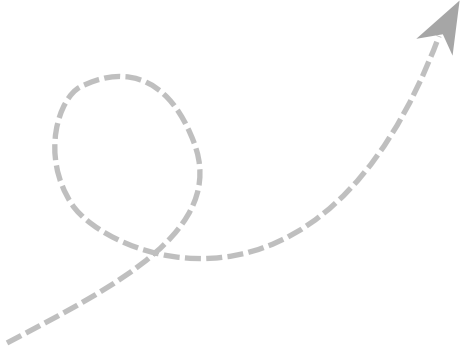
Лекция 4.1

Взаимодействие с устройствами ввода-вывода и DMA.

Lecturer: Alex Maximov

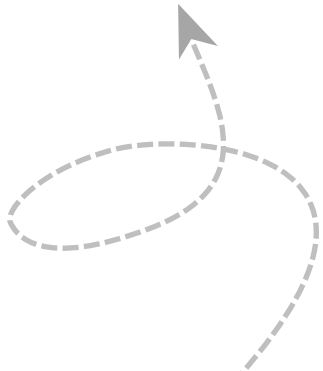
Position: SDC RTSoft - Head of the Software and Advanced Development Department



Version 1.0 2022

Содержание

- Взаимодействие с устройством через память
- DMA



Типы адресов

User virtual addresses – обычный адрес использующийся пользоватьскими программами 32 или 64 бита длины. Каждый процесс имеет свое адресное пространство.

Physical addresses

Адрес который используется между процессором и системной памятью.The addresses used between the processor and the system's memory. Physical addresses 32 или 64-bit. 32-ные системы могут иметь большее физическое пространство в ряде случаев.

Bus addresses

Адрес используемы между перефирийной шиной и памятью. Обычно такой же как физический. Некоторые процессоры имею I/O memory management unit (IOMMU) который транслирует адреса между шиной и основной памятью.

Логический адрес ядра

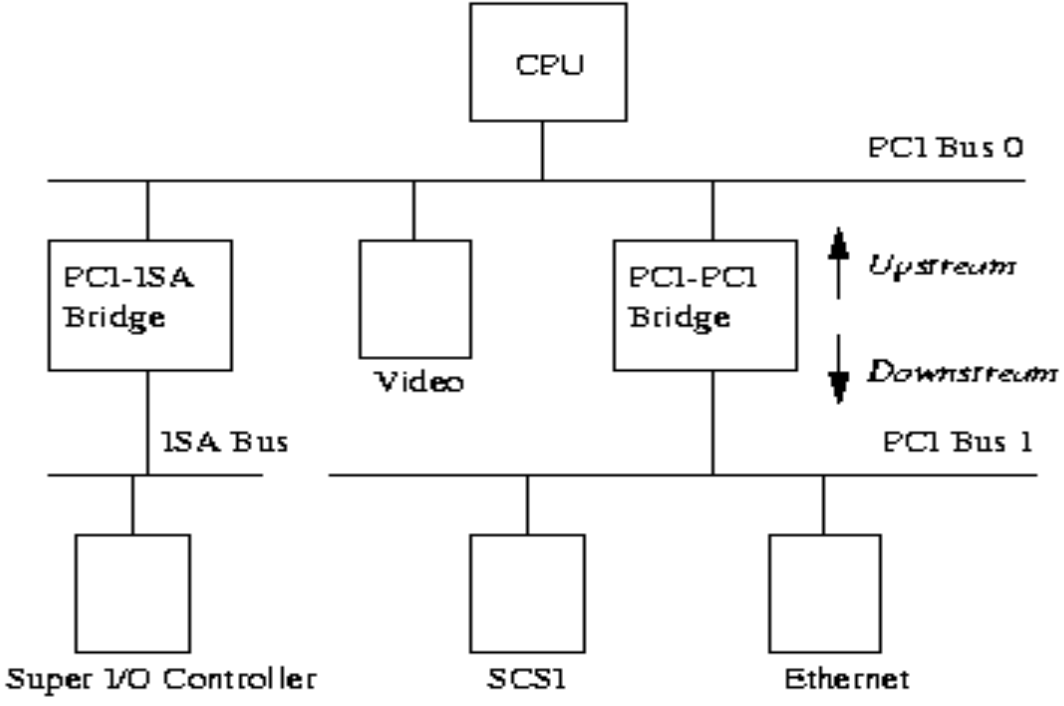
Обычный адрес в пространстве ядра. В большинстве архитектур логический адрес и физический адрес различаются только на константу. В случае наличия большого количества физической памяти логических адресов для доступа к ней может нахватать. kmalloc возвращает логический адрес.

