Sistemas Operacionais

Gestão de entrada/saída - software

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Agosto de 2020

Conteúdo

1 Software de Entrada/saída

2 Drivers

3 Estratégias de interação
- Entrada/saída por programa
- Entrada/saída por eventos
- Entrada/saída por DMA

4 Tratamento de interrupções

Software de entrada/saída

Software que interage com os dispositivos:

- *Drivers* (ou pilotos)
- Abstrações de baixo nível (sockets, blocos, etc)

Grande diversidade de dispositivos:

- Muitos dispositivos distintos = muitos *drivers*
- 60% do código-fonte do núcleo Linux são *drivers*

Estrutura geral do núcleo

Drivers:

- Interagem com os dispositivos
- Acessam portas de E/S e tratam interrupções

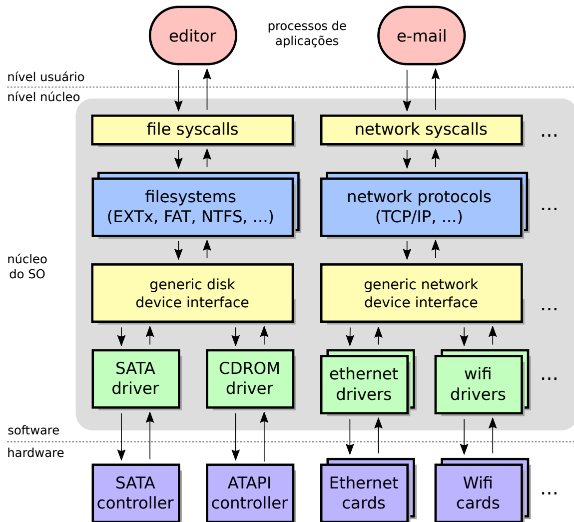
Dispositivos genéricos:

- Visão genérica de dispositivos similares (discos)
  - Discos: vetores de blocos de dados
  - Interfaces de rede (especificação NDIS)

Abstrações: Sistemas de arquivos, protocolos de rede

Chamadas de sistema: interface oferecida aos processos

Estrutura geral



Classes de dispositivos genéricos

Dispositivos orientados a caracteres

- Transferências de dados byte a byte, sequencial
- Ex.: dispositivos USB, *mouse*, teclado, modems

Dispositivos orientados a blocos

- Transferências feitas em blocos de bytes
- Blocos possuem endereços definidos
- Ex.: discos, fitas e dispositivos de armazenamento

Classes de dispositivos genéricos

Dispositivos de rede

- Blocos de dados de tamanho variável (mensagens)
- Envios de blocos de forma sequencial
- NDIS – *Network Device Interface Specification*
- Ex.: Interfaces *Ethernet*, *Bluetooth*

Dispositivos gráficos

- Renderização de texto e gráficos em uma tela
- Usam uma área de RAM compartilhada (*framebuffer*)
- Acesso por bibliotecas específicas (*DirectX*, DRI)

Driver

Componente de baixo nível do SO:

- Executa geralmente em modo núcleo
- Interage com o hardware do dispositivo
- Um driver para cada tipo de dispositivo

Funcionalidades:

- Funções de entrada/saída
- Funções de gerência
- Funções de tratamento de eventos

Estrutura de um driver

Funções de **entrada/saída**:

- Transferência de dados dispositivo $\rightleftharpoons$ núcleo
- Caracteres, blocos, mensagens

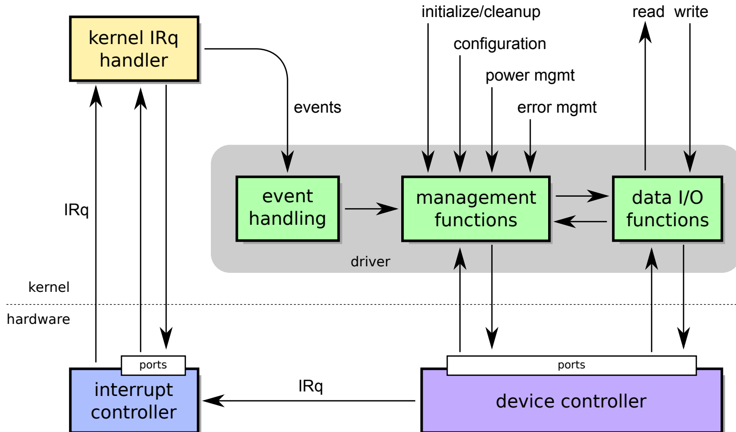
Funções de **gerência**:

