Princípios do SW de E/S

Objetivos do SW de E/S:

1. **Independência do dispositivo**

Deve ser possível escrever um programa que acesse qualquer dispositivo de E/S sem a necessidade de especificar explicitamente a qual dispositivo se refere

Exemplo 1: programa que lê um arquivo de entrada de dispositivos diferentes (HD, CD-ROM, DVD, USB, etc)

Exemplo 2: sort < input > output (qq tipo de entrada e qq tipo de saída)

Problema de tratar diferenças fica com o SO

Princípios do SW de E/S

2. Nomeação Uniforme

Nome do arquivo ou do dispositivo deve ser uma cadeia de caracteres, independentemente do dispositivo acessado

Exemplo: no UNIX, discos são montados no sistema de arquivos de tal forma que fica transparente ao usuário de qual dispositivo veio um arquivo

USB pode ser montada em um diretório /usr/ast/backup. Copiar um arquivo para o subdiretório /usr/ast/backup/tuesday

significa copiar o arquivo para a USB

⇒ ***todos os arquivos são acessados da mesma maneira: nome do caminho***

Princípios do SW de E/S

3. Tratamento de erros

erros devem ser detectados e corrigidos (se possível) o mais próximo possível do HW, de forma transparente às camadas superiores

Exemplo: controlador de disco (HW) deve corrigir um erro de leitura (ou o ***driver*** do dispositivo, caso este não consiga, lendo novamente um bloco)

- ✓ ***Em muitos casos, a camada inferior detecta e corrige o erro sem que as camadas superiores tomem conhecimento do mesmo***
- ✓ ***Somente se as camadas inferiores não conseguirem tratar o erro é que as camadas superiores devem ser informadas***

Princípios do SW de E/S

4. Transferência síncrona ou assíncrona (via interrupções)

maioria é assíncrona, deixando a CPU livre para outra atividade

… mas programas de usuário são mais fáceis de escrever de forma síncrona (ou *bloqueante*, quando o programa fica esperando após uma chamada READ, por exemplo)

⇒ ***SO fica responsável por fazer as operações orientadas a interrupção parecerem bloqueantes ao usuário***

Princípios do SW de E/S

5. Utilização de buffer

Muitas vezes, dados provenientes dos dispositivos não podem ser armazenados no destino final.

Ex1: pacote vindo da rede

Ex2: dispositivos com restrições de tempo real (áudio digitais)

⇒ *Dados devem ser colocados num buffer para separar a taxa de* ***envio / recebimento*** *da taxa com o qual ele é* ***esvaziado / preenchido***

Princípios do SW de E/S

6. Utilização de dispositivos compartilhados e dedicados

- ***Disco***: dispositivo que pode ser compartilhado
- ***Impressora***: dispositivo dedicado
- Uso de dispositivos dedicados gera problemas de impasses

⇒ ***SO deve ser capaz de tratar tanto os dispositivos compartilhados quanto os dispositivos dedicados de uma maneira que evite problemas***

E/S Programada

CPU faz todo o trabalho

Exemplo: impressão da cadeia de caracteres ABCDEFGH

1. Processo do usuário monta a cadeia em um buffer
2. Processo do usuário requisita impressão da cadeia de caracteres através de uma chamada de sistema
3. Se impressora estiver sendo usada, retorna código de erro ou bloqueia o processo até que esteja disponível
4. Qdo a impressora estiver disponível, processo faz chamada de sistema para imprimir cadeia

E/S Programada

5. SO copia a cadeia para um vetor no espaço do núcleo
6. SO verifica se impressora está disponível (registrador específico)
7. SO copia primeiro caracter para registrador de dados da impressora

   ⇒ ***ativa a impressora*** (algumas impressoras armazenam uma linha ou página antes de começar a imprimir)

5. SO verifica registrador de status da impressora para saber se pode enviar mais dados
6. SO imprime caracter seguinte, etc… Ao final, controle retorna para o processo do usuário que solicitou a impressão

