Interfaces e Periféricos 2005/2006

Device drivers

Paulo Pedreiras, V1.0, Fev/2006

Device Drivers: Introdução

- O número de periféricos existentes é muito vasto:
  - Teclado, rato, carta gráfica, carta de rede, carta de som, disco duro, drive de diskettes, scanner, microfone, câmera vídeo, etc.
- Este periféricos apresentam características distintas:
  - **Operações**: leitura, escrita, leitura e escrita
  - **Modo de acesso**: caracter, bloco, etc.
  - **Representação**: ASCII, UNICODE, *Little/Big Endian*, etc.
  - **Largura de banda**: alguns bytes/s a MB/s
  - **Recursos**: *Interrupts*, DMA, portos (I/O, mem mapped), etc.
  - **Implementação**: os membros de uma dada classe de dispositivos apresentam implementações diferentes (e.g. diferentes fabricantes, diferentes modelos) com reflexos profundos na sua operação interna

Device Drivers: Introdução

- Os Sistemas Operativos (SO) **não podem conhecer todos os tipos de dispositivos** passados, actuais e futuros com um nível de detalhe suficiente para realizar o seu controlo a baixo nível!

- Solução:
  - **Separar** a **interacção** com o **hardware** da restante funcionalidade do SO

Device Drivers: Introdução

- **Device driver**

  Um programa que permite a outro programa, tipicamente um Sistema Operativo, interagir com um dado dispositivo de hardware

  - Palavras chave:
    - **Abstracção, uniformização de acesso, independência entre o SO e o hardware**!
  - O **SO** especifica uma **interface** que estabelece como é que o controlo de uma dado tipo de dispositivos é realizado.
    A função do device driver é **traduzir** as **chamadas** realizadas pelo SO em chamadas específicas do dispositivo

Device Drivers: Introdução

- Exemplo: Comunicação série na placa DET188
  - Admitindo que é fornecida uma **livraria** que apresenta a seguinte **interface**:
    - void comDrv_install(unsigned int baud)
    - char comDrv_rxc(void)
    - void comDrv_txc(char ch)
    - void comDrv_release(void)
  - Do ponto de vista da aplicação:
    - É **relevante** qual o **modelo/fabricante** da USART?
    - Se a USART for **substituída** por outra com **arquitectura** interna **distinta**, sendo fornecida uma **livraria** com interface **compatível**, é necessário alterar a aplicação?

Device Drivers: caso de estudo

- Caso de estudo
  - Realização de um device driver para uma USART 8250/16550 em ambiente Linux
  - Porquê Linux?
    - **Código** fonte **disponível**
    - **Utilização** livre e **gratuita**
    - **Documentação** de qualidade disponível
    - **Arquitectura** adequada
  - Porquê USART?
    - Já foi usada nas **aulas práticas** da disciplina
    - **Hardware standard**, disponível em qualquer PC (não portátil ☺)

Device Drivers: driver para USART em Linux

- Módulos do núcleo (Linux Kernel Modules/LKM)
  - Podem ser carregados e descarregados dinamicamente
  - Partilham um único espaço de endereçamento (kernel space)
  - Tipos de módulos
    - ***Device drivers***
    - *Filesystem drivers* (ext2, FAT16/ 32, NFS, …)
    - *System calls*
    - Pilhas protocolares
    - …

Device Drivers: driver para USART em Linux

- Os LKM consistem de pelo menos duas funções básicas:

```
int init_module(void)
{ /*Executada quando o módulo é carregado*/
...
}

void cleanup_module(void)
{ /*Executada quando o módulo é removido*/
...
}
```

Device Drivers: driver para USART em Linux

- Manipulação de LKM
  - **insmod** Inserção de um módulo no kernel
  - **rmmod** Remoção de um módulo
  - **depmod** Determinação das interdependências entre LKMs
  - **ksyms** Símbolos exportados pelos LKMs
  - **lsmod** Lista os LKM carregados
  - **modinfo** Mostra o conteúdo da secção .modinfo de um LKM (ficheiro objecto)
  - **modprobe**
    - Carregamento de módulos com verificação de dependências e carga automática de módulos terceiros. E.g. se o módulo A exporta uma função que o módulo B necessita, o comando *modprobe* carrega automaticamente o módulo A (se não estiver já carregado) quando se solicita a carga do modulo B.

