
# Organização de Memória

objetivos

**Após estudar este capítulo, você deverá ser capaz de:**

- Descrever os diversos tipos de memória de computador
- Definir o conceito de hierarquia de memória
- Expressar a relação entre caching e hierarquia de memória
- Explicar o funcionamento básico de um disco magnético
- Fazer distinção entre programa baseado em processamento e programa baseado em entrada e saída
- Discutir a diferença entre análise de algoritmo em memória interna e análise de algoritmo em memória externa
- Descrever o conceito de localidade de referência e saber usá-lo na prática para melhorar a eficiência de um programa

ESTE CAPÍTULO descreve conceitos e tecnologias relacionados com armazenamento de dados que são importantes para o entendimento de requisitos de implementação de algoritmos e estruturas de dados em diferentes meios de armazenamento.

O conhecimento coberto neste capítulo é importante por duas razões principais: (1) ele mostra que a eficiência de algoritmos que processam dados em memória secundária deve ser julgada de modo bem diferente da eficiência de algoritmos que processam dados contidos em memória principal e (2) ele ensina ao programador como utilizar o conhecimento adquirido sobre hierarquias de memória para melhorar o desempenho de programas.

# 1.1 Meios de Armazenamento

Esta seção descreve os meios de armazenamento mais comuns. Em resumo, as tecnologias básicas de armazenamento discutidas nesta seção são:

- Memória RAM estática (SRAM) é um tipo de memória volátil, cara e extremamente rápida. Esse tipo de memória é usado em memórias cache que podem ou não fazer parte de uma CPU.
- Memória RAM dinâmica (DRAM) é um tipo de memória volátil, bem mais barata e lenta do que memória SRAM (embora DRAM seja ainda bastante rápida). Esse tipo de memória é comumente usada em memória principal e em placas de vídeo.
- Memória ROM é uma memória não volátil tipicamente usada para armazenamento de *firmware*.
- Discos magnéticos (HD) constituem uma memória com grande capacidade de armazenamento de dados. Eles são relativamente baratos, mas muito lentos quando comparados aos demais meios de armazenamento.
- Discos SSD e bastões de USB (*pen drive*) são memórias *flash* não voláteis que podem servir como alternativas para discos magnéticos.

Registradores constituem o meio de armazenamento mais rápido de todos e serão discutidos na **Seção 1.3**.

## 1.1.1 Memória RAM

**Memória RAM**[1] é um tipo de memória que não envolve componentes mecânicos (como ocorre com um disco, por exemplo) e que permite acesso direto aos dados que ela armazena.

Fisicamente, existem dois tipos de memória RAM:

1. **Memória RAM estática** (**SRAM**) é o tipo de memória RAM mais rápido, mas também é o mais caro. Esse tipo de memória é usado para memórias cache internas ou externas a uma CPU.
2. **Memória RAM dinâmica** (**DRAM**). Normalmente, esse é o tipo de memória que compõe a memória principal de um computador e também é usada em placas de vídeo.

Chips de memória DRAM se apresentam em módulos que se encaixam na placa mãe de um computador. A **Figura 1–1** mostra um **módulo de memória** DRAM, popularmente conhecido como **pente de memória** devido à sua aparência similar a um pente. Atualmente, esses módulos de memória possuem 168 pinos (**DIMM**) ou 72 pinos (**SIMM**). A transferência de dados entre o controlador de memória e um módulo DRAM com pinos DIMM é efetuada com pacotes de 64 bits, enquanto, quando pinos SIMM são usados, essa transferência se dá em pacotes de 32 bits.

[1] Em língua inglesa, a denominação *RAM* é derivada, por razões históricas, de *random access memory*, que, traduzido literalmente para português, significa memória de acesso aleatório (ou randômico). Mas tanto *random* em inglês quanto *aleatório* em português são qualificações enganosas para esse tipo de memória. Quer dizer, não existe nada de aleatório no sentido de fortuito em memórias RAM, de modo que a melhor tradução para *random access memory* é realmente memória de acesso direto. A propósito, o único meio de armazenamento discutido (brevemente) neste livro que não permite acesso direto são as fitas magnéticas.

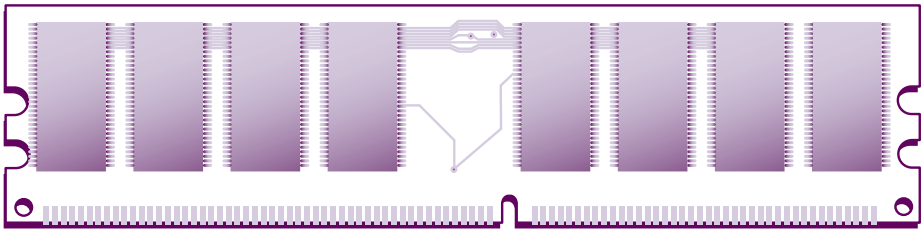

**Figura 1–1: Módulo de Memória DRAM**

Tipicamente, um computador pessoal possui alguns poucos megabytes de memória SRAM e entre 4 e 16 gigabytes de memória DRAM. Memória SRAM é cerca de dez vezes mais rápida e cem vezes mais cara do que memória DRAM. Memória SRAM não é sensível a distúrbios elétricos, como ocorre com memória DRAM. Em compensação, memória SRAM consome mais energia do que memória DRAM.

Memória RAM é **volátil** no sentido de que todos os dados que nela se encontram são perdidos caso ela deixe de estar energizada.

A **Tabela 1–1** apresenta uma breve comparação entre memórias SRAM e DRAM.

| Tipo de Memória | Prós | Contras | Uso |
|---|---|---|---|
| **SRAM** | ❒ Dez vezes mais rápida<br>❒ Insensível a distúrbios elétricos | ❒ Cem vezes mais cara<br>❒ Maior consumo de energia | ❒ Memória cache |
| **DRAM** | ❒ Mais barata<br>❒ Menor consumo de energia | ❒ Mais lenta<br>❒ Sensível a distúrbios elétricos | ❒ Memória principal<br>❒ Placa de vídeo |

**Tabela 1–1: Comparação entre Memórias SRAM e DRAM**

### 1.1.2 Memórias ROM, PROM, EPROM e EEPROM

Memórias **PROM**, **EPROM** e **EEPROM** são utilizadas principalmente para armazenamento de programas de **firmware**, que são programas que controlam ou monitoram certos sistemas. Por exemplo, computadores, periféricos e muitos eletrodomésticos usam *firmware*. Num computador pessoal, as funções básicas de entrada e saída (BIOS) são armazenadas em *firmware*.

Esses tipos de memória se distinguem pelo número de vezes que permitem ser escritas

- ❒ Memória PROM permite uma única escrita.
- ❒ Memória EPROM permite várias escritas (tipicamente, cerca de 1000).
- ❒ Memória EEPROM é semelhante a memória EPROM, mas, ao contrário desse último tipo de memória, ela não requer um dispositivo isolado para ser reprogramada. Em compensação, a capacidade de reescrita de uma memória EEPROM é bem menor do que a de uma memória EPROM.

Coletivamente, por razões históricas, esses tipos de memórias são conhecidos como memórias **ROM**[2]. Elas constituem um tipo de memória **não volátil**, de modo que os dados que elas armazenam persistem na ausência de energia.

Normalmente, memórias ROM não são de interesse em programação de alto nível, de modo que este livro não fará mais referência a elas.

[2] A denominação ROM é derivada de *read only memory* em inglês, que significa memória apenas para leitura (i.e., memória que não permite escrita). Novamente, essa denominação é enganosa como foi discutido acima.

### 1.1.3 Discos Magnéticos

Existem programas (p. ex., alguns bancos de dados, programas que analisam dados científicos) que lidam com enorme quantidade de dados que não podem ser mantidos em memória principal. Esses programas armazenam seus dados em memória secundária, que tipicamente, é representada por um ou mais discos magnéticos.

#### *Características Físicas Importantes*

Um disco magnético é constituído por **pratos** montados uns sobre os outros como mostra a **Figura 1–2**. Cada prato de um disco possui uma **superfície** de cada lado coberta com material magnetizável. Esse material é capaz de armazenar informação de modo não volátil. O disco ilustrado na **Figura 1–2** possui três pratos e seis superfícies. Em geral, um disco com *n* pratos possui *2n* superfícies.

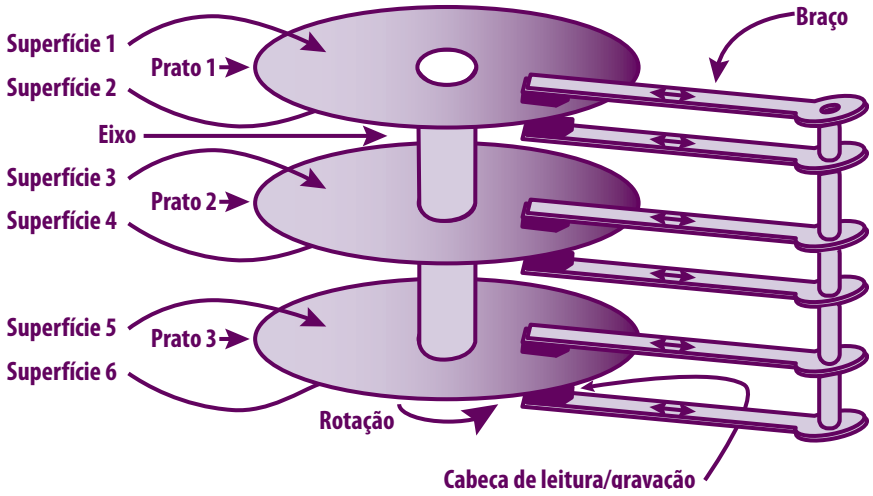


**Figura 1–2: Disco Magnético com Três Pratos e Seis Superfícies**

Durante uma operação de acesso aos dados de um disco magnético, um **eixo** rotatório gira seus pratos a uma taxa constante, que, tipicamente, varia entre *5.400* e *15.000* rotações por minuto (**RPM**). Um **braço** contém em sua extremidade mais próxima ao disco uma **cabeça** responsável por leitura e escrita de dados numa superfície do disco. Existe uma cabeça para cada superfície do disco e isso significa que o número de cabeças (e braços) de um disco é igual ao número de superfícies do disco. As cabeças são alinhadas verticalmente e os braços se movem para frente e para trás na direção indicada pelas setas duplas da **Figura 1–2** e de modo uníssono. Quer dizer, todos os braços se movem de modo harmônico, de modo que, em qualquer instante, todas as cabeças estão posicionadas no mesmo cilindro imaginário (v. adiante). As cabeças de leitura e escrita flutuam sobre um colchão de ar extremamente fino com cerca de *0,1* mícron de espessura.

Quando os braços estão estacionados enquanto o disco gira, cada cabeça traça um círculo imaginário concêntrico em sua respectiva superfície. Esse círculo é chamado **trilha**. As trilhas verticalmente alinhadas a uma dada posição de braço constituem um **cilindro**. Apenas uma cabeça de leitura/escrita de cada vez atua na transmissão de dados entre o disco e a memória principal.

O conjunto formado pelas peças que aparecem na **Figura 1–2** é hermeticamente envolvido por um invólucro metálico (que não aparece na referida figura).

O tipo de disco magnético que acaba de ser descrito é popularmente conhecido como **disco rígido** ou, simplesmente, **HD**.

### *Geometria de Superfície*

A **Figura 1–3** apresenta esquematicamente a geometria de uma superfície de disco magnético. Cada superfície é dividida em anéis concêntricos denominados **trilhas**. Cada trilha é dividida em **setores**.

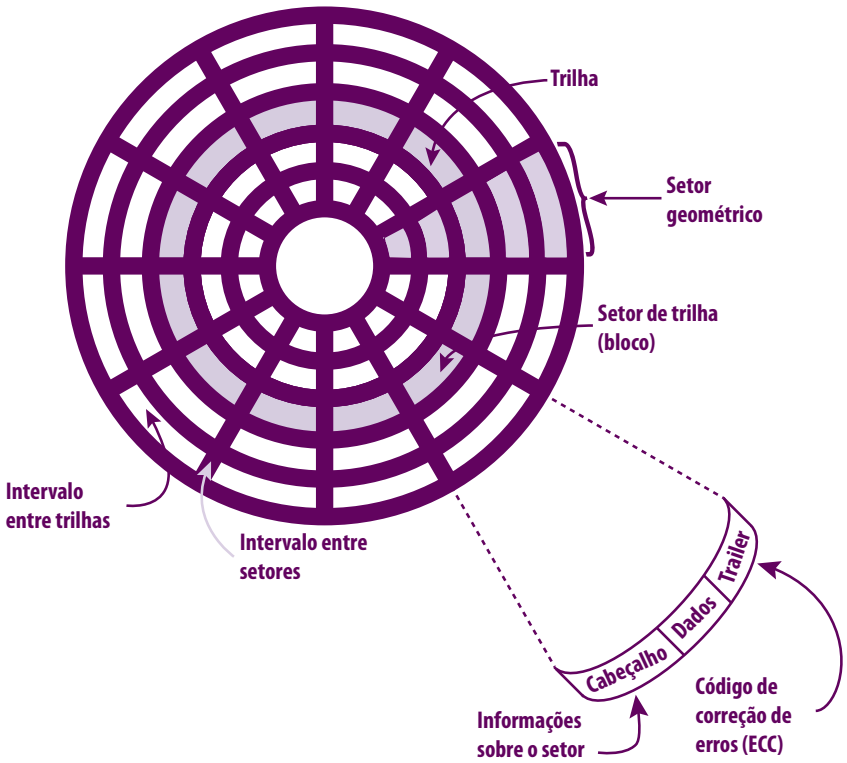


**Figura 1–3: Geometria de Superfície de Disco Magnético**

Cada setor é constituído de três partes básicas: **cabeçalho**, **área de dados** e **extrato** (*trailer*, em inglês).

O cabeçalho de um setor contém informações sobre o próprio setor, que, resumidamente, são as seguintes:

- **Identificação do setor**, que contém um número que identifica o setor e sua localização no disco. A identificação de endereço é usada para assegurar que o disco posicionará a cabeça de leitura/gravação sobre a localização correta.
- **Informação sobre sincronização**, que é usada pelo controlador de disco (v. adiante) para guiar os processos de leitura e escrita.

O cabeçalho pode também incluir um endereço alternativo para ser usado se a área de dados deixar de ser confiável.

O extrato (*trailer*) de um setor contém um **código de correção de erro** (**ECC** — sigla em inglês) que assegura a integridade dos dados. Quer dizer, esse código é baseado na área de dados e é usado para checar e possivelmente corrigir erros que possam ter sido introduzidos nos dados.

A área de dados de um setor contém os dados que efetivamente são úteis para o usuário. Obviamente, quanto maior for o espaço ocupado pelas demais informações, menor será o espaço deixado para armazenar dados úteis para processamento.

**Intervalos** (ou **espaçadores**) são áreas de separação entre trilhas e setores. Esses intervalos, que são desproporcionalmente representados na **Figura 1–3**, não armazenam dados, mas contêm bits que os identificam.

Como foi visto, no contexto de discos magnéticos, setor é uma subdivisão de uma trilha, mas se deve tomar cuidado para não confundir *setor de trilha* com *setor geométrico*. Geometricamente, um setor é a parte de um círculo delimitada por dois raios do círculo e um arco que intercepta esses raios. Em outras palavras, um setor é análogo a uma fatia de pizza [v. **Figura 1–4 (a)**]. Por outro lado, um setor de trilha (doravante, denominado apenas **setor**) é a interseção de um setor geométrico com uma trilha [v. **Figura 1–4 (b)**]. Frequentemente, setor é também denominado *bloco*.

