. инициализация сетевой карты и управление её параметрами;
2. регистрация устройства в ядре операционной системе;
3. обработка прерываний;

4. запись и чтение кадров.

Ядро Linux является монолитным с поддержкой модулей ядра [2]. Модули ядра могут выполнять различные функции от реализации драйверов (модуль «IGB» [3]) и файловых систем (модуль «BTRFS» [4]) до виртуализации (модуль «KVM» [5]), поэтому далее понятия модуля ядра и драйвера будут одним и тем же. Дальнейшее описание работы драйверов сетевых карт будет основано на реализацииЫ модуля «IGB», так как его работу можно эмулировать в системе виртуализации «QEMU» (см. создание песочницы).

Основные технологии

Рассмотрим технологии, которые применяются при работе сетевой карты.

DMA

DMA (Direct Memory Access) — это технология, позволяющая устройствам ввода/вывода читать и записывать данные в оперативную память напрямую, без участия центрального процессора (CPU).

Для настройки работы DMA используются функции «dma_set_mask», которая настраивает маску для потоковых DMA-операций (одиночные передачи), «dma_set_coherent_mask», которая настраивает маску для когерентных DMA-операций (постоянные отображения памяти), или «dma_set_mask_and_coherent», которая настраивает маску и для потоковых, и для когерентных DMA-операций [7].

(Описать картинку отображения памяти).

DCA

Кольцевые очереди

SoftIRQ

NAPI

(Устройства на шине PCI [2]) (Драйверы: сетевой интерфейс [2]) (Из исходного кода/module_deinit)

Установка и удаление

Каждое подключенное устройство, например, по шине PCI (Peripheral component interconnect) или по USB (Universal Serial Bus) имеет два численных индекса [1]:

1. VID (Vendor ID) — численный индекс производителя;
2. PID (Product ID) или DID (Device ID) — численный индекс продукта.

Эта пара индексов отвечает за обнаружение драйвером устройства, с которым он умеет работать. Чтобы понять, с какими устройствами может работать драйвер, необходимо найти список структур «pci_device_id», в котором перечислены пары VID и PID поддерживаемых устройств.

```
// src/e1000_hw.h
// Пример индексов PID
```

```
#define E1000_DEV_ID_I354_BACKPLANE_1GBPS   0x1F40
#define E1000_DEV_ID_I354_SGMII         0x1F41
```

```
// src/igb_main.c
// Пример списка поддерживаемых устройств драйвера «IGB»
static const struct pci_device_id igb_pci_tbl[] = {
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_I354_BACKPLANE_1GBPS) },
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_I354_SGMII) },
    /* ... */
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_82575EB_FIBER_SERDES) },
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_82575GB_QUAD_COPPER) },
    /* required last entry */
    {0, }
};

// Регистрация списка в системе через файл
// /lib/modules/$(uname -r)/modules.alias
MODULE_DEVICE_TABLE(pci, igb_pci_tbl);
```

После загрузки модуля, например, с помощью команд `insmod` или `modprobe` выполняется функция, которая передается в макрос `module_init`.

```
// src/igb_main.c
// Структура с информацией о модуле
static struct pci_driver igb_driver = {
    // Имя модуля
    .name     = igb_driver_name,
    // Поддерживаемые устройства
    .id_table = igb_pci_tbl,
    // Функция регистрация устройства
    .probe    = igb_probe,
    // Функция удаления устройства
    .remove   = __devexit_p(igb_remove),
    // Функция приостановки устройства
    .suspend  = igb_suspend,
    // Функция возобновления работы устройства
    .resume   = igb_resume,
    // Возможны и другие функции
    // в зависимости от конфигурации
    /* ... */
};

// Пример установки модуля
static int __init igb_init_module(void)
{
    /* ... */
    // Установка структуры с данными драйвера
    ret = pci_register_driver(&igb_driver);
    /* ... */
```

```
}

// Регистрация функции установки модуля
module_init(igb_init_module);
```