E/S programada

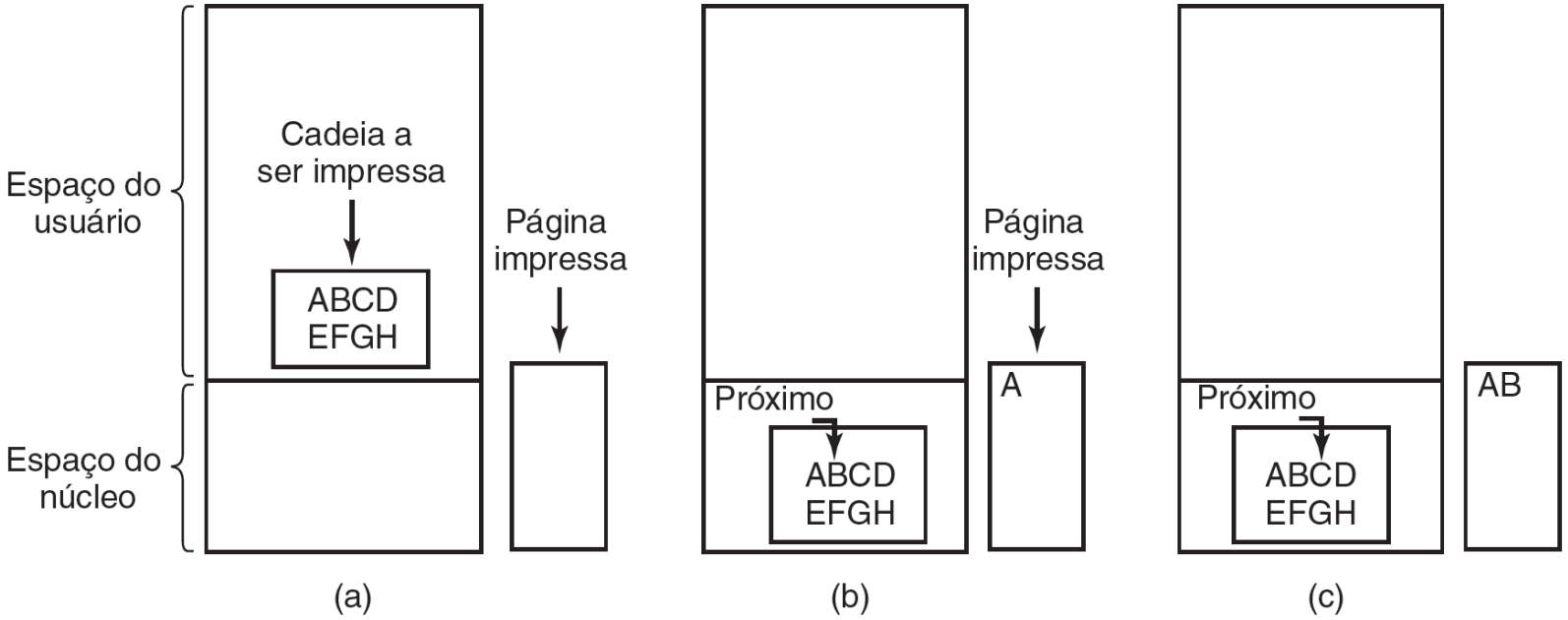

Figura 5.6 Estágios da impressão de uma cadeia de caracteres.

E/S Programada

Problema:

Após imprimir um caracter, CPU fica verificando se a impressora está pronta para aceitar outro

⇒ ***Espera ocupada (ou polling)***

E/S Programada

Desvantagem da E/S programada:

Simples, mas "segura a CPU" até que E/S seja feita (espera ociosa);

Obs:

1. Se tempo da impressão for curto ou for um sistema embarcado, espera ociosa é tolerável

2. Em sistemas mais complexos (CPU tem outras tarefas), espera ociosa é ineficiente

E/S usando interrupção

Motivação:

Exemplo:

Suponha que a impressora pode imprimir 100 caracteres/seg

10 ms por caracter ⇒ **CPU fica 10 ms em espera ociosa!!!!**

Tempo suficiente para chavear o contexto e executar outro processo!!!!!

E/S usando interrupção

Funcionamento:

1. processo que solicitou a impressão da cadeia ABCDEFGH é bloqueado até o fim de toda a cadeia ser impressa
2. buffer é copiado para o espaço do núcleo e primeiro caracter copiado para a impressora qdo esta aceitá-lo
3. a partir deste ponto, CPU escalona outro processo para executar
4. impressora imprime caracter e gera interrupção para aceitar outro (após 10 ms)

E/S usando interrupção

5. CPU interrompe processo atual, salva seu estado e desvia para a rotina de tratamento de interrupções

6. CPU envia o próximo caracter para o registrador de dados da impressora e confirma o recebimento da interrupção

7. CPU retorna para o processo que estava executando antes da interrupção

8. caso não existam mais caracteres para serem impressos, o tratador de interrupções desbloqueia processo que pediu a impressão

E/S usando DMA

Desvantagem do uso de interrupções:

interrupção da CPU a cada caracter impresso

⇒ **tratamento de interrupções custa tempo!**

Solução:

- DMA alimenta os caracteres para a impressora sem que a CPU seja interrompida
- reduz o número de interrupções da CPU de uma por caracter para uma por buffer impresso
- com muitos caracteres e impressão lenta

  ⇒ melhora substancial no desempenho!!

E/S usando DMA

Desvantagem do uso de DMA:

Controlador de DMA é mais lento que a CPU!