Device Drivers: driver para USART em Linux

The big picture

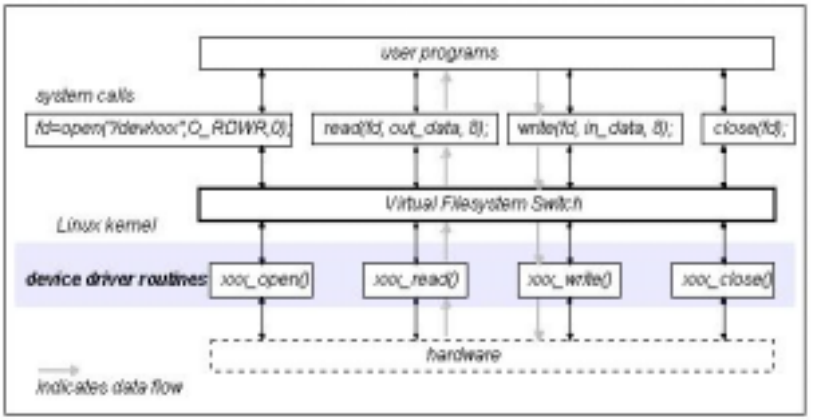
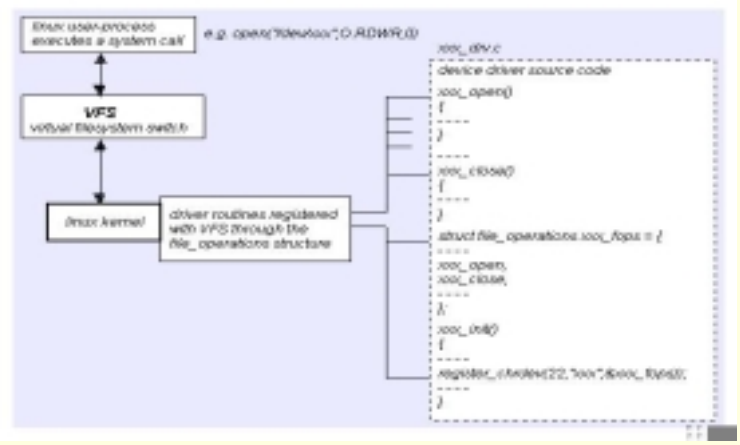


Imagem retirada de "Write a Linux Hardware Device Driver", Andrew O'Shauqhnessy, Unix world

Device Drivers: driver para USART em Linux

Interface entre o device driver e o kernel



Device Drivers: driver para USART em Linux

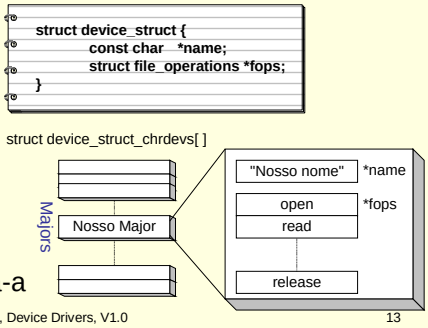
- Estabelecimento da associação entre um device driver e um ficheiro
  - Há "ficheiros especiais" que realizam a interface com os device drivers
    - Tipo: c(har) ou b(lock)
    - Major number: usado pelo VFS para mapear as funções correctas
    - Minor number: identificar sub-dispositivos (e.g. 2 discos IDE partilham o mesmo device driver)
  - Criação de um ficheiro especial
    - mknod NAME TYPE MAJOR MINOR
    - e.g mknod /dev/usart c 240 0

    ls –l /dev/usart

    crw-r—r-- 1 root root 240, 0 Feb 14 18:07 /dev/usart

Device Drivers: driver para USART em Linux

- Estabelecimento da associação entre um device driver e um ficheiro (continuação)
  - Do lado do device driver
    - Preenchimento de uma estrutura (*file ops*) com os endereços das funções implementadas
    - Registo na lista de devices (Major number, nome e fops)
  - Acesso
    - Quando é acedido um ficheiro especial, o kernel obtém o numero e tipo associado, varre a lista de device drivers registados até localizar a entrada correspondente, lê o endereço da função invoca-a



Device Drivers: driver para USART em Linux

- Funções a implementar
  - Device driver
    - **open** registo de utilizadores do módulo (automático)
    - **release** decremento no número de utilizadores do módulo (automático)
    - **write** envio de caracteres pela USART
    - **read** leitura de caracteres vindos da USART
  - Módulo
    - **init** Inicialização, registo do device driver e alocação de recursos (memória p/ buffers, IRQ, …)
    - **cleanup** Libertação de recursos
  - USART
    - Diversas: ISR, envio/recepção de caracteres, configuração

