SCC 503 - ALGORITMOS E ESTRUTURAS DE DADOS II

Arquivos - Fundamentos

Prof.: Leonardo Tórtoro Pereira
leonardop@usp.br

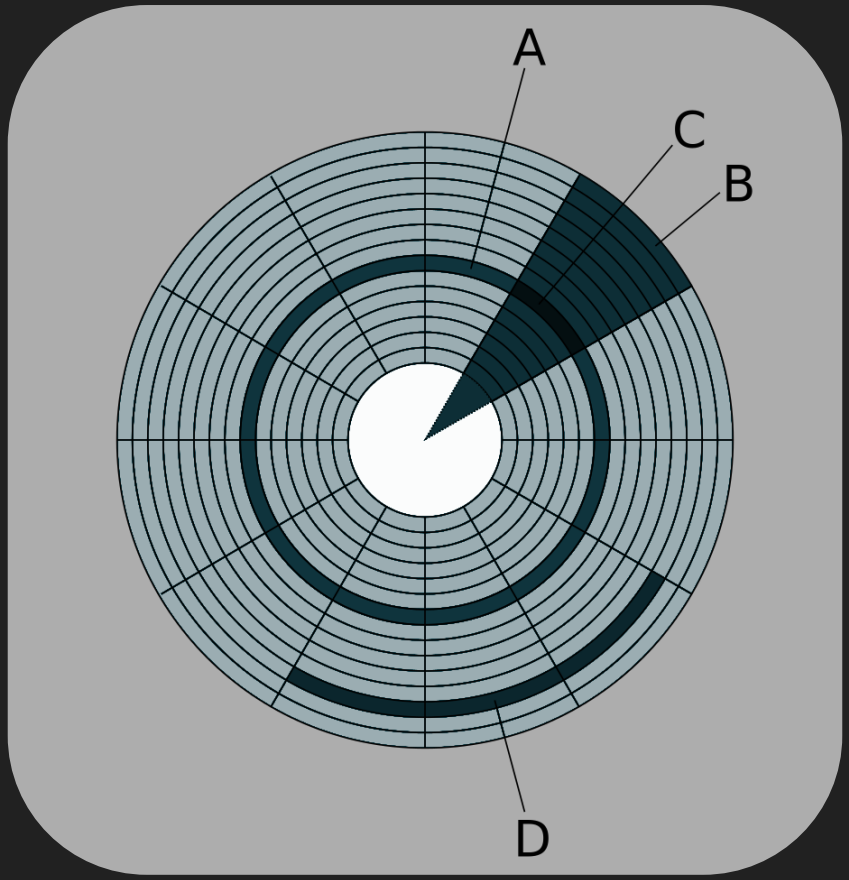
*Material baseado em aulas dos professores: Elaine Parros Machado de Souza, Gustavo Batista, Robson Cordeiro, Moacir Ponti Jr., Maria Cristina Oliveira e Cristina Ciferri.

Recapitulando

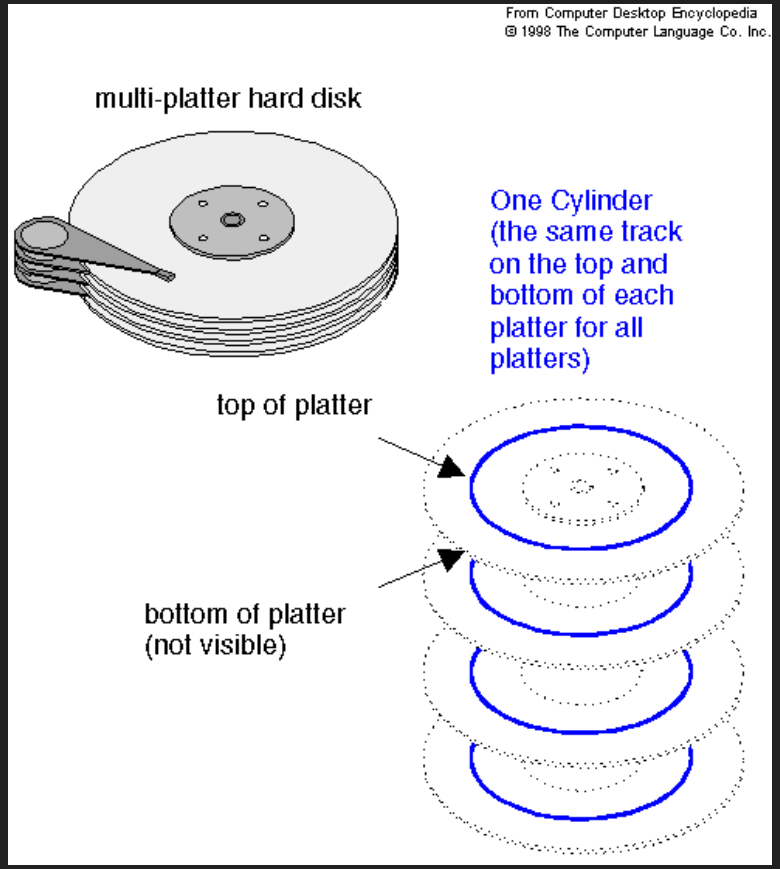
Recapitulando

- ➔ Arquivos são mais baratos e maiores, porém mais lentos
- ➔ Mesmo com os SDDs mais avançados, não é a mesma coisa que acessar a memória principal (pelo menos 1.000 vezes mais lento)
- ➔ RAM é menor e mais cara, porém mais rápida

Visão Geral de um HDD



A - Trilha; B - Setor Geométrico; C - Setor do Disco; D - Cluster
Fonte: https://en.wikipedia.org/wiki/Disk_sector



Um Cilindro - informação em qualquer prato pode ser acessada sem mover o braço!