<asm/page.h>

__pa( ) logical ->physical

__va( ) physical -> logical

Пример архитектуры системы



Взаимодействие с устройством

Устройства взаимодействуют в CPU через специальные регистры:

- управляющий регистр;
- регистр состояния;
- входной регистр;
- выходной регистр.

Регистры могут:

- расположены в специальном адресном пространстве (в пространстве портов ввода/вывода);
- отображаться в память.

Операции чтения обмена с устройствами могут дать неожиданный результат

Получение доступа к портам ввода-вывода


```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,
const char *name);

void release_region(unsigned long start, unsigned long n);

int check_region(unsigned long first, unsigned long n); /*Устарела с версии 2.4.xx*/
```

Посмотреть распределение портов можно:

/proc/ioports

Команды для работы взаимодействия через порты в.в.

- *Bytes*

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);

- Words

unsigned inw(unsigned port);
void outw(unsigned char byte, unsigned port);

- *“Long" integers*

unsigned inl(unsigned port);
void outl(unsigned char byte, unsigned port);

Реализация функций может отличаться на разных платформах.

Команды для работы взаимодействия через порты в.в.

- *byte strings*

void insb(unsigned port, void *addr, unsigned long count);

void outsb(unsigned port, void *addr, unsigned long count);

- *word strings*

void insw(unsigned port, void *addr, unsigned long count);

void outsw(unsigned port, void *addr, unsigned long count);

- long strings

void insl(unsigned port, void *addr, unsigned long count);

void outsl(unsigned port, void *addr, unsigned long count);

Передача нескольких значений. Может быть более эффективно, чем соответствующий цикл на С, если у процессора есть специальные команды

Получение адреса области портов ввода-вывода.

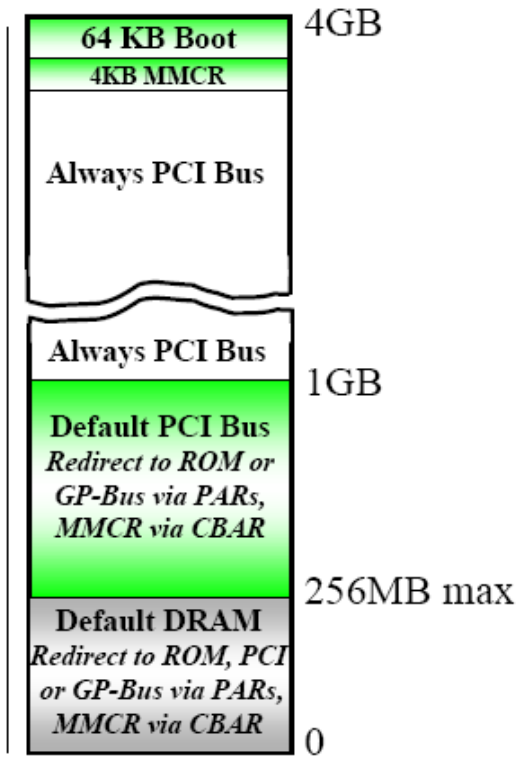
- Адрес области портов ввода вывода может быть получен:
- Через параметры модуля;
- Известен заранее и закодирован в драйвере
- Запрошен у платформы
- Прочитан из области конфигурирования шины PCI

Практический пример.

- Чтение портов ввода-вывода. Реализовать макросы изменения, установки и очистки бит в регистрах.

Обмен через память

- Обмен через область разделяемой памяти - наиболее распространенный способ взаимодействия.
- При взаимодействии важен порядок операций.
- Иногда возникает неожиданный результат



Резервирование области памяти

Область память должна быть выделена до использования. Функции определены в <linux/ioports.h>

struct resource * request_mem_region(unsigned long start,unsigned long len, char *name);

- возвращает NULL, если ошибка

void release_mem_region(unsigned long start,unsigned long len);

Уточнить назначение регионов памяти можно в **/proc/iomem**

Отображение памяти ввода-вывода в виртуальную память ядра

Для обращения к разделяемой памяти драйверу необходим виртуальный адрес области памяти.

