TRABALHO PRÁTICO 3:
Memória Virtual

Sandro Miccoli - 2009052409 - smiccoli@dcc.ufmg.br

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

8 de dezembro de 2012

Resumo. *Este relatório descreve como foi implementado uma versão simplificada de um simulador de memória virtual (SMV). Será descrito como foi modelado o problema, e como cada política de reposição de páginas funciona. Finalmente será detalhado a análise de complexidade dos algoritmos e uma análise de cada política de acordo com a especificação do trabalho, e, por último, uma breve conclusão do trabalho implementado.*

1. INTRODUÇÃO

A memória virtual foi inicialmente criada para possibilitar a um programa ser executado em um computador com uma quantidade de memória principal (física) menor que o tamanho de todo o espaço do utilizado pelo próprio programa. Ou seja, o espaço ocupado pelas instruções, dados e pilha de execução de um programa pode ser maior que o espaço em memória principal disponível.

Memória virtual, é uma técnica que usa a memória secundária como uma cache para armazenamento secundário. Houve duas motivações principais: permitir o compartilhamento seguro e eficiente da memória entre vários programas e remover os transtornos de programação de uma quantidade pequena e limitada na memória principal. [Wikipedia]

O objetivo principal do trabalho é implementar uma versão simplificada de um sistema de memória virtual (SMV), pois usaremos apenas um nível de paginação, sem caches ou otimizações. Quando houver a necessidade de reposição de páginas, foram utilizadas três políticas de reposição:

- **FIFO (FirstIn, FirstOut)** - A página que está residente a mais tempo é escolhida para remoção.
- **LRU (Least Recently Used)** - A página acessada a mais tempo deve ser escolhida para remoção.
- **LFU (Least Frequently Used)** - A página com a menor quantidade de acessos deve ser escolhida para remoção.

O restante deste relatório é organizado da seguinte forma. A Seção 2 descreve como foi feita a modelagem do problema e o armazenamento das páginas virtuais. A Seção 3 descreve como foi feito a manipulação das páginas da memória e detalhes das políticas de reposição. A Seção 4 trata de detalhes específicos da implementação do trabalho: quais os arquivos utilizados; como é feita a compilação e execução; além de detalhar o formato dos arquivos de entrada e saída. A Seção 5 contém a análise de desempenho de cada uma das políticas de reposição. A Seção 6 conclui o trabalho.

2. MODELAGEM

Para simular a memória virtual neste trabalho, foi utilizado um tipo abstrato de dados de lista duplamente encadeada. Foi escolhida esta estrutura pela facilidade de manipulação e pesquisa dos elementos contidos nela.

A abstração da estrutura é a seguinte: a lista representa a memória física e cada uma das suas células representam as páginas que foram carregadas na memória. Assim, utilizando as funções de inserção e remoção da lista encadeada, podemos implementar as políticas de reposição propostas pro trabalho.

3. SOLUÇÃO PROPOSTA

É fornecido na entrada do trabalho o tamanho em bytes da memória física, o tamanho de cada página e N acessos à memória. Cada um desses acesso é referente à posição da memória virtual acessada sequencialmente (para mais detalhes sobre o formato de entrada, consultar o item 4.3.1 deste trabalho). Porém, para saber em qual página cada acesso se enconta, dividimos a posição de acesso pela quantidade de bytes de cada página. Por exemplo, dada a sequência de acesso **0 2 4 2 10 1 6 8**, podemos transformar isso para **0 0 1 0 2 0 1 2**. Assim sabemos qual página deverá estar na memória a cada acesso.

Foi implementado um algoritmo para cada política de reposição (FIFO, LRU e LFU), e todos possuem o mesmo sistema de contagem de *page faults*.

Para o **FIFO**, a escolha da página a ser removida é a que entrou primeiro, ou seja, a primeira da fila. No **LRU**, também removemos as páginas que estão no início da fila, pois toda vez que a página é acessada ela vai pro final da fila, assim, seu início contém as páginas que foram acessadas menos recentemente. No **LFU**, para cada acesso à memória, cada página possui um contador para indicar a quantidade de acessos à ela. Assim, a cada acesso à memória, ordenamos as páginas de acordo com esse contador, então as primeiras páginas da fila serão as que foram menos frequentemente usadas.

3.1. Algoritmos implementados

- *void FIFO(TipoLista * memoria, TipoCelula pagina)*

  Descrição:
  Parâmetros:
  Complexidade: $O(n)$, onde n é .

- *void LRU(TipoLista * memoria, TipoCelula pagina)*

  Descrição:
  Parâmetros:
  Complexidade: $O(n^2)$, onde n é o .

- *void LFU(TipoLista * memoria, TipoCelula pagina)*

  Descrição:
  Parâmetros:
  Complexidade: $O(n^2)$, onde n é o .