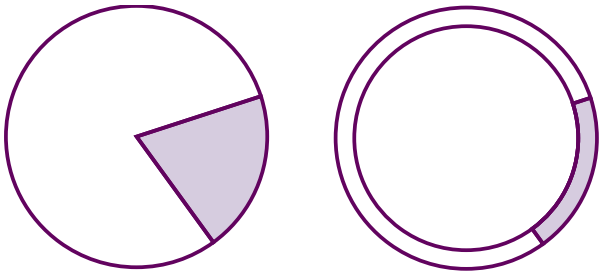


**Figura 1–4: Setor Geométrico e Setor de Trilha**

## *Capacidade*

A **capacidade** de um disco é o número máximo de bytes que podem ser armazenados nele e é determinada pelos seguintes fatores:

- **Densidade de gravação**. Essa propriedade é medida em bits por polegada e representa o número de bits que podem ser armazenados numa polegada de trilha[3].
- **Densidade de trilha**, que é o número de trilhas que cabe numa polegada de raio que parte do centro de uma superfície. Essa propriedade é medida em bits por polegada.
- **Densidade de área**, que é determinado pelo produto da densidade de gravação pela densidade de trilha e é medida em bits por polegada ao quadrado.

Na prática, a capacidade de um disco é obtida por meio da fórmula:

$$capacidade = \frac{bytes}{setor} \times \frac{setores}{trilha} \times \frac{trilhas}{superfície} \times \frac{superfícies}{prato} \times \frac{pratos}{disco} \qquad \textbf{Fórmula 1–1}$$

**Exemplo 1.1** Suponha que um disco tenha as seguintes características:

| | |
|---|---|
| Número de pratos: | 3 |
| Número de bytes por setor: | 512 |
| Número de trilhas por superfície: | 20.000 |
| Número de setores por trilha: | 300 |
| Número de superfícies por prato: | 2 |

Então, usando-se a **Fórmula 1–1**, esse disco terá a seguinte capacidade:

$$capacidade = 512 \times 300 \times 20000 \times 2 \times 3 = 18432000000 = 18.432\ GB$$

Fabricantes de discos consideram 1 GB igual a $10^9$ bytes (em vez de $2^{30}$) e esse fato foi levado em consideração no exemplo acima.

[3] Uma polegada vale 2,54 centímetros.

Antes que um disco possa ser usado ele precisa ser **formatado** pelo controlador do disco. Nesse processo, os intervalos entre setores (v. **Seção 1.2**) são preenchidos com informações que identificam os setores. No processo de formatação, também são identificados cilindros defeituosos, que são logicamente excluídos. Além disso, alguns cilindros saudáveis são logicamente separados dos demais para atuarem como sobressalentes. Por causa desses cilindros defeituosos e sobressalentes, a capacidade de um disco formatado é menor do que aquela anunciada pelo fabricante do disco.

### *Acesso a Setores*

O acesso aos dados armazenados nas superfícies magnéticas de um disco é efetuado por meio de suas cabeças de leitura e escrita. Movendo os braços para frente e para trás em direção ao eixo do disco, essas cabeças podem alcançar qualquer trilha de uma superfície, como ilustra a **Figura 1–5**. Assim, para acessar (i.e., ler ou escrever) dados num determinado setor de um disco, primeiro, o braço deve ser movido de modo que ele seja posicionado sobre a trilha correta e então deve esperar que o setor apareça sob ele enquanto o disco gira.

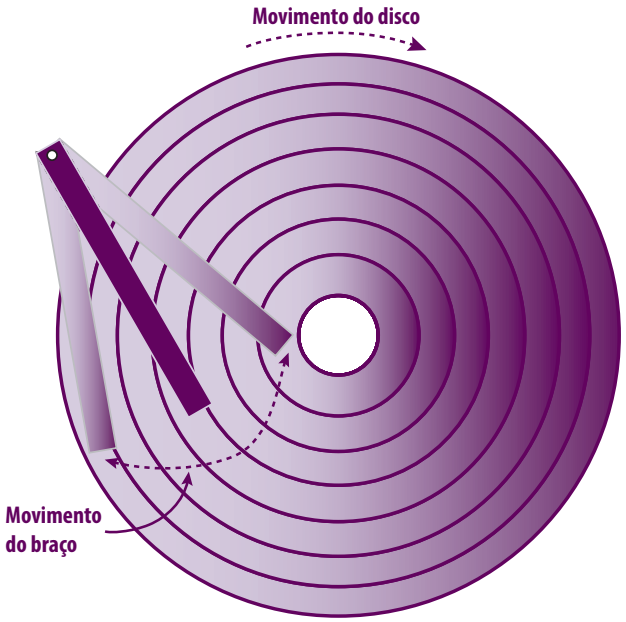


**Figura 1–5: Acesso a Blocos num Disco Magnético**

### *Medidas de Desempenho*

**Tempo de acesso** é o intervalo de tempo decorrido desde o instante em que uma solicitação de leitura ou escrita é emitida até o instante em que a transferência de dados é iniciada. Para **armazenar** (i.e., escrever) ou **recuperar** (i.e., ler) dados em algum local de um disco, as cabeças de leitura/escrita devem posicionar-se mecanicamente sobre o cilindro adequado e esperar que o setor desejado passe por uma cabeça. Leitura ou escrita num disco é efetuada em blocos do tamanho de um setor, que, num disco formatado, contém tipicamente 512 bytes ou 4 KiB de dados. O tempo de acesso depende dos seguintes intervalos de tempo:

- **Tempo de posicionamento**[4] ($T_{posicionamento}$) é o tempo necessário para posicionar a cabeça de leitura e escrita sobre a trilha que contém o setor desejado. Esse tempo depende da posição inicial da cabeça e da velocidade de movimentação do braço. Nos dias atuais, o tempo médio de posicionamento varia entre 3 ms e 9 ms, mas pode chegar a 20 ms, dependendo da distância em que uma trilha se encontra com relação à posição inicial do braço e do modelo de disco. Discos menores tendem a apresentar

[4] *Seek time*, em inglês.

menores tempos de posicionamento, visto que a cabeça de leitura/gravação precisa percorrer uma distância menor. O tempo de posicionamento pode ser economizado se o próximo acesso ocorrer no mesmo cilindro em que se encontra correntemente o braço do mecanismo.

- **Atraso rotacional** (ou **latência rotacional** — $T_{latência}$) é o tempo de espera pela passagem do primeiro bit do setor desejado pela cabeça de leitura e gravação depois de o braço ter sido movido para a trilha desejada. Esse tempo depende da posição da superfície do disco no instante em que a cabeça atinge a trilha que contém o setor desejado e da velocidade de rotação do disco. No pior caso, nesse instante, o setor desejado acabou de passar pela referida cabeça e é preciso esperar por uma rotação completa do disco para que ele passe novamente pela mesma posição. Em média, a metade de uma rotação do disco é requerida para que o início do setor desejado apareça sob a cabeça de leitura/gravação. Assim o **tempo médio de latência** de um disco é a metade do tempo de uma rotação completa dele. Uma vez que o primeiro setor dos dados que serão acessados apareça sob a cabeça de leitura/gravação, a transferência de dados começa.
- O **tempo de transferência** ($T_{transferência}$) depende da velocidade rotacional e do número de setores por trilha, de modo que se pode calcular seu valor, em segundos, como:

$$T_{transferência} = \frac{1}{RPM} \times \frac{1}{setores\ por\ trilha} \times 60$$

A latência de acesso, que é uma combinação de tempo de posicionamento e latência rotacional, é tipicamente da ordem de vários milissegundos. Por outro lado, leva menos do que um nanossegundo para acessar registradores e memória cache L1 (i.e., esse acesso é mais de um milhão de vezes mais rápido do que acesso a um disco magnético).

O tempo médio de acesso ao conteúdo de um setor de disco, que varia de 8 a 20 milissegundos, pode ser calculado como a soma do tempo médio de posicionamento, da latência rotacional e do tempo médio de transferência. Portanto, formalmente, tem-se que o tempo médio de acesso pode ser calculado usando-se a fórmula:

$$T_{acesso} = T_{posicionamento} + T_{latência} + T_{transferência} \qquad \textbf{Fórmula 1–2}$$

**Exemplo 1.2** Nos dias atuais, um disco gira a uma velocidade angular constante entre *5.400* e *15.000* rotações por minuto (RPM). Suponha que um disco gire a *12.000* RPM. Então isso corresponde a *200* rotações por segundo, o que significa que um setor passa pela cabeça de leitura/escrita 200 vezes a cada segundo ou que o atraso rotacional é *0,005* segundos (ou *5 ms*). Nesse exemplo, a latência rotacional é de *2,5 ms*.

**Exemplo 1.3** Considere um disco com as seguintes características:

| | |
|---|---|
| Taxa de rotação (RPM): | 10000 RPM |
| Tempo médio de posicionamento ($T_{posicionamento}$): | 9 ms |
| Número de setores por trilha: | 400 |
| Número de superfícies por prato: | 2 |

Então esse disco apresentará a seguinte latência rotacional:

$$T_{latência} = 1/2 \times 60/10000\ s = 0{,}003\ s = 3{,}0\ ms$$

O tempo médio de transferência desse disco é dado por:

$$T_{transferência} = 60/10000 \times 1/400\ s = 1{,}5 \times 10^{-5}\ s = 0{,}015\ ms$$

Usando a **Fórmula 1–2**, a estimativa de tempo de acesso para o disco deste exemplo é:

$$T_{acesso} = T_{posicionamento} + T_{latência} + T_{transferência} = 9 + 3{,}0 + 0{,}015 = 12{,}01\ ms$$

Esse último exemplo mostra que o tempo de acesso a um setor (*12,01 ms* no exemplo) é dominado pelo tempo de posicionamento (*9 ms*) e pela latência rotacional (*3 ms*). Ou seja, o tempo de transferência (*0,015 ms* no exemplo) tem importância relativamente desprezável nesse processo. Além disso, o acesso ao primeiro byte de um bloco é relativamente lento, mas, em compensação, os demais bytes do mesmo bloco são acessados num intervalo de tempo ínfimo.

**Exemplo 1.4** O tempo de acesso a uma palavra de *32* bits armazenada em memória cache (SRAM) é aproximadamente *4 ns* (ou $4 \times 10^{-6}$ *ms*), enquanto acessar uma palavra do mesmo tamanho armazenada em memória principal (DRAM) leva cerca de *60 ns* (ou $6 \times 10^{-5}$ *ms*). Assim para ler um bloco de 4 KiB (*1024* palavras de *32* bits) em memória cache leva-se cerca de *0,004 ms* e para ler essa mesma quantidade em memória principal leva-se cerca de *0.061 ms*.

Considerando-se um tempo aproximado de acesso a um bloco de disco igual a *12 ms* (como no exemplo anterior), tem-se que esse disco leva quase *3.000* vezes mais tempo para ler a mesma quantidade de dados que uma memória SRAM e cerca de *200* vezes mais tempo para ler a mesma quantidade de dados que uma memória DRAM.

Quando a cabeça de leitura e escrita está posicionada sobre a trilha desejada, à medida que cada bit passa sob a cabeça, ora ele é obtido (em caso de leitura), ora ele é alterado (em caso de escrita). De qualquer modo, o tempo para localizar e ler um bloco em memória secundária é muito grande comparado com o tempo gasto para processá-lo em memória principal.

**Exemplo 1.5** Suponha que a largura do tipo **int** seja *32* bits numa dada implementação de C. O tempo gasto para ler um bloco contendo *1024* elementos do tipo **int** de um array armazenado num disco com rotação de 10000 RPM é em média igual a *12 ms*, como foi visto nos dois últimos exemplos. Nesse intervalo de tempo, uma CPU moderna de alto desempenho é capaz de executar cerca de 8.000.000 (oito milhões) de instruções. Esse tempo é muito mais do que necessário para processar todos os elementos do array (p. ex., somando-os ou ordenando-os).

Apesar de discos serem mais baratos e terem maior capacidade do que memória principal, eles são muito mais lentos devido às suas partes mecânicas móveis. O movimento mecânico tem dois componentes: rotação de prato e movimento de braço. Correntemente, discos comerciais giram a velocidades entre *5.400* e *15.000* rotações por minuto (RPM). Embora *15.000* RPM possa parecer rápido, nesse caso, uma rotação leva *4* milissegundos, que é muito maior do que o tempo de acesso de *60* nanossegundos comumente encontrado em memórias DRAM. Ou seja, no intervalo de tempo de espera por uma rotação completa para um determinado item passar sob a cabeça de leitura/gravação, pode-se acessar a memória principal mais de *60.000* vezes durante esse intervalo. Em média, tem-se que esperar por apenas metade da rotação, mas, ainda assim, a diferença em tempo de acesso de memória de silício comparado com discos magnéticos é enorme. Mover os braços que contêm as cabeças de leitura e gravação também leva um tempo considerável. Correntemente, em discos comerciais, o tempo gasto para completar esse movimento está no intervalo entre 8 e 11 milissegundos.

A **taxa de transferência de dados** é a taxa com a qual os dados podem ser lidos ou armazenados num disco. Discos atuais suportam taxas de transferência máximas de 25 até 100 megabytes por segundo. Taxas de transferência para trilhas internas de um disco são significativamente menores do que as taxas de transferência máximas

do mesmo disco, visto que essas trilhas possuem menos setores. Por exemplo, um disco com a taxa máxima de transferência de 100 megabytes por segundo pode ter uma taxa de transferência de cerca de 30 megabytes por segundo em suas trilhas internas.

Outra medida de desempenho comumente usada para discos é o **tempo médio de falha** (**MTTF**[5]), que é uma medida de confiabilidade do disco. O tempo médio de falha de um disco mede quanto tempo, em média, se pode esperar que o disco seja usado continuamente sem qualquer falha. De acordo com afirmações de fabricantes, o tempo médio de falha de discos hoje em dia varia de 500.000 a 1.200.000 horas — cerca de 57 a 136 anos. Na prática, o tempo médio de falha assegurado para um disco é calculado sobre uma probabilidade de falha quando ele é novo. Por exemplo, dados 1.000 discos relativamente novos, se o MTTF deles for 1.200.000 horas, em média, um deles irá falhar em 1200 horas. Portanto um tempo médio de falha de 1.200.000 horas de um disco não quer dizer que se possa esperar que ele funcione por 136 anos.

### *Armazenamento de Arquivos*

**Cluster** é o menor espaço que pode ser alocado para conter um arquivo. Ou seja, *cluster* é o conjunto de setores que constitui a menor unidade de alocação capaz de ser endereçada num disco magnético. Entretanto os setores que constituem um *cluster* não precisam ser necessariamente contíguos em disco. O armazenamento de arquivos num sistema de arquivos que usa tamanhos muito grandes de *cluster* causa desperdício de espaço (i.e., gera **espaço ocioso**). Quanto maior for o tamanho de um *cluster* maior será o desperdício de espaço em disco, mas, em compensação, um *cluster* maior reduz a manutenção e a fragmentação, o que, em consequência, reduz o tempo de acesso aos arquivos. O tamanho típico de um *cluster* varia de um setor (*512* bytes ou *4* KiB) a *128* setores (*64* KiB).

Um bloco é uma unidade lógica consistindo de um número fixo de setores contíguos e sistemas de arquivos dividem memória secundária em blocos do mesmo tamanho, que varia entre 512 bytes e 4 KB. Neste contexto, bloco é uma sequência de bytes que apresenta certo tamanho. O tamanho de um bloco é prefixado pelo sistema de arquivos e não pode ser alterado. Num sistema de arquivos, o tamanho de um bloco pode ser igual ou múltiplo do tamanho físico de um bloco.