⇒ se trabalho do controlador não é realizado em sua velocidade máxima e CPU não tem nada para fazer, melhor usar interrupções ou E/S programada

Obs: normalmente, usar o DMA é mais vantajoso

Camadas de software E/S

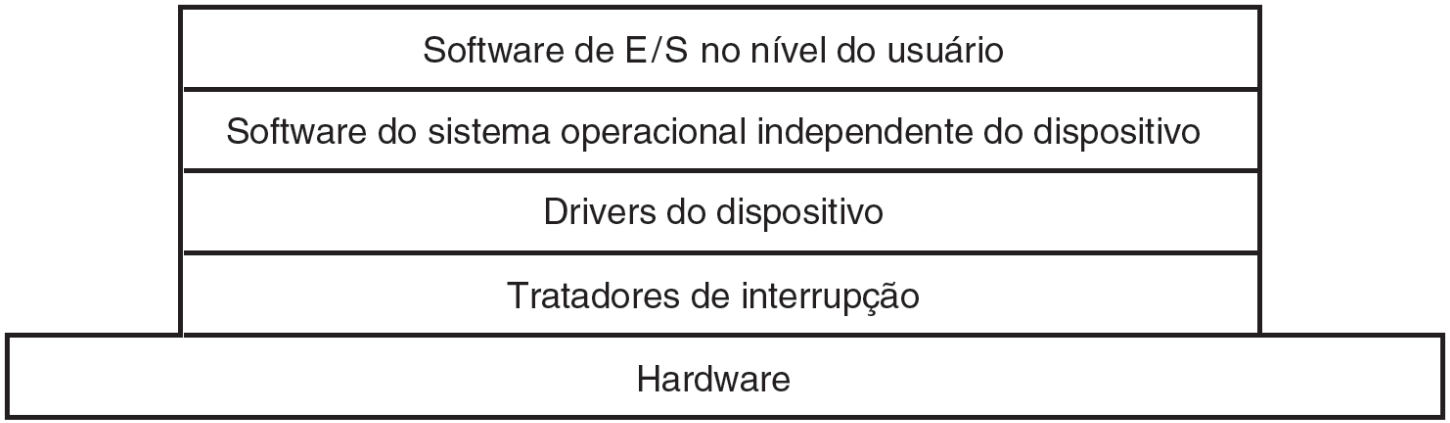

Figura 5.10 Camadas do software de E/S.

Camadas de software E/S

Cada camada:

- Função bem definida
- Interface bem definida para as camadas adjacentes
- Funcionalidade e interface diferem de sistema para sistema

Tratadores de interrupção

- interrupções são consideradas desagradáveis e, portanto, devem ser **"escondidas" pelo SO**
- melhor maneira de esconder: bloquear o *driver* que iniciou a operação de E/S até que esta se complete e a interrupção ocorra
- *O driver* pode se autobloquear executando um *down* em um semáforo, por exemplo (rotina de interrupção o desbloqueia, fazendo um *up* no semáforo, por exemplo)
- *Os drivers* devem ser processos do núcleo (com seus próprios estados, pilhas e PCs)

Processamento de uma interrupção em SW (após interrupção do HW)

1. Salva quaisquer registradores que ainda não foram salvos pelo hardware de interrupção.

2. Estabelece um contexto para a rotina de tratamento da interrupção (pode envolver TLB, MMU e tabela de páginas)

3. Estabelece uma pilha para a rotina de tratamento da interrupção.

4. Sinaliza o controlador de interrupção. Se não existe um HW controlador de interrupção centralizado, reabilita as interrupções.

5. Copia os registradores de onde foram salvos para a tabela de processos.

Processamento de uma interrupção em SW (após a interrupção do HW)

6. Executa a rotina de tratamento da interrupção. Ela extrai informações dos registradores do controlador do dispositivo que gerou a interrupção (informações colocadas pela CPU).
7. Escolhe o próximo processo a ser executado.
8. Estabelece o contexto da MMU (e TLB) para o próximo processo a ser executado.
9. Carrega os registradores do novo processo, incluindo sua PSW (*Program Status Word*).
10. Inicia a execução do novo processo.

Drivers dos dispositivos

Dispositivos de E/S têm características diferentes (nº de registradores, comandos, etc), recebem e fornecem informações diferentes

⇒ códigos controladores (*drivers*) diferentes

- ✓ Escritos e fornecidos pelos fabricantes do dispositivo para diferentes SOs
- ✓ Podem ser parte do núcleo do SO, acessando mais facilmente os registradores dos controladores
- ✓ Podem executar no espaço do usuário, com chamadas de sistema para leitura e escrita nos registradores

Drivers dos dispositivos

Vantagem de implementação no espaço do usuário:

Isolar o núcleo dos *drivers* e estes entre si, eliminando problemas causados ao SO por *drivers* defeituosos. Ex: MINIX 3

… porém, a maioria dos SOs operam com drivers no modo núcleo!!!!