- *void ordenaPorAcessos(TipoLista * memoria)*

  Descrição:
  Parâmetros:
  Complexidade: $O(n^2)$, onde n é o .

- *TipoApontador resideEmMemoria(TipoLista * memoria, int pagina)*

  Descrição:
  Parâmetros:
  Complexidade: $O(n^2)$, onde n é o .

4. IMPLEMENTAÇÃO

4.1. Código

4.1.1. Arquivos .c

- **tp3.c** Arquivo principal do programa. Lê os arquivos de entrada, calcula os *page faults* pra cada política e escreve o resultado em um arquivo de saída.
- **lista.c** TAD da uma lista duplamente encadeada. Contém funções de manipulação (criação, inserção, remoção e liberação de memória), e também de impressão e cópia.
- **smv.c** Contém a implementação das políticas de reposição FIFO, LRU, LFU.

4.1.2. Arquivos .h

- **lista.h** TAD da uma lista duplamente encadeada. Contém a definição da estrutura e das funções.
- **smv.h** Contém a definição das políticas de reposição FIFO, LRU, LFU.

4.2. Compilação

O programa deve ser compilado através do compilador GCC através dos seguintes comandos

Para programação dinâmica:

```
gcc -Wall -Lsrc src/tp3.c src/lista.c src/arquivos.c src/smv.c -o tp3
```

Ou através do comando $make$.

4.3. Execução

A execução do programa tem como parâmetros:

- Um arquivo de entrada contendo várias instâncias a serem simuladas.
- Um arquivo de saída que irá receber a quantidade de *page faults* pra cada política de reposição

O comando para a execução do programa é da forma:

```
./tp3 <arquivo_de_entrada> <arquivo_de_saída>
```

4.3.1. Formato da entrada

A primeira linha do arquivo de entrada contém o valor k de instâncias que o arquivo contém. As k instâncias são definidas em duas linhas cada uma. A primeira linha contem três inteiros: o tamanho em bytes da memória física, o tamanho em bytes de cada página, e o número N de acessos. A linha seguinte contém N inteiros representando as N posições da memória virtual acessadas sequencialmente.

A seguir um arquivo de entrada de exemplo:

```
1
8 4 10
0 2 4 2 10 1 0 0 6 8
```

4.3.2. Formato da saída

O arquivo de saída consiste em k linhas, cada uma representando o resultado de uma instância. Cada linha contém um inteiro que representa o número de falhas utilizando FIFO, LRU e LFU, necessariamente nessa ordem. Um exemplo é mostrado abaixo:

```
6 5 5
```

5. AVALIAÇÃO EXPERIMENTAL

Para testar os algoritmos foi feito o seguinte: um script foi criado que, utilizando um dicionário de aproximadamente 2000 palavras, geraria uma sequência de palavras, concatenadas ou não, para testar cada tipo de cenário.

Foi realizado uma série de testes, com entradas crescentes, da maneira que é explicado nas próximas seções.

5.1. Máquina utilizada

Segue especificação da máquina utilizada para os testes:

```
model name:     Intel(R) Core(TM) i3 CPU       M 330  @ 2.13GHz
cpu MHz:        933.000
cache size:     3072 KB
MemTotal:       3980124 kB
```

5.2. Testes

Então foi pensado em cinco testes a serem executados, um com apenas uma palavra pra ser testada, outro com 5 palavras concatenadas, um com 10 palavras concatenadas, um com 50 palavras concatenadas e, por último, um teste com 100 palavras concatenadas. Assim teríamos testes com palavras relativamente pequenas e com uma série de caracteres relativamente grandes. Assim poderíamos ver o comportamento de cada paradigma para tanto entradas pequenas quando grandes.

O gráfico que será mostrado em cada teste mostra especificamente a quantidade de caracteres (em porcentagem) a mais que o algoritmo guloso teve de inserir que o algoritmo

de programação dinâmica. Assim podemos fazer uma análise entre a solução não-ótima e a solução ótima.

Dentro de cada subseção será dado um exemplo de entrada para cada teste e será detalhado melhor as características de cada um.