Device Drivers: inicialização (1)

```
/* Called after insmod */
static int usart_init(void)
{
  int result;

  /* Get access to IO region */
  if ( !request_region(usart_base, USART_NPORTS, "usart") ) {
    printk(KERN_INFO "usart: can't get I/O port address 0x%lx\n", usart_base);
    return -ENODEV;
  }

  /* Device registration */
  /* Automatic allocation of device numbers */
  result = alloc_chrdev_region(&usart_devnum, 0,1,"usart");
  if (result < 0) {
    printk(KERN_INFO "usart: can't get device number\n");
    release_region(usart_base, USART_NPORTS);
    return result;
  }
```


Device Drivers: inicialização (2)

```
/* Fill in cdev structure */
  cdev_init(&dev.cdev, &usart_fops);
  dev.cdev.owner = THIS_MODULE;
  result = cdev_add(&dev.cdev, usart_devnum, 1);

  if (result < 0) {
    printk(KERN_INFO "usart: can't add cdev structure\n");
    unregister_chrdev_region(usart_devnum, 1);
    release_region(usart_base, USART_NPORTS);
    return result;
  }

  /* Buffer allocation */
  /* Initialize the input buffer. */
  usart_in_buffer = (unsigned char *)kmalloc(BUFF_SIZE,GFP_KERNEL);
  usart_in_head = usart_in_tail = usart_in_buffer;

  /* Initialize the output buffer. */
  usart_out_buffer = (unsigned char *)kmalloc(BUFF_SIZE,GFP_KERNEL);
  usart_out_head = usart_out_tail = usart_out_buffer;

  if( (usart_in_buffer == NULL) || (usart_out_buffer == NULL))
    usart_cleanup();
```

Device Drivers: inicialização (3)

```
/* IRQ set-up */
result = request_irq(usart_irq, usart_interrupt, 0, "usart", NULL);
if (result) {
  printk(KERN_INFO "usart: can't get assigned irq %i\n", usart_irq);
  usart_cleanup ();
  return result;
}

/* Set-up the USART */
usart_setup(usart_baud, usart_dl, usart_par, usart_sb);

return 0;
}
```


Device Drivers: remoção do módulo

```
/* Called after rmmod (Free allocated resources) */
static void usart_cleanup(void)
{
  /* Turn off interrupts */
  outb(0, usart_base + USART_IER);
  free_irq(usart_irq, NULL);

  /*  Free cdev and device numbers */
  cdev_del(&dev.cdev);
  unregister_chrdev_region(usart_devnum, 1);

  /* Release IO region */
  release_region(usart_base, USART_NPORTS);

  /* Free buffer memory */
  if(usart_in_buffer)
    kfree(usart_in_buffer);

  if(usart_out_buffer)
    kfree(usart_out_buffer);

}
```

Device Drivers: funções de interface (1)

```
/* ------------------------------------------------------------------------------------------
 * Device Driver Interface Primitives (open, close, read, write)
 * ------------------------------------------------------------------------------------------ */
static int usart_open(struct inode *inode, struct file *filp)
{
  struct usart_dev *dev;
  /* Place in filp.private_data a pointer to the usart_dev, that will */
  /* be directly accessed by the other methods */
  dev=container_of(inode->i_cdev, struct usart_dev, cdev);
  filp->private_data = dev;

  return 0;
}

static int usart_release(struct inode *inode, struct file *filp)
{
  MSG("usart_release called\n");
  return 0;
}
```


Device Drivers: funções de interface (2)

```
static ssize_t usart_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
  int bufncar, bufncar_end, bufncar_beg;

  disable_irq(usart_irq); /* get exclusive access to buffer */

  bufncar = (int)(usart_in_tail - usart_in_head); /*compute # of data bytes in input buffer*/

  if (bufncar >= 0) {
    enable_irq(usart_irq);

    if (bufncar < count) /* Adjust the number of bytes returned */
      count = bufncar;

    if (copy_to_user(buf, (char *)usart_in_head, count)) /* Copy data to user space */
      return -EFAULT;

    usart_in_head+=count; /* Adjust buffer head pointer */
  }
  else { /* Wrapped */
   ..... }
  return count;
}
```