O Que Queremos?

O Que Queremos?

➔ Precisamos encontrar uma maneira de otimizar o acesso aos arquivos, especialmente reduzindo o tempo de busca, se quisermos usar bem os arquivos!

O Que Queremos?

- ➔ Idealmente, conseguir a informação com 1 acesso ao disco
- ➔ Se não for possível em 1 acesso, queremos uma estrutura que permita encontrar a informação buscada com o mínimo possível de acessos
- ➔ Queremos informações agrupadas, aumentando a chance de pegar tudo que precisamos com 1 única `visita´ ao disco

Sistema de Arquivos

Sistema de Arquivos

- ➔ Formatação Física
  - ◆ Organização do disco em setores/trilhas (vem da fábrica)
- ➔ Formatação Lógica
  - ◆ 'Instala' o sistema de arquivos no disco:
    - Subdivide o disco em regiões endereçáveis
    - Grava estruturas de gerenciamento dos arquivos
    - *Overhead* de espaço ocupado com informações para gerenciamento

Sistema de Arquivos

- ➔ Faz parte do sistema operacional (S.O.)
- ➔ Fornece a infraestrutura básica para a manipulação de arquivos em memória secundária via software
- ➔ Oferece um conjunto de operações para a manipulação de arquivos
  - ➔ Que operações são essas?

Operações Sistema de Arquivos

criar (create, open)	destruir ou remover (delete)
renomear (rename)	abrir (open)
fechar (close)	ler dados (read)
escrever dados (write)	escrever dados no final (append)
posicionar (seek)	…

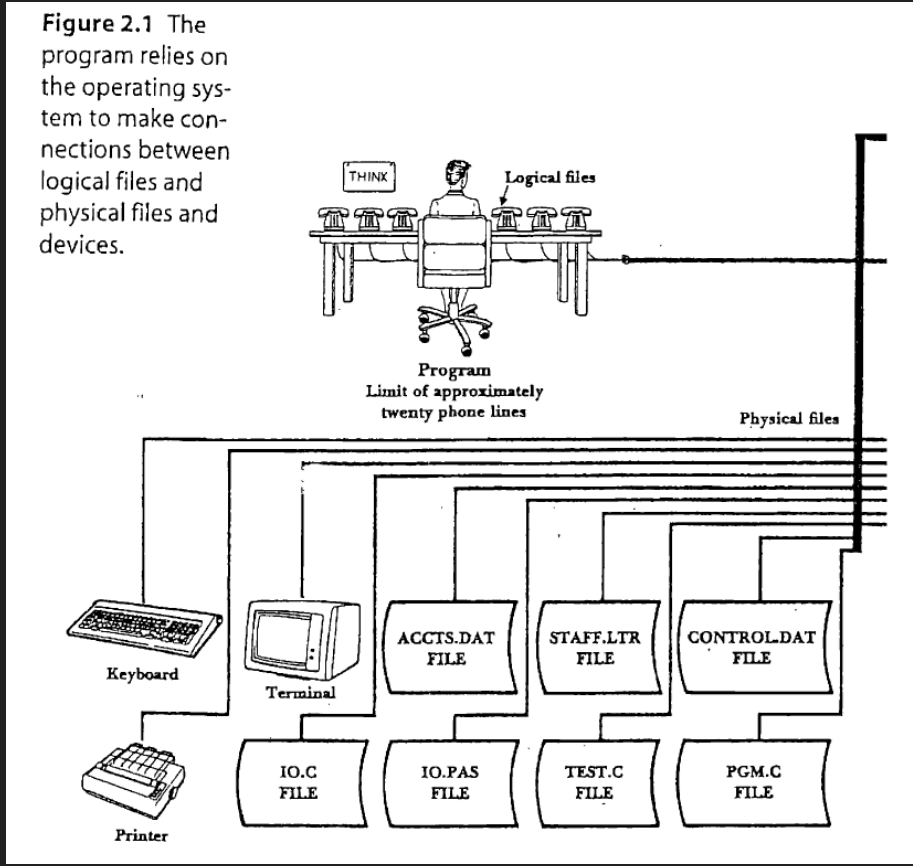
Arquivos Físico e Lógico

Arquivos Físico e Lógico

- Arquivo Físico:
  - Sequência de bytes armazenada no disco em blocos distribuídos em trilhas/setores.
- Arquivo Lógico:
  - Arquivo como visto pelo aplicativo que o acessa – sequência contínua de registros ou bytes.
- Associação arquivo físico/arquivo lógico:
  - Gerenciada pelo Sistema de Arquivo/S.O.