После того, как модуль будет установлен в ядро, запустится функция `igb_probe` (`igb_driver.probe`) для каждого поддерживаемого устройства, которая выполнит их инициализацию. Для драйвера «IGB» устройствами будут являться сетевые интерфейсы. Их инициализация состоит из следующих шагов:

1. Инициализация PCI-устройства [6];
2. Установка маски DMA (см. DMA) [7];
3. Резервирование участков памяти [6];
4. Захват шины PCI для управления устройством [6];
5. Создание и заполнение структуры «net_device» для регистрации сетевого интерфейса [8];
6. Регистрирация поддерживаемых функций ethtool (см. настройка и тестирование);
7. Настройка прерываний и подсистемы NAPI (см. NAPI) [9];
8. Настройка наблюдателя, который перезагружает сетевой интерфейс в случае проблем;
9. Установка новых настроек интерфейса;
10. Регистрация интерфейса в сетевой части ядра;
11. Получение прямого доступа к управлению интерфейсом;
12. Инициализация технологии DCA (см. DCA) [10];
13. Множество других настроек в зависимости от конфигурации и устройства.

```
// src/igb/igb_main.c
// Структура с операциями над сетевым интерфейсом
static const struct net_device_ops igb_netdev_ops = {
    .ndo_open       = igb_open,
    .ndo_stop       = igb_close,
    .ndo_start_xmit     = igb_xmit_frame,
    .ndo_get_stats      = igb_get_stats,
    .ndo_set_rx_mode    = igb_set_rx_mode,
    .ndo_set_mac_address    = igb_set_mac,
    /* ... */
};
```

```
// src/igb/igb_ethtool.c
// Структура с операциями над сетевым интерфейсом при помощи ethtool
static const struct ethtool_ops igb_ethtool_ops = {
    /* ... */
    .get_drvinfo            = igb_get_drvinfo,
    .get_regs_len           = igb_get_regs_len,
    .get_regs               = igb_get_regs,
    .get_wol                = igb_get_wol,
    .set_wol                = igb_set_wol,
    .get_msglevel           = igb_get_msglevel,
    .set_msglevel           = igb_set_msglevel,
```

```
    .nway_reset             = igb_nway_reset,
    .get_link               = igb_get_link,
    .get_eeprom_len         = igb_get_eeprom_len,
    .get_eeprom             = igb_get_eeprom,
    .set_eeprom             = igb_set_eeprom,
    .get_ringparam          = igb_get_ringparam,
    .set_ringparam          = igb_set_ringparam,
    .get_pauseparam         = igb_get_pauseparam,
    .set_pauseparam         = igb_set_pauseparam,
    .self_test              = igb_diag_test,
    .get_strings            = igb_get_strings,
    /* ... */
};
```

```
// src/igb/igb_main.c
// Пример инициализации устройства
static int igb_probe(struct pci_dev *pdev,
                     const struct pci_device_id *ent)
{
    /* ... */
    struct net_device *netdev;
    struct igb_adapter *adapter;
    int err;
    /* ... */
    // Инициализация PCI-устройства
    err = pci_enable_device_mem(pdev);
    if (err)
        return err;

    // Установка маски DMA
    err = dma_set_mask(pci_dev_to_dev(pdev), DMA_BIT_MASK(64));