5.2.1. Testes pequenos - Uma palavra

A quantidade média de caracteres neste teste ficou entre aproximadamente 10. A seguir um exemplo das palavras utilizadas no teste:

```
abbreviations
abandoned
youthfulness
...
```

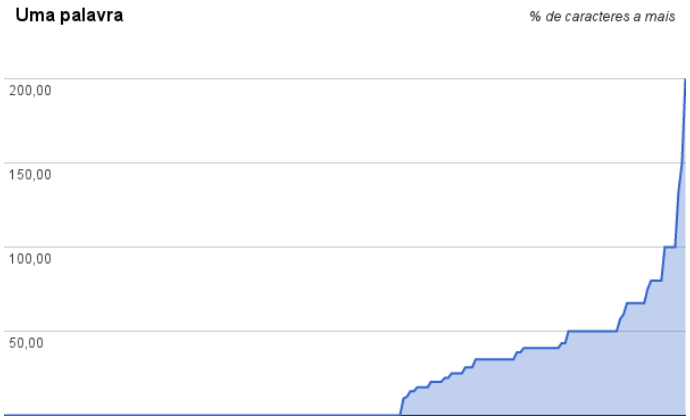

Figura 1. Uma palavra - Quantidade de caracteres a mais (em %)

Na Figura 1 é perceptível que a grande maioria dos testes não houve diferença entre a solução gulosa e a solução ótima. Porém, quando houve, a diferença de caracteres cresceu bastante, chegando até a 200%.

Para não alterar demais a visualização do gráfico, um resultado foi removido pois a diferença de caracteres era de 800%; mas, de uma série de 200 palavras testadas, apenas uma que teve esse comportamento. Isso ocorreu quando o algoritmo de programação dinâmica teve de inserir apenas uma letra e o guloso inseriu 9.

5.2.2. Testes médios - Cinco palavras concatenadas

Na Figura 2 pode-se ver o resultado do teste que utilizou uma série de palavras concatenadas, neste caso, cinco palavras. Cada instância ficou com aproximadamente 30 até 40 caracteres. A seguir um exemplo das palavras usadas no teste:

```
abatedabnormallywondrousxeroxingzero
deifiedwonabolishmentsabbottzenith
yankeewizardwoefulabaseswrangle
...
```

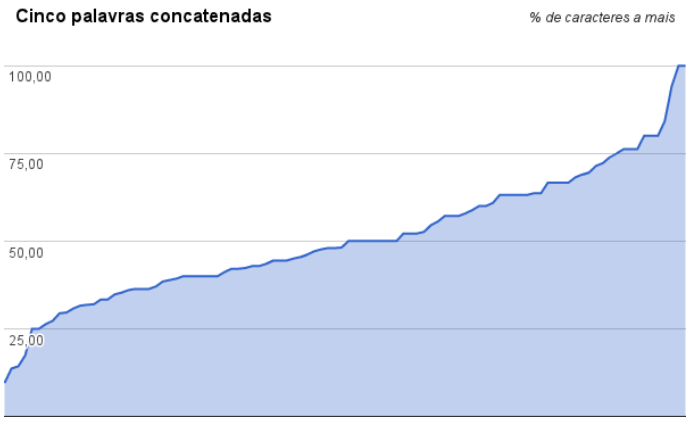

Figura 2. Cinco palavras - Quantidade de caracteres a mais (em %)

5.2.3. Testes médios - Dez palavras concatenadas

Na Figura 3 consta o resultado do teste que utilizou uma série de caracteres que eram formados por palavras concatenadas umas nas outras, neste caso, 10 palavras no total. Cada instância ficou com aproximadamente 60 até 80 caracteres. A seguir um exemplo das palavras usadas no teste:

```
woveyugoslavianyoungsterreferyuridevovedwonderabashingaboveyellowed
wontwritesabaseszuluwrestyeayesterdayabbreviatingzoologicallyzone
yorktownwrittenaberrationsaibohphobiaababawrappedwolffaberrantablewrote
...
```

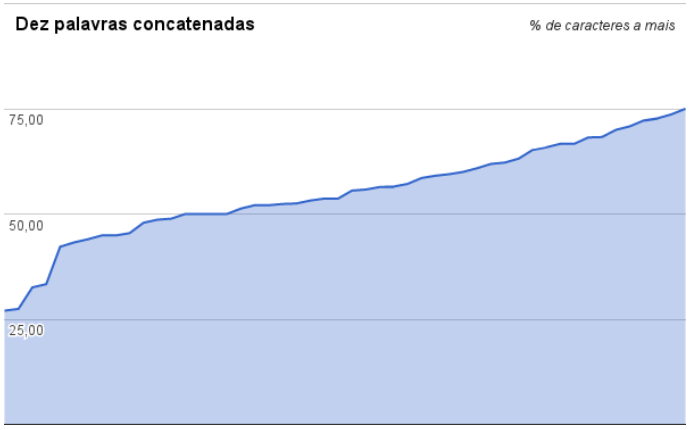

Figura 3. Dez palavras - Quantidade de caracteres a mais (em %)

5.2.4. Testes grandes - Cinquenta palavras concatenadas

Na Figura 4 aumentamos a quantidade de palavras concatenadas para cinquenta. Neste teste não iremos mostrar as palavras concatenadas, mas cada uma ficou com 350 até 450 caracteres.