Arquivos Físico e Lógico

- ➔ Para uma aplicação, um arquivo é como uma conexão de telefone em uma rede de telefonia
- ➔ O programa pode receber e enviar bytes pela linha
  - ◆ Mas não sabe para onde eles vão ou da onde eles vem
- ➔ O programa sabe apenas sobre o fim da sua linha
- ➔ Essa linha é o “arquivo lógico”

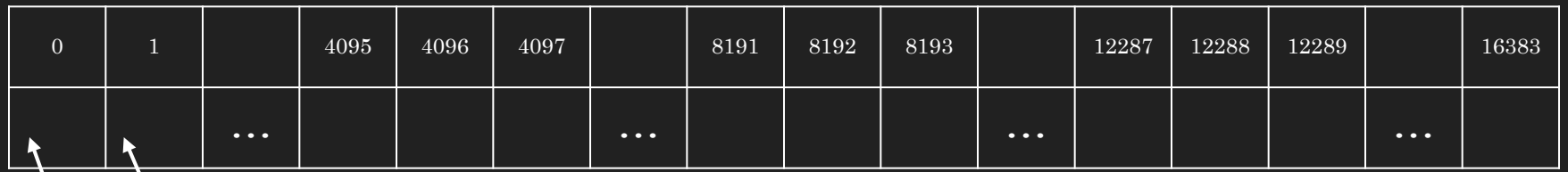


Analogia de arquivo lógico e físico

Fonte: M. J. Folk, B. Zoellick and G. Riccardi. File Structures: An object-oriented approach with C++, Addison Wesley, 1998.

Arquivo Físico

➔ Sequência de bytes



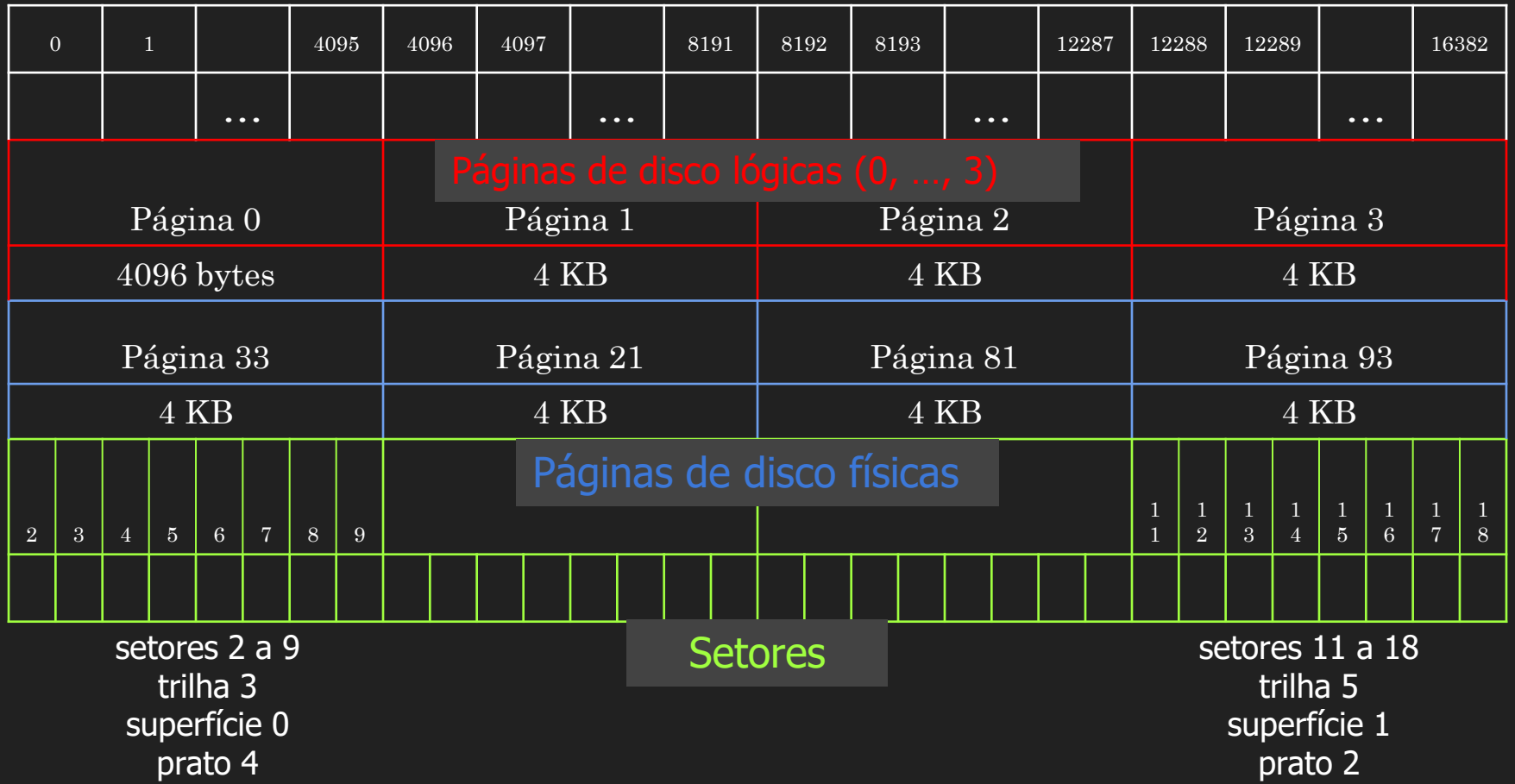
tamanho:
16.384 bytes ou 16KB

Página de Disco

- ➔ Conjunto de setores logicamente contíguos no disco
- ➔ Um arquivo é visto pelo sistema de arquivos como um conjunto de páginas de disco
  - ◆ Arquivos são alocados em uma ou mais páginas de disco
- ➔ Unidade de transferência de dados entre a RAM e o disco

