学习情况

2014-5-8

周永强

六院八隊

Contents

说明

1 学习情况

这两周做的事情不多，大部分时间花在了分布式课程设计Android的开发上面，另外就是继续分析内核的网卡驱动，主要是分析PCI驱动和DMA技术，但是PCI很不熟悉，几种不同的地址空间搞得有点乱，具体还要花一段时间估计才能有比较透彻的理解。

上一次主要是分析网卡驱动中的NAPI技术，随后又画了下流程图，搞清楚具体的数据处理流程。另外着重分析下其中DMA的使用。

千兆网卡中DMA的使用

1 DMA简介

DMA是直接内存访问，数据传送过程中不需要cpu的干预，当数据传输结束后设备会向cpu发送中断信号，通知数据传送完成。驱动程序中使用DMA技术能够显著提高性能。

Intel的千兆网卡采用PCI与系统互联，其相关的DMA操作封装成与PCI相关的函数。

2 DMA分配

若要使用DMA，首先要分配一段连续的空间，供DMA使用，DMA的分配函数如下：

Program 1: DMA分配

```
//功能：分配一致性dma内存，返回这块内存的虚拟地址EA，
//这块内存的物理地址保存在 dma_handle
//dev:  NULL也行
//size: 分配空间的大小
//dma_handle: 用来保存内存的总线地址（物理地址）
void *dma_alloc_cohrent(struct device *dev, size_t size,
                        dma_addr_t *dma_handle, int flag);
```

3 DMA地址映射

内核中对于DMA地址映射主要提供了下面几个函数。

Program 2: DMA地址映射函数

```
// File: linux/include/asmgeneric/pcidmacompact.h
static inline dma_addr_t
pci_map_single(struct pci_dev *hwdev, void *ptr, size_t size, int
direction)
{
  return dma_map_single(hwdev == NULL ? NULL : &hwdev->dev,
  ptr, size, (enum dma_data_direction)direction);
}

// File: /linux/include/asmi386/dmamapping.h
static inline dma_addr_t
dma_map_single(struct device *dev, void *ptr, size_t size,
               enum dma_data_direction direction)
{
  BUG_ON(direction == DMA_NONE);
  flush_write_buffers();
  return virt_to_phys(ptr);
}
static inline void
dma_unmap_single(struct device *dev, dma_addr_t dma_addr,
               size_t size, enum dma_data_direction direction)
{
  BUG_ON(direction == DMA_NONE);
}
```

DAM对应一块内存区域，供外设和CPU共同使用，当然不会同时使用，外设访问需要使用物理地址，因此调用dma_map_single返回ptr的物理地址供驱动程序使用，dma_unmap_single正好相反，执行这个之后，CPU就可以操作DMA缓存区的数据。一般典型的过程是驱动程序通过dma_map_single函数找到DMA共享区的物理地址，将数据写入，写入完毕后然后调用dma_unmap_single，CPU就可以操作读取数据了。

4 NAPI定义

NAPI 使用结构体 napi_struct 描述，如果驱动程序要使用该技术的话，可以在自己的结构体中内嵌实现。在e1000驱动adapter的定义中可以看到napi_struct的存在。

Program 3: e1000.h

```
/*e1000.h*/
struct e1000_adapter {
  /*...*/
```

```
  /*嵌入napi结构*/
  struct napi_struct napi;
  /*...*/
}
```

5 NAPI注册与初始化

napi_struct在定义以后，必须初始化和注册才能够使用。其初始化与注册是在e1000_probe函数中完成，该函数是探测函数，会在。。（PCI routine）。。执行。注册函数为：

netif_napi_add(dev, &napi, poll, weight)

前两个参数是设备和napi结构指针，重要的是第三个参数，第三个参数是函数指针，指向使用napi时要使用的轮询函数，在注册的时候必须指明该参数，然后当使用轮询接收数据时，会回调该函数。在e1000驱动中，轮询函数为e1000_clean。

Program 4: e1000_probe

```
/*e1000_main.c*/
static int e1000_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
  struct net_device *netdev;
  struct e1000_adapter *adapter;
  /*...*/
  SET_NETDEV_DEV(netdev, &pdev->dev);
  pci_set_drvdata(pdev, netdev);
  adapter = netdev_priv(netdev);
  adapter->netdev = netdev;
  adapter->pdev = pdev;
  /*...*/
  //初始化网卡驱动的相关操作，包括open，close等
  netdev->netdev_ops = &e1000_netdev_ops;
  e1000_set_ethtool_ops(netdev);
  /*...*/
  //napi初始化与注册，这里面要指明轮询函数为e1000_clean
  netif_napi_add(netdev, &adapter->napi, e1000_clean, 64);
  /*...*/
}
```

6 调度NAPI

在讲napi的调度之前必须先说明一下中断，中断在e1000_open中注册完成，注册完成后，每当网卡有数据到达，便会产生中断，然后调用中断处理程序，在中断处理程序

中禁用中断，并且调度napi轮询接收数据。napi的调度函数为：
void napi_schedule(struct napi_struct *napi);
或：
if (napi_schedule_prep(napi))
__napi_schedule(napi);
二者的效果一样，只不过后者是先测试当前已经在进行napi调度了。