- Inicialização e configuração do dispositivo
- Inicialização e configuração do *driver*
  - Syscalls: `ioctl()`, `DeviceIoControl()`

Funções de **tratamento de eventos**:

- Interrupções geradas pelo dispositivo

Estrutura de um driver



UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

Estratégias de interação

Como o núcleo interage com os dispositivos?

Através dos *drivers*!

Os *drivers* implementam **estratégias de interação**:

- Por programa (ou por *polling*)
- Por eventos (ou por interrupções)
- Por acesso direto à memória (DMA)

Entrada/saída por programa

Estratégia de entrada/saída mais simples.

- Também chamada **varredura** ou **polling**.

O *driver* pede a operação e aguarda sua conclusão:

1. Espera o dispositivo estar pronto (*status*)
2. Escreve dado na porta de saída (*data out*)
3. Escreve comando (*control*)
4. Espera dispositivo concluir a operação (*status*)

Exemplo: interface paralela simples

I/O port 378_H : P_0 (*data port*)

- 8 bits de dados

I/O port 379_H : P_1 (*status port*)

6. $\overline{\texttt{ack}}$: o dado em P_0 foi recebido (*acknowledge*)
7. `busy`: o controlador está ocupado

I/O port $37A_H$: P_2 (*control port*)

0. $\overline{\texttt{strobe}}$: o *driver* diz que há um dado em P_0

Entrada/saída por programa

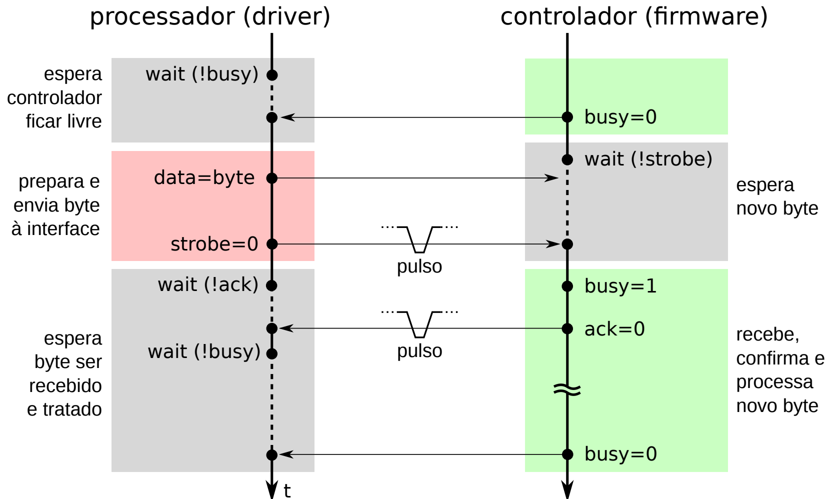
```
1  // portas da interface paralela LPT1
2  #define DATA   0x0378      # porta de dados (P0)
3  #define STAT   0x0379      # porta de status (P1)
4  #define CTRL   0x037A      # porta de controle (P2)
5
6  // bits de controle e status das portas
7  #define ACK   6            # bit 6 da porta de status
8  #define BUSY  7            # bit 7 da porta de status
9  #define STROBE 0           # bit 0 da porta de controle
10
11 // operações em bits individuais de bytes (n: 0...7)
12 #define BIT_SET(a,n) (a | 1 << n)  // n-esimo bit de a = 1
13 #define BIT_CLR(a,n) (a & ~(1 << n)) // n-esimo bit de a = 0
14 #define BIT_TEST(a,n) (a & 1 << n) // testa n-esimo bit de a
```