Для реализации этой задачи используется функция ioremap:

```
#include <asm/io.h>;
void *ioremap(unsigned long phys_addr,unsigned long size);
```

(реализация ioremap_nocash аналогична ioremap на большинстве платформ)

Функция возвращает виртуальный адрес, соответствующий заданному физическому или NULL в случае неудачи.

```
void iounmap(void *address);
```

Отличие от обычной памяти

- Запросы чтения и записи могут быть кэшированы
- Компилятор может произвести оптимизацию и использовать регистры вместо памяти
- Компилятор может произвести изменение порядка команд
- CPU может переупорядочить команды

Решение проблем с памятью

- Кэширование I/O портов и памяти в.в. запрещается аппаратно или ядром Linux
- Можно использовать квалификатор volatile
- Для указания компилятору того, что нельзя использовать регистры вместо записи в память.
- Для предотвращения изменения порядка можно использовать барьеры памяти (Memory barriers).

Memory barriers. Логика работы

Барьер вставляется между операциями, которые должны быть видны аппаратуре в определенном порядке.

Пример:

```
writel(dev->registers.addr, io_destination_address);  // Подготовка данных
writel(dev->registers.size, io_size);                  // Подготовка данных
writel(dev->registers.operation, DEV_READ);            // Подготовка данных
wmb( );   // БАРЬЕР НА ЗАПИСЬ – гарантирует, что все команды записи, указанные до него выполнены
writel(dev->registers.control, DEV_GO);
```

Memory barriers

Барьер вставляется между операциями, которые должны быть видны аппаратуре в определенном порядке.

Аппаратно независимые. Влияют только на поведение компилятора. Не влияют не переупорядочивание CPU

```
#include <asm/kernel.h>
void barrier(void);
```

Аппаратно зависимые

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

Безопасны на всех архитектурах.

Еще один пример использования барьеров


```
while (count--) {
outb(*(ptr++), port);
  wmb( );
}
```

Запись необходимо произвести сейчас, без оптимизации.

Функции обмена через разделяемую память в.в.

Для обеспечения портабельности кода лучше использовать специальные функции:

unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);

Чтение или запись нескольких байт:

void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);

Дополнительные утилиты:

void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);

Непосредственное чтение или запись по адресу полученному от ioremap может не работать на некоторых архитектурах.

Как узнать параметры области память в.в

- Современные периферийные устройства для PC не используют фиксированных адресов памяти
- Устройство получает физический адрес неиспользованной области памяти CPU в процессе конфигурирования системы (обычно это обязанность BIOS)
- Расположение и размер области памяти запоминается в специальной non-volatile battery-powered RAM ('configuration memory')

Пример работы с памятью в.в

```
printk ("Get virtual BAR...\t\t");
  device->virtual=ioremap_nocache (device->real,device->size);
  if (device->virtual==0) {printk ("failed.\n"); return -1;} else  printk
("%u...OK.\n",(uint32)device->virtual);

  printk ("Request region BAR...\t\t");
  if (request_mem_region (device->real,device->size,CAN527_NAME)) printk ("OK.\n"); else
{printk ("failed.\n"); return -1;}

  printk ("Set IRQ...");
  status=request_irq(device->irq,device->irq_handler,SA_SHIRQ,CAN527_NAME,device);
  if (status!=0) {printk ("failed.\n"); return -1;} else  printk ("%i...OK.\n",device->irq);
```

Пример


```
#include <linux/pci.h>
static struct pci_device_id rtl8139_pci_tbl[] = {
     {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
     {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0,}
};
MODULE_DEVICE_TABLE (pci, rtl8139_pci_tbl)

static struct pci_driver rtl8139_pci_driver = {
     .name       =  "foo8139",
     .id_table    =  rtl8139_pci_tbl,
     .probe       =  foo_probe,
     .remove     =  foo_remove,
};

static int init_module (void) {
  return pci_register_driver(&rtl8139_pci_driver);
}
```