- SOs devem ter uma arquitetura bem definida, modelo bem definido do que faz o ***driver*** e como ele interage com o restante do SO

- ***Drivers*** são posicionados abaixo do SO, logo acima dos controladores dos dispositivos

SISTEMAS OPERACIONAIS MODERNOS
3ª EDIÇÃO

Drivers dos dispositivos

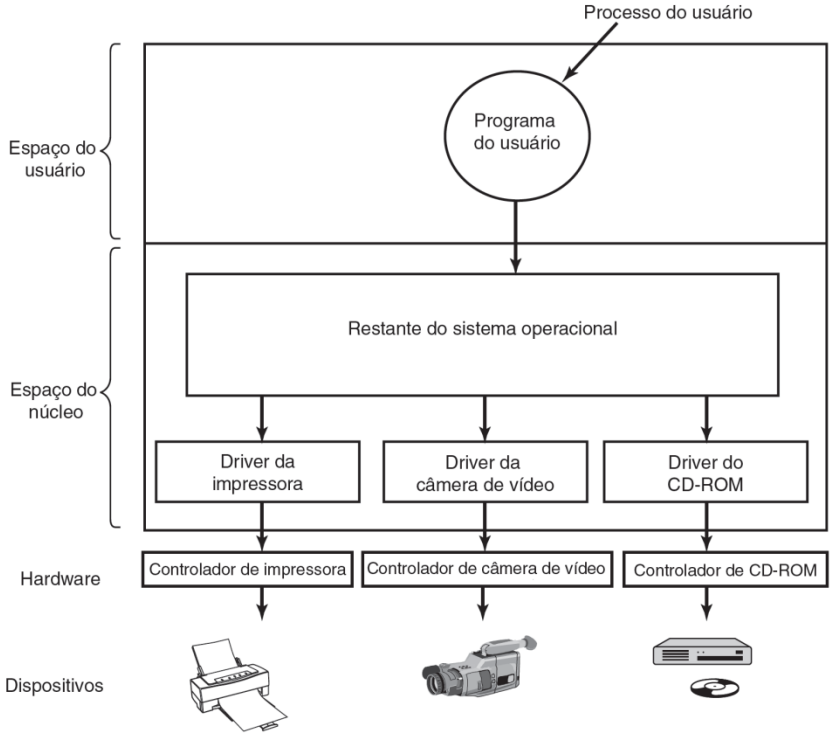

Figura 5.11 Posicionamento lógico dos drivers de dispositivos. Na verdade, toda comunicação entre os drivers e os controladores passa pelo barramento.

Drivers dos dispositivos

- maioria dos SOs define uma interface padrão para os *dispositivos de blocos* e outra para os *dispositivos de caracteres*:

  interfaces: série de rotinas que o SO utiliza para fazer o *driver* trabalhar

  Exemplos dessas rotinas:

  → leitura de blocos

  → escrita de uma cadeia de caracteres

- nos sistemas UNIX antigos os drivers eram compilados (e raramente trocados) com o SO
- nos PCs atuais, usuários normalmente não têm habilidade para recompilar o SO

  ⇒ *drivers* são carregados no SO dinamicamente, durante a execução

Drivers dos dispositivos

Funções dos *drivers*:

- aceitar requisições de leitura ou gravação, provindas de um software independente de dispositivo. Ex.: traduzir um nº de bloco para nº do cabeçote, trilha, setor, cilindro
- iniciar o dispositivo, se necessário
- verificar se são válidos os parâmetros de entrada do dispositivo
- retornar erro se necessário
- gerenciar necessidades de energia

Drivers dos dispositivos

- registrar eventos (msgs de erro, por exemplo)
- verificar se dispositivo está em uso e, se for o caso, enfileirar uma requisição
- se dispositivo ocioso, examinar status do hardware para ver se requisição pode ser tratada imediatamente
- enviar sequência de comandos ao dispositivo

  ⇒ **escrever os comandos nos registradores dos controladores**

- verificar se controlador aceitou comando e se está pronto para aceitar o próximo... até todos os comandos serem emitidos

Drivers dos dispositivos

Após comandos emitidos:

1º caso: ***Driver*** espera o controlador terminar o trabalho solicitado
⇒ **se autobloqueia e espera o dispositivo gerar uma interrupção para desbloqueá-lo**

2º caso: operação finaliza rapidamente sem necessidade de autobloqueio.
Ex: rolar a tela de um vídeo em modo caracter (ns)

- ***Driver*** verifica ocorrência de erros: se não houver erros, passa os dados (se for o caso) para a camada superior
- seleciona próxima requisição pendente, se houver, caso contrário se autobloqueia e espera próxima requisição

Drivers dos dispositivos

Observações sobre ***drivers***:

- Os *drivers* devem ser ***reentrantes***, ou seja, podem ser chamados uma segunda vez antes que a primeira chaamada tenha sido concluída. Ex: enquanto um *driver* de rede está processando um pacote, outro chega

- SO pode informar ao *driver* que o dispositivo sendo usado foi removido

⇒ transferência deve ser abortada sem danificar quaisquer estruturas de dados do núcleo

⇒ requisições pendentes para o dispositivo retirado devem ser removidas

⇒ enviar msgs de erro aos processos que as requisitaram

Software E/S independente de dispositivo

Funções básicas:

1. executar funções de E/S comuns para todos os dispositivos

2. fornecer uma interface uniforme para o SW no nível do usuário

Software E/S independente de dispositivo

Uniformizar interfaces para os drivers de dispositivos
Armazenar no buffer
Reportar erros
Alocar e liberar dispositivos dedicados
Providenciar um tamanho de bloco independente de dispositivo

Tabela 5.2 Funções do software de E/S independente de dispositivo.