Fisicamente, um **arquivo** é uma sequência de blocos físicos e o custo de processamento dele é determinado pelo número de blocos lidos no arquivo e armazenados em memória principal ou lidos em memória principal e escritos no arquivo. Idealmente, cada bloco contém um número inteiro de registros[6], o que pode causar desperdício de espaço, mas, em compensação, isso evita que um registro faça parte de dois blocos.

Um bloco pode conter uma parte de um arquivo, o que causa ineficiência de uso de espaço devido à fragmentação, pois nem sempre um arquivo tem tamanho que é múltiplo do tamanho de um bloco. Portanto os últimos blocos de um arquivo são parcialmente preenchidos, acarretando espaço ocioso. Em média, metade de um bloco por arquivo é espaço ocioso.

### *Otimização de Acesso a Blocos*

Solicitações para operações de entrada e saída (E/S) num disco são geradas pelo sistema operacional. Cada solicitação especifica o número (endereço) do bloco a ser acessado no disco. Dados são transferidos entre disco e memória principal em **blocos**, que se tornam, assim, **unidades de transferência**. O termo **página** é frequentemente usado em lugar de *bloco*, embora em alguns contextos esses termos possam se referir a conceitos diferentes.

Uma sequência de solicitações de blocos num disco pode ser classificada como um **padrão de acesso sequencial** ou um **padrão de acesso direto**. Num padrão de acesso sequencial, solicitações sucessivas são referentes

[5] Em inglês, esse acrônimo significa *Mean Time To Failure*.

[6] Um registro é uma subdivisão lógica de um arquivo, como será visto na **Seção 2.11.2**.

a números sucessivos de blocos, que estão na mesma trilha ou em trilhas adjacentes. Para acessar blocos sequencialmente, um posicionamento da cabeça de leitura/gravação pode ser requerido para o primeiro bloco, mas solicitações sucessivas não requerem outro posicionamento ou requererem um posicionamento para uma trilha adjacente, o que é bem mais rápido do que um posicionamento para uma trilha que está mais distante.

Em contraste, num padrão de acesso direto, solicitações sucessivas se referem a blocos que estão aleatoriamente distribuídos no disco[7], de sorte que cada tal solicitação pode requer um posicionamento bem diferente da cabeça de leitura/escrita. O número de acessos diretos a blocos que pode ser satisfeito por um disco num segundo depende do tempo de posicionamento e é tipicamente cerca de 100 a 200 acessos por segundo. Como apenas uma pequena quantidade de dados (i.e., exatamente um bloco) é acessada em cada posicionamento, a taxa de transferência é significativamente menor para um padrão de acesso direto do que para um padrão de acesso sequencial.

Várias técnicas foram desenvolvidas para acelerar o acesso a blocos, dentre as quais:

- **Buffering**. Blocos que são lidos de um disco são armazenados temporariamente num buffer em memória principal para satisfazer futuras solicitações. Buffering é levado a efeito pelo sistema operacional. O funcionamento de buffering é semelhante ao funcionamento básico de caching (v. **Seção 1.4**), mas, rigorosamente, esses dois conceitos são diferentes.
- **Leitura antecipada**. Quando um bloco é acessado em disco, blocos consecutivos da mesma trilha são lidos para um buffer em memória principal, mesmo que não haja solicitação pendente para outros blocos. No caso de acesso sequencial, essa leitura antecipada assegura que muitos blocos já estejam em memória quando eles forem requisitados e isso minimiza o tempo gasto em posicionamento e latência rotacional por bloco lido. Leitura antecipada, entretanto, não é muito útil para acesso direto.
- **Organização de arquivo**. Para reduzir o tempo de acesso aos blocos que constituem um arquivo, eles podem ser organizados em disco de uma maneira que corresponda aproximadamente ao modo que se espera que os dados do arquivo sejam acessados. Por exemplo, quando se espera que um arquivo seja acessado sequencialmente, deve-se idealmente manter todos os blocos do arquivo em cilindros adjacentes. Sistemas operacionais modernos ocultam a organização de discos de seus usuários e gerenciam a alocação de blocos internamente. No decorrer do tempo, os blocos de um arquivo podem se tornam espalhados no disco, o que é denominado **fragmentação**. Para reduzir fragmentação, alguns sistemas proveem utilidades que efetuam varreduras no disco e reposicionam blocos para melhorar o desempenho no acesso aos arquivos.

### 1.1.4 Memórias Flash

**Memória flash** é um tipo de memória não volátil cuja tecnologia é baseada em EEPROM (v. **Seção 1.1.2**). Recentemente, esse tipo de memória tem sido empregado em vários equipamentos, incluindo câmeras digitais, telefones celulares e computadores. Além disso, um novo tipo de disco denominado **disco de estado sólido** (**SSD**) tem sido cada vez mais utilizado como memória secundária em computadores pessoais.

#### *Discos SSD*

Um disco de estado sólido (SSD) é uma tecnologia de armazenamento de dados que consiste em chips de memória flash (v. **Seção 1.1.2**). De fato, essa tecnologia não tem nada a ver com disco (pelo menos, no sentido geométrico). Quer dizer, um *disco* SSD não contém nenhum dos componentes descritos na **Seção 1.1.3**. Ele é conectado ao barramento de entrada e saída e comporta-se como um disco rígido, sendo que os chips de memória flash substituem as partes mecânicas de um HD. Assim um disco SSD funciona como um disco rígido em muitas situações e, como a expressão disco SSD já se tornou popular, ela continuará a ser utilizada aqui.

[7] Por isso, acesso direto é frequentemente denominado **acesso aleatório**.

A **Figura 1–6** apresenta esquematicamente um disco SSD. Nessa figura, a camada de tradução flash é um dispositivo que traduz solicitações de acesso a blocos em acessos aos chips de memória flash.

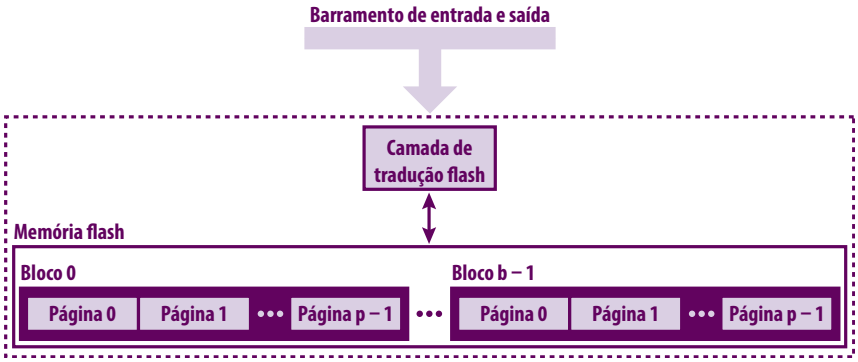


**Figura 1–6: Configuração de um Disco SSD**

Discos SSD apresentam desempenhos diferentes de discos magnéticos. Quando blocos são acessados sequencialmente, a escrita é ligeiramente mais lenta do que a leitura, mas quando o acesso é direto, operações de escrita são cerca de dez vezes mais lentas do que operações de leitura. Isso ocorre devido ao modo de funcionamento dessas operações nessa tecnologia.

Como mostra a **Figura 1–6**, a memória flash que constitui um disco SSD consiste em *b* blocos, cada um dos quais contendo *p* páginas. O tamanho de uma página varia entre *512* bytes e *4* KB e o número de blocos varia entre *32* e *128* páginas, com tamanhos de bloco variando entre *16* KB e *512* KB. A unidade de acesso a um disco é a página. Isso significa que uma página inteira é lida ou escrita em cada operação elementar de entrada ou saída de dados num disco SSD. Agora, uma página só pode ser escrita após o bloco à qual ela pertence ter sido apagado. No entanto, uma vez que um bloco tenha sido apagado, todas as páginas que fazem parte desse bloco podem ser escritas sem que seja preciso apagar o bloco novamente. A escrita num disco SSD é mais demorada do que a leitura porque apagar um bloco antes de escrever nele leva mais tempo do que acessá-lo.

Um bloco de um disco SSD torna-se desgastado após aproximadamente 100.000 operações de escrita. Quando isso ocorre, o bloco não pode mais ser utilizado.

Discos SSD apresentam vantagens e desvantagens com relação a discos magnéticos. Como discos SSD não possuem componentes mecânicos e por isso oferecem acessos mais rápidos do que discos magnéticos e consomem menos energia. Por outro lado, a principal desvantagem de discos SSD é que eles se desgastam mais rapidamente com o tempo de uso do que HDs. Além disso, o custo por byte de um disco SSD ainda é muito maior do que o custo correspondente de um disco rígido.

Assim como discos magnéticos, discos SSD também podem falhar. O problema com discos SSD é que eles raramente dão sinal de que irão falhar. Quer dizer, a falha repentina de um disco SSD é mais provável do que no caso de discos rígidos que frequentemente sinalizam (p. ex., por meio de ruídos e lentidão) que irão falhar em breve.

### *Pen Drive*

Um **bastão de memória**, popularmente conhecido e doravante referido como **pen drive**, é mostrado na **Figura 1–7**. Esse é um dispositivo de armazenamento de dados portátil que inclui memória flash com uma interface USB integrada. Como são portáteis, eles são tipicamente usados para backups e transferência de arquivos entre computadores.


**Figura 1–7: Bastão de Memória USB (Pen Drive)**

Como esses dispositivos são constituídos de memória flash, eles apresentam as mesmas vantagens e desvantagens que os discos SSD. Eles apresentam como vantagem adicional o fato de serem portáteis e apresentarem pequenas dimensões. Mas essa mesma vantagem pode ser uma desvantagem, visto que, devido às suas dimensões reduzidas, eles são facilmente perdidos, esquecidos ou furtados. Pen drives usam estruturas de blocos semelhantes àquelas vistas para discos SSD.

### 1.1.5 Fitas Magnéticas

**Fitas magnéticas** constituem um dos mais antigos meios de armazenamento. Antigamente, essas fitas eram usadas como memória secundária com a mesma intensidade que os atuais discos magnéticos. Hoje em dia, fita magnética é mais utilizada como meio de armazenamento de reserva (**backup**) para grandes volumes de dados, de modo que raramente um programador precisa se preocupar em como manipular dados armazenados em fita.

Fitas magnéticas continuam sendo uma alternativa para meios de armazenamento mais modernos devido ao seu baixo custo por byte, o que constitui uma grande vantagem quando se lidam com enormes quantidades de dados. Por outro lado, fitas magnéticas são mais lentas do que discos magnéticos.

## 1.2 Acesso a Dispositivos de Entrada e Saída

### 1.2.1 Arquitetura Básica de um Computador

A **Figura 1–8** mostra de modo simplificado a arquitetura clássica de um computador. Os componentes principais mostrados na **Figura 1–8** são: a **unidade central de processamento** (**CPU**), a **ponte de entrada e saída** e a **memória principal**, composta de módulos DRAM. Esses componentes são conectados por dois barramentos: (1) o **barramento de sistema**, que conecta a CPU com a ponte de entrada e saída, e (2) o **barramento de memória**, que conecta a ponte de entrada e saída com a memória principal.

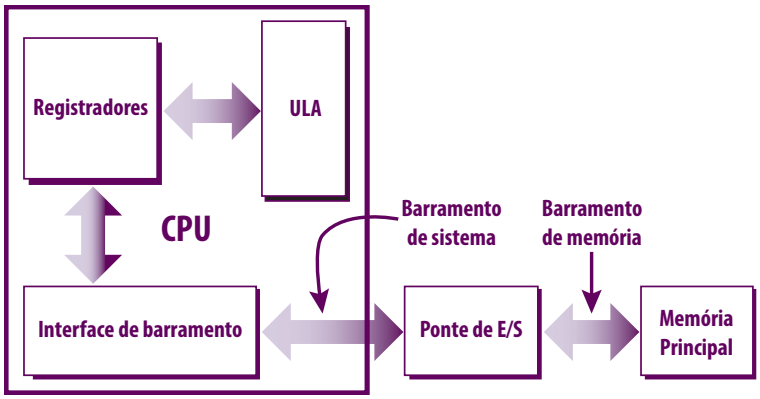


**Figura 1–8: Acesso a Memória Principal**

Um **barramento** é um conjunto de fios elétricos paralelos que interliga componentes de um computador e através do qual fluem dados, endereços e sinais de controle. Um único barramento pode ser compartilhado por dois ou mais dispositivos.

A ponte de entrada e saída traduz sinais elétricos do barramento de sistema em sinais elétricos do barramento de memória. Essa ponte também conecta os barramentos de memória e de sistema com um **barramento de entrada e saída**, que é compartilhado por dispositivos de entrada ou saída (v. adiante).

### 1.2.2 Dispositivos de Entrada e Saída

Um **dispositivo de entrada** é um equipamento periférico que permite que dados sejam transferidos dele para a memória principal. Exemplos de dispositivos de entrada são mouse e teclado. Por outro lado, um **dispositivo de saída** permite que dados residentes em memória principal sejam escritos nele. Podem ser citados como dispositivos de saída: monitor de vídeo e impressora. Existem ainda dispositivos que ora são de entrada ora são de saída, como, por exemplo, um disco rígido.

A **Figura 1–9** mostra como dispositivos de entrada ou saída são conectados à memória principal. O responsável por essa conexão é o barramento de entrada e saída, que, diferentemente do que ocorre com os barramentos de sistema e de memória, não depende de uma CPU específica. Além disso, um barramento de entrada e saída é bem mais lento do que um barramento de sistema ou de memória.

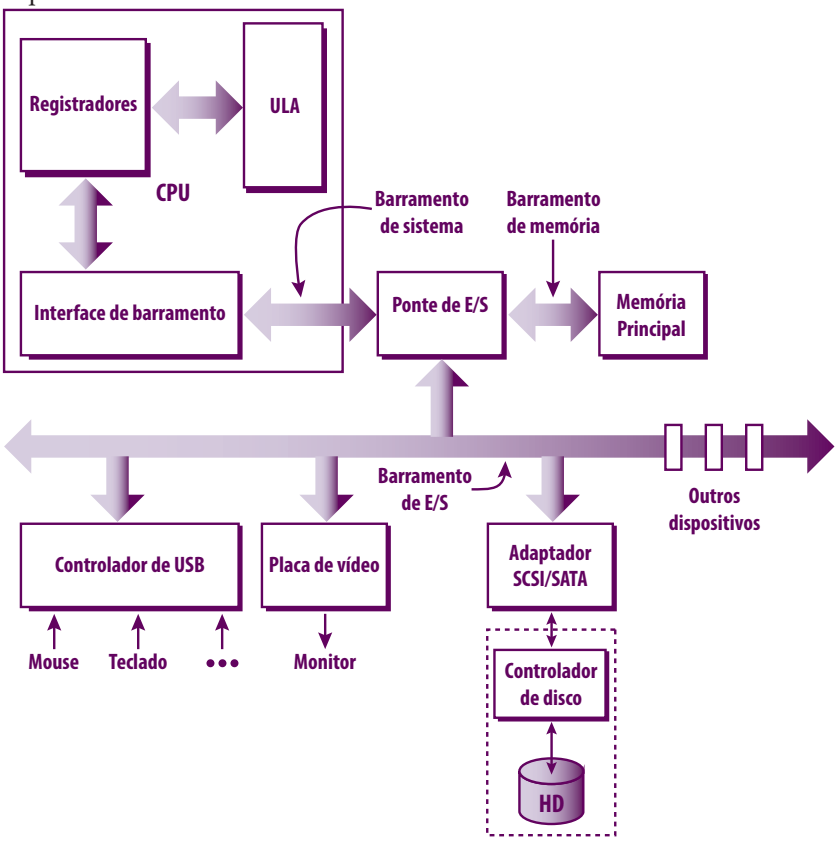


**Figura 1–9: Conexão de Dispositivos de Entrada e Saída**

O barramento de entrada e saída apresentado na **Figura 1–9** é baseado na tecnologia PCI do fabricante Intel e conecta CPU, memória principal e dispositivos de entrada e saída. Ele é capaz de acomodar uma grande variedade de dispositivos de entrada e saída, conforme é ilustrado nessa figura.

