Pesquisa em Memória Secundária

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados 2 (AE43CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

DAINF
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CÂMPUS PATO BRANCO

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Sumário

Introdução

- Pesquisa em memória primária
  - Pesquisa sequencial
  - Pesquisa binária
  - Árvores de pesquisa
    - Binária
    - AVL
    - Vermelha-preta
  - *Hashing*

Introdução

- Pesquisa em memória secundária envolve mais registros do que a memória interna pode suportar
  - Custo para acessar registro é maior: minimizar acessos
  - Métodos eficientes de pesquisa dependem das características de hardware e sistema operacional
  - Medida de complexidade: custo de transferir dados entre a memória principal e secundária (minimizar o número de transferências)
  - Apenas um registro pode ser acessado por vez
    - Fitas magnéticas: acesso de forma sequencial
    - Discos: acesso direto, mas todo bloco deve ser trazido à memória

Sumário

Modelo Computacional para Memória Secundária

- Memória virtual
  - Normalmente implementado como uma função do sistema operacional
  - Uso de uma pequena quantidade de memória principal e uma grande quantidade de memória secundária
  - Boa estratégia para algoritmos com pequena localidade de referência
  - Organização do fluxo entre a memória principal e secundária é extremamente importante
  - Programador pode endereçar grandes quantidades de dados, deixando para o sistema a responsabilidade de transferir o dado da memória secundária para a principal

Modelo Computacional para Memória Secundária

- Memória virtual: funções básicas
  - Realocação
  - Proteção
  - Paginação

Modelo Computacional para Memória Secundária

- Sistema de paginação:
  - Implementação mais utilizada
  - Espaço de endereçamento dividido em páginas de igual tamanho (geralmente múltiplos de 512 bytes)
  - Memória principal dividida em molduras de páginas (*frames*) de igual tamanho
  - Molduras de páginas contêm algumas páginas ativas
  - Páginas inativas estão na memória secundária

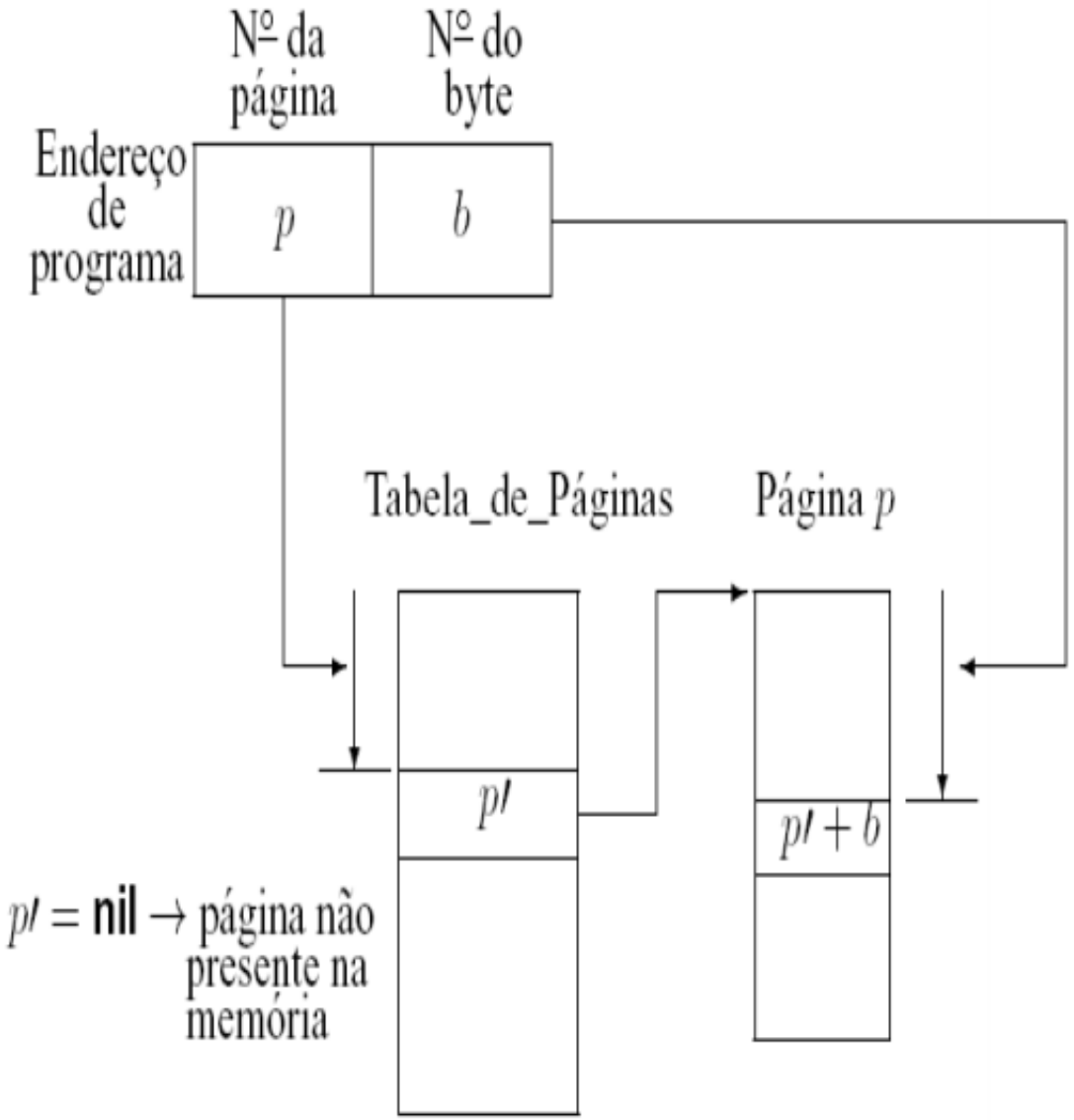
Modelo Computacional para Memória Secundária

