Sistemas Operacionais

Gestão de entrada/saída - software

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Agosto de 2020

Conteúdo

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

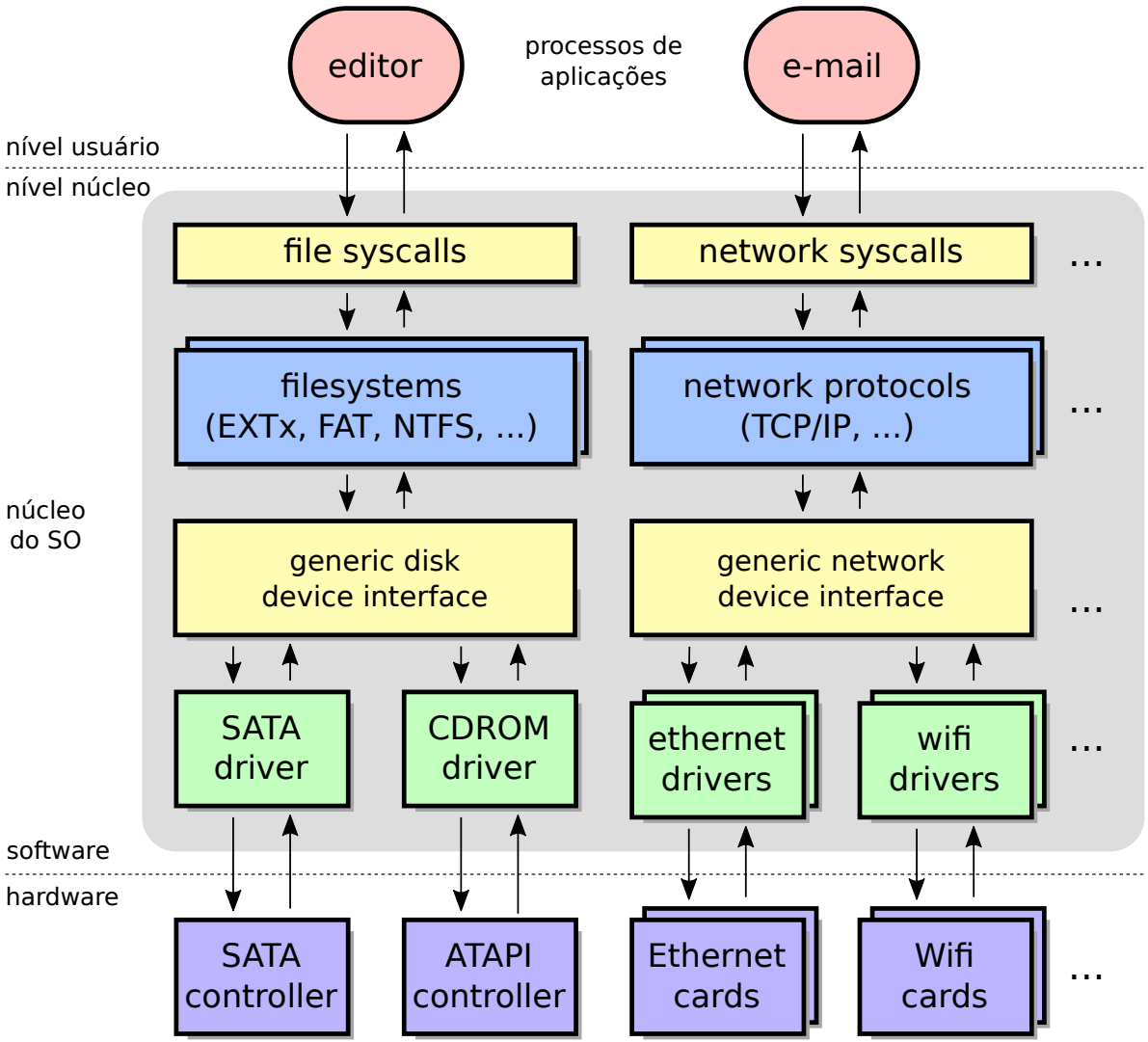
Dispositivos genéricos:

- 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

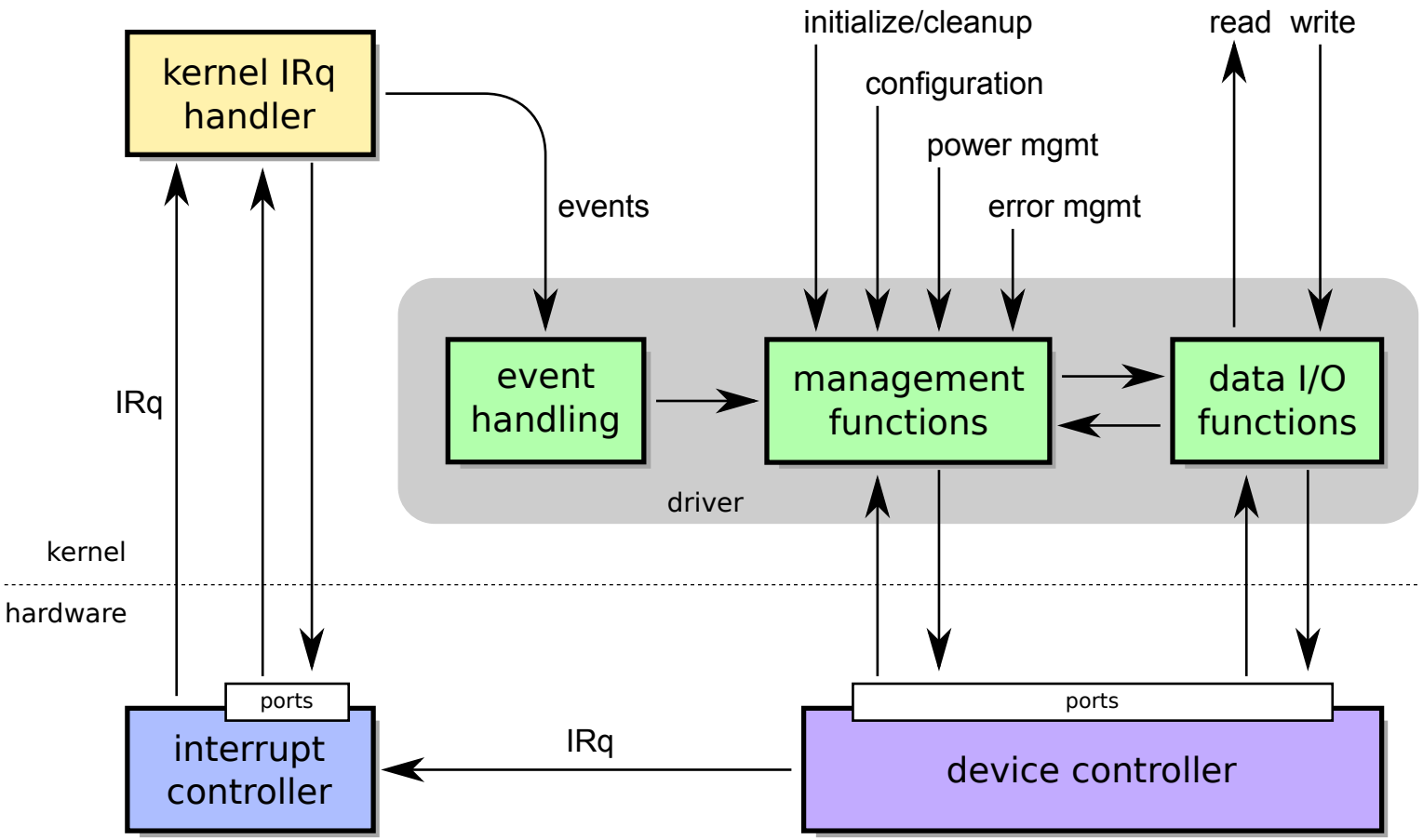
Funções de **gerência**:

- 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



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. $\overline{\texttt{busy}}$: 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

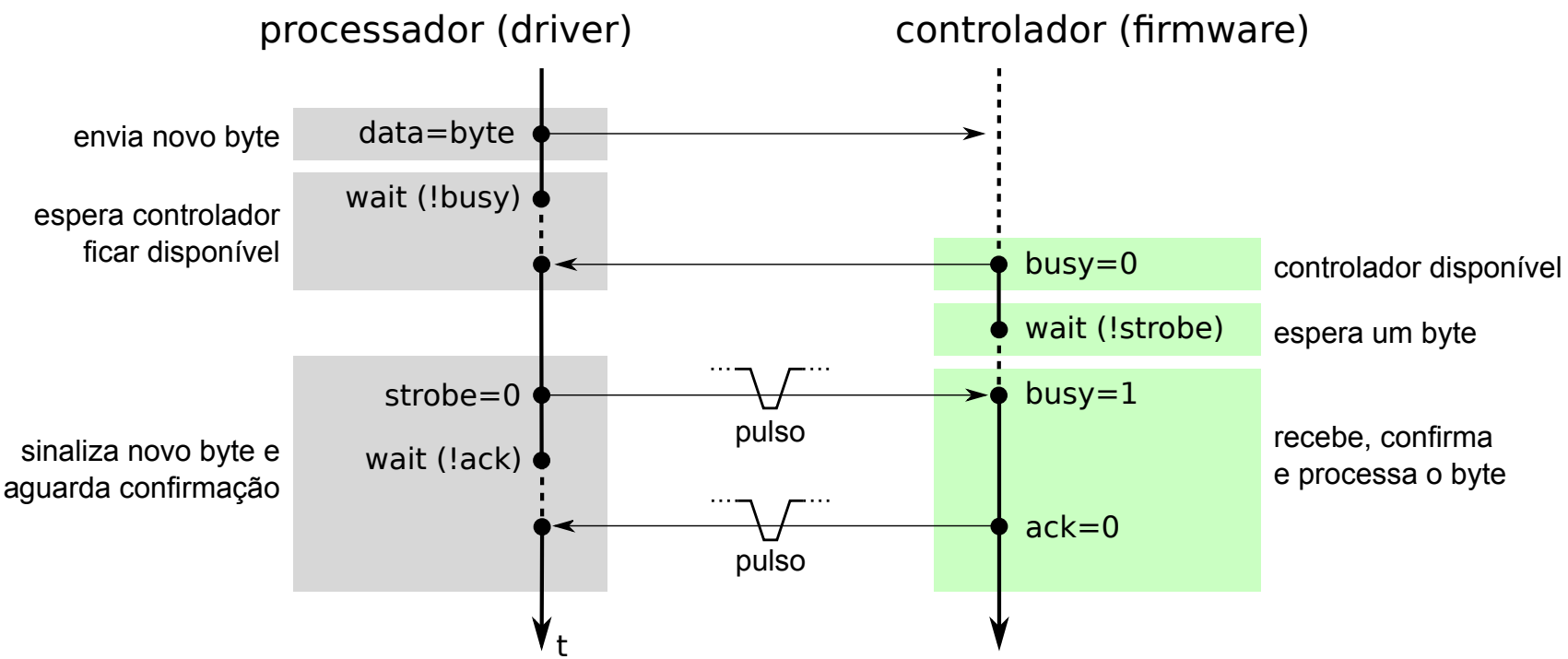
Entrada/saída por programa

```
1  // interface paralela LPT1
2  #define LPT1  0x0378        // endereço da interface paralela LPT1
3  #define DATA  (LPT1 + 0)    // porta de dados
4  #define STAT  (LPT1 + 1)    // porta de status
5  #define CTRL  (LPT1 + 2)    // porta de controle
6
7  // bits de controle e status
8  #define ACK   6             // bit 6 (porta de status)
9  #define BUSY  7             // bit 7 (porta de status)
10 #define STROBE 0            // bit 0 (porta de controle)
11
12 // operações em bits: seta/limpa/testa o N-ésimo bit de A
13 #define BIT_SET(A,N) (A | (1 << N))      // seta (= 1)
14 #define BIT_CLEAR(A,N) (A & ~(1 << N))   // limpa (= 0)
15 #define BIT_TEST(A,N) (A & (1 << N))     // testa
16
17 // acesso à s portas de E/S (instruções IN e OUT)
18 unsigned char inb (unsigned short int port) ;
19 void outb (unsigned char value, unsigned short int port) ;
```