Também chamada de bloco de disco ou cluster

Página de Disco



Mapeamentos

- ➔ Páginas lógicas ⇔ páginas físicas
  - ◆ Depende da técnica de alocação de espaço em disco
  - ◆ Ex.: alocação contígua, alocação encadeada e alocação indexada
- ➔ Páginas físicas ⇔ setores
  - ◆ Feito pelo *device driver* (componente do SO)
  - ◆ O setor é a menor unidade endereçável de um disco
    - ● Um *read* recupera todo o conteúdo de um setor e salva esse conteúdo em um *buffer*

Posição Atual no Arquivo

- Abstração que permite a especificação de uma chamada do sistema para indicar em que ponto exatamente um arquivo deve ser lido ou escrito

Posição Atual no Arquivo

➔ Características

- A leitura e escrita acontecem a partir da posição atual
- A posição atual é então avançada para imediatamente após o último byte lido ou escrito
- É possível informar um endereço específico a ser lido, o qual faz com que a posição atual seja a informada no endereço

Clusters

Tamanho de Página de Disco (Cluster)

- ➔ Definido pelo S.O. na formatação do disco
- ➔ Exemplo
  - ◆ (FAT Windows): sempre uma potência de 2
    - 2, 4, 8, 16 ou 32KB
  - ◆ Determinado pelo máximo que a FAT consegue manipular, e pelo tamanho do disco
    - FAT16: pode endereçar 2^{16} clusters (páginas) = 65.536 clusters

Quanto maior o tamanho da página de disco, maior a fragmentação interna, porém o número de acessos a disco tende a ser menor!

Ex: FAT– File Allocation Table

- Cada entrada na tabela possui a localização física do cluster (página) associado a um arquivo lógico
- 1 *seek* para localizar 1 cluster:
  - Todos os setores do cluster são lidos sem necessidade de *seeks* adicionais.

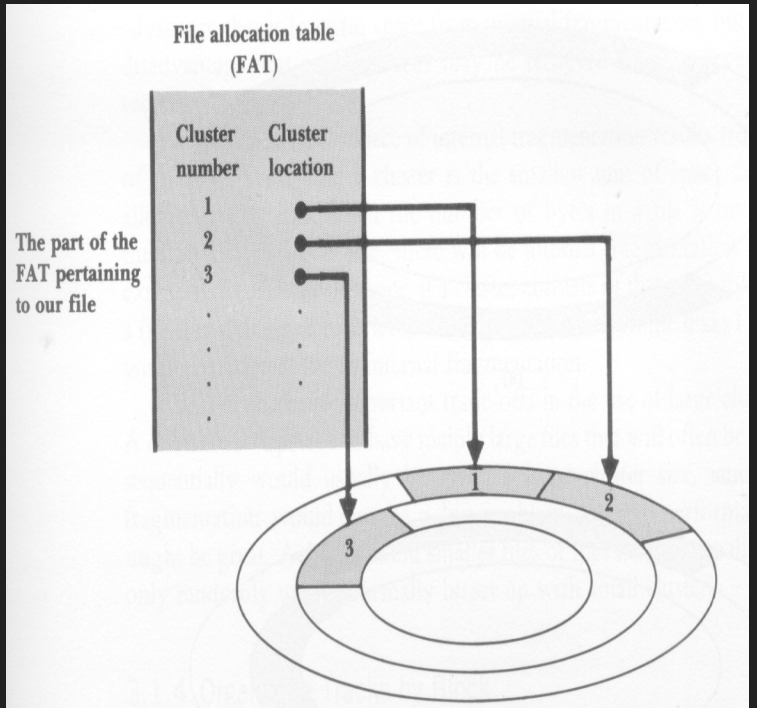


FIGURE 3.5 The file manager determines which cluster in the file has the sector that is to be accessed.

Ex: Sistemas de arquivos

- ➔ FAT32 (Windows 95 e posteriores):
  - clusters de tamanho menor, endereça mais clusters, menos fragmentação interna
- ➔ NTFS (New Technology File System):
  - Sistemas OS/2 (IBM) e Windows NT;
  - Mais eficiente: a menor unidade de alocação é o próprio setor de 512 bytes

Ex: Sistemas de arquivos

- ➔ EXT (Extended File System)
  - ◆ GNU/Linux
  - ◆ ..EXT3, EXT4
- ➔ HFS (Hierarchical File System)
  - ◆ MAC
- ➔ ...