Program 5: e1000_open

```
/*e1000_open*/
static int e1000_open(struct net_device *netdev)
{
    struct e1000_adapter *adapter = netdev_priv(netdev);
    struct e1000_hw *hw = &adapter->hw;
    /*...*/
    //网卡设置，其中apapter->clean_rx 在其中进行设置
    e1000_configure(adapter);
    /*...*/
    //申请中断
    err = e1000_request_irq(adapter);
    if (err)
        goto err_req_irq;
    /*...*/
    napi_enable(&adapter->napi);
    e1000_irq_enable(adapter);
    /*...*/
}

//中断注册
static int e1000_request_irq(struct e1000_adapter *adapter)
{
    struct net_device *netdev = adapter->netdev;
    //中断处理函数设定
    irq_handler_t handler = e1000_intr;
    int irq_flags = IRQF_SHARED;
    int err;
    //申请中断，中断处理函数为e1000_intr
    err = request_irq(adapter->pdev->irq, handler, irq_flags, netdev->name, netdev);
    /*...*/
}

static irqreturn_t e1000_intr(int irq, void *data)
{
    /*...*/
```

```
    /* disable interrupts, without the synchronize_irq bit */
    ew32(IMC, ~0);
    E1000_WRITE_FLUSH();
    //调度napi进行数据接收
    if (likely(napi_schedule_prep(&adapter->napi))) {
        adapter->total_tx_bytes = 0;
        adapter->total_tx_packets = 0;
        adapter->total_rx_bytes = 0;
        adapter->total_rx_packets = 0;
        __napi_schedule(&adapter->napi);
    }
    /*...*/
}
```

当napi调度以后，后面的工作便是轮询接收数据，轮询函数已经在e1000_probe中设定为e1000_clean。

Program 6: e1000_clean

```
static int e1000_clean(struct napi_struct *napi, int budget)
{
    struct e1000_adapter *adapter = container_of(napi, struct e1000_adapter, napi);
    int tx_clean_complete = 0, work_done = 0;
    //发送数据报文，这是发送流程里面的，暂时可以不关心
    tx_clean_complete = e1000_clean_tx_irq(adapter, &adapter->tx_ring[0]);
    //接收数据，这里clean_rx 为函数指针，指向数据接收函数
    //budget为接收数据量的限制
    adapter->clean_rx(adapter, &adapter->rx_ring[0], &work_done, budget);

    if (!tx_clean_complete)
        work_done = budget;

    /* If budget not fully consumed, exit the polling mode */
    //如果budget没有用完，也即接收的数据量没有超多限度，
    //那么说明轮询可以暂时结束了，重新启用中断，响应后续到来的数据
    if (work_done < budget) {
        if (likely(adapter->itr_setting & 3))
            e1000_set_itr(adapter);
        napi_complete(napi);
        if (!test_bit(__E1000_DOWN, &adapter->flags))
            //重新启用中断
            e1000_irq_enable(adapter);
    }
```

```
    return work_done;
}
```

最后还要看一下clean_rx函数，这个函数之所以设定成指针是为了灵活使用接收函数，不同型号的网卡可能不一样，但是原理都是一样的，该指针的初始化在e1000_open函数中的e1000_configure()中进行设定，而该函数又调用e1000_configure_rx进行设定接收函数，假定接收函数被设为e1000_clean_rx_irq（设成其他函数也一样，原理是相同的），接收函数完成功能见代码。

Program 7: e1000_clean_rx_irq

```
static bool e1000_clean_rx_irq(struct e1000_adapter *adapter,
                               struct e1000_rx_ring *rx_ring,
                               int *work_done, int work_to_do)
{
  /*...*/
  i = rx_ring->next_to_clean;
  rx_desc = E1000_RX_DESC(*rx_ring, i);
  buffer_info = &rx_ring->buffer_info[i];

  while (rx_desc->status & E1000_RXD_STAT_DD) {
    //sk_buff代表报文，下面接收完毕就上交协议栈做下一步的处理
    //数据也就从网卡驱动进入内核协议栈进行解析了
    struct sk_buff *skb;
    u8 status;

    if (*work_done >= work_to_do)
      break;
    (*work_done)++;
   /* read descriptor and rx_buffer_info after status DD */
    rmb();

    skb = buffer_info->skb;
    buffer_info->skb = NULL;

    prefetch(skb->data - NET_IP_ALIGN);
    /*...*/

    length = le16_to_cpu(rx_desc->length);
    /*...*/
    process_skb:
      total_rx_bytes += (length - 4); /* don' t count FCS */
      total_rx_packets++;
```

```
    /*...*/
    skb_put(skb, length);
    /*...*/
    e1000_receive_skb(adapter, status, rx_desc->special, skb);

    next_desc:
      rx_desc->status = 0;
    /*...*/
    }
}
```

7 删除NAPI

消注册在网卡消注册的时候进行调用，调用函数为netif_napi_del()，具体的代码不再分析。

Bibliography

[1] http://www.linuxfoundation.org/node/add/wiki?gids[]=5066.

[2] Linux kernel 2.6.34.