Uniformizar interfaces para os drivers de dispositivo

1. Se cada dispositivo possuir uma interface diferente, o SO deverá ser modificado (e recompilado) para cada novo dispositivo que aparece

2. Interface diferente

   ⇒ funções disponíveis para cada *driver* além de funções do núcleo que cada *driver* precisa são diferentes de *driver* para *driver*

   ⇒ demanda grande esforço de programação para cada novo driver

Uniformizar interfaces para os drivers de dispositivo

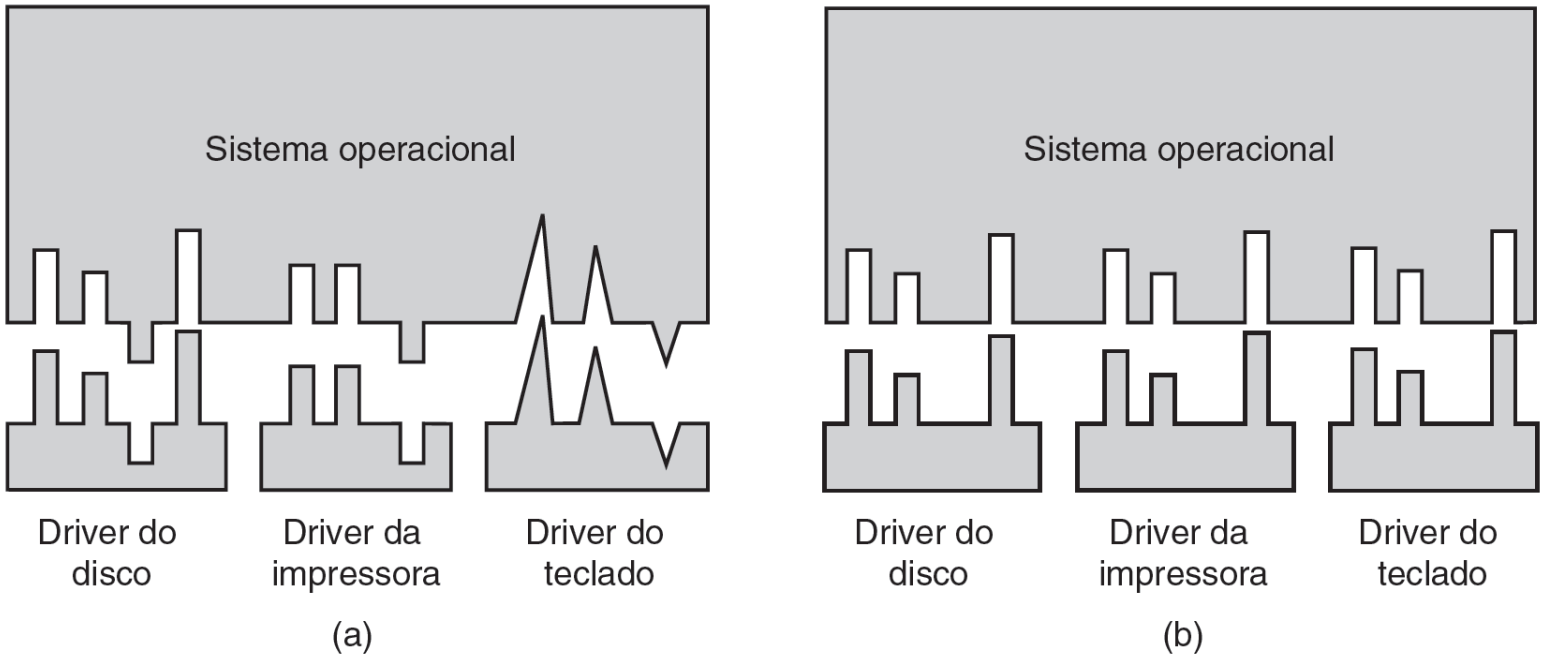

Figura 5.12 (a) Sem uma interface-padrão para o driver. (b) Com uma interface-padrão para o driver.

Uniformizar interfaces para os drivers de dispositivo

1. todos os ***drivers*** devem possuir a mesma interface (ou semelhantes)

2. ***driver*** deve se adequar ao sistema operacional (e não o contrário, como no primeiro caso)

3. programadores dos ***drivers*** sabem quais funções do ***driver*** devem ser fornecidas e quais funções do núcleo eles podem chamar

Uniformizar interfaces para os drivers de dispositivo

Aspectos essenciais para o funcionamento de uma interface

1. Existência de um conjunto de funções para cada ***classe de dispositivo (discos, impressoras, etc)***

2. Nomeação padrão para os dispositivos

3. Proteção para cada um dos dispositivos contra acessos não autorizados

Uniformizar interfaces para os drivers de dispositivo

1. Existência de um *conjunto de funções*

Para cada *classe de dispositivos*, o SO define um conjunto de funções básicas que deve ser complementado pelo ***driver***

Ex.: disco - leitura e escrita, formatação, ligar/desligar

- ***driver*** contém uma tabela com ponteiros cada uma de suas funções
- quando o ***driver*** é carregado, o SO registra endereço da tabela para futuro acesso às funções do ***driver***
- tabela é a ***interface*** entre o ***driver*** e o SO
- todos os dispositivos de uma ***classe*** devem seguir este procedimento

2. Nomeação Padrão:

O SW independente de dispositivo cuida do mapeamento de nomes simbólicos de dispositivos no *driver* apropriado

Ex.: UNIX – /dev/disk0 especifica o i-node para um arquivo especial.