Extent

Extent

- ➔ Sequência de páginas consecutivas no disco, alocadas para o mesmo arquivo.
- ➔ 1 *seek* recupera 1 *extent*
- ➔ Situação ideal:
  - ◆ 1 arquivo ocupar 1 *extent*
  - ◆ Normalmente não é possível, o arquivo é espalhado em vários *extents* pelo disco

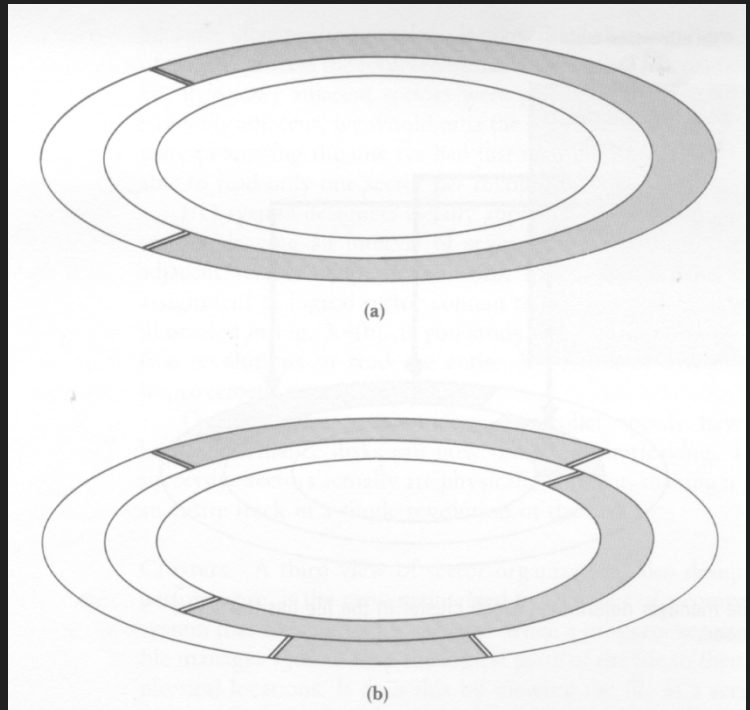


FIGURE 3.6 File extents (shaded area represents space on disk used by a single file).

Fragmentação Interna

- ➔ Perda de espaço útil decorrente da organização do arquivo em páginas de disco de tamanho fixo
- ➔ Exemplo
  - ◆ Página de disco de 4K (4.096 bytes)
  - ◆ Arquivo que ocupa 1 byte: desperdício de 4.095 bytes nessa página de disco

Cálculo de Tempo de Acesso

Exemplo de Cálculo de Tempo de Acesso

- Supor duas situações:
  - O arquivo está gravado sequencialmente em um disco
  - O arquivo está espalhado pelo disco, cada página em uma trilha diferente
- Objetivo:
  - Comparar os tempos de acesso e a influência do tempo de busca (*seek*)

Exemplo de Cálculo de Tempo de Acesso

- ➔ Arquivo:
  - ◆ 8.704 Kbytes
  - ◆ 34.000 registros de 256 bytes
- ➔ HD/S.O.:
  - ◆ 1 setor = 512 bytes
  - ◆ 1 página = 8 setores = 4.096 bytes
  - ◆ 1 trilha = 170 setores

Exemplo de Cálculo de Tempo de Acesso

➔ São necessárias:

- Quantas páginas?
- Quantas trilhas?

Exemplo de Cálculo de Tempo de Acesso

- ➔ Resposta:
  - ◆ São necessários:
    - 2.125 páginas ou
    - 100 trilhas

Primeira Situação

Exemplo de Cálculo de Tempo de Acesso

- Primeira situação (sequencial):
  - O arquivo está disposto em 100 trilhas
- Supor os seguintes tempos:
  - Tempo de busca médio (*seek*): 8 ms
  - *Delay* de rotação (latência): 3 ms
  - Tempo de leitura de 1 trilha (transf.): 6 ms
- Tempo total ???

Exemplo de Cálculo de Tempo de Acesso

- ➔ Tempo total = número de trilhas * Total de processamento por trilha
- ➔ Processamento por trilha
  - ◆ *Seek* + Latência + Transferência
- ➔ Tempo total = 100 * (8 + 3 + 6) = 100 * 17
- ➔ Tempo total = 1.700ms = 1,7s

Segunda Situação

Exemplo de Cálculo de Tempo de Acesso

- ➔ Segunda situação (espalhados):
  - ◆ 2.125 páginas alocadas dispersas pelo disco
- ➔ Supor os seguintes tempos:
  - ◆ Tempo de busca médio: 8 ms
  - ◆ *Delay* de rotação: 3 ms
  - ◆ Tempo de leitura de 1 página: 0,28 ms
- ➔ Tempo total = ????

Exemplo de Cálculo de Tempo de Acesso

- ➔ Tempo total = número de páginas * Total de processamento por página
- ➔ Processamento por página
  - ◆ *Seek* + Latência + Transferência
- ➔ Tempo total = 2125 * (8 + 3 + 0,28) = 2.125 * 11,28
- ➔ Tempo total = 23.900ms = 23,9s