Acesso às portas: `val = in(port)` e `out(port, val)`

Entrada/saída por programa (cont.)

```
1  void send_byte (char c)
2  {
3    // espera o controlador ficar livre, testando a porta de status
4    while (BIT_TEST (in(STAT), BUSY)) ;
5
6    // escreve o byte "c" a enviar na porta de dados
7    out (DATA, c) ;
8
9    // gera pulso de 1 us em 0 no bit STROBE da porta de controle,
10   // para indicar ao controlador que há um novo dado
11   out (CTRL, BIT_CLR (in(CTRL), STROBE)) ; // bit STROBE = 0
12   usleep (1) ;                             // aguarda 1 us
13   out (CTRL, BIT_SET (in(CTRL), STROBE)) ; // bit STROBE = 1
14
15   // espera controlador receber o dado (pulso em 0 no bit ACK)
16   while (BIT_TEST (in(STAT), ACK)) ;
17
18   // espera o controlador concluir a operação solicitada
19   while (BIT_TEST (in(STAT), BUSY)) ;
20 }
```

Entrada/saída por programa



Entrada/saída por eventos

Estratégia básica:

1. Requisitar a operação desejada
2. Suspender o fluxo de execução (tarefa atual)
3. O dispositivo gera uma IRq ao concluir
4. O *driver* retoma a operação de E/S

A operação de E/S pelo *driver* é dividida em duas etapas:

1. Uma função de E/S inicia a operação
2. Uma função de tratamento de evento a continua

Entrada/saída por eventos

Parte 1: inicia a transferência de dados

```
// saída de dados por evento: solicitação de operação
void send_byte (char c)
{
  // espera o controlador ficar livre, testando a porta de status
  while (BIT_TEST (in(STAT), BUSY)) ;

  // escreve o byte "c" a enviar na porta de dados
  out (DATA, c) ;

  // gera pulso de 1 us em 0 no bit STROBE da porta de controle,
  // para indicar ao controlador que há um novo dado
  out (CTRL, BIT_CLR (in(CTRL), STROBE)) ; // bit STROBE = 0
  usleep (1) ;                               // aguarda 1 us
  out (CTRL, BIT_SET (in(CTRL), STROBE)) ; // bit STROBE = 1
```

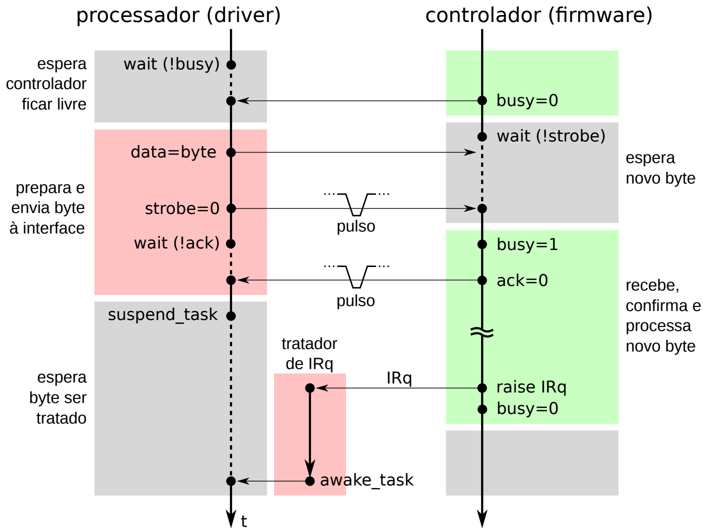
Entrada/saída por eventos

```
15
16    // espera controlador receber o dado (pulso em 0 no bit ACK)
17    while (BIT_TEST (in(STAT), ACK)) ;
18
19    // suspende a execução, liberando o processador para outras tarefas
20    // enquanto o controlador está ocupado processando o dado recebido.
21    suspend_task () ;
22  }
```

Parte 2: trata as interrupções

```
1  // saída de dados por evento: tratamento da interrupção
2  void event_handle ()
3  {
4    // o controlador concluiu a operação, acorda a tarefa solicitante.
5    awake_task () ;
6  }
```