Пример


```
void* ptr_pci;
int foo_probe(struct pci_dev *dev, const struct pci_device_id *id) // Реализация probe
{
    unsigned long  mem_addr;
    unsigned long  mem_len;
    mem_addr = pci_resource_start(dev,0);

   if (pci_resource_flags (dev,0)&IORESOURCE_MEM)
       printk ("OK.\n");
   else {printk ("failed.\n"); return -1;}
      printk ("Get virtual BAR0...");
   mem_len = pci_resource_len   (dev,0);
   ptr_pci = ioremap(mem_addr, mem_len);

   major = register_chrdev(0,"MyPCI",&fops);

   printk(KERN_INFO "Load driver PCI %d\n",major);
   return 0;
}
```

Пример


```
void foo_remove(struct pci_dev *dev)
{
    unregister_chrdev(major,"MyPCI");
}

void cleanup_module(void)
{
   pci_unregister_driver(&rtl8139_pci_driver);
}
```

Задание

1. Распечатать в шестнадцатеричном виде области памяти сетевой карты.
2. Реализовать чтение MAC адреса через IOCTL.

DMA

- Синхронный и асинхронный DMA
- Обобщенный уровнь работы с DMA
- Когерентное DMA
- Потоковое DMA
- Scater/gathering

Типы адресов

User virtual addresses – обычный адрес использующийся пользоватьскими программами 32 или 64 бита длины. Каждый процесс имеет свое адресное пространство.

Physical addresses

Адрес который используется между процессором и системной памятью.The addresses used between the processor and the system's memory. Physical addresses 32 или 64-bit. 32-ные системы могут иметь большее физическое пространство в ряде случаев.

Bus addresses

Адрес используемы между перефирийной шиной и памятью. Обычно такой же как физический. Некоторые процессоры имею I/O memory management unit (IOMMU) который транслирует адреса между шиной и основной памятью.

Логический адрес ядра

Обычный адрес в пространстве ядра. В большинстве архитектур логический адрес и физический адрес различаются только на константу. В случае наличия большого количества физической памяти логических адресов для доступа к ней может нахватать. kmalloc возвращает логический адрес.

<asm/page.h>

__pa( ) logical ->physical

__va( ) physical -> logical

DMA

DMA механизм, позволяющий перефирийным устройствам передавать дынные в память и читать из памяти без участия центрального процессора.

Синхроный и асинхронный DMA

Синхронный DMA.

1. В ответ на команду read пользовательского процесса драйвер выделяет DMA Драйвер

Ограничения на работу с памятью

- Желательно использовать непрерывную область памяти в физическо адресном пространстве
- Для выделения памяти можно использовать kmalloc (выделяет до 128 KB)
- Для выделения больших объемов памяти можно использовать __get_free_pages (до 8MB).
- Можно использовать блочный I/O и сетевые буферы
- Невозможно использовать память выделенную vmalloc.

Резервирование памяти для DMA

Можно зарезервировать память для DMA заранее

Например, есть 32 MB RAM, и необходимо 2 MB для DMA:

Загружите ядро с параметром **mem=30**

Ядро будет использовать первые 30 MB RAM.

Драйвер сможет занять оставшиеся 2 MB:

```
dmabuf = ioremap (
0x1e00000, /* Start: 30 MB */
0x200000 /* Size: 2 MB */
);
```

Проблемы с синхронизацией памяти

Кеширование памяти может вызывать проблемы при работе с DMA

До того как выполнять DMA обмен с устройством необходимо:

- Убедиться что все записи в буфер DMA произведены;

(Проблема кеширования)

После выполнения DMA:

До того как драйвер будет читать из области DMA необходимо обеспечить запись кешей.

Двунаправленный DMA

Необходимо

Необходима синхронизация кешей до и после выполнения DMA передачи.