Esse i-node contém o *nº principal do dispositivo*, usado para localizar o *driver* apropriado, e o *nº secundário do dispositivo*, usado para especificar a unidade a ser lida.

Todos os dispositivos têm um nº principal e um nº secundário!

3. Proteção para os dispositivos

Tanto no UNIX qto no Windows: ***dispositivos aparecem como nome de arquivos!***

⇒ ***administrador simplesmente ajusta as permissões adequadas para cada dispositivo como faria para um arquivo comum***

Armazenamento no buffer

Exemplo: fluxo de caracteres lidos de um modem

1ª Opção: fig 5.13a reiniciar o processo do usuário a cada caracter que chega (ineficiente):

Chamada de Sistema, bloqueia à espera, interrupção, passa o caracter, desbloqueia, …. (chega outro caracter, ….)

2ª Opção: fig 5.13b processo do usuário disponibiliza um buffer de tamanho ***n*** e a rotina de tratamento de interrupção só acorda o processo após preencher o buffer

Problema! Página do buffer pode ir para o disco! Trancá-la na memória??

⇒ muitos processos podem fazer o mesmo ⇒ ***diminui nº de molduras disponíveis na memória!***

Armazenamento no buffer

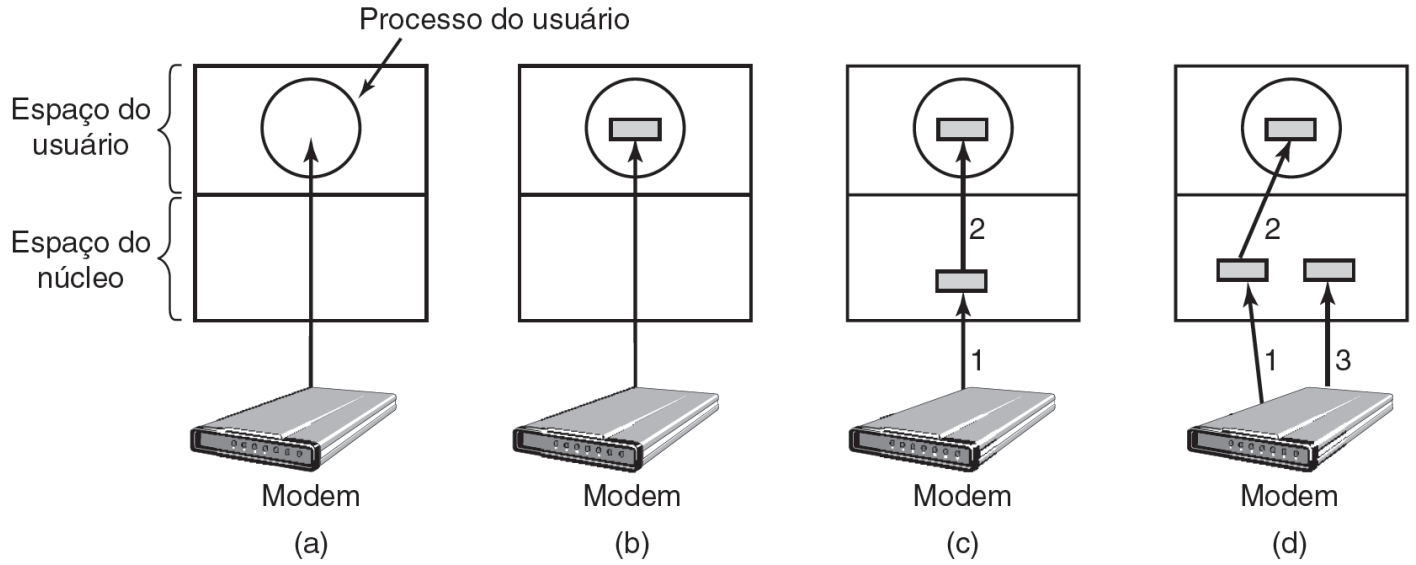

Figura 5.13 (a) Entrada não enviada para buffer. (b) Utilização de buffer no espaço do usuário. (c) Utilização de buffer no núcleo, seguido da cópia para o espaço do usuário. (d) Utilização de buffer duplicado no núcleo.

Armazenamento no buffer

3ª Opção: criar um buffer no núcleo. Quando este buffer estiver cheio, trazer página do buffer do usuário para a memória e realizar cópia. Mais eficiente.