O **controlador de USB**[8] mostrado na **Figura 1–9** é um dispositivo que serve como interface para dispositivos conectados a um barramento de USB. Hoje em dia, existem inúmeros dispositivos que podem ser conectados via USB, dentre os quais mouse, teclado, câmeras e telefones celulares.

[8] *USB* é um acrônimo para *Universal Serial Bus*. Para os propósitos deste livro você não precisa se preocupar com o exato significado dessa sigla.

O **adaptador SCSI/SATA** que aparece na **Figura 1–9** é uma interface para discos rígidos do tipo SCSI ou do tipo SATA. Interface de disco é um mecanismo de comunicação entre o disco e o resto do sistema. Exemplos de interfaces de disco são SATA, ATA e SCSI. Tipicamente, **discos SCSI** são mais rápidos e mais caros do que **discos SATA**. Um adaptador SCSI pode servir para vários discos, diferentemente do que ocorre com um adaptador SATA que suporta apenas um disco.

Como mostra a **Figura 1–9**, vários outros dispositivos podem ser conectados ao barramento de entrada e saída por meio de adaptadores. Na prática, esses adaptadores são inseridos na placa mãe de um computador que provê conexão elétrica com o referido barramento.

### 1.2.3 Operações de Entrada e Saída

Uma **operação de entrada** consiste na transferência de dados de um dispositivo de entrada para a memória principal. Por sua vez, uma **operação de saída** consiste no caminho inverso para um dispositivo de saída.

Operações de entrada e saída de um programa são efetuadas por intermédio do **sistema operacional** que serve como hospedeiro para o programa. Por exemplo, um programa escrito em C pode solicitar ao sistema operacional, por intermédio da função **fopen()**, a abertura de um arquivo.

Quando o sistema operacional executa uma operação de entrada ou saída, tal como a leitura de um bloco de um disco, ele inicia a seguinte sequência de eventos:

1. O sistema operacional envia um comando para o **controlador de disco** (v. **Figura 1–9**) solicitando-o que leia o bloco cuja numeração única lhe é passada. Esse bloco é exatamente aquele que contém os dados que precisam ser lidos (i.e., o bloco cujos dados serão copiados para a memória principal).
2. O controlador de disco usa informações armazenadas em *firmware* para traduzir a numeração de bloco que lhe foi passada numa tripla constituída por informações sobre superfície, trilha e setor que identificam o bloco desejado.
3. O hardware do controlador interpreta essa tripla e provoca o movimento da cabeça de leitura e gravação para o setor desejado.
4. A leitura é efetuada conforme foi descrito na **Seção 1.1.3** e os dados lidos são copiados numa área de armazenamento temporário (buffer) do controlador.
5. Os dados são finalmente movidos do buffer do controlador para a memória principal.

A **Figura 1–10** ilustra resumidamente o processo de leitura de dados num disco. Nessa figura, a CPU emite um comando para um dispositivo de entrada ou saída. Neste caso, o comando ilustrado é referente a leitura num disco. Enquanto a solicitação é processada, se for o caso, a CPU prossegue com a execução de suas tarefas. É importante salientar que uma CPU com 1 GHz é capaz de executar 14 milhões de instruções no intervalo de tempo de cerca de *12* ms que leva para ler um bloco (v. **Seção 1.1.3**). Uma eternidade em termos de tempo de processamento que é desperdiçada.

Depois que o controlador de disco recebe o comando de leitura emitido pela CPU, ele lê o conteúdo do bloco e o transfere para memória principal conforme foi descrito na **Seção 1.1.3**. Como mostra a **Figura 1–11**, esse processo não sofre nenhuma intervenção da CPU e recebe a denominação **acesso direto à memória** (**DMA**). Após concluída a transferência dos dados lidos para a memória principal, o controlador envia um **sinal de interrupção** para a CPU, como mostra a **Figura 1–12**. Após receber esse sinal, a CPU interrompe suas tarefas e passa a executar o procedimento iniciado pelo sistema operacional. Esse procedimento registra o fato de a operação de entrada/saída ter terminado e retorna o controle para o ponto no qual a CPU foi interrompida.

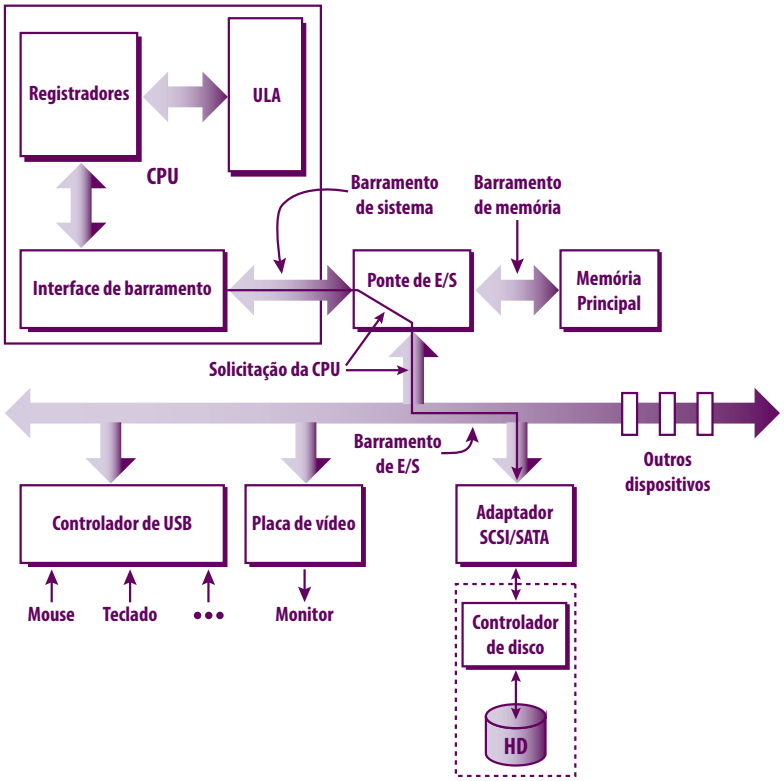


**Figura 1–10: Leitura de um Setor: CPU Faz a Solicitação**

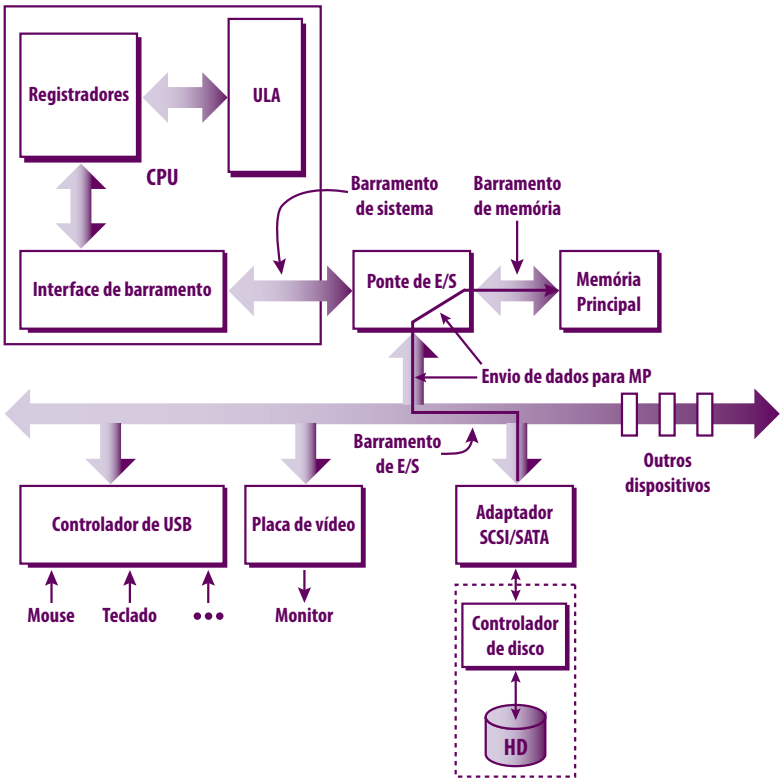


**Figura 1–11: Leitura de um Setor: Envio de Dados para Memória Principal**

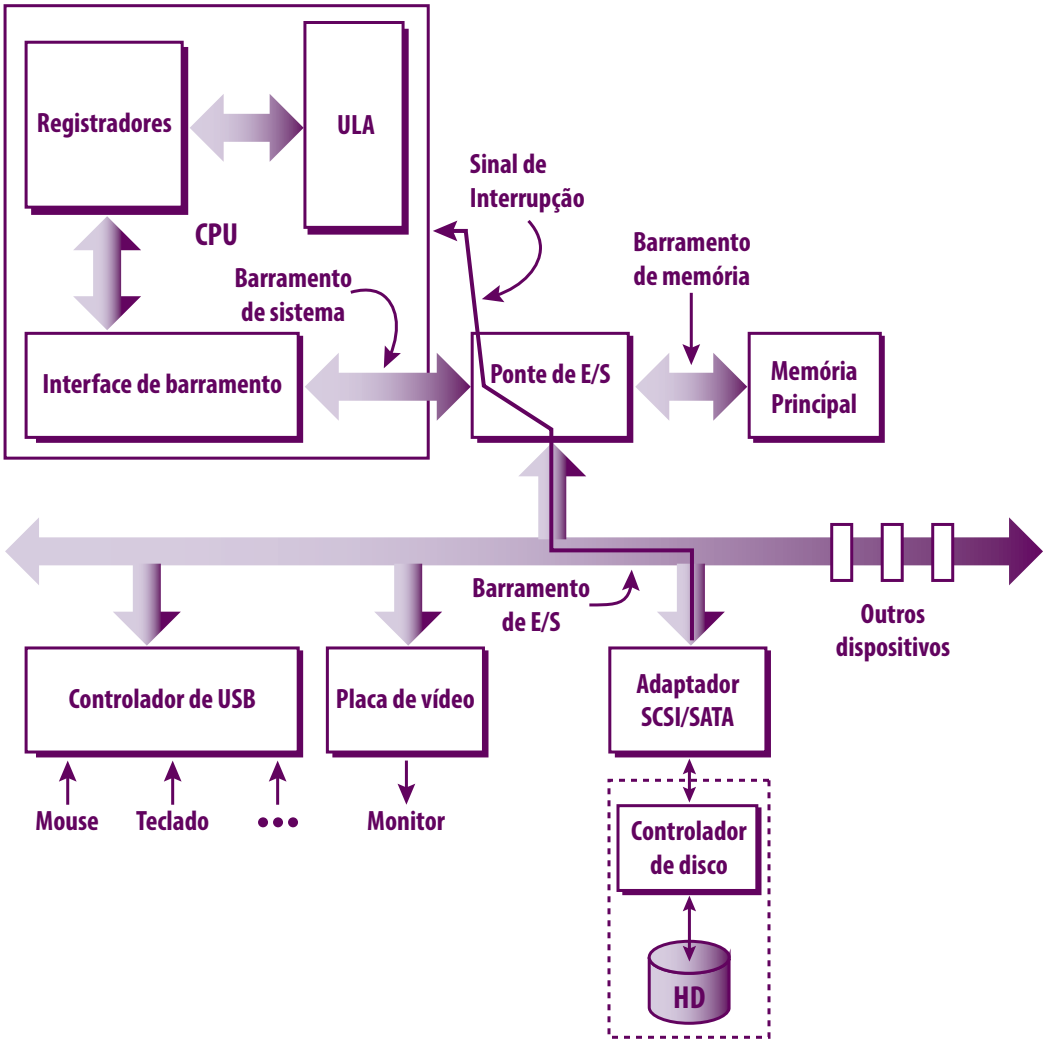


FIGURA 1–12: LEITURA DE UM SETOR: ENVIO DE SINAL DE INTERRUPÇÃO PARA CPU

## 1.3 Hierarquias de Memória

Tecnologias de armazenamento diferentes possuem diferentes preços e desempenhos. Memória SRAM é mais rápida do que memória DRAM, que é bem mais rápida do que discos magnéticos. Por outro lado, memórias mais rápidas são mais caras do que memórias mais lentas.

O espaço total utilizado por um computador para armazenamento de dados pode ser visualizado como uma **hierarquia de memória** imposta pelo tempo de resposta, pela capacidade e pelo custo de cada componente. Quer dizer, componentes mais caros e com tempos de acesso e capacidades menores ocupam postos superiores nessa hierarquia, como mostra a **Figura 1–13**. Ou seja, dispositivos de armazenamento tornam-se mais lentos, mais baratos e com maior capacidade à medida que se desce do mais elevado para o mais baixo nível da hierarquia.

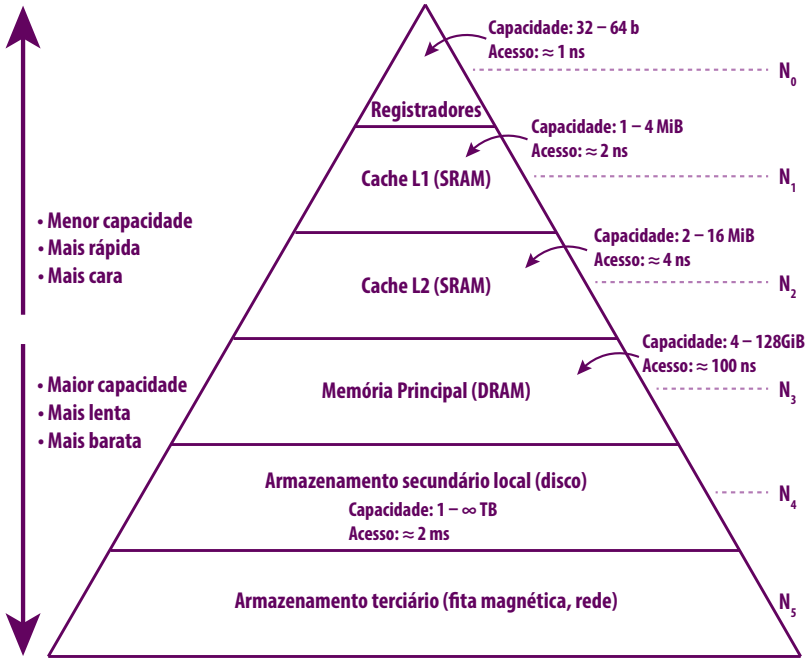


**Figura 1–13: Hierarquia de Memória**

O objetivo de uma hierarquia de memória é fazer com que um sistema de memória funcione com os tempos de acesso dos níveis mais altos, mas com os custos e as capacidades dos níveis mais baixos.

