Capítulo 20

Software de entrada/saída

20.1 Introdução

O sistema operacional é responsável por oferecer acesso aos dispositivos de entrada/saída às aplicações e, em consequência, aos usuários do sistema. Prover acesso eficiente, rápido e confiável a um conjunto de periféricos com características diversas de comportamento, velocidade de transferência, volume de dados produzidos/consumidos e diferentes interfaces de hardware é um enorme desafio. Além disso, como cada dispositivo define sua própria interface e modo de operação, o núcleo do sistema operacional deve implementar o código necessário para interagir com milhares de tipos de dispositivos distintos. Como exemplo, cerca de 70% das 20 milhões de linhas de código do núcleo Linux na versão 4.3 pertencem a código de *drivers* de dispositivos de entrada/saída.

Este capítulo apresenta uma visão geral da organização e do funcionamento dos componentes do sistema operacional responsáveis por interagir com os dispositivos de hardware de entrada/saída, e como o acesso a esses dispositivos é provido às aplicações em modo usuário.

20.2 Arquitetura de software de entrada/saída

Para simplificar o uso e a gerência dos dispositivos de entrada/saída, o código do sistema operacional é estruturado em camadas, que levam da interação direta com o hardware (como o acesso às portas de entrada/saída, interrupções e operações de DMA) às interfaces de acesso abstratas e genéricas oferecidas às aplicações, como arquivos e *sockets* de rede.

Uma visão conceitual dessa estrutura em camadas pode ser vista na Figura 20.1. Nessa figura, a camada inferior corresponde aos dispositivos físicos propriamente ditos, como discos rígidos, teclados, etc. A camada logo acima corresponde aos controladores de dispositivos (discos SATA, USB, etc.) e aos controladores de DMA e de interrupções implementados no *chipset* do computador.

A primeira camada de software no núcleo do sistema operacional corresponde aos *drivers* de dispositivos, ou simplesmente *drivers*[1], que são os componentes de código que interagem diretamente com cada controlador, para realizar as operações

[1]Em Portugal se utiliza o termo "piloto", que eu aprecio, mas optei por não utilizá-lo neste texto porque o termo em inglês é omnipresente no Brasil.