    if (!err) {
        err = dma_set_coherent_mask(pci_dev_to_dev(pdev),
            DMA_BIT_MASK(64));
        if (!err)
            pci_using_dac = 1;
    } else {
        err = dma_set_mask(pci_dev_to_dev(pdev), DMA_BIT_MASK(32));
        if (err) {
            err = dma_set_coherent_mask(pci_dev_to_dev(pdev),
                DMA_BIT_MASK(32));
            if (err) {
                IGB_ERR(
                  "No usable DMA configuration, aborting\n");
                goto err_dma;
            }
        }
    }

    /* ... */
```

```
    // Резервирование участков памяти
    err = pci_request_selected_regions(pdev,
                      pci_select_bars(pdev,
                              IORESOURCE_MEM),
                      igb_driver_name);
    if (err)
        goto err_pci_reg;

    /* ... */

    // Захват шины PCI для управления устройством
    pci_set_master(pdev);

    /* ... */

    // Создание и заполнение структуры net_device для регистрации сетевого
интерфейса
    netdev = alloc_etherdev_mq(sizeof(struct igb_adapter),
                   IGB_MAX_TX_QUEUES);

    /* ... */

    if (!netdev)
        goto err_alloc_etherdev;

    SET_MODULE_OWNER(netdev);
    SET_NETDEV_DEV(netdev, &pdev->dev);

    pci_set_drvdata(pdev, netdev);
    adapter = netdev_priv(netdev);
    adapter->netdev = netdev;
    adapter->pdev = pdev;
    hw = &adapter->hw;
    hw->back = adapter;
    adapter->port_num = hw->bus.func;
    adapter->msg_enable = GENMASK(debug - 1, 0);

    /* ... */

#ifdef HAVE_NET_DEVICE_OPS
    netdev->netdev_ops = &igb_netdev_ops;
#endif /* HAVE_NET_DEVICE_OPS */

    // Регистрирация поддерживаемых функций ethtool
    igb_set_ethtool_ops(netdev);

    /* ... */

    // Настройка прерываний и подсистемы NAPI
    // igb_sw_init вызывает igb_init_interrupt_scheme
    // igb_init_interrupt_scheme вызывает igb_alloc_q_vectors
    // igb_alloc_q_vectors вызывает igb_alloc_q_vector
    // igb_alloc_q_vector вызывает netif_napi_add
    err = igb_sw_init(adapter);
```

```
    if (err)
        goto err_sw_init;

    /* ... */

    // Настройка наблюдателя
    INIT_WORK(&adapter->reset_task, igb_reset_task);
    INIT_WORK(&adapter->watchdog_task, igb_watchdog_task);

    /* ... */

    // Установка новых настроек интерфейса
    igb_reset(adapter);

    /* ... */

    // Получение прямого доступа к управлению интерфейсом
    igb_get_hw_control(adapter);

    // Регистрация интерфейса в сетевой части ядра
    strscpy(netdev->name, "eth%d", IFNAMSIZ);
    err = register_netdev(netdev);
    if (err)
        goto err_register;

    /* ... */

    // Инициализация технологии DCA
#ifdef IGB_DCA
    if (dca_add_requester(&pdev->dev) == E1000_SUCCESS) {
        adapter->flags |= IGB_FLAG_DCA_ENABLED;
        dev_info(pci_dev_to_dev(pdev), "DCA enabled\n");
        igb_setup_dca(adapter);
    }
#endif

    /* ... */

}
```