Problema: o que acontece com os caracteres que chegam enquanto a página do usuário está sendo trazida para a memória? (lembrar que o buffer do núcleo está cheio!).

Solução: Manter um segundo buffer no núcleo. Buffers trabalham alternadamente!

SISTEMAS OPERACIONAIS MODERNOS
3ª EDIÇÃO

Exemplo de utilização de um buffer na saída de dados

Desvantagem: se dados forem copiados muitas vezes o desempenho cai pois todas as cópias para os buffers são feitas sequencialmente

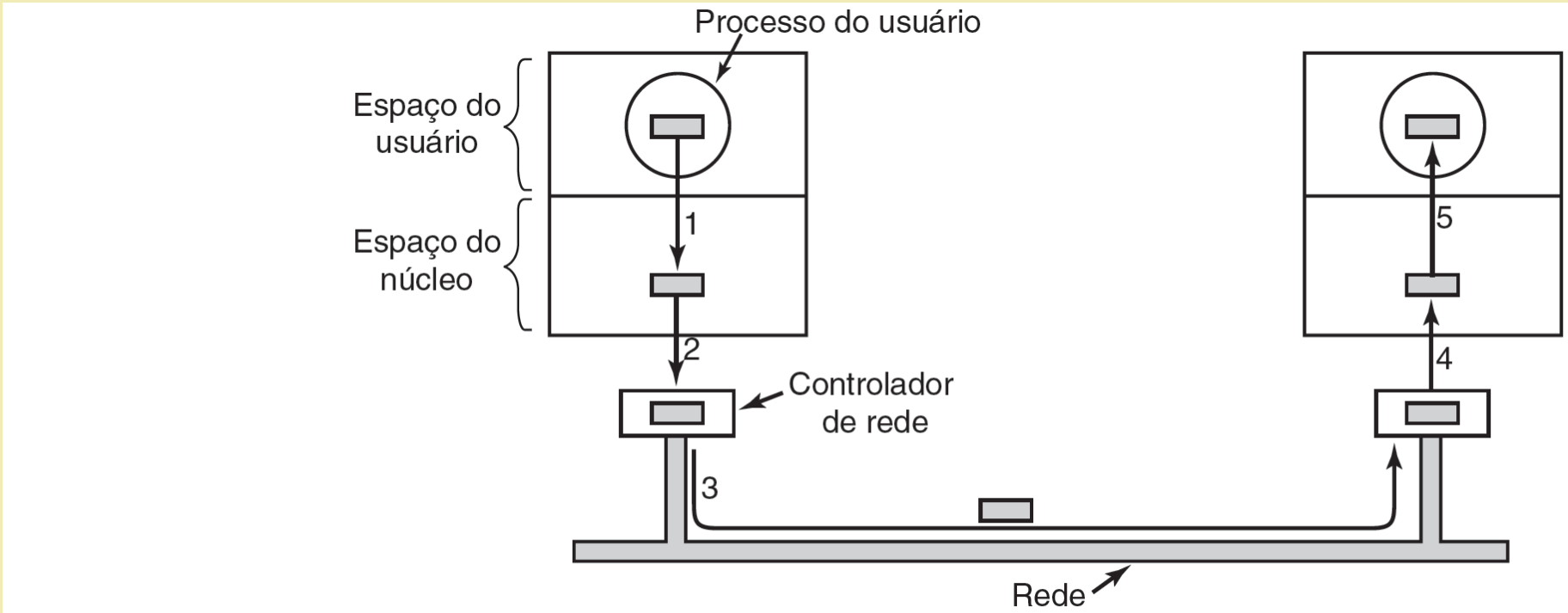

Figura 5.14 O trânsito na rede pode envolver muitas cópias de um pacote.

Reportar erros

- Comuns no contexto de E/S
- Se específico do dispositivo ⇒ tratado pelo driver apropriado
- Modelo para tratamento de erros é independente do dispositivo
- 3 classes de erros:
  - ✓ Erros de programação (Ex.: tentar escrever em dispositivo de entrada)
  - ✓ Endereço de buffer inválido ou especificar dispositivo inválido

    ⇒ **Devolver Código de erro para o chamador**
  - ✓ Tentar escrever em bloco de disco danificado

    ⇒ abre caixa de diálogo ou gera código de erro

Alocar e liberar dispositivos dedicados

SO deve ser capaz de examinar as requisições de acesso aos dispositivos dedicados (ex: impressora) e decidir se as aceita ou não, dependendo da disponibilidade

Duas abordagens:

1. Processos devem usar a chamada de sistema OPEN quando desejarem abrir arquivos especiais, associados aos dispositivos

   OPEN pode falhar (dispos. indisp.) ou não; **CLOSE** libera dispositivo

2. Ao invés de causar uma falha, o processo que chamou o dispositivo que está ocupado é bloquedo e colocado numa fila. Quando o dispositivo for liberado, o primeiro processo da fila é atendido

Providenciar um tamanho de bloco independente de dispositivo

Discos diferentes podem ter tamanhos de setores diferentes

- SW independente de dispositivo deve tornar transparente o tamanho de bloco para as camadas superiores, agrupando setores, por exemplo