Сервисы подсистемы DMA в Linux

Выделение буфера памяти в кеш-когерентной области (синхронные DMA)
Операции прямой синхронизации кеша
Согласование работы DMA с IOMMU

Подсистема PCI и USB содержат дополнительные API для работы с DMA

Для подключения API для работы с PCI необходимо подключить

```
#include <linux/pci.h>
#include <linux/dma-mapping.h>
```

Оботображение (maping) области DMA

- DMA maping — является комбинацией выделения DMA буфера и генерации адреса по которому к этому буферу может обращаться устройство.
- Выделяют два типа мапинга:
- Когерентный
- Потоковый

Когерентный мапинг

- Ядро выделяет подходящий буфер и устанавливает отображения для драйвера устройства.
- Буфер может быть одновременно доступен для CPU и для устройства
- Буфер находится в кеш-когерентной области памяти
- Обычно буфер выделяется на все время существования модуля.
- На некоторых платформах использовани когерентного мепинга затратно

Потоковый мапинг (Streaming mappings)

- Streaming mappings
- Ядро организует мапинг для буфера предоставленного драйвером.
- Используется буфер уже выделенный драйвером.
- Отображение устанавливается для каждой транзакции передачи данных.
- DMA регистры используются более эффективно.
- Возможно оптимизация процесса передачи.
- (Рекомендуемое решение)

Выделение буфера для когерентного отображения.

Ядро выделяет буфер и осуществляет мапинг:

```
include <asm/dmamapping.h>
void *  dma_alloc_coherent(
   struct device *dev, /* device structure */
   size_t size, /* Needed buffer size in bytes */
   dma_addr_t *handle, /* Output: DMA bus address */
   gfp_t gfp /* Standard GFP flags */
);
```

Результат — адрес буфера или NULL.

```
void dma_free_coherent(struct device *dev,
size_t size, void *cpu_addr, dma_addr_t handle);
```

Когерентный мапинг

Выделение области консистентной памяти. Запись в данную память процессором или устройством может быть сразу же считана опонентом без эффектов кеширования.

void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, int flag)

void * pci_alloc_consistent(struct pci_dev *dev, size_t size, dma_addr_t *dma_handle)

Утилита выделяет регион <size> bytes когерентной памяти и возвращает <dma_handle>, который может быть преобразован в unsigned int такой же разрядности, какя шина используется и использоваться как физический адрес региона.

Освобождение:

void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr dma_addr_t dma_handle)

void pci_free_consistent(struct pci_dev *dev, size_t size, void *cpu_addr dma_addr_t dma_handle)
Пример.

Использовани буфера для когерентного мепинга


```
truct rtl8139_private {
        struct pci_dev *pci_dev;  /* PCI device */
        void *mmio_addr; /* memory mapped I/O addr */
        unsigned long regs_len; /* length of I/O or MMI/O region */
        unsigned int tx_flag;
        unsigned int cur_tx;
        unsigned int dirty_tx;
        unsigned char *tx_buf[NUM_TX_DESC];   /* Tx bounce buffers */
        unsigned char *tx_bufs;        /* Tx bounce buffer region. */
        dma_addr_t tx_bufs_dma;
        struct net_device_stats stats;
        unsigned char *rx_ring;
        dma_addr_t rx_ring_dma;
        unsigned int cur_rx;
};
static int rtl8139_open(struct net_device *dev) {
        int retval;        struct rtl8139_private *tp = dev->priv;
        /* get the IRQ second arg is interrupt handler third is flags, 0 means no IRQ sharing          */
        retval = request_irq(dev->irq, rtl8139_interrupt, 0, dev->name, dev);
        if(retval)        return retval;
        /* get memory for Tx buffers memory must be DMAable      */
        tp->tx_bufs = pci_alloc_consistent(tp->pci_dev, TOTAL_TX_BUF_SIZE, &tp->tx_bufs_dma);
         if(!tp->tx_bufs) {    free_irq(dev->irq, dev);    return -ENOMEM;        }
        tp->tx_flag = 0;        rtl8139_init_ring(dev);       rtl8139_hw_start(dev);
        return 0;
}
```

Использовани буфера для когерентного мепинга (2)


```
static void rtl8139_init_ring (struct net_device *dev){
        struct rtl8139_private *tp = dev->priv;
        int i;
        tp->cur_tx = 0;         tp->dirty_tx = 0;

        for (i = 0; i < NUM_TX_DESC; i++)
              tp->tx_buf[i] = &tp->tx_bufs[i * TX_BUF_SIZE];
        return;
}
static void rtl8139_hw_start (struct net_device *dev) {
        struct rtl8139_private *tp = dev->priv;
        void *ioaddr = tp->mmio_addr;
        u32 i;
        rtl8139_chip_reset(ioaddr);
        writeb(CmdTxEnb, ioaddr + CR);          /* Must enable Tx before setting transfer thresholds! */
        /* tx config */
        writel(0x00000600, ioaddr + TCR); /* DMA burst size 1024 */
        /* init Tx buffer DMA addresses */
        for (i = 0; i < NUM_TX_DESC; i++) {
              writel(tp->tx_bufs_dma + (tp->tx_buf[i] - tp->tx_bufs),
                                ioaddr + TSAD0 + (i * 4));
        }
        writew(INT_MASK, ioaddr + IMR); /* Enable all known interrupts by setting the interrupt mask. */
        netif_start_queue (dev);
        return;
}
```