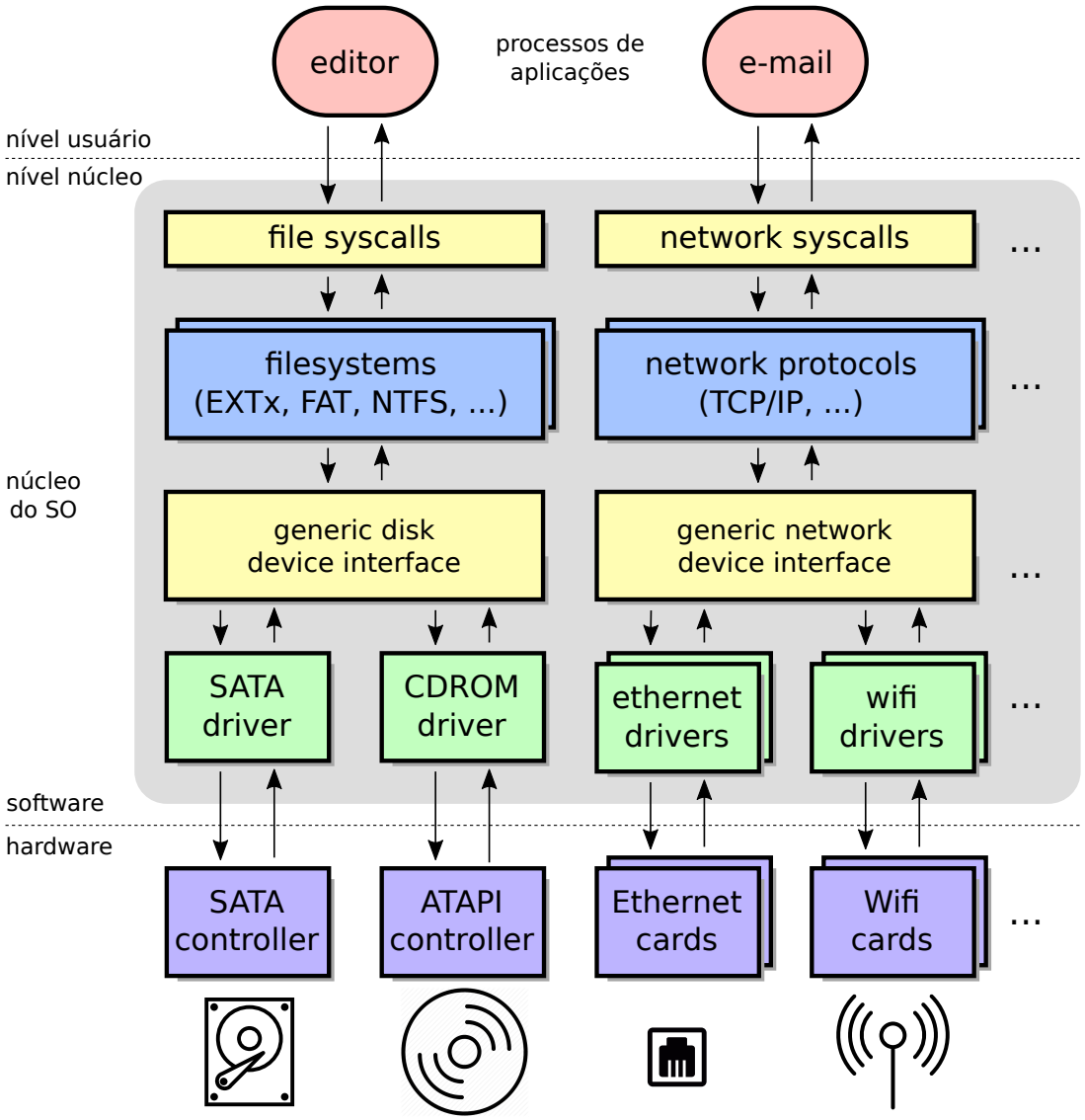

Figura 20.1: Estrutura em camadas do software de entrada/saída.

de entrada/saída, receber as requisições de interrupção e fazer o gerenciamento do dispositivo correspondente. Dentro de um mesmo grupo de dispositivos similares, como as placas de rede, há centenas de modelos de diferentes fabricantes, com interfaces distintas. Para cada dispositivo é então necessário construir um *driver* específico.

Acima dos *drivers* existe uma camada de código, denominada *generic device interface*, cuja finalidade é construir uma visão genérica de dispositivos similares, para que o restante do sistema operacional não precise ter consciência das peculiaridades de cada dispositivo, mas possa tratá-los por famílias ou classes, como dispositivos de armazenamento, interfaces de rede, de vídeo, etc.

Acima da camada de interface genérica de dispositivos, uma ou mais camadas de código estão presentes, para implementar abstrações mais complexas, como sistemas de arquivos e protocolos de rede. Finalmente, no topo da arquitetura de software, são implementadas as chamadas de sistema fornecidas às aplicações para acessar as abstrações construídas pelas camadas inferiores, como arquivos, diretórios e *sockets* de rede, etc.

20.3 Classes de dispositivos

Para simplificar a construção de aplicações e das camadas mais elevadas do próprio sistema operacional, os dispositivos de entrada/saída são geralmente agrupados em classes ou famílias com características similares, para os quais uma interface genérica pode ser definida. Por exemplo, discos rígidos SATA, discos SSD e DVD-ROMs têm características mecânicas e elétricas distintas, mas servem basicamente para o mesmo propósito: armazenar arquivos. O mesmo pode ser afirmado sobre interfaces de rede *Ethernet* e *Wifi*: embora usem tecnologias distintas, ambas permitem a comunicação entre computadores.

Nos sistemas de padrão UNIX os dispositivos são geralmente agrupados em quatro grandes famílias[2]:

Dispositivos orientados a caracteres: são aqueles cujas transferências de dados são sempre feitas byte por byte, em sequência. Um dispositivo orientado a caracteres pode ser visto como um fluxo contínuo de entrada ou de saída de bytes. A característica sequencial faz com que não seja possível alterar o valor de um byte que já foi enviado. Dispositivos ligados às interfaces paralelas e seriais do computador, como *mouse* e teclado, são os exemplos mais clássicos desta família. Os terminais de texto e *modems* de transmissão de dados por linhas seriais (como as linhas telefônicas) também são considerados dispositivos orientados a caracteres.

Dispositivos orientados a blocos: são aqueles dispositivos em que as operações de entrada ou saída de dados são feitas usando blocos de bytes de tamanho fixo. Esses blocos são lidos ou escritos em posições específicas do dispositivo, ou seja, são endereçáveis. Conceitualmente, um dispositivo orientado a blocos pode ser visto como um vetor de blocos de bytes de mesmo tamanho. Discos rígidos, CDROMs, fitas magnéticas e outros dispositivos de armazenamento são exemplos típicos desta família.

Dispositivos de rede: estes dispositivos permitem enviar e receber mensagens entre processos e computadores distintos. As mensagens são blocos de dados de tamanho variável, com envio e recepção feitas de forma sequencial (não é possível alterar o conteúdo de uma mensagem que já foi enviada). As interfaces *Ethernet*, *Wifi*, *Bluetooth* e GPRS são bons exemplos desta classe de dispositivos.

Dispositivos gráficos: permitem a renderização de texto e gráficos em terminais de vídeo. Devido aos requisitos de desempenho, sobretudo para jogos e filmes, estes dispositivos exigem um alto desempenho na transferência de dados. Por isso, sua interface genérica é constituída por funções para consultar e configurar o dispositivo gráfico e uma área de memória compartilhada entre o processador e o dispositivo, usualmente denominada *frame buffer*, que permite acesso direto à memória de vídeo. Programas ou bibliotecas que interagem diretamente com o dispositivo gráfico têm acesso a essa área de memória através de bibliotecas específicas, como *DirectX* em ambientes Windows ou *DRI – Direct Rendering Engine* no Linux.

[2]Nos sistemas *Windows* os dispositivos são agrupados em um número maior de categorias.

Vários dispositivos não se enquadram diretamente nas categorias acima, como receptores de GPS, sensores de temperatura e interfaces de áudio. Nestes casos, alguns sistemas operacionais optam por criar classes adicionais para esses dispositivos (como o Windows), enquanto outros buscam enquadrá-los em uma das famílias já existentes (como os UNIX). No Linux, por exemplo, os dispositivos de áudio são acessados pelas aplicações como dispositivos orientados a caracteres: uma sequência de bytes enviada ao dispositivo é tratada como um fluxo de áudio a ser reproduzido, geralmente em formato PCM. Cabe ao driver do dispositivo e à camada de interface genérica transformar essa interface orientada a caracteres nas operações de baixo nível necessárias para reproduzir o fluxo de áudio desejado.