Device Drivers: funções de interface (3)

```
static ssize_t usart_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
  int buffree, buffree_beg, buffree_end;

  if(usart_out_tail < usart_out_head) { /* Tail before Head (info in buffer wraps) */
    buffree = usart_out_head - usart_out_tail - 1; /* Compute available space */

    enable_irq(usart_irq); /* release exclusive access to buffer */

    if(buffree == 0) /* No room for any data (in non-blocking mode)*/
      return -ENOSPC;

    if(count > buffree)
      count = buffree; /* If not enough free space, truncate bytes to write */
```


Device Drivers: funções de interface (4)

```
    (continuação)

    if (copy_from_user((char *) usart_out_tail, buf, count))
      return -EFAULT;

    usart_out_tail += count; /* Update tail pointer (cannot wrap, since Head after) */
  }
  else { /* Head before tail. */
    ..... }

  if(count != 0) {
    usart_thbe_int_enable(); /* Enable generation of Empty Tx holding buffer interrupts */
    usart_tx_data(); /* Try to put one char in tx buffer. Other ones txmited via tx interrupts */
  }

  return count;
}
```

Device Drivers: funções associadas à USART (1)

```
/* USART interrupt handler (just TX/RX ints handled) */
static irqreturn_t usart_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  unsigned char usart_status;

  usart_status = (unsigned char)inb(usart_base + USART_IIR); /* Get status */

  /* Data received INT? */
  if (usart_status & USART_IIR_RXDATA) {
    usart_rx_data();
  }

  /* TX buffer empty INT? */
  if (usart_status & USART_IIR_THREMPT) {
    usart_tx_data();
  }

  return IRQ_HANDLED;
}
```


Device Drivers: funções associadas à USART (2)

```
/* Upload data from the USART RX buffer to the input buffer */
static void usart_rx_data(void)
{
  unsigned char data, usart_status;
  volatile unsigned char *bufptr;

  do {

    usart_status = inb(usart_base + USART_LSR); /* Get USART status */

    if(usart_status & USART_LSR_DR) { /* Data ready? */
      data = inb(usart_base + USART_RXBUF); /* Read USART data */

      bufptr= [ "Next Position in Circular Buffer"]

      if ( bufptr != usart_in_head) { /* Input buffer full? */
        *usart_in_tail = data;
        usart_in_tail = bufptr;
      }
      else { /* Buffer Full! Drop data */ …}
    }
  } while(usart_status & USART_LSR_DR); /* Do while data in USART RX buffer */

  return;
}
```

Device Drivers: funções associadas à USART (3)

```
/* Pick data from output buffer and dowload it to USART TX buffer */
static void usart_tx_data(void)
{  unsigned char usart_status;

   disable_irq(usart_irq); /* Get exclusive access to buffer. (Note: function may be called
                              both from the ISR and from the usart_write() file op */

   usart_status = (unsigned char)inb(usart_base + USART_LSR); /* Get USART status */
   if(usart_status & USART_LSR_ETHR ) { /* If USART TX Buffer empty */

     if(usart_out_head != usart_out_tail) { /* Output buffer not empty */
       outb(*usart_out_head, usart_base + USART_TXBUF);
       [ update head pointer ]
      }
      else /* Output Buffer Empty! */
        usart_thbe_int_disable(); /* Disab. generation of Empty Tx holding buffer interrupts */
      }
    else { /*TX Buffer not avail. May happen when function is called from usart_write()) */ … }
  }

   enable_irq(usart_irq); /* Release exclusive access to buffer */

   return;
}
```


Device Drivers: comentários

- Device driver **testado** para o Linux Kernel 2.6.15
- O **driver nativo** do Linux **não pode estar carregado**
  - Caso o driver nativo não esteja definido como módulo pode ser necessário recompilar o kernel
    - Verificar "Device Drivers / Character Devices / Serial Drivers / 8250/16550 and compatibles“
- O device driver apresentado é **minimalista**, tendo sido escrito apenas para ilustrar o conceito. Há **diversos componentes** em falta (ficam como desafio), e.g.
  - Configuração da USART não parametrizada
  - Implementação de filas de espera
  - Partição do *Interrupt handler* (*bottom-half processing*)
  - Implementação da função IOCTL para configuração da USART

Device Drivers: bibliografia

- **Linux**
  - Linux Device Drivers, 3rd Edition, O'Reilly, J. Corbet, A. Rubini, G. Kroah-Hartman
  - Beginning Linux Programming, 3rd Edition, Wrox, N.Matthew, R. Stones
  - **Código fonte do driver nativo**
- **USART**
  - http://www.beyondlogic.org/serial/serial.htm (Web, 16/Feb/2006)