Далее с помощью функций из структур `net_device_ops` и `ethtool_ops` происходит настройка сетевого интерфейса из пространства пользователя. Так при выполнении команды `ip link set up` выполняется функция `igb_open` (`net_device_ops.ndo_open`). Она выполняет следующее:

1. Получение настроек устройства;
2. Отключение несущей, чтобы исключить параллельную работу интерфейса;
3. Создание колец отправки и получения пакетов;
4. Включение несущей;
5. Инициализация и настройка различных параметров интерфейса;
6. Установка количества очередей отправки и получения пакетов;
7. Перевод интерфейса в включенное состояние;

8. Включение NAPI для каждой очереди;
9. Включение прерываний;
10. Включение возжности отправки пакетов;
11. Запуск наблюдателя.

```
// src/igb_main.c
// Пример запуска интерфейса
static int __igb_open(struct net_device *netdev, bool resuming)
{
    // Получение настроек устройства
    struct igb_adapter *adapter = netdev_priv(netdev);
    struct e1000_hw *hw = &adapter->hw;

    /* ... */

    // Отключение несущей
    netif_carrier_off(netdev);

    // Создание колец отправки пакетов
    err = igb_setup_all_tx_resources(adapter);
    if (err)
        goto err_setup_tx;

    // Создание колец получения пакетов
    err = igb_setup_all_rx_resources(adapter);
    if (err)
        goto err_setup_rx;

    // Включение несущей
    igb_power_up_link(adapter);

    // Инициализация и настройка различных параметров интерфейса
    igb_configure(adapter);

    // Настройка прерывааний
    err = igb_request_irq(adapter);
    if (err)
        goto err_req_irq;

    // Установка количества очередей отправки пакетов
    netif_set_real_num_tx_queues(netdev,
                 adapter->vmdq_pools ? 1 :
                 adapter->num_tx_queues);

    // Установка количества очередей получения пакетов
    err = netif_set_real_num_rx_queues(netdev,
                   adapter->vmdq_pools ? 1 :
                   adapter->num_rx_queues);
    if (err)
        goto err_set_queues;

    // Перевод интерфейса в включенное состояние
```

```
    clear_bit(__IGB_DOWN, adapter->state);

    // Включение NAPI для каждой очереди
    for (i = 0; i < adapter->num_q_vectors; i++)
        napi_enable(&(adapter->q_vector[i]->napi));
    igb_configure_lli(adapter);

    /* ... */

    // Включение прерываний
    igb_irq_enable(adapter);

    /* ... */

    // Включение возжности отправки пакетов
    netif_tx_start_all_queues(netdev);

    /* ... */

    // Запуск наблюдателя
    hw->mac.get_link_status = 1;
    schedule_work(&adapter->watchdog_task);

    return E1000_SUCCESS;

    /* ... */

}

int igb_open(struct net_device *netdev)
{
    return __igb_open(netdev, false);
}
```

Соответсвенно при выполнении команды `ip link set down` будет вызвана функция `igb_close` (`net_device_ops.ndo_close`), которая выполнит следующее:

1. Получение настроек устройства;
2. Переключение устройства в состояние DOWN;
3. Отключение несущей;
4. Остановка отправки пакетов;
5. Отключение NAPI;
6. Отключение прерываний;
7. Обновдение статистики интерфейса;
8. Обновление настроек;
9. Очистка колец отправки и приема пакетов;
10. Освобождение прямого управления устройством;
11. Освобождения памяти дли прерываний;
12. Освобождение памяти для колец отправки и получения пакетов.

```
// src/igb_main.c
// Пример выключения интерфейса
void igb_down(struct igb_adapter *adapter)
{
    /* ... */

    // Переключение устройства в состояние DOWN
    set_bit(__IGB_DOWN, adapter->state);

    /* ... */

    // Отключение несущей
    netif_carrier_off(netdev);
    // Остановка отправки пакетов
    netif_tx_stop_all_queues(netdev);

    /* ... */

    // Отключение NAPI
    for (i = 0; i < num_q_vectors; i++)
        napi_disable(&(adapter->q_vector[i]->napi));

    // Отключение прерываний
    igb_irq_disable(adapter);

    /* ... */

    // Обновдение статистики интерфейса
    igb_update_stats(adapter);

    adapter->link_speed = 0;
    adapter->link_duplex = 0;

    /* ... */

    // Обновление настроек
    igb_reset(adapter);

    /* ... */

    // Очистка колец отправки и приема пакетов
    igb_clean_all_tx_rings(adapter);
    igb_clean_all_rx_rings(adapter);

#ifdef IGB_DCA
    // Обновление DCA
    igb_setup_dca(adapter);
#endif
}

static int __igb_close(struct net_device *netdev, bool suspending)
{
    // Получение настроек устройства
```

```
    struct igb_adapter *adapter = netdev_priv(netdev);
#ifdef CONFIG_PM_RUNTIME
    struct pci_dev *pdev = adapter->pdev;
#endif /* CONFIG_PM_RUNTIME */

    /* ... */

    igb_down(adapter);

    // Освобождение прямого управления устройством
    igb_release_hw_control(adapter);

    // Освобождения памяти дли прерываний
    igb_free_irq(adapter);

    /* ... */

    // Освобождение памяти для колец отправки и получения пакетов
    igb_free_all_tx_resources(adapter);
    igb_free_all_rx_resources(adapter);

    /* ... */

    return 0;
}

int igb_close(struct net_device *netdev)
{
    return __igb_close(netdev, false);
}
```