- Mecanismo de paginação tem duas funções:
  1. Mapeamento dos endereços: determinar qual página está sendo usada e encontrar a moldura (se existir)
  2. Transferência das páginas entre memória primária e secundária
- Referência de página
  - Uma parte dos bits do endereço é definida como um número de página
  - Outra parte é o número do byte do item dentro da página
  - Exemplo: se o espaço de endereçamento é de 24 bits
    - A memória virtual terá 2^{24} bytes
    - Se a página é de 512 bytes (2^9): nove bits são usados para o número do byte dentro da página ($\boldsymbol{b}$) e o restante (15 bits) representa o número da página ($\boldsymbol{p}$)

Modelo Computacional para Memória Secundária

- O mapeamento de endereços é feito utilizando uma tabela de páginas
  - $f(p, b) = p' + b$
  - A p-ésima entrada da tabela contém a localização p' da moldura de página (*frame*) que contém a página número p, caso a mesma esteja na memória principal
  - A tabela de páginas pode ser representada como um vetor do tamanho de número possível de páginas
  - Se a página número p não estiver em um *frame* na memória principal, p' terá valor nulo

Modelo Computacional para Memória Secundária



Modelo Computacional para Memória Secundária

- Caso um programa necessitar de uma página que não esteja na memória principal ($p = NULL$), essa página $\boldsymbol{p}$ deve ser recuperada da memória secundária para a primária
  - Também, a tabela de páginas deve ser atualizada

- Se não houver uma moldura de página vazia, uma página deverá ser removida da memória principal
  - O ideal é a remoção da página que não será referenciada pelo período de tempo mais longo no futuro

  - Como não há como prever o futuro, o mesmo é inferido a partir do comportamento passado

Modelo Computacional para Memória Secundária

- Algoritmos para escolha da página a ser removida:
  - Menos recentemente utilizada (LRU – "*least recently used*")
  - Menos frequentemente utilizada (LFU – "*least frequently used*")
  - Ordem de chegada (FIFO – *first in first out*)

Modelo Computacional para Memória Secundária

- Menos recentemente utilizada (LRU)
  - Um dos algoritmos mais utilizados
  - Remove a página menos recentemente utilizada
  - Princípio: comportamento futuro deve seguir o passado recente
  - Implementação por meio de filas
    - Após a utilização de uma página, ela é colocada no fim da fila
    - A página no início da fila é que foi menos recentemente utilizada
    - A nova página trazida da memória secundária deve ser colocada no *frame* que contém a página menos recentemente utilizada

Modelo Computacional para Memória Secundária

- Menos frequentemente utilizada (LFU)
  - Remove a página menos frequentemente utilizada
  - Custo extra: registrar número de acessos a cada página
  - Desvantagem: uma página recentemente trazida da memória secundária tem um baixo número de acessos e pode ser brevemente removida
- Ordem de chegada (FIFO)
  - Remove a página que está na memória principal há mais tempo
  - Algoritmo mais simples e barato de se manter
  - Desvantagem: ignora que a página mais antiga pode ser a mais referenciada