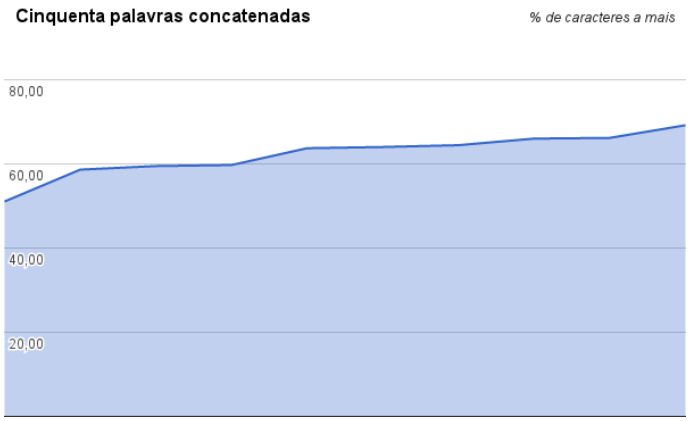

Figura 4. Cinquenta palavras - Quantidade de caracteres a mais (em %)

5.2.5. Testes grandes - Cem palavras concatenadas

Na Figura 5, no último teste, concatenamos cem palavras. Não iremos mostrar as palavras concatenadas, mas cada uma ficou com uma média de 900 caracteres.

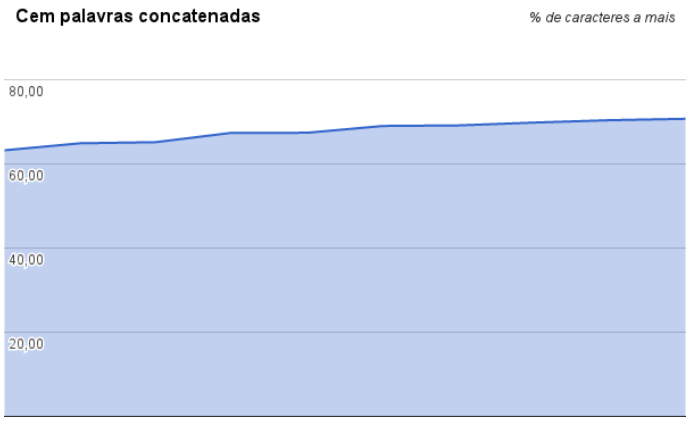

Figura 5. Cem palavras - Quantidade de caracteres a mais (em %)

5.3. Resultado

Como é possível perceber pela série de testes na última seção, quanto maior a cadeia de caracteres a ser computada pelos algoritmos, mais homogênea e regular a diferença entre cada solução fica. Nos primeiros teste, de apenas uma palavra 5.2.1, e de 5 palavras 5.2.2, a variação é gritante. No primeiro varia de 0% até 800%, no segundo a variação diminui mas ainda é relativamente grande. Podemos ver na Figura 2, que a quantidade de caracteres vai de aproximadamente 15% para 100%.

Nos últimos testes a curva de variação de caracteres vai se tornando cada vez mais estável. No teste com dez palavras concatenadas, na Figura 3, a variação vai de aproximadamente 25% para 75%. Mas quando os caracteres vão para a casa das centenas e milhares, essa variação se torna cada vez menor, se localizando em torno dos 60%, 70%.

6. CONCLUSÃO

Os dois algoritmos implementados solucionam o problema do palíndromo apresentado. O primeiro algoritmo, utilizando o paradigma de programação dinâmica, possui solução ótima, ou seja, sempre encontrará o melhor resultado. Já o algoritmo guloso, possui solução não-ótima, pois toma as decisões que julga ser melhor naquele momento específico, porém sem considerar vários outros aspectos do problema.

Foi visto na Avaliação Experimental, que quanto maior a cadeia de caracteres, mais constante ficava a diferença entre as soluções ótima e não-ótima. Claro que a solução ótima ainda conseguia resolver o problema com uma média de 60% a menos de caracteres inseridos, mas ainda assim era um resultado que não era esperado. Acreditava que a solução não-ótima do algoritmo guloso teria um rendimento muito pior.

Caso fosse acatar o resultado do teste 5.2.1, poderia dizer que o teto observável para a quantidade de caracteres a ser inseridos seria 800%, porém esse foi um único caso dentre centenas de outros testados. Observando os resultados dos testes maiores, poderia dizer que o teto observável ficaria em torno de 75%.

O problema de gerar palíndromos de forma eficiente foi solucionado com sucesso. Além disso, duas soluções, uma ótima e uma não-ótima, foram implementadas, obtendo um bom rendimento nos inúmeros testes realizados. Nos testes comparativos foi possível perceber a diferença gritante entre uma solução ótima e uma não-ótima.

Referências

Wikipedia. Memória virtual. `http://pt.wikipedia.org/wiki/Memria_virtual`.