Os níveis da hierarquia mostrada na **Figura 1–13** são:

- **Nível N0**. No topo da hierarquia encontra-se um pequeno número de registradores que a CPU acessa com um único ciclo de instrução[9].
- **Níveis N1 e N2**. Memórias cache (SRAM) com capacidades (entre 6 KiB e 128 MiB) muito maiores do que a capacidade do conjunto de registradores, mas são mais lentas (entre 700 GB/s e 40 GB/s). Essas memórias cache atuam como armazenamento temporário para dados e instruções armazenados em memória principal, que é mais lenta. Num sistema com memórias cache múltiplas, identificadas como L1, L2 etc., essas memórias são progressivamente mais lentas, têm maior capacidade e são mais baratas. Tipicamente, essas memórias cache são entre três e dez vezes mais rápidas do que a memória principal. Um computador moderno pode conter até cinco níveis de memória cache, mas, na hierarquia ilustrada na **Figura 1–13** há apenas dois níveis de memória cache.
- **Nível N3**. A **memória interna** ou **memória principal** (DRAM) é bem maior do que a memória cache, mas requer um tempo maior de acesso (i.e., dezenas ou centenas de ciclos de instrução). A memória principal age como espaço transitório para dados armazenados em disco, que é um dispositivo de acesso bem mais lento, apesar de apresentar uma capacidade relativamente grande.
- **Nível N4**. A memória secundária é representada por discos com enorme capacidade, mas relativamente muito lentos. Os discos servem como área de armazenamento temporário para fitas magnéticas ou outros discos acessados via rede.
- **Nível N5**. Memórias terciárias representadas por fitas magnéticas e discos que se encontram em servidores remotos acessados via rede.

[9] Um ciclo de instrução é o ciclo de operação básica de uma CPU. Para entender os detalhes de funcionamento de uma CPU, consulte um texto básico sobre arquitetura de computadores.

Obviamente, existem outras combinações para composição de uma hierarquia de memória. Por exemplo, um disco SSD pode ser interposto entre a memória principal e discos magnéticos.

Os níveis de hierarquia que interessam ao programador dependem do tamanho dos dados processados. Para a maioria dos programas, existem apenas dois níveis: aquele que armazena os dados do programa e o nível logo abaixo. Por exemplo, se todos os dados cabem em memória principal, os níveis de interesse são a memória interna e a memória cache. Se os dados não cabem completamente em memória principal, os níveis de interesse são memória principal e memória secundária. A diferença em termos de acesso entre as memórias principal e secundária é mais dramática do que entre as memórias principal e cache.

## 1.4 Caching

Em computação, **memória cache** (ou, simplesmente, **cache**) consiste em hardware ou software dedicado a estocar dados ou instruções para uso futuro, de modo que eles possam ser acessados mais rapidamente do que se estivessem em seus espaços originais em memória. Em outras palavras, memória cache é um meio de armazenamento menor e mais rápido que serve como área de armazenamento temporário para dados armazenados num meio de armazenamento maior e mais lento. Cache também armazena resultados de operações recentes de modo que elas não precisem ser refeitas. **Caching** refere-se ao uso de memória cache.

A ideia central na qual se baseia o uso de memória cache é que um meio de armazenamento no nível $n$ numa hierarquia de memória serve como cache para o meio de armazenamento no nível $n + 1$ nessa hierarquia. Por exemplo, um disco rígido pode servir de cache para arquivos obtidos via Web e, por sua vez, a memória principal serve como cache para o disco local.

Computadores antigos possuíam apenas três níveis de memória: registradores, memória principal e disco magnético. Mas devido à grande diferença entre tempo de acesso a registradores e a memória principal, decidiu-se acrescentar uma pequena memória cache SRAM, denominada **cache L1**, entre os registradores e a memória principal, como mostra a **Figura 1–14**. Uma memória cache L1 pode ser acessada quase tão rapidamente quanto um registrador; i.e., o tempo de acesso para esse tipo de cache situa-se entre 2 e 4 ciclos de instrução[10]. Além disso, computadores mais modernos podem incluir memórias cache denominadas **L2** (acessadas em 10 ciclos) e **L3** (acessadas em 30 ou 40 ciclos) que são progressivamente maiores e mais lentas do que a memória cache **L1**, mas, mesmo assim, são bem mais rápidas do que a memória principal.

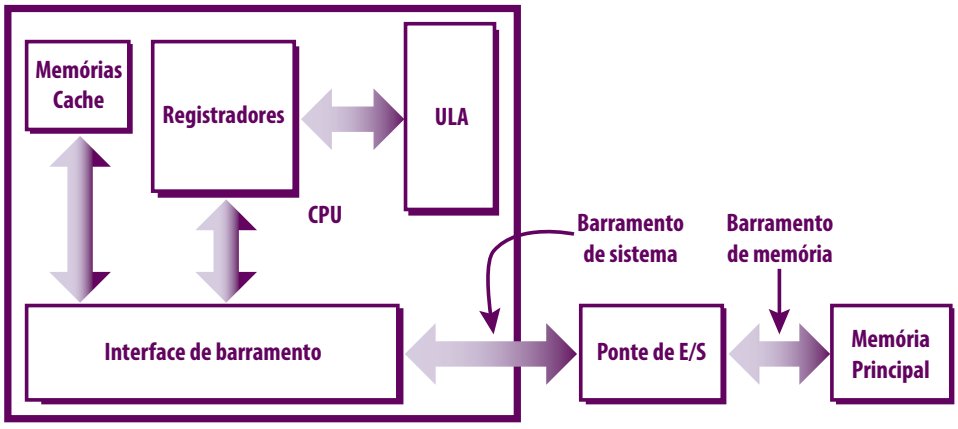


**Figura 1–14: Memórias Cache**

[10] É importante notar que *cache* é um conceito, mas aqui neste caso, esse conceito é representado por um dispositivo físico. Isso pode ser um tanto ambíguo. Neste texto, quando se fala em cache de CPU ou cache nível L1, L2 etc. se está fazendo referência a um dispositivo físico. Em outras situações, cache pode se referir a qualquer meio de armazenamento que aparece num nível superior a outro numa hierarquia de memória.

A **Figura 1–15** mostra o funcionamento de caching numa hierarquia de memória. Nessa figura, dados armazenados no nível *n* + *1* são divididos em blocos, cada um dos quais possui um endereço único[11]. Usualmente, o tamanho desses blocos é fixo, mas também pode ser variável (p. ex., arquivos da Web). Dados armazenados no nível *n* também são particionados em blocos e esses blocos têm o mesmo tamanho daqueles do nível *n* + *1*.

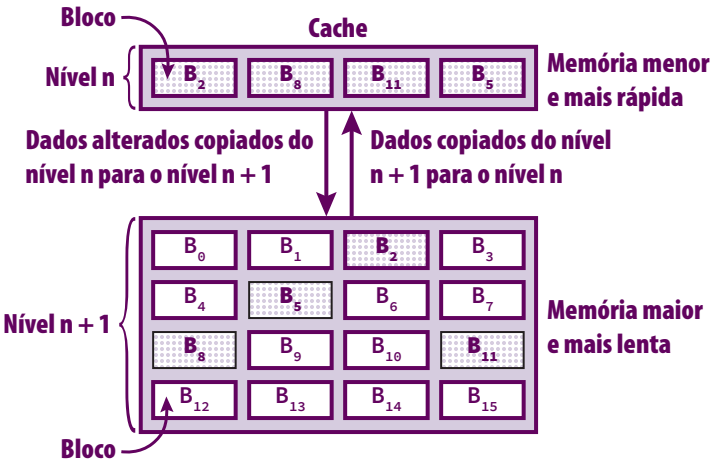


**Figura 1–15: Caching numa Hierarquia de Memória**

Como ilustra a **Figura 1–15**, a memória cache no nível *n* armazena alguns blocos do nível *n* + *1*. Nessa figura, os blocos $B_2$, $B_8$, $B_{11}$ e $B_5$ do meio de armazenamento no nível *n* + *1* se encontram armazenados na memória cache do nível *n*. A referida figura indica ainda que dados são copiados entre os meios de armazenamento nos níveis *n* e *n* + *1*, sendo que dados que se encontram no nível *n* são copiados para o nível *n* + *1* apenas se eles tiverem sido alterados.

Em geral, meios de armazenamento que ocupam as posições mais baixas numa hierarquia de memória tendem a usar blocos maiores para compensar a relativa lentidão desses dispositivos.

Quando um programa precisa acessar um dado que se encontra no nível *n* + *1* numa hierarquia de memória, ele primeiro procura-o em algum bloco que se encontra no nível *n* dessa hierarquia. Quando o referido dado é encontrado no nível *n*, diz-se que ocorreu um **acerto de cache**. Em caso contrário, diz-se que ocorreu um **lapso de cache**. Por exemplo, considerando-se a configuração apresentada na **Figura 1–15**, se um programa tenta acessar um dado que se encontra no bloco $B_{11}$, ocorre um acerto de cache. Por outro lado, se ele tenta acessar um dado que se encontra no bloco $B_0$, ocorre um lapso de cache.

Quando ocorre um acerto de cache, o programa acessa o dado desejado diretamente no nível *n* que, em virtude da natureza da hierarquia de memória, permite acesso mais rápido do que o nível *n* + *1*. Por outro lado, quando ocorre um lapso de cache, o sistema obtém o bloco que contém o dado no nível *n* + *1* e armazena esse bloco no nível *n*. Nesse processo, é possível que um bloco que já se encontre armazenado no nível *n* precise ser sobrescrito se a memória cache desse nível já estiver cheia.

O processo de sobrescrita de um bloco em memória cache é conhecido como **desalojamento de bloco**. A **Figura 1–16** ilustra um exemplo de lapso de cache e desalojamento de bloco. Nesse exemplo, um programa procura um dado *d* e não o encontra na memória cache no nível *n*. Entretanto esse dado se encontra no bloco $B_3$ que se acha no nível *n* + *1*. Então esse bloco deve ser carregado na memória cache no nível *n*. Mas como essa memória se encontra lotada, um dos blocos que lá se encontram precisa ser sobrescrito. Nesse exemplo, o bloco desalojado é o bloco $B_5$, mas poderia ser outro (v. adiante). Depois que a memória cache do nível *n*

[11] No corrente contexto, esses blocos são também denominados **linhas de cache**.

obtém o bloco $B_3$ no nível $n + 1$, o programa pode ler o dado $d$ no nível $n$, conforme foi descrito acima. Uma vez que o bloco $B_3$ é carregado no nível $n$, ele permanece lá com a expectativa de que será novamente acessado.

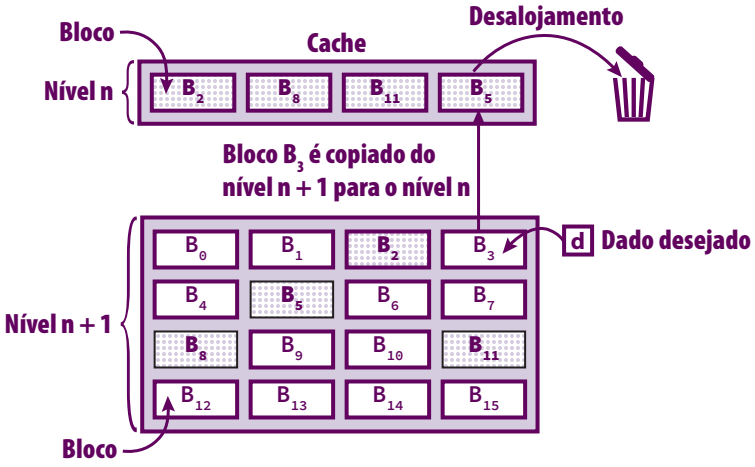


**Figura 1–16: Lapso de Cache e Desalojamento de Bloco**

**Heurísticas** são usadas para tentar reduzir o número de lapsos de cache[12]. Quer dizer, a decisão sobre qual bloco desalojar quando ocorre um lapso de cache é ditada por uma heurística denominada **política de substituição** (ou **política de desalojamento**) de memória cache. A implementação dessa política envolve diferentes algoritmos e estruturas de dados. Existem três políticas básicas de substituição de blocos de memória cache:

- **Política aleatória**. Essa política causa o desalojamento de um bloco escolhido aleatoriamente e é ilustrada na **Figura 1–17**. Essa abordagem é muito fácil de implementar e tem custo temporal $\theta(1)$, mas, em compensação, ela não leva em consideração localidade temporal ou espacial (v. **Seção 1.5.1**).

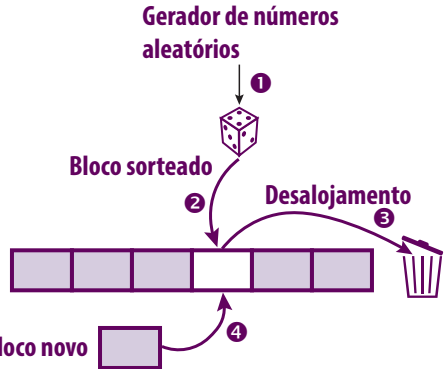


**Figura 1–17: Política Aleatória de Desalojamento de Cache**

- **Política FIFO**. Nessa política de desalojamento, os blocos são armazenados numa fila, de modo que essa abordagem remove o bloco mais antigo na fila (i.e., aquele que se encontra na frente da fila)[13]. A **Figura 1–18** ilustra essa política, que também apresenta custo temporal $\theta(1)$. A política FIFO também não leva em consideração localidade temporal ou espacial (v. **Seção 1.5.1**).

[12] *Heurística* é um dos termos mais mal-empregados em computação e esta é uma ótima oportunidade para tentar esclarecer o conceito ao qual esse termo se refere. Um algoritmo que *sempre* resolve um determinado problema não pode ser classificado como *heurístico*. Um **algoritmo heurístico** é aquele que *pode* resolver um problema numa dada situação, mas não há garantia de que isso ocorrerá. Se a heurística na qual tal algoritmo se baseia é bem fundamentada em experimentos, a probabilidade de que isso ocorra é grande. Heurística tem tudo a ver com cache e localidade de referência, que será discutido na próxima seção.

[13] A denominação dessa política torna-se óbvia se você estudou filas adequadamente, pois essas estruturas de dados são também denominadas de *estruturas FIFO*, como foi visto no **Volume 1** desta obra.

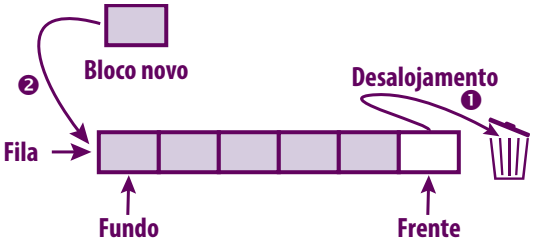


**Figura 1–18: Política FIFO de Desalojamento de Cache**

- **Política MRU**. MRU é um acrônimo para (bloco) **menos recentemente usado**, o que significa que o bloco desalojado nessa abordagem é aquele cujo acesso foi o mais antigo. De acordo com as pesquisas, essa é a melhor abordagem, pois ela leva em conta localidade temporal, que será discutida na **Seção 1.5.1**. Essa abordagem pode ser implementada por meio de uma fila de prioridade (v. **Capítulo 10**), que apresenta custo temporal $\theta(n \cdot log\ n)$ no melhor caso. Essa política é ilustrada na **Figura 1–19**.

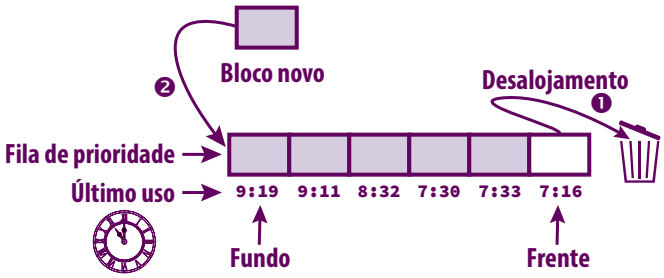


**Figura 1–19: Política MRU de Desalojamento de Cache**

Existem muitos detalhes referentes a caching que não são abordados aqui, pois eles estão além do escopo deste texto. Além disso, caching é um tópico intimamente relacionado com gerenciamento de memória virtual, que também está além do alcance deste livro.