20.4 *Drivers* de dispositivos

Um *driver* de dispositivo, ou simplesmente *driver*, é um componente do sistema operacional responsável por interagir com um controlador de dispositivo. Cada tipo de dispositivo possui seu próprio *driver*, muitas vezes fornecido pelo fabricante do mesmo. Cada *driver* é geralmente capaz de tratar um único tipo de dispositivo, ou uma família de dispositivos correlatos do mesmo fabricante. Por exemplo, o *driver* RTL8110SC(L), da empresa *Realtek Corp.*, serve somente para as interfaces de rede RTL8110S, RTL8110SB(L), RTL8169SB(L), RTL8169S(L) e RTL8169 desse fabricante.

Internamente, um *driver* consiste de um conjunto de funções que são ativadas pelo núcleo do sistema operacional conforme necessário. Existem basicamente três grupos de funções implementadas por um *driver*, ilustradas na Figura 20.2:

Funções de entrada/saída: responsáveis pela transferência de dados entre o dispositivo e o sistema operacional; essas funções recebem e enviam dados de acordo com a classe dos dispositivo: caracteres (bytes), blocos de tamanho fixo (discos), blocos de tamanho variável (pacotes de rede) ou áreas de memória compartilhadas entre o dispositivo e a CPU (imagens/vídeo e outros).

Funções de gerência: responsáveis pela gestão do dispositivo e do próprio *driver*. Além de funções para coordenar a inicialização e finalização do *driver* e do dispositivo, geralmente são fornecidas funções para configurar o dispositivo, para desligar ou colocar em espera o dispositivo quando este não for usado, e para tratar erros no *dispositivo*. Algumas dessas funções podem ser disponibilizadas aos processos no espaço de usuário, através de chamadas de sistema específicas como `ioctl` (UNIX) e `DeviceIoControl` (Windows).

Funções de tratamento de eventos: estas funções são ativadas quando uma requisição de interrupção é gerada pelo dispositivo. Conforme apresentado na Seção 19.6, toda requisição de interrupção gerada pelo dispositivo é encaminhada ao controlador de interrupções do hardware, que a entrega ao núcleo do sistema operacional. No núcleo, um tratador de interrupções (*IRq handler*) reconhece e identifica a interrupção junto ao controlador e em seguida envia uma notificação de evento a uma função do *driver*, para o devido tratamento.

Além das funções acima descritas, um *driver* mantém estruturas de dados locais, para armazenar informações sobre o dispositivo e as operações em andamento.

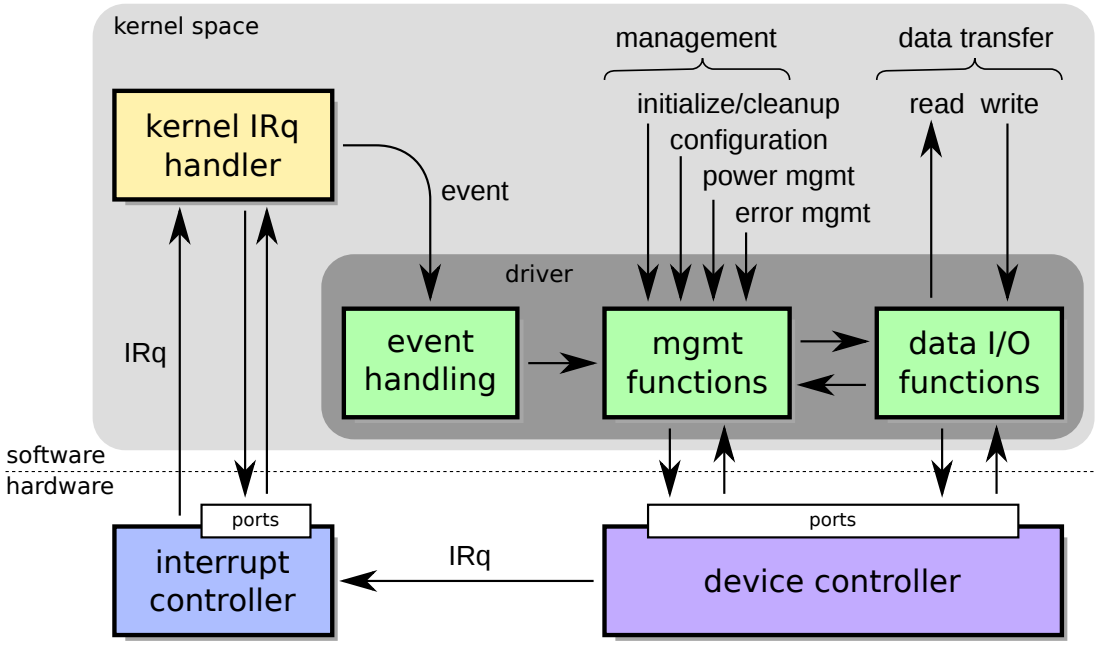

Figura 20.2: Visão geral de um *driver* de dispositivo.