Arquivo Lógico

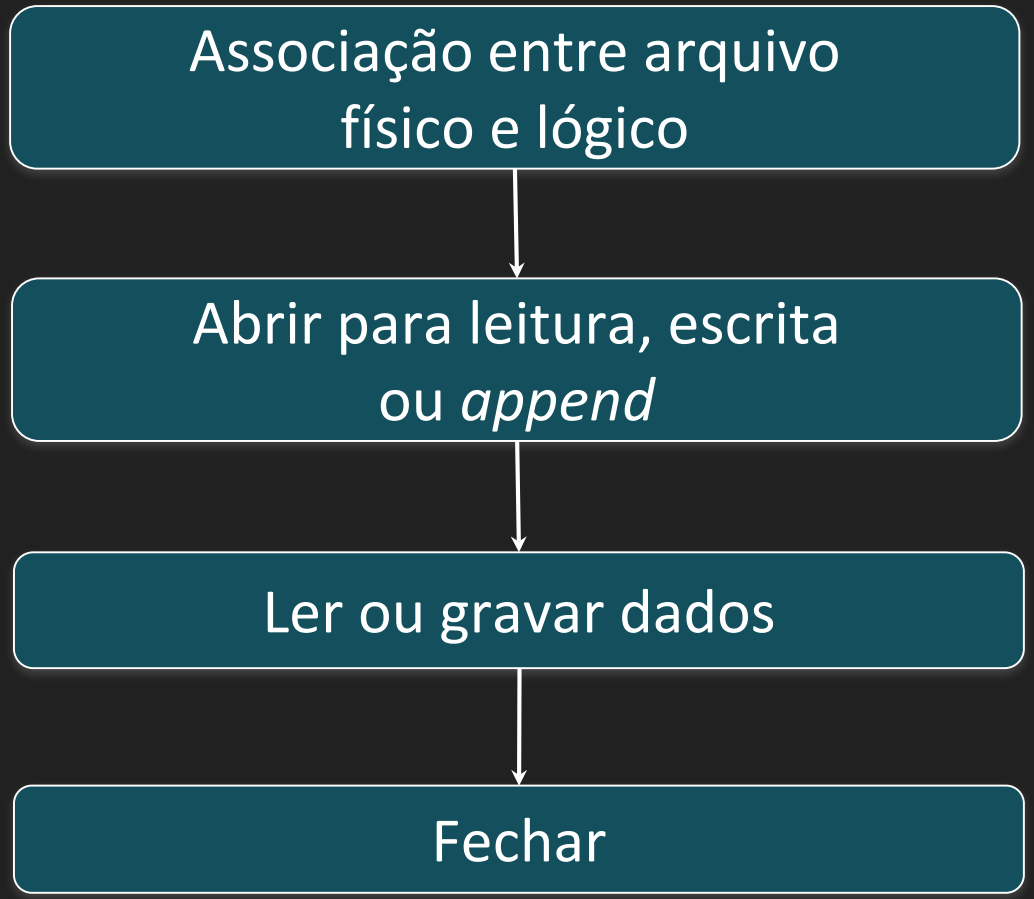
Arquivo Lógico

- ➔ Maioria das linguagens de programação
  - ◆ Informações lidas e gravadas em arquivos como *streams* de bytes
    - Gerenciar arquivos
      - Gerenciar esses *streams*
- ➔ Não importa qual a fonte e organização do arquivo físico!
- ➔ Tudo isso é abstraído pelo SO (ainda bem!)

Arquivo Lógico

➔ Operações comuns sobre arquivos:

- Abrir / Fechar
- Ler/Gravar
- Verificar situações de erro
- …



Operações Sobre Arquivos

Associação entre arquivo físico e lógico

➔ Relembrando.... em C:

- Associa e abre para leitura

```
FILE *p_arq;
if  ((p_arq=fopen(      "meuarq.dat"     ,   "r"   ))==  NULL)
        printf(      "Erro na abertura do arquivo            \ n" );
else
```

Abertura de Arquivos

- ➔ Arquivo novo (escrita) ou arquivo já existente (leitura ou escrita)
  - ◆ Em C
    - `fopen` – comando da linguagem (stdio.h)
    - Parâmetros indicam o modo de abertura

Função Open

- fd=fopen(<filename>,<flags>)
  - filename: nome do arquivo a ser aberto;
  - flags: controla o modo de abertura;

Função Open

r	Abre para leitura. O arquivo precisa existir
w	Cria um arquivo vazio para escrita
a	Adiciona conteúdo ao final arquivo (append)
r+	Abre o arquivo para leitura e escrita
w+	Cria um arquivo vazio para leitura e escrita
a+	Abre um arquivo para leitura e adição (append)
t	Modo texto (default)
b	Modo binário – o fim do arquivo é o último byte

Fechamento de Arquivos

➔ Encerra a associação entre arquivos lógico e físico

- Todas as informações são atualizadas e salvas
- Conteúdo dos buffers de E/S enviados para o arquivo.

Fechamento de Arquivos

- ➔ S.O. fecha o arquivo se o aplicativo não o fizer ao final da execução do programa
- ➔ Interessante para:
  - ◆ Garantir que os buffers sejam descarregados
  - ◆ Liberar os espaços alocados ao arquivo para outros arquivos
- ➔ É importante fechar ainda assim para evitar perda de dados por interrupção e liberar os nomes lógicos cedo

Fechamento de Arquivos

```
int     main ()
{
    FILE * pFile;
    pFile = fopen (         "myfile.txt"          , "w" );
    if     (pFile!=NULL)
    {
        fclose (pFile);
    }
    return      0;
}
```