## 1.5 Localidade de Referência

Diz-se que um programa é **dirigido por processamento** quando, durante a maior parte de sua execução, ele executa instruções. Por outro lado, um programa é **dirigido por entrada e saída** se, na maior parte de seu tempo de execução, ele efetua operações de leitura ou escrita de dados em memória secundária. Por exemplo, um programa que ordena um array completamente contido em memória principal é dirigido por processamento, enquanto um programa que ordena um conjunto de milhões de registros mantidos num arquivo em disco é dirigido por entrada e saída (v. **Capítulo 12**).

Uma maneira de melhorar o desempenho de um programa é reduzir o acesso aos níveis inferiores de uma hierarquia de memória, como aquela apresentada na **Seção 1.3**. Computadores (hardware) e sistemas operacionais possuem mecanismos para acelerar a maioria dos acessos de memória. Por exemplo, o hardware é responsável pelo movimento de dados entre memória cache (SRAM) e memória principal (DRAM) para acelerar o acesso a dados. Essa memória cache armazena dados e instruções usados mais recentemente. Por outro lado, sistemas operacionais usam a memória principal como cache para o bloco de um disco magnético mais recentemente acessado. Isso evita que a CPU desperdice tempo demais aguardando a finalização de operações de entrada e saída. Esses mecanismos usados por hardware e software são baseados em localidade de referência, que será discutida nesta seção.

Programas com boa **localidade de referência** tendem a acessar os mesmos dados repetidamente ou a acessar dados adjacentes em memória. Programas com boa localidade também tendem a acessar dados que se encontram

em porções superiores da hierarquia de memória. Assim boa localidade é uma qualidade desejável de um programa, pois essa qualidade faz com ele seja executado mais rapidamente. Um bom programa tende a acessar dados próximos a outros dados recentemente acessados e essa tendência é conhecida como **princípio de localidade**.

### 1.5.1 Localidades Temporal e Espacial

Existem duas categorias de localidade de referência: localidade temporal e localidade espacial. **Localidade temporal** diz respeito ao uso repetido de um determinado conjunto de dados ou instruções durante um certo intervalo de tempo, enquanto **localidade espacial** se refere ao uso de um determinado conjunto de dados cujos elementos estão localizados em endereços próximos entre si.

Memórias cache favorecem localidade temporal com base no seguinte fato experimental: se um programa acessa certo espaço em memória, é provável que ele torne a acessá-lo muitas vezes em breve. Um exemplo de constatação desse fato é o uso de uma variável de contagem em diversas instruções num laço de contagem. Por outro lado, o benefício do uso de memórias cache para localidade espacial baseia-se em outro resultado experimental: se um programa acessa certo espaço em memória, é provável que ele também acesse em breve outras posições vizinhas. Em geral, o uso de memórias cache com intuito de favorecer a localidade de referência de programas baseia-se num fato experimental que é uma espécie de **Princípio de Pareto** (ou **regra 80/20**) de utilização de dados[14]:

***Na maioria dos programas, cerca de 80% do acesso à memória incide sobre 20% de dados.***

Por exemplo, de acordo com esse princípio, se um programa utiliza *100* variáveis, em média, ele acessa apenas *20* delas *80%* das vezes.

Agora, considere como exemplo de localidade de referência, a função `SomaArray()`, definida abaixo, que acessa sequencialmente os elementos de um array com o objetivo de somá-los. Nessa função, a variável `soma` é acessada a cada passagem no laço **for**. Portanto essa função apresenta uma boa localidade temporal com respeito à variável `soma`. Por outro lado, como essa variável não é um elemento de uma variável estruturada, não existe localidade espacial com relação a essa variável.

```
int SomaArray(int ar[], int n)
{
   int i, soma = 0;

   for (i = 0; i < n; i++)
      soma += ar[i];

   return soma;
}
```

Na função `SomaArray()`, os elementos do array `ar[]` são acessados sequencialmente exatamente na ordem em que eles são armazenados em memória. Portanto a função `SomaArray()` possui boa localidade espacial com relação à variável `ar[]`, mas tem má localidade temporal com relação a essa mesma variável pois cada elemento desse array é acessado exatamente uma única vez. Como a função `SomaArray()` apresenta boa localidade espacial ou temporal com relação a cada variável no corpo do laço, em geral, ela tem boa localidade.

Uma função tal qual `SomaArray()` possui **padrão de referência sequencial** ou **padrão de referência de ordem 1**. Em geral, quando se acessa cada enésimo elemento de um array unidimensional classifica-se esse acesso como **padrão de referência** (**de ordem**) **n**. À medida que a ordem de um padrão de referência aumenta, assim diminui a localidade espacial. Portanto o melhor que se pode obter num acesso a elementos de um array é o padrão de referência 1 (ou **padrão sequencial**). Por exemplo, suponha que o parâmetro `ar[]` da função `SomaArray()` represente um array com *5* elementos (i.e., o parâmetro `n` dessa função vale `5`), que seu primeiro

[14] Este é mais um exemplo de heurística.

elemento possui endereço `e` e que a largura de cada elemento é igual a 4 bytes. Desse modo, a **Tabela 1–2** mostra como os elementos do array recebido como parâmetro pela referida função são acessados.

| Endereço | `e` | `e + 4` | `e + 8` | `e + 12` | `e + 16` |
|---|---|---|---|---|---|
| Conteúdo | `ar[0]` | `ar[1]` | `ar[2]` | `ar[3]` | `ar[4]` |
| Ordem de acesso | 1 | 2 | 3 | 4 | 5 |

**Tabela 1–2: Padrão de Referência Sequencial em Array Unidimensional**

Para melhor entender o processo, serão adotadas as seguintes simplificações adicionais:

- O array está alinhado com um bloco em memória principal. Quer dizer, o endereço do elemento `ar[0]` coincide com o endereço inicial de um bloco em memória principal.
- O tamanho de cada bloco entre a memória principal e a memória cache exatamente acima dela na hierarquia de memória é de 16 bytes. Ou seja, cabem precisamente quatro elementos do array em questão em cada um desses blocos.
- A referida memória cache está inicialmente vazia (**memória cache fria**).

Levando em consideração essas simplificações, durante a execução da função `SomaArray()`, ocorre a seguinte sequência de eventos:

1. Na primeira passagem pelo corpo do laço **for** da função `SomaArray()`, ocorre um lapso de cache, uma vez que, por hipótese, a memória cache está fria (i.e., vazia). Assim o bloco que contém os quatro primeiros elementos do array `ar[]` é carregado na memória cache. Esse fato é ilustrado na **Figura 1–20**.

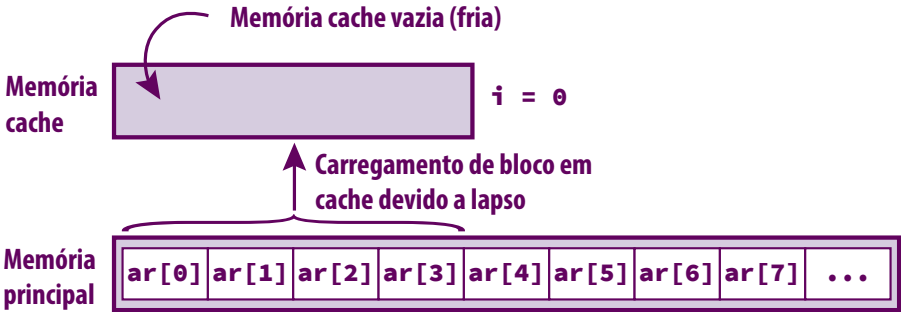


**Figura 1–20: Padrão de Referência 1: Primeiro Lapso de Cache**

2. Nas três passagens seguintes pelo laço **for** sob discussão, ocorrem acertos de cache porque os elementos do array que são acessados já se encontram na memória cache, como mostra a **Figura 1–21**. Portanto nesse caso, não ocorre movimentação de bloco entre a memória principal e a memória cache.

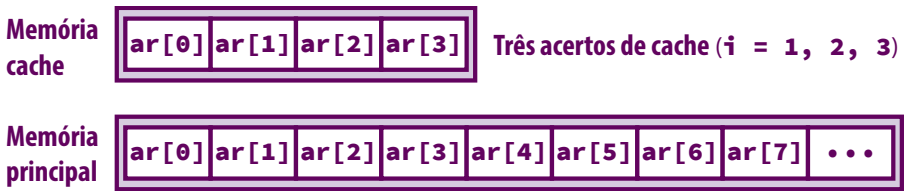


**Figura 1–21: Padrão de Referência 1: Três Acertos de Cache**

3. Como mostra a **Figura 1–22**, na quinta passagem pelo corpo do laço **for** da função `SomaArray()`, ocorre um novo lapso de cache. Dessa vez, a memória cache não está fria, mas o elemento `ar[4]` não se encontra nela. Desse modo, um novo bloco precisa ser carregado na memória cache.

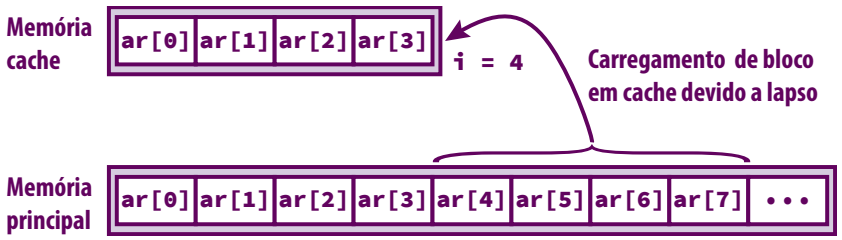


**Figura 1–22: Padrão de Referência 1: Segundo Lapso de Cache**

Nas próximas passagens pelo corpo do laço **for** da função `SomaArray()`, o padrão apresentado acima irá se repetir (v. **Figura 1–23**). Logo com as suposições feitas acima, ocorre um lapso de cache a cada quatro acessos (ou 75% de acertos de cache), o que é considerado muito bom em termos de eficiência.

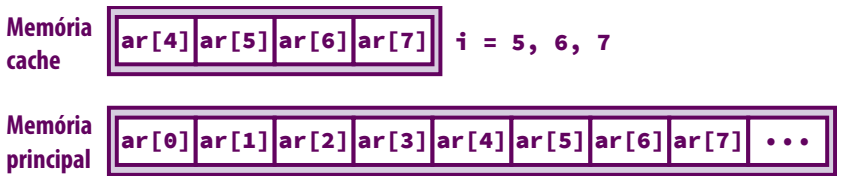


**Figura 1–23: Padrão de Referência 1: Mais Três Acertos de Cache**

Padrões de referência também são importantes quando se acessam elementos de arrays multidimensionais. Considere, por exemplo, a função `SomaArrayBi1()` apresentada a seguir[15].

```
int SomaArrayBi1(int a[][N], int m)
{
   int i, j, soma = 0;

   for (i = 0; i < m; i++)
      for (j = 0; j < N; j++)
         soma += a[i][j];

   return soma;
}
```

A função `SomaArrayBi1()` soma os elementos de um array bidimensional e retorna o valor obtido. Os laços **for** dessa função acessam os elementos do array `a[][]` **por linha**; i.e., o laço interno acessa os elementos da primeira linha, depois os elementos da segunda linha e assim por diante. Esse padrão de acesso coincide exatamente com o modo como um array bidimensional é armazenado em memória, como ilustra a **Figura 1–24** que mostra como um array com três linhas e três colunas é armazenado em memória. Examinado-se essa figura, percebe-se que a função `SomaArrayBi1()` exibe ótima localidade espacial, visto que ela acessa os elementos do array na mesma ordem em que eles são armazenados em memória. Portanto o que se tem aqui é um padrão de referência 1, o que garante à função `SomaArrayBi1()` uma ótima localidade espacial.

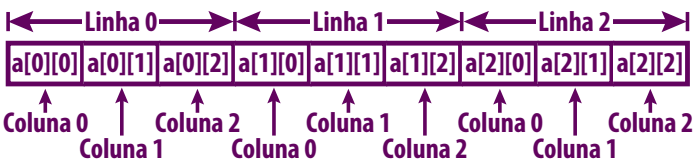


**Figura 1–24: Armazenamento de um Array Bidimensional em Memória**

Para entender melhor a razão pela qual a função `SomaArrayBi1()` exibe essa localidade espacial, suponha que o parâmetro `a[][]` casa com um array com três linhas e três colunas (nesse caso, o parâmetro `m` e a constante

[15] Neste e no próximo exemplos, `N` é uma constante simbólica previamente definida.

`N` valem `3`). A **Tabela 1–3** mostra como a função `SomaArrayBi1()` acessa os elementos do array recebido como parâmetro.

| Endereço | e | e + 4 | e + 8 | e + 12 | e + 16 | e + 20 |
|---|---|---|---|---|---|---|
| Conteúdo | a[0][0] | a[0][1] | a[0][2] | a[1][0] | a[1][1] | a[1][2] |
| Ordem de acesso | 1 | 2 | 3 | 4 | 5 | 6 |

**Tabela 1–3: Padrão de Referência 1 em Array Bidimensional**

Agora considere a função `SomaArrayBi2()` a seguir.

```
int SomaArrayBi2(int a[][N], int m)
{
   int i, j, soma = 0;

   for (j = 0; j < N; j++)
      for (i = 0; i < m; i++)
         soma += a[i][j];

   return soma;
}
```

A função `SomaArrayBi2()` é funcionalmente equivalente à função `SomaArrayBi1()`. Quer dizer, ambas as funções fazem o mesmo. Entretanto elas o fazem de modo diferente. Note que, como foi dito acima, a função `SomaArrayBi1()` acessa os elementos do array que ela recebe como parâmetro por linha, enquanto a função `SomaArrayBi2()` acessa esses mesmos elementos **por coluna**. Objetivamente, note atentamente que o que diferencia essas funções é o modo como os laços de repetição **for** são escritos. Para entender melhor, suponha que, como no exemplo anterior, o parâmetro `a[][]` é um array com três linhas e três colunas e (i.e., os parâmetros `m` e `n` valem `3`). A **Tabela 1–4** mostra como a função `SomaArrayBi2()` acessa os elementos que ela recebe como parâmetro.

| Endereço | e | e + 4 | e + 8 | e + 12 | e + 16 | e + 20 |
|---|---|---|---|---|---|---|
| Conteúdo | a[0][0] | a[0][1] | a[0][2] | a[1][0] | a[1][1] | a[1][2] |
| Ordem de acesso | 1 | 3 | 5 | 2 | 4 | 6 |

**Tabela 1–4: Padrão de Referência n em Array Bidimensional**

A função `SomaArrayBi2()` tem padrão de referência de ordem *n*, em que *n* é o número de elementos na segunda dimensão (colunas) do array recebido como parâmetro. Se esse array não couber inteiramente em memória cache, cada acesso causará um lapso de cache.

A **Figura 1–25** ilustra o conceito de padrão de referência para acesso a elementos de um array.