Os *drivers* normalmente executam dentro do núcleo do sistema operacional, em modo privilegiado. Por ser código de terceiros executando com acesso total ao hardware, eles constituem um dos maiores riscos à estabilidade e segurança do sistema operacional. *Drivers* mal construídos ou mal configurados são fontes frequentes de problemas como travamentos ou reinicializações inesperadas.

20.5 Estratégias de interação

Cada *driver* deve interagir com seu respectivos dispositivo de entrada/saída para realizar as operações desejadas, através das portas de seu controlador. Esta seção aborda as três estratégias mais frequentemente usadas pelos *drivers* para essa interação, que são a entrada/saída controlada por programa, a controlada por eventos e o acesso direto à memória, detalhados a seguir.

20.5.1 Interação controlada por programa

A estratégia de entrada/saída mais simples, usada com alguns tipos de dispositivos, é a interação controlada por programa, também chamada **varredura**, **polling** ou PIO – *Programmed I/O*. Nesta abordagem, o *driver* solicita uma operação ao controlador do dispositivo, usando as portas *control* e *data-out* (ou *data-in*) de sua interface, e aguarda a conclusão da operação solicitada, monitorando continuamente os bits da respectiva porta de status.

Considerando as portas da interface paralela descrita na Seção 19.4, o comportamento do *driver* em uma operação de saída na porta paralela usando essa abordagem é descrito (simplificadamente) pelos trechos de código a seguir. O primeiro trecho de código contém definições de macros C úteis para o restante do código:

```
// portas da interface paralela LPT1 (endereço inicial em 0378h)
#define P0    0x0378        # porta de dados
#define P1    0x0379        # porta de status
#define P2    0x037A        # porta de controle

// bits de controle e status das portas
#define BUSY  7             # bit 7 da porta de status
#define ACK   6             # bit 6 da porta de status
#define STROBE 0            # bit 0 da porta de controle

// operações em bits individuais de bytes
#define BIT_SET(a,n) (a |= 1 << n)       // muda n-esimo bit de a para 1
#define BIT_CLR(a,n) (a &= 1 << n)       // muda n-esimo bit de a para 0
#define BIT_TEST(a,n) (a & 1 << n)       // testa n-esimo bit de a
```

O código a seguir contém a implementação simplificada de uma operação de saída de caractere. As leituras e escritas de bytes nas portas do dispositivo são representadas respectivamente pelas funções `in(port)` e `out(port,value)`:

```
// saída de dados por programa
void polling_output (char c)
{
  // espera o controlador ficar livre, testando a porta de status (P1)
  while (BIT_TEST (in(P1), BUSY)) ;

  // escreve o byte "c" a enviar na porta de dados (P0)
  out (P0, c) ;

  // gera pulso em 0 no bit STROBE da porta de controle (P2),
  // para indicar ao controlador que há um novo dado em P0
  out (P2, BIT_CLR (in(P2), STROBE)) ;    // bit STROBE de P2 = 0
  usleep (1) ;                            // aguarda 1 us
  out (P2, BIT_SET (in(P2), STROBE)) ;    // bit STROBE de P2 = 1

  // espera controlador receber o dado (pulso em 0 no bit ACK de P1)[3]
  while (BIT_TEST (in(P1), ACK)) ;

  // espera o controlador concluir a operação solicitada
  while (BIT_TEST (in(P1), BUSY)) ;
}
```

Observa-se que o processador e o controlador executam ações coordenadas e complementares: o processador espera que o controlador esteja livre antes de lhe enviar um dado; por sua vez, o controlador espera que o processador lhe envie um novo dado para processar, e assim por diante. Essa interação é ilustrada na Figura 20.3.

Nessa estratégia, o controlador pode ficar esperando por novos dados, pois só precisa trabalhar quando há dados a processar, ou seja, quando o bit *strobe* de sua porta P_2 indicar que há um novo dado em sua porta P_0. Entretanto, manter o processador esperando até que a operação seja concluída é indesejável, sobretudo se a operação solicitada for demorada. Além de constituir uma situação clássica de desperdício de recursos por **espera ocupada**, manter o processador esperando pela resposta de um

[3] Este laço de espera sobre o bit `ACK` é desnecessário, pois o próximo laço testa o bit `BUSY`, que indica a conclusão da operação solicitada. Contudo, ele foi mantido no código para maior clareza didática.