Entrada/saída por programa

```
1  void send_char_polling (unsigned char c)
2  {
3    // escreve o byte "c" a enviar na porta de dados
4    outb (c, DATA) ;
5
6    // espera a interface ficar livre (bit BUSY = 1)
7    while (! BIT_TEST (inb (STAT), BUSY)) ;
8
9    // gera um pulso 0 no bit STROBE (bit STROBE = 0) durante 1 us,
10   // para indicar à interface que há um novo dado em DATA
11   aux = inb (CTRL) ;
12   outb (BIT_CLEAR (aux, STROBE), CTRL) ; // bit STROBE = 0
13   usleep (1) ;                           // aguarda 1 us
14   outb (BIT_SET (aux, STROBE), CTRL) ;   // bit STROBE = 1
15
16   // espera a interface receber o dado (pulso em 0 no bit ACK)
17   while (BIT_TEST (inb (STAT), ACK)) ;
18 }
```

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

```
1  // saída de dados por eventos (simplificado, sem controle de erros)
2  void send_char_irq (unsigned char c)
3  {
4    // escreve o byte "c" a enviar na porta de dados
5    outb (c, DATA) ;
6
7    // espera a interface ficar livre (bit BUSY = 1)
8    while (! BIT_TEST (inb (STAT), BUSY)) ;
9
10   // gera um pulso 0 no bit STROBE (bit STROBE = 0) durante 1 us,
11   // para indicar à interface que há um novo dado em DATA
12   aux = inb (CTRL) ;
13   outb (BIT_CLEAR (aux, STROBE), CTRL) ;  // bit STROBE = 0
14   usleep (1) ;                            // aguarda 1 us
15   outb (BIT_SET (aux, STROBE), CTRL) ;    // bit STROBE = 1
16
17   // continua ...
```