Entrada/saída por eventos



Envio de buffer de dados por eventos

```
7   // envia um buffer de dados de tamanho buffsize
8   void send_buffer (char *buffer)
9   {
10    // envia o primeiro byte do buffer
11    pos = 0 ;
12    send_byte (buffer[pos]) ;
13
14    // suspende a tarefa, liberando a CPU para outras tarefas
15    suspend_task () ;
16  }
17
18  // trata interrupções da interface paralela
19  void event_handle ()
20  {
21    pos++ ;                              // avança posição no buffer
22    if (pos < bufsize)                   // o buffer terminou?
23      send_byte (buffer[pos]) ;          // não, envia o próximo byte
24    else
25      awake_task () ;                    // sim, acorda a tarefa
26  }
```

Acesso direto à memória

Transferência direta entre dispositivo e RAM

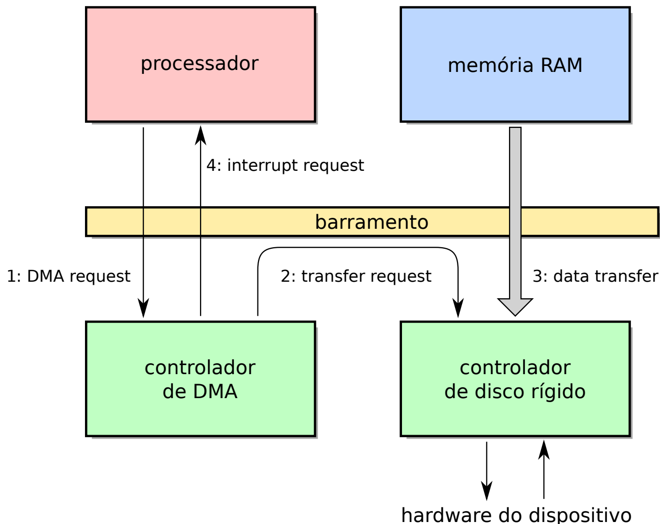
Os dados não precisam passar pela CPU (desempenho)

Muito usado para transferir grandes volumes de dados

Passos:

1. A CPU programa o controlador DMA com os parâmetros (endereços e tamanho da transferência)
2. O controlador de DMA interage com o controlador do disco para transferir os dados da RAM
3. O controlador do disco recebe os dados da RAM
4. No final, o controlador de DMA interrompe a CPU

Acesso direto à memória



Acesso direto à memória

```
// requisição da operação de saída através de DMA
void dma_output ()
{
  // solicita uma operação DMA, informando os dados da transferência
  // através da estrutura "dma_data"
  request_dma (dma_data) ;

  // suspende o processo solicitante, liberando o processador
  suspend_task () ;
}

// rotina de tratamento da interrupção do controlador de DMA
void interrupt_handle ()
{
  // informa o controlador de interrupções que a IRq foi tratada
  acknowledge_irq () ;

  // saída terminou, acordar o processo solicitante
  awake_task (...) ;
}
```

Tratamento de interrupções

Interrupções são tratadas por *handlers* (tratadores):

- Compõem a estrutura dos *drivers*
- Operam usualmente com interrupções inibidas
- Devem ser muito rápidos, portanto simples

A maioria dos SOs trata interrupções em **dois níveis**:

- FLIH - *First-Level Interrupt Handler*
- SLIH - *Second-Level Interrupt Handler*

Tratamento de interrupções

- FLIH - *First-Level Interrupt Handler*
  - Tratador primário (rápido)
  - Recebe a IRq e a reconhece junto ao PIC
  - Registra dados da IRq em uma fila de eventos
  - Outros nomes: *Hard, fast, top-half IH*
- SLIH - *Second-Level Interrupt Handler*
  - Tratador secundário (lento)
  - Trata os eventos da fila de eventos
  - *Pool de threads* de núcleo escalonadas
  - Outros nomes: *Soft, slow, bottom-half IH*

Tratadores de interrupções FLIH e SLIH