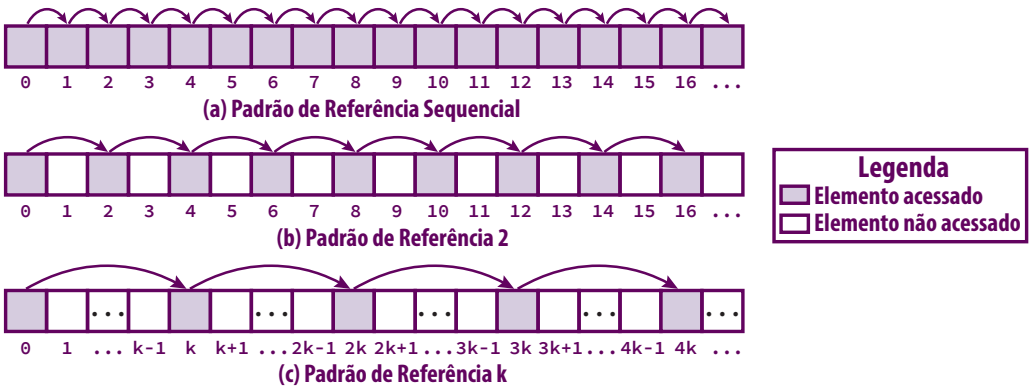


**Figura 1–25: Padrões de Referência para Acesso a Arrays**

Como mostram os exemplos acima, após um bloco ter sido carregado em memória cache em decorrência de um lapso de cache, espera-se que ocorram vários acertos de cache com esse bloco. Além disso, como uma memória cache que se encontra num nível *n* na hierarquia de memória é mais rápida do que a memória que se encontra no nível *n* + *1* da mesma hierarquia, os acertos de cache subsequentes compensam o referido lapso de cache.

Altas taxas de lapso de cache têm um grande impacto negativo sobre a eficiência de um programa. Por exemplo, a função `SomaArrayBi1()` pode apresentar um desempenho duas vezes maior do que a função `SomaArrayBi2()`. Em suma, os programadores devem ter ciência sobre localidade quando codificam para serem capazes de explorá-la.

É importante salientar que um array se beneficia bem mais com o uso de memória cache do que uma estrutura encadeada (p. ex., lista encadeada ou árvore) porque os elementos de um array são armazenados contiguamente. Por outro lado, os elementos de uma estrutura encadeada são alocados individualmente e, provavelmente, se situam em posições dispersas em memória.

### 1.5.2 Localidade de Acesso a Instruções

Instruções em **linguagem de máquina** são armazenadas em memória e, antes de serem executadas, precisam ser lidas pela CPU. Além disso, memórias cache podem não apenas armazenar dados como também instruções. Portanto um programa também pode ser julgado de acordo com a facilidade de acesso a suas instruções. Por exemplo, na função `SomaArray()` apresentada na **Seção 1.5.1**, as instruções no corpo do laço **for** são acessadas de modo sequencial, de maneira que o laço apresenta boa localidade espacial. Além disso, como o corpo do referido laço é executado várias vezes, ele também apresenta boa localidade temporal.

### 1.5.3 Blocagem e Memória Virtual

**Blocagem** é o processo de organizar dados em blocos e é usado para facilitar o processamento de dados armazenados em memória secundária. Quando os dados que se encontram numa certa localização em memória secundária são acessados, transfere-se um grande bloco de bytes contíguos que inclui os dados desejados. Essa ideia é motivada por localidade espacial (v. **Seção 1.5.1**). Ou seja, ela se baseia na expectativa de que outros dados vizinhos àqueles almejados serão em breve acessados. Neste contexto, esses blocos são também denominados **páginas**.

Especificamente, se os dados armazenados numa localização *l* em memória secundária são acessados, então carrega-se em memória principal um grande bloco de dados contíguos que incluem a localização *l* (v. **Figura 1–26**). Essa ação é motivada pela expectativa de que outros locais em memória secundária próximas a *l* serão acessados em breve.

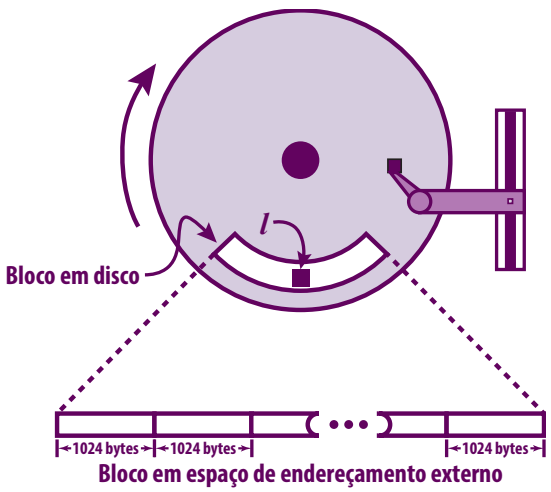


**Figura 1–26: Bloco em Disco e em Espaço de Endereçamento Externo**

Blocagem para discos também é motivada pelas propriedades de funcionamento dos próprios discos. Quer dizer, como foi visto na **Seção 1.1.3**, uma cabeça de leitura de um disco leva um tempo relativamente longo para se posicionar para leitura numa certa localização, mas, uma vez que o braço esteja posicionado, ele pode rapidamente ler muitos locais contíguos, porque o disco que está sendo lido continua girando em alta velocidade.

Em programação, *bloco* tem múltiplos significados dependendo do contexto. No contexto de armazenamento de dados em disco e para um sistema de arquivos, um bloco é uma abstração que engloba um ou vários setores de um disco.

Na prática, pode-se verificar qual é o **tamanho de um bloco** usado por um sistema de arquivos utilizando a interface de comandos (terminal ou console) do sistema operacional ora em uso. Por exemplo, num sistema operacional da família Unix, pode-se digitar o seguinte comando numa janela de console:

```
touch teste.txt
```

Esse comando irá criar no diretório corrente um arquivo chamado `teste.txt` que não contém nada (pois nada foi escrito nele, de modo que o arquivo existe apenas como uma entrada no diretório). Se você usar seu sistema operacional para obter informações sobre esse arquivo verá que ele contém 0 KiB (i.e., ele ainda não contém nenhum dado armazenado). Entretanto se você abrir esse arquivo num editor de texto, introduzir um único caractere e salvá-lo, apesar de o único caractere que constitui o conteúdo do arquivo ocupar apenas um byte, o arquivo passará a ocupar 4 KiB no disco. Quer dizer, ele agora está ocupando um bloco no disco[16]. À medida que você acrescenta mais caracteres ao arquivo, o tamanho dele, obviamente, cresce, mas ele continuará ocupando os mesmos 4 KiB de espaço em disco até que o número de caracteres no arquivo ultrapasse 4 KiB. Nesse último caso, outro bloco será alocado para o arquivo em disco e o espaço utilizado em disco pelo mesmo arquivo passará a ser 8 KiB.

Isso significa, por exemplo, que se for utilizado um disco (hipotético) com espaço disponível de 40 KiB e formatado em blocos de 4 KiB, pode-se torná-lo repleto criando-se 10 arquivos contendo apenas 1 byte cada, de modo que serão desperdiçados aproximadamente 39.990 bytes que poderiam ser usados em novos arquivos. Nesse caso, é claro que se pode acrescentar mais dados em cada arquivo já criado até o limite de espaço disponível, mas não será possível acrescentar mais arquivos a esse disco hipotético. Assim pode-se concluir que blocos de 4 KiB são bem ineficientes para um disco de 40 KiB. Na vida real, isso não constitui um problema desde que tenham discos tão grandes que o número de blocos disponíveis exceda em grande quantidade o número de arquivos que se deseja armazenar em disco.

**Memória virtual** consiste em prover um espaço de endereçamento tão grande quanto a capacidade da memória secundária e transferir dados e instruções para a memória principal à medida que eles são acessados. Desse modo, o uso de memória virtual não limita o programador ao tamanho da memória interna. Trazendo-se dados e instruções para a memória principal, espera-se que eles sejam acessados novamente em breve. Portanto essa ação é motivada por localidade temporal (v. **Seção 1.5.1**).

Quando implementada usando caching e blocagem, memória virtual permite muitas vezes que se perceba memória secundária como sendo mais rápida do que ela realmente o é. Agora, sabe-se que, normalmente, memória principal é muito menor do que memória secundária. Além disso, comumente um programa chega a um ponto em que ele precisa de dados ou instruções que se encontram originalmente em memória secundária (virtual), mas que não se encontram em memória principal porque ela já está repleta de blocos. Para satisfazer essa necessidade, algum bloco que se encontra na memória principal deve ser removido para abrir espaço para um novo bloco proveniente da memória secundária. Existem inúmeros algoritmos e estruturas de dados que

[16] Supondo, evidentemente, que um bloco ocupa 4 KiB.

podem ser usados para lidar com esse problema, cuja solução é semelhante às políticas de desalojamento usadas com memória cache vistas na **Seção 1.4**.

### 1.5.4 Uso de Registradores

Todo computador possui um pequeno número de registradores que são pequenas unidades de armazenamento dentro da CPU. O computador utiliza seus registradores para efetuar operações lógicas e aritméticas. Por exemplo, a seguinte instrução em C:

```
z = x + y;
```

pode fazer com que os valores de `x` e `y` sejam carregados em dois registradores. Então o computador adiciona os valores contidos nestes dois registradores e armazena o valor resultante na posição de memória representada por z.

Geralmente, operações envolvendo registradores são muito mais rápidas do que operações envolvendo dados armazenados em memória principal. Infelizmente, o número de registradores em qualquer computador é muito limitado se comparado com a capacidade de memória do computador. Também, muito frequentemente, o número de variáveis em uso num programa é muito maior do que o número de registradores disponíveis. Portanto normalmente, não é possível manter todas as variáveis de um programa em registradores, como seria ideal. Bons compiladores são dotados de estratégias para decidir quais variáveis manter em registradores de forma a minimizar o acesso à memória principal.

Em C, a palavra-chave **register** serve para o programador sugerir ao compilador as variáveis que devem ser armazenadas em registradores. Entretanto o compilador tem liberdade para aceitar ou não tal sugestão. Os compiladores comportam-se de maneiras bastante variadas nesse aspecto. Por exemplo, alguns compiladores admitem que o programador utilize uma opção de compilação que especifica se o compilador deve tentar seguir a sugestão do programador ou usar sua própria estratégia de uso de registradores. Outros compiladores podem rejeitar categoricamente qualquer sugestão do programador e utilizar suas próprias estratégias de utilização de registradores.

Uma variável definida com **register** pode nunca ter um endereço em memória (i.e., pode ser que ela seja mantida num registrador durante todo seu tempo de vida). Portanto como registradores não possuem endereço, não se pode fazer referência ao endereço de uma variável declarada com o especificador **register**. Isto resultaria em erro de compilação, independentemente do fato de a variável ser realmente armazenada num registrador ou não.

O especificador **register** pode ser utilizado apenas com parâmetros de funções ou variáveis de duração automática e deve ser usado quando eles são acessados com muita frequência. Um caso típico de uso do especificador **register** é aquele de variáveis utilizadas como contadores em laços **for**, como, por exemplo:

```
register  int  i;
for  (i = 0; i <= 10000; i++){
   ...
}
```

Em princípio, não existe nenhum limite quanto ao número de variáveis ou parâmetros que podem ser qualificados com **register**. Na prática, se houver um número maior de variáveis e parâmetros qualificados com **register** do que o número de registradores disponível e o compilador aceitar as sugestões de alocação de registradores, ele irá considerar apenas as primeiras qualificações dessa natureza até que o número de registradores disponíveis seja atingido.

Nos dias atuais, muitos compiladores são suficientemente *inteligentes* para utilizar ótimas estratégias de alocação de registradores, de modo que o programador raramente precisa se preocupar com o uso da palavra-chave **register**.

### 1.5.5 Como Melhorar a Localidade de Referência de um Programa

Um programa com boa localidade de referência apresenta menos lapsos de cache e, assim, é mais rápido. Ou seja, programas que obtêm seus dados em níveis mais elevados de uma hierarquia de memória são executados mais rapidamente do que aqueles que assim o fazem em níveis inferiores dessa hierarquia. Portanto o programador deve escrever programas que sejam favoráveis ao uso de memória cache; i.e., programas que possuam boa localidade de referência. Esta seção apresenta algumas regras para julgamento de programas quanto a localidade de referência e sugestões sobre como o programador pode melhorar seus programas nesse aspecto.

Programas com boa localidade de referência satisfazem pelo menos os seguintes requisitos:

- ❒ Eles fazem referência às mesmas variáveis repetidamente e, assim, exibem uma boa localidade temporal.
- ❒ Quando processam arrays, eles usam o menor padrão de referência possível para obterem a melhor localidade espacial.
- ❒ Eles usam laços de repetição com corpos pequenos, pois quanto menor for o corpo do laço e o número de iterações, melhores serão as localidades espacial e temporal com relação a instruções.

O programador que conhece bem o funcionamento de hierarquias de memória usa esse conhecimento para escrever programas eficientes, independentemente da organização específica de memória utilizada. Programadores conscientes podem acelerar seus programas usando localidade espacial e temporal seguindo as seguintes recomendações:

- ❒ Concentre-se na otimização de laços internos, nos quais a maior carga de processamento ocorre.
- ❒ Melhore a localidade espacial acessando dados sequencialmente com padrão de referência 1 na ordem em que eles são armazenados em memória. Padrões de referência 1 são bons porque todas as memórias cache de uma hierarquia de memória armazenam dados em blocos contíguos.
- ❒ Melhore a localidade espacial usando dados repetidamente logo após eles terem sido acessados pela primeira vez.
- ❒ Variáveis locais e parâmetros usados repetidamente devem ser qualificados com a palavra-chave **register** para sugerir ao compilador que eles sejam carregados em registradores. Mas bons compiladores não precisam desse auxílio.

## 1.6 Análise de Algoritmos em Memória Secundária

No **Capítulo 6** do **Volume 1** desta obra, mostra-se como analisar custos temporais e espaciais de algoritmos que lidam com dados disponíveis exclusivamente em memória interna. Resumidamente, nesse caso, o custo temporal de um algoritmo é proporcional ao número de instruções que ele requer para processar dados de um certo tamanho $n$. O acesso aos dados em si é irrelevante nessa análise, visto que se considera que os dados estão prontamente disponíveis em memória. Além disso, o custo espacial de um algoritmo, medido em termos do espaço adicional que ele usa para processar seus dados, pode ser importante porque, conforme foi visto na **Seção 1.3**, o custo desse espaço em memória interna é relativamente dispendioso. Por sua vez, o custo espacial de algoritmos que atuam em memória secundária é, normalmente, negligenciado.

A análise convencional de algoritmos não leva localidade de referência em consideração e assume que tempos de acesso a dados são iguais independentemente de onde eles se encontram armazenados numa hierarquia de memória. Existem duas justificativas para essa simplificação:

1. Informações sobre uma organização específica de memória são dependentes de hardware e difíceis de obter.
2. Sistemas operacionais possuem mecanismos baseados em localidade de referência que facilitam o acesso rápido a dados, como foi descrito na **Seção 1.5**.

A análise assintótica deixa de fazer sentido quando grande parte dos dados processados é mantida em memória secundária, pois esse tipo de análise assume que todas as operações possuem o mesmo custo de processamento. Ocorre, porém, que a análise de algoritmos que precisam acessar dados armazenados em memória secundária deve ser baseada em outros parâmetros devido a dois fatores principais:

1. Como foi visto na **Seção 1.1**, o custo de transferência de dados entre memória principal e memória secundária é muito maior do que o custo de execução de uma instrução. De fato, no tempo gasto para acessar um bloco armazenado em disco, por exemplo, podem ser executadas bilhões de instruções. Assim efetuar um número elevado de operações em memória principal torna-se irrelevante em face de um único acesso a disco.
2. Análise de custo espacial tradicional tem pouca importância aqui, porque o custo de espaço em memória secundária é relativamente muito baixo.

Deve-se ainda levar em consideração que, quando se lê um byte em memória secundária, um bloco inteiro, cujo tamanho típico é 4 KiB, é transferido dessa memória para a memória principal. Assim sendo, se os dados forem organizados cuidadosamente, como será mostrado no **Capítulo 6**, cada acesso à memória secundária pode resultar no carregamento em memória principal de 4 KiB que serão úteis no complemento de qualquer operação que esteja em andamento.