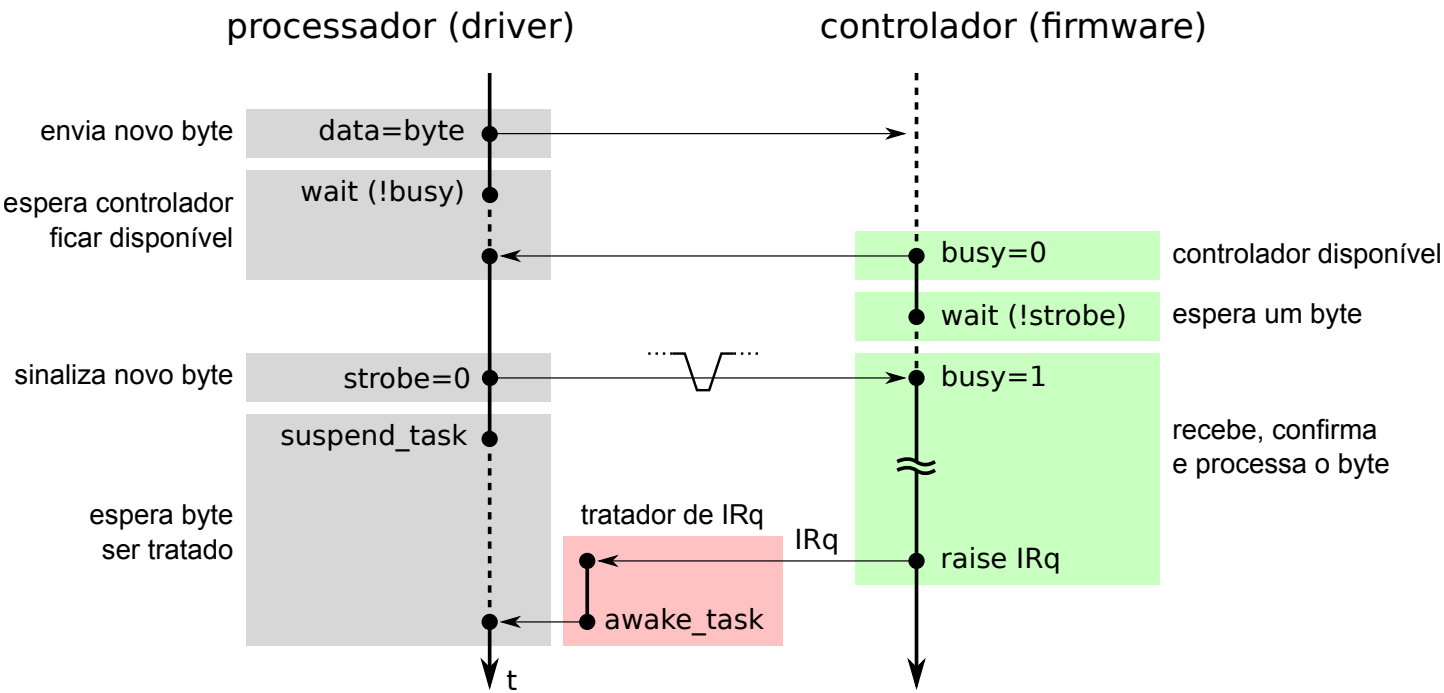
Entrada/saída por eventos

```
18    // não espera o ACK da interface
19
20    // suspende a execução da tarefa atual e libera a CPU
21    // para outras tarefas enquanto a interface está ocupada
22    // processando o último dado recebido.
23    task_suspend (...) ;
24  }
```

Parte 2: trata as interrupções

```
1   // tratamento da interrupção gerada pela interface paralela
2   void irq_handle ()
3   {
4     // a interface concluiu a operação e sinalizou com uma interrupção;
5     // acordar a tarefa que solicitou a última operação
6     task_awake (...) ;
7   }
```

Entrada/saída por eventos



Envio de buffer de dados por eventos

```
1  // buffer de bytes a enviar
2  char *buffer ;
3  int buff_size, pos ;
4
5  // envia um buffer de dados de tamanho buff_size (simplificado)
6  void send_buffer (char *buffer)
7  {
8    // envia o primeiro byte do buffer
9    pos = 0 ;
10   send_char_irq (buffer[pos]) ;
11 }
12
13 // tratamento da interrupção gerada pela interface paralela
14 void irq_handle ()
15 {
16   pos++ ;                          // avança posição no buffer
17   if (pos >= buff_size)            // o buffer terminou?
18     task_awake (...) ;             // sim, acorda a tarefa
19   else
20     send_char_irq (buffer[pos]) ;  // não, envia o próximo byte
21 }
```