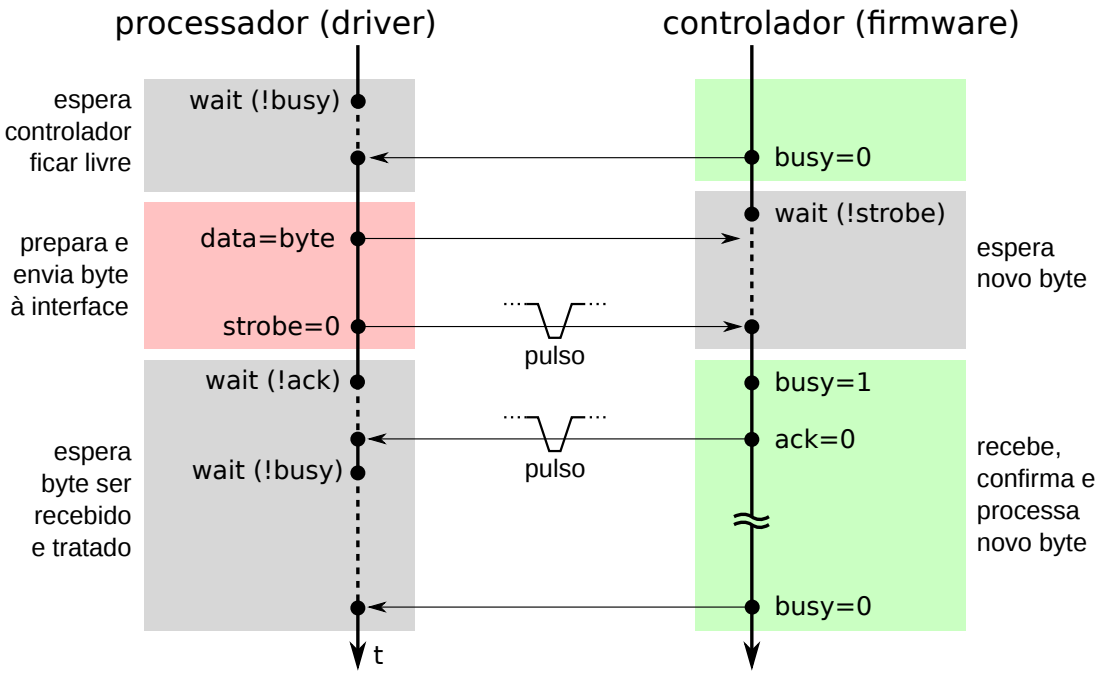

Figura 20.3: Entrada/saída controlada por programa.

controlador lento pode prejudicar o andamento de outras atividades importantes do sistema, como a interação com o usuário.

O problema da espera ocupada torna a estratégia de entrada/saída por programa pouco eficiente, sobretudo se o dispositivo for lento. Por isso, ela é pouco usada em sistemas operacionais de propósito geral. Seu uso se concentra sobretudo em sistemas embarcados dedicados, nos quais o processador só tem uma atividade (ou poucas) a realizar. A estratégia básica de varredura pode ser modificada, substituindo o teste contínuo do status do dispositivo por um teste periódico (por exemplo, a cada 1*ms*), e permitindo ao processador executar outras tarefas enquanto o dispositivo estiver ocupado. Todavia, essa abordagem implica em uma menor taxa de transferência de dados para o dispositivo e, por essa razão, só é usada em dispositivos com baixa vazão de dados.

20.5.2 Interação controlada por eventos

Uma forma mais eficiente de interagir com dispositivos de entrada/saída consiste em efetuar a requisição da operação desejada e suspender o fluxo de execução corrente, liberando o processador para tratar outras tarefas. Quando o dispositivo tiver terminado de processar a operação solicitada, seu controlador irá gerar uma requisição de interrupção (IRq) para notificar o respectivo *driver*, que poderá então retomar a execução daquele fluxo de instruções. Essa estratégia de ação é denominada **interação controlada por eventos** ou por interrupções, pois as interrupções têm um papel fundamental em sua implementação.

Na estratégia de entrada/saída por eventos, uma operação de entrada ou saída é dividida em dois blocos de instruções: um bloco que inicia a operação, ativado pelo *driver* a pedido de um processo ou *thread*, e uma **rotina de tratamento de interrupção** (*interrupt handler*), ativada a cada interrupção, para informar sobre a conclusão da última operação solicitada.

Considerando novamente a interface paralela descrita na Seção 19.4, o código a seguir representa o lançamento da operação de E/S pelo *driver* e a rotina de tratamento de interrupção (subentende-se as constantes e variáveis definidas na listagem anterior). Conforme visto na Seção 19.4, o controlador da interface paralela pode ser configurado para gerar uma interrupção através do flag *Enable_IRq* de sua porta de controle (porta P_2).

```
1  // saída de dados por evento: solicitação de operação
2  void event_output (char c)
3  {
4    // espera o controlador ficar livre, testando a porta de status (P1)
5    while (BIT_TEST (in(P1), BUSY)) ;
6
7    // escreve o byte "c" a enviar na porta de dados (P0)
8    out (P0, c) ;
9
10   // gera pulso em 0 no bit STROBE da porta de controle (P2),
11   // para indicar ao controlador que há um novo dado em P0
12   out (P2, BIT_CLR (in(P2), STROBE)) ;    // bit STROBE de P2 = 0
13   usleep (1) ;                            // aguarda 1 us
14   out (P2, BIT_SET (in(P2), STROBE)) ;    // bit STROBE de P2 = 1
15
16   // espera controlador receber o dado (pulso em 0 no bit ACK de P1)
17   while (BIT_TEST (in(P1), 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 }
23
24 // saída de dados por evento: tratamento da interrupção
25 void event_handle ()
26 {
27   // o controlador concluiu sua operação, acordar a tarefa solicitante.
28   awake_task () ;
29 }
```

Nesse exemplo, percebe-se que o *driver* inicia a transferência de dados para a interface paralela e suspende a tarefa solicitante (chamada *suspend_task*), liberando o processador para outras atividades. Ao ocorrer uma interrupção, a rotina de tratamento do *driver* é ativada pelo SO e acorda a tarefa solicitante. O diagrama da Figura 20.4 ilustra de forma simplificada a estratégia de entrada/saída usando interrupções.