Exemplo de Implementação com Paginação

Exemplo de Implementação com Paginação

- Exemplo com paginação:

```
#define ITENSPAGINA 5
#define MAXTABELA 20

typedef struct{
  long RA;
  char nome[70];
  int codigo_curso;
  float coef;
}Aluno;

typedef struct{
  Aluno itens[ITENSPAGINA];
}Pagina;
```

Exemplo de Implementação com Paginação

- Exemplo com paginação:

```
int buscar(Aluno* aluno, FILE *arq){
   Pagina pag;
   int i = 0;
   int tam = tamanho_arquivo(arq);

   fseek(arq, 0, SEEK_SET);

   if (tam > 0){
     do{
       fread(pag, sizeof(Pagina), 1, arq);

       for (i = 0; (i < ITENSPAGINA) && (aluno->RA < pag->itens[i]->RA); j++);

       if ((i < ITENSPAGINA) && (aluno->RA == pag->itens[i]->RA)){
         *aluno = pag->itens[i];

         return 1;
       }else
         fseek(arq, sizeof(Pagina), SEEK_CUR);
     }while (ftell(arq) < tam);
   }

   return 0;
}
```

Exemplo de Implementação com Paginação

- Exemplo com paginação:

```
int tamanho_arquivo(FILE *arq){
  fseek(arq, 0, SEEK_END);

  return ftell(arq);
}
```

Exemplo de Implementação com Paginação

- Complexidade:
  - Carregar M páginas: $O(M)$
  - Percorrer N itens em cada página: $O(N)$
  - Custo total: $O(M * N)$

Acesso Sequencial Indexado

Acesso Sequencial Indexado

- Utiliza o princípio da pesquisa sequencial
  - Cada item é lido sequencialmente até encontrar uma chave maior ou igual a chave de pesquisa

- Providências necessárias para aumentar a eficiência da pesquisa sequencial:
  - O arquivo deve estar ordenado pelo campo chave do item
  - Um arquivo de índice de páginas, contendo pares de valores (x, p), deve ser criado, onde
    - x é a chave de um item
    - p é o endereço da página na qual o primeiro item contém a chave x

Acesso Sequencial Indexado

- Exemplo para um conjunto de 15 registros (itens)
  - Cada página tem capacidade para armazenar 4 itens do disco
  - Cada entrada do índice de páginas armazena a chave do 1º item de cada página e o endereço de tal página no disco

3	14	25	41
1	2	3	4

1 | 3 5 7 11 | 2 | 14 17 20 21 | 3 | 25 29 32 36 | 4 | 41 44 48 |

- Para se pesquisar por um item, deve-se:
  - Localizar, no índice de páginas, a página que pode conter o item desejado de acordo com sua chave de pesquisa
  - Realizar uma pesquisa sequencial na página localizada

Acesso Sequencial Indexado

- Implementação do acesso sequencial indexado:

```
#define ITENSPAGINA 4

// Uma entrada da tabela de índice das páginas
typedef struct {
  int posicao;
  int chave;
}Indice;

typedef struct{
  int chave;
  char descricao[51];
  float preco;
  /* outros componentes */
}Item;
```

Acesso Sequencial Indexado

- Implementação do acesso sequencial indexado:

```
int buscar(Indice tab[], int tam, Item* item, FILE *arq){
   Item pagina[ITENSPAGINA];
   int i, n_itens;
   long desloc;

   for (i = 0; i < tam && tab[i].chave <= item->chave; i++);

   if ((i == 0) && (tab[i].posicao < item->chave)) return 0;
   else {
      if (i < tam) n_itens = ITENSPAGINA;
      else {
         fseek(arq, 0, SEEK_END);
         n_itens = (ftell(arq) / sizeof(Item)) % ITENSPAGINA;
      }

      desloc = (tab[i - 1].posicao) * ITENSPAGINA * sizeof(Item);
      fseek (arq, desloc, SEEK_SET);
      fread(&pagina, sizeof(Item), n_itens, arq);

      for (i = 0; i < n_itens; i++)
         if (pagina[i].chave == item->chave){
            *item = pagina[i];
            return 1;
         }

      return 0;
   }
}
```

Acesso Sequencial Indexado

- Complexidade:
  - Percorrer a tabela de índice de tamanho m
  - Percorrer a página de tamanho n
  - Custo total: $O(n + m)$

Acesso Sequencial Indexado

Disco magnético

Fonte: `https://www.indiamart.com/proddetail/2tb-seagate-hard-disk-3264950212.html`