Использовани буфера для когерентного мепинга (3)


```
static int rtl8139_start_xmit(struct sk_buff *skb, struct net_device *dev){
        struct rtl8139_private *tp = dev->priv;
        void *ioaddr = tp->mmio_addr;
        unsigned int entry = tp->cur_tx;
        unsigned int len = skb->len;
#define ETH_MIN_LEN 60  /* minimum Ethernet frame size */
        if (len < TX_BUF_SIZE) {
                if(len < ETH_MIN_LEN)
                        memset(tp->tx_buf[entry], 0, ETH_MIN_LEN);
                skb_copy_and_csum_dev(skb, tp->tx_buf[entry]);
                dev_kfree_skb(skb);
        } else {
                dev_kfree_skb(skb);
                return 0;
        }
        writel(tp->tx_flag | max(len, (unsigned int)ETH_MIN_LEN),
                        ioaddr + TSD0 + (entry * sizeof (u32)));
        entry++;
        tp->cur_tx = entry % NUM_TX_DESC;

        if(tp->cur_tx == tp->dirty_tx) {
                netif_stop_queue(dev);
        }
        return 0;
}
```

Использовани буфера для когерентного мепинга (4)


```
tstatic void rtl8139_chip_reset (void *ioaddr) {
        int i;      /* Soft reset the chip. */
        writeb(CmdReset, ioaddr + CR);
        for (i = 1000; i > 0; i--) {        /* Check that the chip has finished the reset. */
              barrier();
              if ((readb(ioaddr + CR) & CmdReset) == 0)
                    break;
              udelay (10);
        }
        return;
}
```

DMA pool

dma_alloc_coherent обычно выделяет память при помощи __get_free_pages (минимально 1 page).
Для выделения меньших когерентных буферов можно использовать pool

<include linux/dmapool.h>
Создать DMA pool:

```
struct dma_pool *
dma_pool_create (
const char *name, /* Name string */
struct device *dev, /* device structure */
size_t size, /* Size of pool buffers */
size_t align, /* Hardware alignment (bytes) */
size_t allocation /* Address boundaries not to be crossed */
);
```

DMA pool (продолжение)

Выделить буфер из пула

void * dma_pool_alloc (struct dma_pool *pool,gfp_t mem_flags,
dma_addr_t *handle);

Осовободить буфер в пул

void dma_pool_free (struct dma_pool *pool,void *vaddr,dma_addr_t dma);

Удалить пул. Предварительно необходимо удалить все буферы.

void dma_pool_destroy (struct dma_pool *pool);

Потоковые DMA

Отображает фрагмент виртуальной памяти драйвера и возращает физический адрес.
Работает с уже выделленными буферами.

(include linux/dmapool.h)

```
dma_addr_t dma_map_single(struct device *dev, void *cpu_addr, size_t size,
    enum dma_data_direction direction)
dma_addr_t pci_map_single(struct device *dev, void *cpu_addr, size_t size,
    int direction)
```

```
DMA_NONE                = PCI_DMA_NONE          нет неправления(для отладки)
DMA_TO_DEVICE       = PCI_DMA_TODEVICE     из памяти в устйротво
DMA_FROM_DEVICE     = PCI_DMA_FROMDEVICE  из устройства в память
DMA_BIDIRECTIONAL= PCI_DMA_BIDIRECTIONAL     направление неизвестно
```

Пример. 8139cp.c

```
static int cp_rx_poll(struct napi_struct *napi, int budget)
...
573             mapping = dma_map_single(&cp->pdev->dev, new_skb->data, buflen,
574                                      PCI_DMA_FROMDEVICE);
```

DMA unmap

После того, как передача данных завершена мапинг должен быть удален

```
void
dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
 enum dma_data_direction direction)
void
pci_unmap_single(struct pci_dev *hwdev, dma_addr_t dma_addr,
 size_t size, int direction)
```

Отмапливание региона. Параметры аналогичны.