Uma variante mais eficiente da operação por eventos consistiria em fornecer ao *driver* um *buffer* com N bytes a enviar, como mostra o código a seguir:

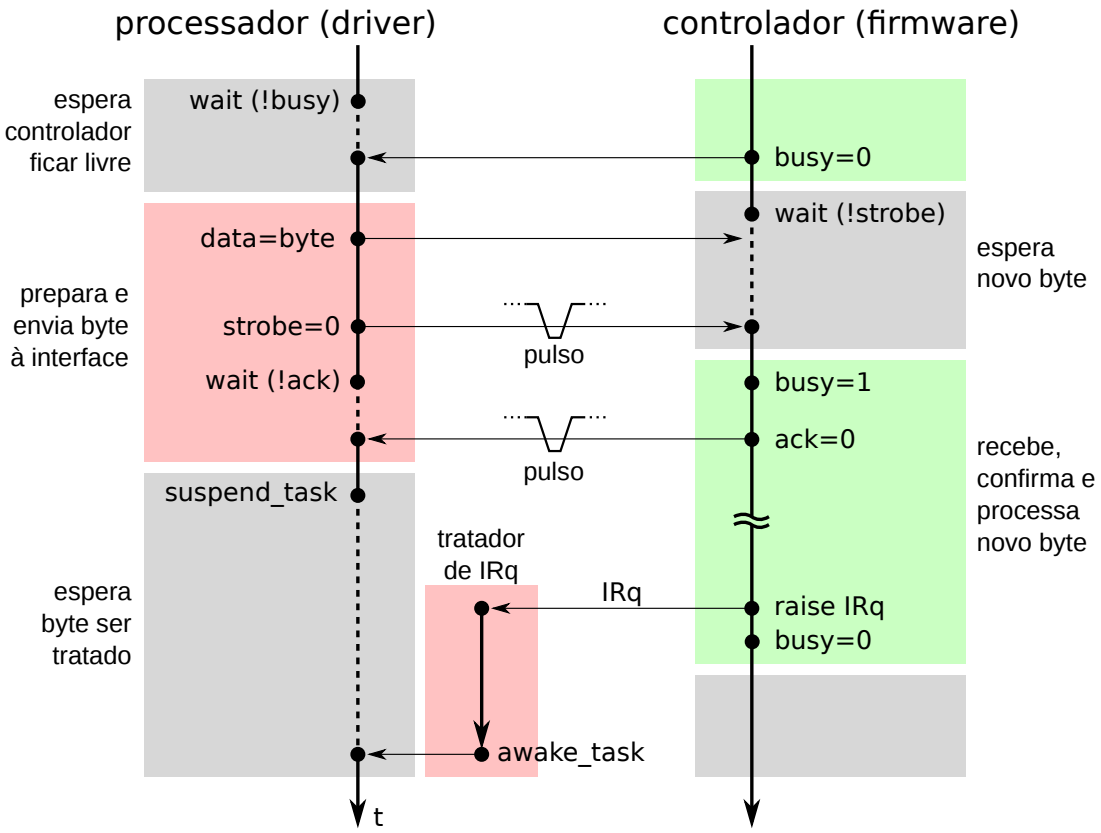

Figura 20.4: Entrada/saída controlada por eventos (interrupções).

```
1  // buffer de bytes a enviar
2  char *buffer ;
3  int bufsize, pos ;
4
5  // envia um byte à porta paralela
6  void send_byte (char c)
7  {
8    // espera o controlador ficar livre, testando a porta de status (P1)
9    while (BIT_TEST (in(P1), BUSY)) ;
10
11   // escreve o byte a enviar na porta de dados (P0)
12   out (P0, c) ;
13
14   // gera pulso em 0 no bit STROBE da porta de controle (P2),
15   // para indicar ao controlador que há um novo dado em P0
16   out (P2, BIT_CLR (in(P2), STROBE)) ;    // bit STROBE de P2 = 0
17   usleep (1) ;                            // aguarda 1 us
18   out (P2, BIT_SET (in(P2), STROBE)) ;    // bit STROBE de P2 = 1
19
20   // espera controlador receber o dado (pulso em 0 no bit ACK de P1)
21   while (BIT_TEST (in(P1), ACK)) ;
22 }
```

```
23 // lançamento da operação de saída de dados
24 void event_output ()
25 {
26   // envia o primeiro byte do buffer
27   pos = 0 ;
28   send_byte (buffer[pos]) ;
29
30   // suspende a execução, liberando o processador para outras tarefas
31   // enquanto o controlador está ocupado processando o dado recebido.
32   suspend_task () ;
33 }
34
35 // rotina de tratamento de interrupções da interface paralela
36 void event_handle ()
37 {
38   pos++ ;                          // avança posição de envio no buffer
39   if (pos >= bufsize)              // o buffer terminou?
40     awake_task () ;                // sim, acorda a tarefa solicitante
41   else
42     send_byte (buffer[pos]) ;      // não, envio o próximo byte
43 }
```

20.5.3 Acesso direto à memória

Na maioria das vezes, o tratamento de operações de entrada/saída é uma operação lenta, pois os dispositivos periféricos são mais lentos que o processador. Além disso, o uso do processador principal para intermediar essas operações é ineficiente, pois implica em transferências adicionais (e desnecessárias) de dados: por exemplo, para transportar um byte de um *buffer* da memória para a interface paralela, esse byte precisa antes ser carregado em um registrador do processador, para em seguida ser enviado ao controlador da interface. Para resolver esse problema, a maioria dos computadores atuais, com exceção de pequenos sistemas embarcados dedicados, oferece mecanismos de **acesso direto à memória** (DMA - *Direct Memory Access*), que permitem transferências diretas entre a memória principal e os controladores de entrada/saída.