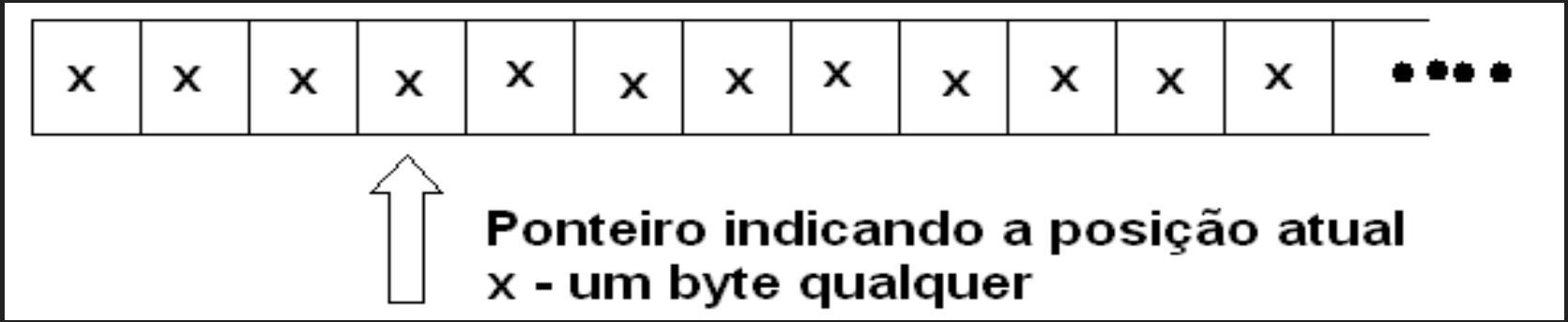
Leitura e Escrita

- Várias funções para L/E sequencial...
  - fputs(), fgets()
  - fwrite(), fread()
  - fprintf(), fscanf()

- Acesso a posições arbitrárias no arquivo
  - fseek()

Seeking

O Ponteiro no Arquivo Lógico



Acesso Sequencial vs Aleatório

- ➔ Leitura sequencial: ponteiro de leitura avança byte a byte (ou por blocos), a partir de uma posição inicial
- ➔ Acesso aleatório (direto): posicionamento em um byte ou registro arbitrário

Seeking

- fseek(*stream*, *offset* , *origin*)
  - Coloca o indicador de posição associado ao *stream* numa nova posição
  - Para *streams* abertas no modo binário, a nova posição é definida adicionando um *offset* para uma posição de referência especificada por *origin*

Seeking

- fseek(*stream*, *offset* , *origin*)
  - Constantes de *origin*
    - **SEEK_SET**- Parte do início do arquivo e avança <offset> bytes
    - **SEEK_END**- Parte do final do arquivo e retrocede <offset> bytes
    - **SEEK_CUR**- Parte do local atual e avança <offset> bytes

Seeking

- fseek(*stream*, *offset* , *origin*)
  - Para *streams* em modo texto, o *offset* é um valor inteiro (*long int*) (pode ser 0)
  - *ftell ()* retorna a posição atual do ponteiro
  - *v. exemplo*

Leitura e Escrita

Leitura e Escrita

- fputs(*str*, *stream*)
  - Escreve a string apontada por *str* no arquivo *stream*
  - Copia até encontrar o caractere null ‘\0’
    - Não copia o ‘\0’
  - Difere do *puts()* ao deixar definir a *stream* e ao não escrever automaticamente ‘\n’ ao fim

Leitura e Escrita

```
int main (){
    FILE * pFile ;
    pFile = fopen ( "example.txt" , " wb" );
    fputs ( " This is an apple ." , pFile );
    fseek ( pFile , 9 , SEEK_SET );
    fputs ( " sam" , pFile );
    fclose ( pFile );
    return 0;
}
```

Leitura e Escrita

- fgets(*str*, *num*, *stream*)
  - Lê caracteres de uma *stream* e salva como uma string em C em *str* até (*num*-1) caracteres serem lidos
    - OU até encontrar um '\n' OU EOF
    - O '\n' é incluído na string
  - '\0' é adicionado ao fim da string automaticamente

Leitura e Escrita

```
int main () {
    FILE * pFile ;
    char buffer [ 100 ];
    pFile = fopen ( "file.txt" , "r" );
    if ( pFile == NULL) perror ( " Error opening file" );
    else {
        while ( ! feof ( pFile ) ) {
            if ( fgets (buffer , 100 , pFile ) == NULL ) break ;
            fputs (buffer , stdout );
        }
        fclose ( pFile );
    }
    return 0;
}
```

Leitura e Escrita

- fprintf(*stream*, format, ...)
  - Escreve a string apontada por *format* no *stream*
    - Segue os princípios do *printf*
- fscanf(*stream*, format, ...)
  - Lê de *stream* os dados e salva de acordo com a formatação de *format*
    - Segue os princípios do *scanf*

Leitura e Escrita

```
int    main   (){
      FILE *     pFile   ;
      int     n;
      char    name  [ 100 ];
      pFile     =  fopen    ( "myfile.     txt   " , "w" );
      for    (n= 0   ; n<   3   ; n++)   {
         puts    ( " please   ,   enter    a  name: "    );
         gets    ( name);
         fprintf       ( pFile   ,   " Name %d [%- 10.10s]   \ n" ,n+ 1,name);
      }
      fclose     ( pFile   );
      return     0;
}
```

Leitura e Escrita

```
int     main   (){
        FILE *      pFile   ;
        int     n;
        char    name  [ 100 ];
        pFile     =  fopen    ( "myfileprintf.          txt   " , "r"   );
        while   ( fscanf    ( pFile    ,    " Name %d %[^ \ n]"   ,& n,name ) != EOF){
                fgetc   ( pFile    );
                printf     ( " Name %d %s\ n" , n,      name);
        }
        fclose      ( pFile    );

        return      0;
}
```