Пример.

```
static int cp_rx_poll(struct napi_struct *napi, int budget)
...
 dma_unmap_single(&cp->pdev->dev, mapping,
563                              buflen, PCI_DMA_FROMDEVICE);
```

Замечания по использования потоковых DMA

- Буфер должен использовать только для передачи в том направлении в котором он выделен;
- После того, как буфер замаплен он принадлежит устройству. До тех пор пока буфер не размаплен (dma_unmap_single) драйвер не должен использовать его содержимое;
- Буфер для записи на устройство не должен быть размаплен до того как заполнен данными;
- Буфер не должен размапливаться до тех пор пока DMA активно;

Захват буфера

Если драйвер должен получить доступ к буферу без размапливание -

void dma_sync_single(struct device *dev, dma_addr_t dma_handle, size_t size, enum dma_data_direction direction)

void pci_dma_sync_single(struct pci_dev *hwdev, dma_addr_t dma_handle, size_t size, int direction)

До того, как устройство сможет получить доступ к буферу владение должно быть передану устройству.

Пример. (drivers/media/video/vino.c)

```
dma_sync_single(NULL,
        fb->desc_table.dma_cpu[VINO_PAGE_RATIO * i],
        PAGE_SIZE, DMA_FROM_DEVICE);
```

skatter/gather отображение

skatter/gather отображение — специальный тип потокового мапинга используется если есть несколько буферов которые которыми необходимо обмениваться.

Причины использования:

- Устройство поддерживает массив указателей и позволяет передавать в одной DMA операции (zero copy);
- Техника работает, если буфера равны размеру страницы (за исключением первого и последнего)

Создание skatter/gather отображение

1. Создать и заполнить массив структур struct scatterlist
2. Задать поля page, length и offset (внутри страницы) для каждой структуры scatterlist (для каждого буфера)
3. Вызвать dma_map_sg
4. Для извлечения шинного адреса из scatterlist

Заголовочные файлы: asm\scatterlist.h

Создание skatter/gather отображение

int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction)

int pci_map_sg(struct pci_dev *hwdev, struct scatterlist *sg, int nents, int direction)

После того, как передача завершена — размапливаем (потоковый мапинг)

void

dma_unmap_sg(struct device *dev, struct scatterlist *sg, int nhwentries,

enum dma_data_direction direction)

void

pci_unmap_sg(struct pci_dev *hwdev, struct scatterlist *sg,

int nents, int direction)

Дополнительно по sg

Переменосимый доступ к длине и адресу

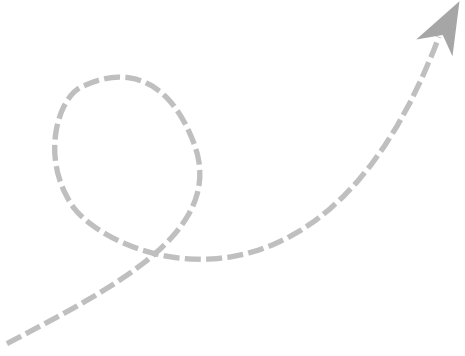
```
sg_dma_address(sg)      = videobuf_to_dma_contig(vb);
      sg_dma_len(sg)        = vb->size;
```

Захват

```
void dma_sync_sg(struct device *dev, struct scatterlist *sg, int nelems,
  enum dma_data_direction direction)
void pci_dma_sync_sg(struct pci_dev *hwdev, struct scatterlist *sg,
      int nelems, int direction)
```

Литература

1. 1. Documentation/DMA-API.txt — документация
(Находится в папке документации к ядру или может быть найдено в интернете, например тут http://devresources.linux-foundation.org/dev/robustmutexes/src/fusyn.hg/Documentation/DMA-API.txt)
2. 2. Linux Cross Reference http://lxr.free-electrons.com/ident?i=dma_set_mask
3. 3. Mohan Lal Jangir. Writing Network Device Drivers for Linux.



Спасибо

Have you any questions?