При завершении работы сетевого драйвера для каждого интерфейса вызывается функция `igb_remove` (`igb_driver.remove`), которая освобождает управление сетевым интерфейсом. Её выполнение состоит из следующих шагов:

1. Переключение устройства в состояние DOWN;
2. Отключение наблюдателя;
3. Отключение технологии DCA;
4. Отключение прямого управления устройством;
5. Удаление интерфейса из сетевой части ядра;
6. Освобождение используемых прерываний;
7. Освободение PCI-устройства.

```
// src/igb_main.c
// Пример удаления устройства
static void igb_remove(struct pci_dev *pdev)
{
    struct net_device *netdev = pci_get_drvdata(pdev);
    struct igb_adapter *adapter = netdev_priv(netdev);
    struct e1000_hw *hw = &adapter->hw;
```

```
    /* ... */

    // Переключение устройства в состояние DOWN
    set_bit(__IGB_DOWN, adapter->state);
    // Отключение наблюдателя
    del_timer_sync(&adapter->watchdog_timer);
    if (adapter->flags & IGB_FLAG_DETECT_BAD_DMA)
        del_timer_sync(&adapter->dma_err_timer);
    del_timer_sync(&adapter->phy_info_timer);

    cancel_work_sync(&adapter->reset_task);
    cancel_work_sync(&adapter->watchdog_task);

    // Отключение технологии DCA
#ifdef IGB_DCA
    if (adapter->flags & IGB_FLAG_DCA_ENABLED) {
        dev_info(pci_dev_to_dev(pdev), "DCA disabled\n");
        dca_remove_requester(&pdev->dev);
        adapter->flags &= ~IGB_FLAG_DCA_ENABLED;
        E1000_WRITE_REG(hw, E1000_DCA_CTRL, E1000_DCA_CTRL_DCA_DISABLE);
    }
#endif

    /* ... */

    // Отключение прямого управления устройством
    igb_release_hw_control(adapter);

    // Удаление интерфейса из сетевой части ядра
    unregister_netdev(netdev);

    // Освобождение используемых прерываний
    igb_clear_interrupt_scheme(adapter);

    /* ... */

    // Освобождение используемой памяти
    pci_release_selected_regions(pdev,
                    pci_select_bars(pdev, IORESOURCE_MEM));

    /* ... */

    kfree(adapter->mac_table);
    kfree(adapter->shadow_vfta);
    free_netdev(netdev);

    /* ... */

    // Освободение PCI-устройства
    pci_disable_device(pdev);
}
```

Получение сетевых пакетов

(https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/)
(https://habr.com/ru/companies/vk/articles/314168/)

Отправка сетевых пакетов

(https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-sending-data/)

Сокеты ядра Linux

(https://man7.org/linux/man-pages/man7/socket.7.html) (https://habr.com/ru/articles/886058/)

Полезные материалы

- Стандарт ISO/IEC 7498 или ГОСТ Р ИСО/МЭК 7498-1-99.
- How To Write Linux PCI Drivers

Источники

1. Цилюрик О. И. Расширения ядра Linux: драйверы и модули. — СПб.: БХВ-Петербург, 688 с.: ил. ISBN 978-5-9775-1719-5
2. Определение сетевой карты
3. Документация ядра «Linux» о модуле «IGB»
4. Документация ядра «Linux» о модуле «BTRFS»
5. Документация ядра «Linux» о модуле «KVM»
6. Документация ядра «Linux» о работе с PCI
7. Документация ядра «Linux» о работе с DMA
8. Документация ядра «Linux» о работе с сетевыми интерфейсами
9. Документация ядра «Linux» о работе с NAPI
10. Ram Huggahalli, Ravi Iyer, Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O - 2005
11.