Leitura e Escrita

- ➔ fwrite(*ptr*, *size*, *count*, *stream*)
  - ◆ Escreve o dado apontado por *ptr* no *stream*
    - O quanto do dado é escrito é calculado pelo tamanho de cada elemento (*size*) vezes o *count*
- ➔ fread(*ptr*, *size*, *count*, *stream*)
  - ◆ Lê o dado do *stream* e salva em *ptr*
    - O quanto do dado é lido é calculado pelo tamanho de cada elemento (*size*) vezes o *count*

Leitura e Escrita

```
#define VECTORSIZE 10
int     main (){
        float       vector[VECTORSIZE];
        FILE *   fp ;
        srand  (time(    NULL));
        for   ( int     i =     0; i < VECTORSIZE; ++i)
                vector[i] =          rand  ()/(    float    )(RAND_MAX/100 );
        fp    =  fopen  ( "data/numbers.bin"          , " wb" );
        fwrite    (vector,       sizeof    ( float    ), VECTORSIZE,     fp  );
        fclose    ( fp  );
        return      0;
}
```

Leitura e Escrita

```
int     main(){
        float       *vector;
        int     vectorSize;
        long    fileSize;
        FILE *fp;
        fp = fopen(       "data/numbers.bin"          , "rb"    );
        fseek(fp,         0,    SEEK_END);
        fileSize = ftell(fp);
        fseek(fp,         0,    SEEK_SET);
        vectorSize = fileSize/              sizeof    ( float     );
        vector = (        float     *) malloc(vectorSize*             sizeof    ( float     ));
        fread(vector,           sizeof    ( float     ), vectorSize, fp);
        for   ( int     i =     0; i < vectorSize; ++i)
                printf(       "%f "    , vector[i]);
        printf(       " \ n" );
        fclose(fp);
        return      0;
}
```

Disco como Gargalo

- ➔ Discos são muito mais lentos que as redes locais ou a CPU
- ➔ Muitos processos são “disk-bound”, i.e, CPU e rede têm que esperar pelos dados do disco

Técnicas p/ Minimizar

- ➔ Multiprogramação: CPU trabalha em outro processo enquanto aguarda o disco
- ➔ *Stripping*: o arquivo é repartido entre vários drives (paralelismo)
- ➔ RAID: (redundant array of inexpensive disks):
  - ◆ Stripping (RAID 0), Mirroring (RAID 1)

Técnicas p/ Minimizar

- ➔ Disk cache: blocos de memória RAM configurados para conter páginas de dados do disco
  - ◆ Ao ler dados de um arquivo, o cache é verificado primeiro. Se a informação desejada não é encontrada lá, é feito um acesso ao disco e o novo conteúdo é carregado no cache

Técnicas p/ Minimizar

- ➔ RAM Disk: simula em memória o comportamento do disco mecânico
  - ◆ Carrega arquivos muito usados, dados descompactados, etc.

E...
Organização de Arquivos!

Organização de Arquivos

- ➔ Meta: criar estruturas para minimizar as desvantagens do uso da memória secundária
- ➔ Objetivo: minimizar o tempo de acesso ao dispositivo de armazenamento secundário

Organização de Arquivos

- ➔ Ideal
  - ◆ Encontrar a informação em um único acesso
- ➔ Se não for possível
  - ◆ Encontrar em poucos acessos
    - ● Ex: uma busca binária não é eficiente o suficiente (~16 acessos para 50.000 registros)

Organização de Arquivos

- ➔ Estruturas de dados eficientes em memória principal são inviáveis em memória secundária
- ➔ Uma das dificuldades de obter uma estrutura de dados adequada é a constante necessidade de alterações nos arquivos

Organização de Arquivos

- ➔ Exemplo:
  - ◆ Árvores AVL (1963) são consideradas rápidas para pesquisa em memória principal
- ➔ Em disco:
  - ◆ Mesmo com árvores balanceadas, são necessários dezenas de acessos ao disco

Organização de Arquivos

- ➔ Árvores-B (1971)
  - ◆ Abordagem diferente de outras árvores
    - ● ex: árvores-B têm um crescimento bottom-up
  - ◆ Base para implementações comerciais de diversos sistemas , incluindo SGBDs
  - ◆ Tempo de acesso proporcional a $\log_k N$
    - ● N => número de entradas (chaves)
    - ● k => número de entradas em um único nó (bloco).

Organização de Arquivos

- ➔ Na prática árvores-B permitem encontrar uma entrada entre milhões em 3 ou 4 acessos a disco.
- ➔ Existem diversas variações de árvores-B (B+ tree, B* tree, B-link tree), e técnicas de otimização para cada passo do algoritmo
  - ◆ Normalmente específicos para um conjunto de aplicações pontual

Organização de Arquivos

➔ Outras técnicas:

- Hashing externo
  - Acesso constante para arquivos que não sofrem muitas alterações (hashing estático)
- Hashing dinâmico
  - Acesso rápido em grandes arquivos, com recuperação da informação em 2 acessos

Referências

- M. J. Folk, B. Zoellick and G. Riccardi. File Structures: An object-oriented approach with C++, Addison Wesley, 1998.
- ZIVIANI, N. Projeto de Algoritmos, Thomson, 2a. Edição, 2004.