Acesso Sequencial Indexado

Disco magnético

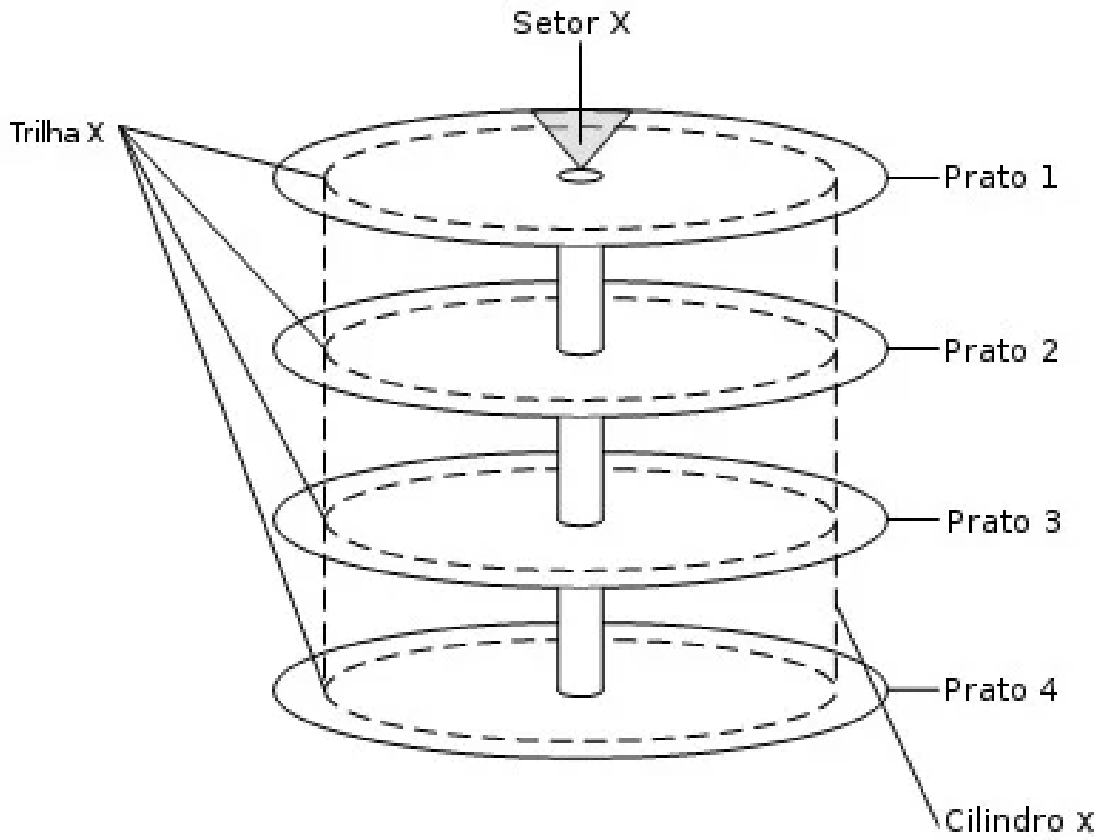
Fonte: `https://www.123rf.com/photo_51017724_hard-disk-drive-hdd-connected-to-the-sata-cable-on-a-white-background.html`

Acesso Sequencial Indexado

Disco magnético

Fonte: https://produto.mercadolivre.com.br/MLB-985991927-hd-pc-8-tb-8000-gb-desktop-seagate-samsung-armazenamento-alt-_JM

- Funcionamento do disco magnético:
  `https://www.youtube.com/watch?v=lpYfep68xnA`



Fonte: `https://canaltech.com.br/hardware/Como-funcionam-os-discos-rigidos/`

Acesso Sequencial Indexado

Disco magnético

- Acesso sequencial indexado em disco magnético:
  - Possibilita acesso sequencial ou randômico
  - Adequado apenas para aplicações com baixa frequência de inserção e remoção
  - Vantagem: garantia de acesso com apenas um deslocamento da cabeça de gravação
  - Desvantagem: inflexibilidade

Referências I


Assis, G. T.
Pesquisa Externa. BCC203 – Algoritmos II.
Notas de Aula. Ciência da Computação. DECOM/UFOP, 2018.


Deitel, H. M. e and Deitel, P. J.
C: Como Programar.
Pearson, 2011.


Schidildt, H.
C Completo e Total.
Pearson, 2011.


Ziviani, N.
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.