O funcionamento do mecanismo de acesso direto à memória em si é relativamente simples. Como exemplo, a seguinte sequência de passos seria executada para a escrita de dados de um *buffer* em memória RAM para o controlador de um disco rígido:

1. o processador acessa as portas do controlador de DMA associado ao dispositivo desejado, para informar o endereço inicial e o tamanho da área de memória RAM contendo os dados a serem escritos no disco. O tamanho da área de memória deve ser um múltiplo do tamanho dos blocos físicos do disco rígido (512 ou 4096 bytes);

2. o controlador de DMA solicita ao controlador do disco a transferência de dados da RAM para o disco e aguarda a conclusão da operação;

3. o controlador do disco recebe os dados da memória;

4. a operação anterior pode ser repetida mais de uma vez, caso a quantidade de dados a transferir seja maior que o tamanho máximo de cada transferência feita pelo controlador de disco;

5. a final da transferência de dados, o controlador de DMA notifica o processador sobre a conclusão da operação, através de uma requisição de interrupção (IRq).

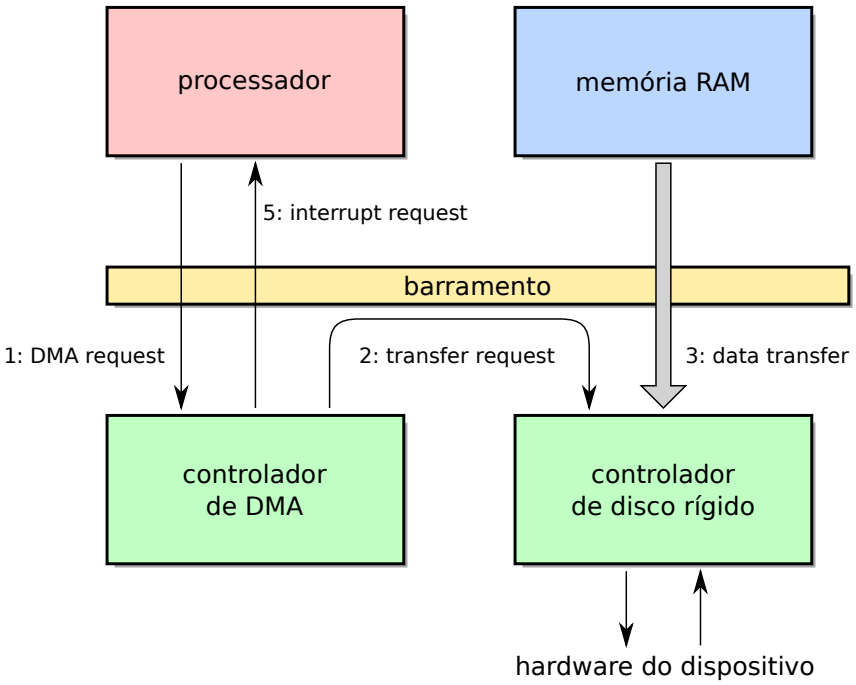

Figura 20.5: Funcionamento do acesso direto à memória.

A dinâmica dessas operações é ilustrada de forma simplificada na Figura 20.5. Uma vez efetuado o passo 1, o processador fica livre para outras atividades[4], enquanto o controlador de DMA e o controlador do disco se encarregam da transferência de dados propriamente dita. O código a seguir representa uma implementação hipotética de rotinas para executar uma operação de entrada/saída através de DMA.

[4]Obviamente pode existir uma contenção (disputa) entre o processador e os controladores no acesso aos barramentos e à memória principal, mas esse problema é atenuado pelo fato do processador poder acessar o conteúdo das memórias *cache* enquanto a transferência DMA é executada. Além disso, circuitos de *arbitragem* intermedeiam o acesso à memória para evitar conflitos. Mais detalhes sobre contenção de acesso à memória durante operações de DMA podem ser obtidos em [Patterson and Henessy, 2005].

```
// 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 (...) ;
}
```

A implementação dos mecanismos de DMA depende da arquitetura e do barramento considerado. Computadores mais antigos dispunham de um controlador de DMA único, que oferecia vários **canais de DMA** distintos, permitindo a realização de transferências DMA simultâneas. Já os computadores pessoais usando barramento PCI não possuem um controlador DMA central; ao invés disso, cada controlador de dispositivo conectado ao barramento pode assumir o controle do mesmo para efetuar transferências DMA sem depender do processador principal [Corbet et al., 2005], gerenciando assim seu próprio canal.

No exemplo anterior, a ativação do mecanismo de DMA é dita **síncrona**, pois é feita explicitamente pelo processador, provavelmente em decorrência de uma chamada de sistema. Contudo, a ativação também pode ser **assíncrona**, quando ativada por um dispositivo de entrada/saída que dispõe de dados a serem transferidos para a memória, como ocorre com uma interface de rede ao receber dados provindos da rede.