O que foi exposto acima constitui a ideia que norteia o modelo de análise de algoritmos que atuam em memória secundária ilustrado na **Figura 1–27**. Nesse modelo, o computador tem acesso a uma memória secundária de grande capacidade na qual os dados residem. Essa memória é dividida em blocos de memória, sendo que cada um deles contém $B$ bytes. O computador também possui memória interna de capacidade limitada na qual os dados devem ser carregados logo antes de ser processados. A transferência de um bloco entre as memórias interna e externa leva um tempo considerado constante (apesar de ser relativamente enorme). Por outro lado, o custo temporal de processamento efetuado em memória interna é considerado irrelevante.

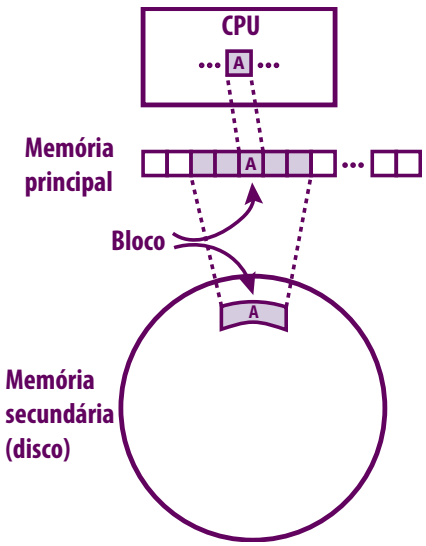


**Figura 1–27: Leitura de uma Variável em Memória Secundária**

À primeira vista, o fato de se considerar o processamento em memória interna sem custo temporal pode parecer um tanto estranho, mas isso simplesmente enfatiza o fato de acessos à memória secundária serem bem mais lentos do que acessos à memória interna. Desse modo, a diferença em termos de tempo de acesso entre disco e memória interna é tão grande que é aceitável executar um número considerável de acessos à memória interna se eles permitem evitar algumas poucas transferências de dados de memória secundária.

Concluindo, um algoritmo que processa dados armazenados em memória secundária deve tentar efetuar o menor número possível de acessos, mesmo que isso se dê a custa da execução de um maior número de instruções.

# 1.7 Exemplo de Programação

## 1.7.1 Medindo Tempo de Execução

**Problema:** Escreva uma função que mede intervalos de tempo entre duas chamadas consecutivas dela.

**Solução:** A função `MedidaDeTempo()` apresentada abaixo atende aquilo que foi solicitado. Seu único parâmetro é a informação que será apresentada na tela acompanhando a medida de tempo.

```
void MedidaDeTempo(const char *info)
{
   static int    numeroDeChamadas = 0;
   static time_t t0; /* Instante inicial */
   time_t        tf; /* Instante final   */
   double        intervalo;

   if (info) {
      printf("\n>>> %s\n", info);
      fflush(stdout);
   }

   ++numeroDeChamadas; /* Esta função foi chamada mais uma vez */

      /* Se o número da chamada for ímpar, esta chamada é a primeira de um par */
   if (numeroDeChamadas%2) { /* Calcula o instante inicial */
      t0 = time(NULL);
   } else{/* Calcula e apresenta o intervalo de tempo entre as duas últimas chamadas */
      tf = time(NULL);

      intervalo = difftime(tf, t0);

      if (intervalo < 60)
         printf("\n\t*** Tempo gasto na operacao: %5.2f segundos ***\n", intervalo);
      else
         printf("\n\t*** Tempo gasto na operacao: %5.2f minutos ***\n", intervalo/60);
   }
}
```

Um programa que contenha a função `MedidaDeTempo()` precisa incluir o cabeçalho `<time.h>` para permitir o uso das funções **time()** e **difftime()**. Note que as variáveis `numeroDeChamadas` e `t0` têm duração fixa, pois elas precisam manter seus valores entre uma chamada da função e a próxima chamada.

# 1.8 Exercícios de Revisão

### Meios de Armazenamento (Seção 1.1)

1. O que é memória RAM?
2. (a) O que é memória SRAM? (b) Em que dispositivos memórias SRAM são tipicamente usadas?
3. (a) O que é memória DRAM? (b) Em que dispositivos memórias DRAM são tipicamente usadas?
4. (a) Que vantagens apresenta uma memória SRAM? (b) Quais são suas desvantagens?
5. (a) Que vantagens apresenta uma memória DRAM? (b) Quais são suas desvantagens?

**6.** (a) O que é um meio de armazenamento volátil? (b) O que é um meio de armazenamento não volátil? (c) Quais são os meios de armazenamento voláteis apresentados neste capítulo? (d) Quais são os meios de armazenamento apresentados neste capítulo que não são voláteis?

**7.** Por que *memória de acesso aleatório* pode ser uma denominação enganosa para memória RAM?

**8.** Em que diferem as memórias PROM, EPROM e EEPROM?

**9.** Por que *memória apenas para leitura* é uma denominação enganosa para as memórias ROM atuais?

**10.** Uma cabeça de leitura/escrita de um disco pode ser movida independentemente das demais cabeças de leitura/escrita do mesmo disco? Explique sua resposta.

**11.** Descreva os seguintes componentes de um disco magnético:
- (a) Prato
- (b) Superfície
- (c) Braço
- (d) Cabeça de leitura/gravação

**12.** Descreva os seguintes componentes geométricos de um disco magnético:
- (a) Trilha
- (b) Setor
- (c) Cilindro
- (d) Bloco
- (e) Espaçamento
- (f) Intervalo

**13.** Qual é a diferença entre setor geométrico e setor de trilha?

**14.** No contexto de discos magnéticos, o que é um cluster?

**15.** O que é um sistema de arquivos?

**16.** Descreva os seguintes conceitos relativos a discos magnéticos:
- (a) Densidade de gravação
- (b) Densidade de trilha
- (c) Densidade de área

**17.** Como se determina a capacidade de um disco magnético?

**18.** Determine a capacidade de um disco magnético com as seguintes características:

| | |
|---|---|
| Número de pratos | 3 |
| Número de superfícies por prato | 2 |
| Número de bytes por setor | 512 |
| Número de trilhas por superfície | 20.000 |
| Número de setores por trilha | 300 |

**19.** (a) O que é tempo de posicionamento num disco magnético? (b) O que é atraso rotacional num disco magnético? (c) O que é latência rotacional num disco magnético?

**20.** Considerando o acesso a discos magnéticos, descreva os seguintes intervalos de tempo:
- (a) Tempo de posicionamento
- (b) Atraso rotacional
- (c) Latência rotacional

(d) Tempo de transferência

**21.** Como se calcula o tempo de transferência de dados num disco magnético?

**22.** O que leva mais tempo para ser acessado um arquivo cujos blocos constituintes fazem parte de um mesmo cilindro ou um arquivo cujos blocos constituintes fazem parte de uma mesma superfície de um disco? Explique sua resposta.

**23.** O que é memória flash?

**24.** O que é um disco de estado sólido (SSD)?

**25.** Por que um disco SSD não pode ser considerado exatamente um *disco*?

**26.** Em que diferem discos magnéticos de discos SSD?

**27.** Tecnicamente, o que é um bastão de memória USB (*pen drive*)?

**28.** Em que situações fitas magnéticas são usadas atualmente?

**29.** Unidades de fitas magnéticas são dispositivos periféricos muito antigos, lentos e só permitem acesso sequencial. O que ainda justifica seus usos nos dias atuais?

**30.** Determine a capacidade de um disco rígido com dois pratos, *10.000* cilindros, *400* setores por trilha e *512* bytes por setor.

**31.** Estime o tempo médio em milissegundos que leva para acessar um bloco de um HD com as seguintes propriedades:

| | |
|---|---|
| Taxa rotacional | *12.000* RPM |
| Tempo médio de posicionamento | *8* ms |
| Número médio de setores por trilha | 600 |

**32.** O que é taxa de transferência de dados?

**33.** O que é é o tempo médio de falha (MTTF)?

**34.** Um fabricante de disco afirma que um de seus modelos apresenta MTTF de 1.000.000 horas. O que isso significa?

**35.** Como um sistema de arquivos interpreta o conceito de bloco?

**36.** Fisicamente, o que é um arquivo?

**37.** Qual é a unidade de transferência de dados entre memória principal e um disco?

**38.** O que é um padrão de acesso sequencial? O que é um padrão de acesso direto?

**39.** O que é uma página no contexto de processamento de arquivos?

**40.** Por que acesso sequencial é bem mais rápido do que acesso direto?

**41.** Por que acesso direto é frequentemente denominado *acesso aleatório*?

**42.** Descreva as técnicas utilizadas para acelerar o acesso a blocos de um disco.

**43.** O que é fragmentação de arquivo?

**44.** Por que é desejável que um registro faça parte de apenas um bloco?

### Acesso a Dispositivos de Entrada e Saída (Seção 1.2)

**45.** (a) O que é um dispositivo de entrada de dados? (b) O que é um dispositivo de saída de dados?

**46.** Qual é o papel desempenhado pelo barramento de entrada e saída num computador?

**47.** Para que serve o controlador de USB de um computador?

**48.** O que é adaptador SCSI/SATA?

**49.** (a) O que é operação de entrada? (b) O que é operação de saída?

**50.** Descreva a sequência de eventos que ocorre quando o sistema operacional efetua uma leitura num disco magnético.

**51.** (a) O que significa formatar um disco magnético? (b) O que ocorre durante esse processo?

**52.** O que é acesso direto à memória (DMA)?

**53.** O que é um sinal de interrupção?

**54.** (a) O que é um buffer? (b) Qual é a diferença entre buffer e cache?

### Hierarquias de Memória (Seção 1.3)

**55.** O que é uma hierarquia de memória?

**56.** Apresente duas hierarquias de memória diferentes daquela apresentada no texto.

**57.** Por que um programador deve entender o funcionamento de hierarquias de memória?

**58.** Qual é a diferença entre memória externa, memória secundária e disco magnético?

### Caching (Seção 1.4)

**59.** (a) O que é memória cache? (b) O que é caching?

**60.** Qual é a relação entre caching e hierarquia de memória?

**61.** (a) O que é acerto de cache? (b) (a) O que é lapso de cache? (c) O que é memória cache fria?

**62.** Por que muitas vezes lapsos de cache são inevitáveis?

**63.** O que é desalojamento de bloco?

**64.** O que é política de substituição de memória cache?

**65.** Descreva as principais abordagens de implementação de políticas de substituição de memória cache apresentando as vantagens e desvantagens de cada uma delas.

### Localidade de Referência (Seção 1.5)

**66.** O que é localidade de referência?

**67.** (a) O que é localidade espacial? (b) O que é localidade temporal?

**68.** O que garante a um programa uma boa localidade espacial?

**69.** O que diz o Princípio de Pareto aplicado a processamento de dados?

**70.** Apresente três técnicas que um programador pode usar para melhorar a localidade espacial de seus programas.

**71.** (a) O que é padrão de referência sequencial? (b) O que é padrão de referência de ordem *n*?

**72.** (a) Para que serve a palavra-chave **register** em C? (b) Quando é recomendável que o programador use essa palavra-chave?

**73.** O que é localidade de acesso a instruções?

**74.** O que é blocagem?

**75.** Como se determina o tamanho de um bloco num sistema de arquivos?

**76.** (a) Qual é o tamanho de um arquivo logo após ele ser criado? (b) Qual é o tamanho de um arquivo logo após um byte ter sido acrescentado a ele?

**77.** A função `Transposta()` a seguir calcula a transposta da matriz representada pelo segundo parâmetro. (a) Qual é o padrão de referência dessa função? (b) Esse padrão de referência pode ser melhorado? Explique seu raciocínio.

```
typedef int tMatriz[200][200];

void Transposta(tMatriz transposta, tMatriz matriz)
{
   int i, j;

   for (i = 0; i < 200; i++)
      for (j = 0; j < 200; j++)
         transposta[j][i] = matriz[i][j];
}
```

**78.** As três funções a seguir são funcionalmente equivalentes, mas elas executam as mesmas operações com diferentes graus de localidade espacial. Ordene essas funções com relação à localidade espacial que elas apresentam. Explique seu raciocínio.

```
typedef struct {
           int x[3];
           int y[3];
        } tPonto;

void F1(tPonto *p, int n)
{
   int i, j;

   for (i = 0; i < n; i++) {
      for (j = 0; j < 3; j++)
         p[i].x[j] = 0;

      for (j = 0; j < 3; j++)
         p[i].y[j] = 0;
   }
}
void F2(tPonto *p, int n)
{
   int i, j;

   for (i = 0; i < n; i++)
      for (j = 0; j < 3; j++) {
         p[i].x[j] = 0;
         p[i].y[j] = 0;
      }
}
void F3(tPonto *p, int n)
{
   int i, j;

   for (j = 0; j < 3; j++) {
      for (i = 0; i < n; i++)
         p[i].x[j] = 0;

      for (i = 0; i < n; i++)
         p[i].y[j] = 0;
   }
}
```

**79.** Por que a palavra-chave **register** perdeu seu significado prático?

**80.** (a) O que é memória virtual? (b) Qual é a relação entre memória virtual e caching?

**81.** O que há de comum entre acesso sequencial a disco e acesso sequencial aos elementos de um array?

**82.** Supondo que as constantes simbólicas `M` e `N` sejam definidas antes das funções a seguir, verifique se cada uma delas apresenta boa localidade de referência.

**(a)**

```
int SomaLinhas(int a[M][N])
{
   int i, j, soma = 0;

   for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
         soma += a[i][j];

   return soma;
}
```

**(b)**

```
int SomaColunas(int a[M][N])
{
   int i, j, soma = 0;

   for (j = 0; j < N; j++)
      for (i = 0; i < M; i++)
         soma += a[i][j];

   return soma;
}
```

**83.** É possível rearranjar os laços **for** da função `SomaArrayTri()` apresentada abaixo de modo que essa função exiba padrão de referência sequencial no acesso ao array tridimensional recebido como parâmetro?

```
#define N 10

int SomaArrayTri(int ar[][N][N], int n1, int n2, int n3)
{
   int  i, j, k, soma = 0;

   for (i = 0; i < n1; i++)
      for (j = 0; j < n2; j++)
         for (k = 0; k < n3; k++)
            soma += ar[k][i][j];

   return soma;
}
```

**84.** A função `MultiplicaMatrizes()` a seguir multiplica duas matrizes representadas por arrays bidimensionais. Como se pode melhorar o padrão de referência dessa função?

```
#define N 100

void MultiplicaMatrizes(int C[N][N], const int A[N][N], const int B[N][N])
{
   int i, j, k;

   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
         for (k = 0; k < N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
```

**85.** Qual é o tipo de programa que não precisa se preocupar com localidade de referência?

### Análise de Algoritmos em Memória Secundária (Seção 1.6)

**86.** O que justifica o fato de a análise de algoritmos convencional não levar em conta localidade de referência?

**87.** Por que a análise de algoritmos tradicional em termos de análise assintótica não é conveniente para algoritmos que processam dados armazenados em memória secundária?

**88.** Por que a análise de algoritmos que processam dados armazenados em memória secundária considera o custo de temporal de execução de instruções igual a zero?

**89.** Como se avalia a eficiência de um algoritmo de memória secundária?

### Exemplo de Programação (Seção 1.7)

**90.** Descreva o funcionamento da função `MedidaDeTempo()`.

**91.** Por que as variáveis `numeroDeChamadas` e `t0` da função `MedidaDeTempo()` precisam ser definidas com static?