  ⇒ camadas superiores usarão sempre o mesmo ***tamanho de bloco lógico***

- o mesmo pode acontecer com unidades que entregam 1 caracter por vez (modem) enquanto outras, como interfaces de redes, entregam vários caracteres por vez

Software de E/S do espaço do usuário

- ✓ composta por uma coleção de rotinas de bibliotecas ligadas aos programas de usuários

- ✓ chamadas de sistema (E/S) são feitas por rotinas de biblioteca
  - Ex: contador = write(fd, buffer, nbytes);
  - Rotina ***write*** é ligada com o programa
  - Conjunto das rotinas de biblioteca é parte do sistema de E/S
  - Outros exemplos: *printf, scanf*

Software de E/S do espaço do usuário

Outro exemplo de SW do espaço do usuário: *spooling e daemon*

Exemplos:

1. Impressoras: cria-se um processo especial, o *daemon*, e um diretório especial, chamado *spool*
   - Processo gera arquivo e o coloca no diretório *spool*
   - *Daemon* (único processo autorizado) imprime arquivos que se encontram no diretório
2. Rede:
   - Usuário coloca arquivo no diretório de *spool* da rede
   - *Daemon* o retira do diretório e o transmite

SISTEMAS OPERACIONAIS MODERNOS
3ª EDIÇÃO

Resumo do Sistema de E/S

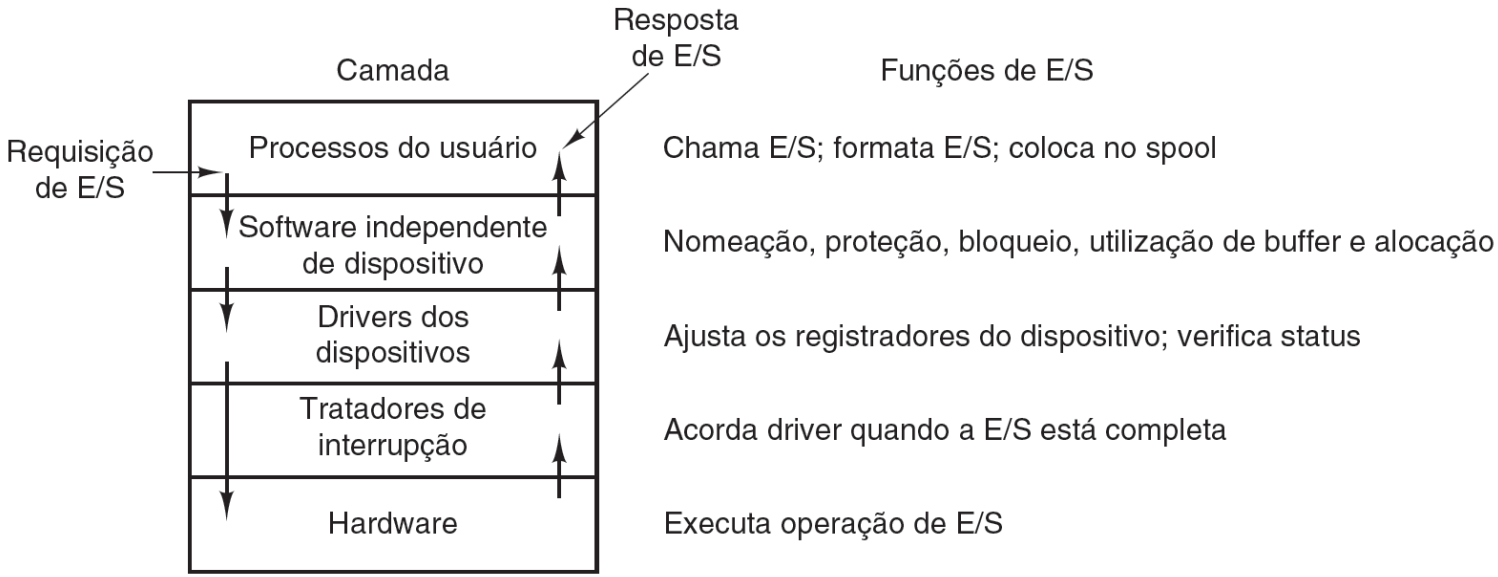

Figura 5.15 Camadas do sistema de E/S e as principais funções de cada camada.

Exemplo de fluxo de controle

1. Programa do usuário quer ler um bloco de um arquivo, requisita o SO para realizar uma chamada de E/S;
2. SW independente de dispositivo procura bloco na *cache do buffer*;
3. Se bloco não está lá, ele chama o *driver* para emitir uma requisição a fim de que o HW obtenha o bloco no disco;
4. Processo é bloqueado até operação do disco terminar e dados estejam disponíveis no *buffer*;
5. Quando disco termina, HW gera uma interrupção ⇒ tratador de interrupção é executado;
6. Tratador obtém o status do dispositivo e acorda o processo que estava dormindo para finalizar a requisição de E/S; processo prossegue....