O mecanismo de DMA é utilizado para transferir grandes blocos de dados diretamente entre a memória RAM e as portas dos dispositivos de entrada/saída, liberando o processador para outras atividades. Todavia, como a configuração de cada operação de DMA é complexa, para pequenas transferências de dados acaba sendo mais rápido e simples usar o processador principal [Bovet and Cesati, 2005]. Por essa razão, o mecanismo de DMA é usado sobretudo nas operações de entrada/saída envolvendo dispositivos que produzem ou consomem grandes volumes de dados, como discos, interfaces de rede, entradas e saídas de áudio, interfaces gráficas e discos.

20.6 Tratamento de interrupções

Durante a execução de uma rotina de tratamento de interrupção, é usual inibir novas interrupções, para evitar a execução aninhada de tratadores de interrupção, o que tornaria o código dos *drivers* (e do núcleo) bem mais complexo e suscetível a erros. Entretanto, manter interrupções inibidas durante muito tempo pode ocasionar perdas de dados ou outros problemas. Por exemplo, uma interface de rede gera uma interrupção

quando recebe um pacote vindo da rede; esse pacote fica em seu buffer interno e deve ser transferido dali para a memória principal antes que outros pacotes cheguem, pois esse buffer tem uma capacidade limitada. Por essa razão, o tratamento das interrupções deve ser feito de forma muito rápida.

Para obter rapidez, a maioria dos sistemas operacionais implementa o tratamento de interrupções em dois níveis: um **tratador primário** (FLIH - *First-Level Interrupt Handler*) e um **tratador secundário** (SLIH - *Second-Level Interrupt Handler*). O tratador primário, também chamado *hard/fast interrupt handler* ou ainda *top-half handler*, é ativado a cada interrupção recebida e executa rapidamente, com as demais interrupções desabilitadas. Sua tarefa consiste em reconhecer a ocorrência da interrupção junto ao controlador de interrupções, criar um *descritor de evento* contendo os dados da interrupção ocorrida, inserir esse descritor em uma fila de eventos pendentes junto ao *driver* do respectivo dispositivo e notificar o *driver*.

O tratador secundário, também conhecido como *soft/slow interrupt handler* ou ainda *bottom-half handler*, tem por objetivo tratar os eventos pendentes registrados pelo tratador primário. Ele geralmente é escalonado como uma *thread* de núcleo com alta prioridade, executando quando um processador estiver disponível. Embora mais complexa, esta estrutura em dois nível traz vantagens: ao permitir um tratamento mais rápido de cada interrupção, ela minimiza o risco de perder interrupções simultâneas; além disso, a fila de eventos pendentes pode ser analisada para remover eventos redundantes (como atualizações consecutivas de posição do *mouse*).

No Linux, cada interrupção possui sua própria fila de eventos pendentes e seus próprios *top-half* e *bottom-half*. Os tratadores secundários são lançados pelos respectivos tratadores primários, sob a forma de *threads* de núcleo especiais (denominadas *tasklets* ou *workqueues*) [Bovet and Cesati, 2005]. O núcleo Windows NT e seus sucessores implementam o tratamento primário através de rotinas de serviço de interrupção (ISR - *Interrupt Service Routine*). Ao final de sua execução, cada ISR agenda o tratamento secundário da interrupção através de um procedimento postergado (DPC - *Deferred Procedure Call*) [Russinovich et al., 2008]. O sistema *Symbian* usa uma abordagem similar a esta.

Por outro lado, os sistemas Solaris, FreeBSD e MacOS X usam uma abordagem denominada *interrupt threads* [Mauro and McDougall, 2006]. Cada interrupção provoca o lançamento de uma *thread* de núcleo, que é escalonada e compete pelo uso do processador de acordo com sua prioridade. As interrupções têm prioridades que podem estar acima da prioridade do escalonador ou abaixo dela. Como o próprio escalonador também é uma *thread*, interrupções de baixa prioridade podem ser interrompidas pelo escalonador ou por outras interrupções. Por outro lado, interrupções de alta prioridade não são interrompidas pelo escalonador, por isso devem executar rapidamente.

Exercícios

1. Explique o que é e como funciona o tratamento de interrupções em dois níveis pelo sistema operacional.

2. Considere um dispositivo físico cujo controlador tem quatro portas de 8 bits:

porta	sentido	função	valor
data_in	*dev* → *cpu*	leitura de dados	byte lido
data_out	*cpu* → *dev*	escrita de dados	byte a escrever
status	*dev* → *cpu*	estado do dispositivo	`IDLE`, `BUSY` ou `ERROR`
control	*cpu* → *dev*	operação a efetuar	`READ`, `WRITE` ou `RESET`

Essas portas são acessadas através das operações "`val=in(port)`" e "`out(port,val)`".

Escreva em pseudocódigo as funções "`char readchar()`" e "`void writechar(char c)`" do *driver* desse dispositivo, que implementam respectivamente as operações de **leitura** e de **escrita** de caracteres individuais, usando a estratégia de *polling* (interação por programa).

3. Considerando o dispositivo da questão anterior, escreva em pseudocódigo a função "`char readchar()`" do *driver* desse dispositivo, que implementa a operação de **leitura** de caracteres individuais, usando a estratégia de **interação por eventos** (interrupções).

Referências

D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd edition*. O'Reilly Media, Inc, 2005.

J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc, 2005.

J. Mauro and R. McDougall. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice-Hall PTR, 2006.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.