Acesso direto à memória

Transferência direta entre dispositivo e RAM

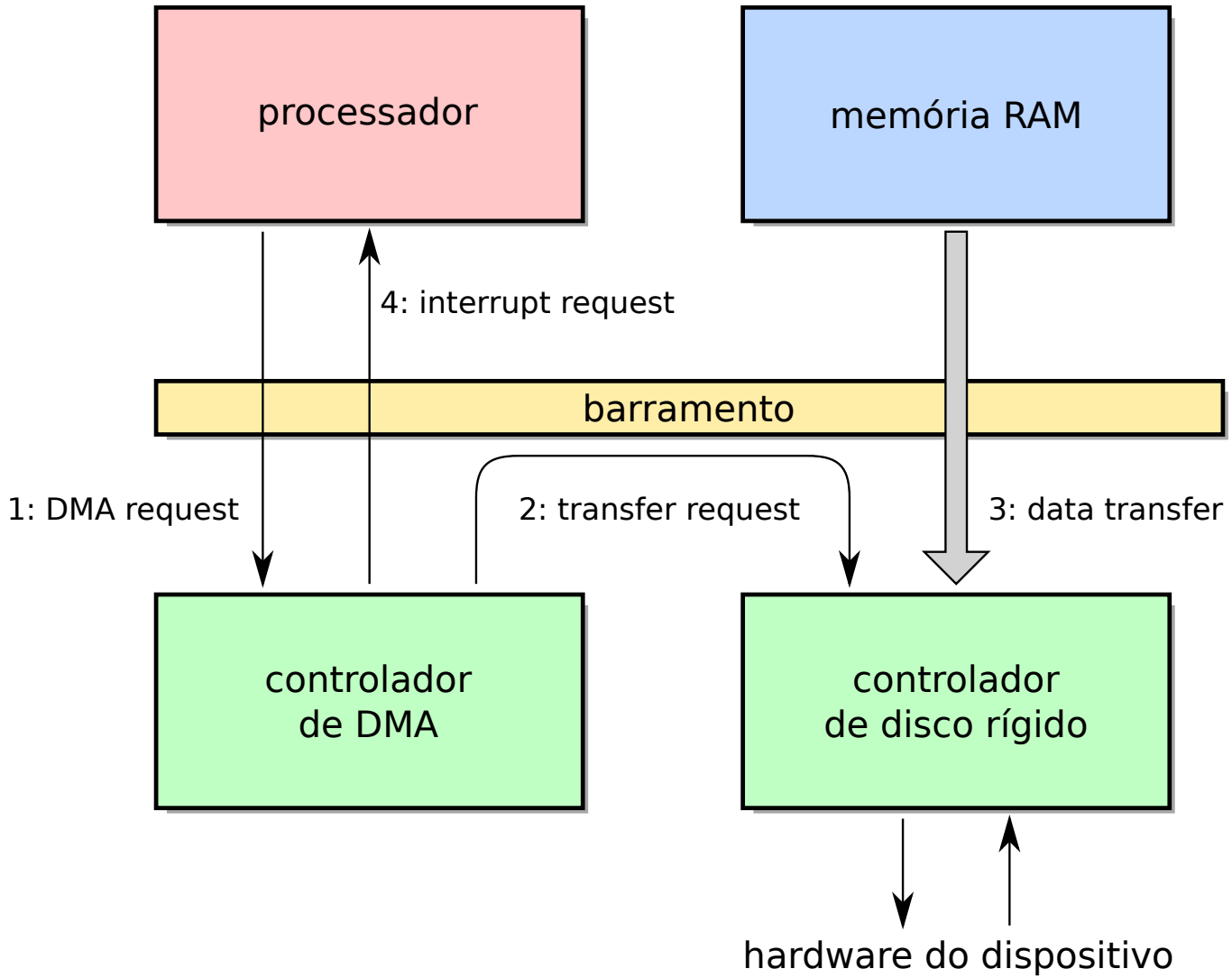
Os dados não precisam passar pela CPU (desempenho)

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

```
1  // requisição da operação de saída através de DMA (simplificado)
2  void dma_output (...)
3  {
4    // solicita uma operação DMA, informando os dados da transferência
5    // através da estrutura "dma_data"
6    dma_request (dma_data) ;
7
8    // suspende a tarefa solicitante, liberando o processador
9    task_suspend (...) ;
10 }
11
12 // tratamento da interrupção gerada controlador de DMA
13 void irq_handle ()
14 {
15   // informa o controlador de interrupções que a IRq foi tratada
16   irq_acknowledge () ;
17
18   // saída terminou, acordar a tarefa solicitante
19   task_awake (...) ;
20 }
```

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