TRABALHO PRÁTICO 3:
Memória Virtual

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

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

9 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 simplificar isso para **0 0 1 0 2 0 1 2**. Assim sabemos qual página deverá estar na memória a cada acesso.

Na situação descrita acima, chegamos às páginas **0 0 1 0 2 0 1 2**; porém, com apenas 8 bytes na memória principal, podemos carregar simultâneamente apenas duas páginas, pois cada uma possui 4 bytes de tamanho. Por esse motivo precisamos adotar políticas que removam uma certa página e a troquem por outra da memória principal. Para solucionar esse problema, foi implementado um algoritmo para cada política de reposição (FIFO, LRU e LFU).

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: "First In, First Out"; a página que entrou primeiro na memória é a primeira a ser removida.
  Parâmetros: Memória principal e página atual que deveria estar na memória
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

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

  Descrição: "Least Recently Used"; a página que foi acessada menos recentemente que será a escolha pra ser removida.
  Parâmetros: Memória principal e página atual que deveria estar na memória
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

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

  Descrição: "Least Frequently Used"; a página que é utilizada menos frequentemente que será removida.
  Parâmetros: Memória principal e página atual que deveria estar na memória
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

- *void removeMenosAcessada(TipoLista * memoria)*

  Descrição: Procura pela página com menor número de acessos e a remove da memória.
  Parâmetros: Memória princial
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

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

  Descrição: Percorre toda a memória para verificar se a página se encontra na memória. Retorna um apontador pra essa página.
  Parâmetros: Memória principal e um inteiro que representa a página a ser encontrada
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

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

As análises feitas para para este trabalho foram todas detalhadas na especificação. O que foi pedido foi o seguinte:

- Calcular a localidade de referência temporal.
- Calcular a localidade de referência espacial.
- Gerar o histograma das distâncias de acessos.
- Gerar o histograma das distâncias de pilha.
- Gerar um gráfico ”Tamanho da página”x ”Bytes movimentados”.
- Gerar um gráfico ”Tamanho da memória”x ”Falhas”.

Para realizar essas análise foi diponibilizado um arquivo com uma configuração diferente no fórum da disciplina. Esse arquivo possuia apenas acessos à memória, sem informações de tamanho da memória ou da página. Foi criado dois scripts que auxiliaram na execução destas análises. O primeiro de localidade espacial e outro de localidade temporal.

Nas próximas seções iremos descrever a máquina utilizada para os testes e o resultado de cada item descrito acima.

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. Localidade de referência temporal

A Tabela 1 mostra os resultados que obtivemos para o cálculo da localidade de referência temporal.

Instância	Temporal
1	19.67
2	14.10
3	21.08
4	10.67

Tabela 1. Localidade de referência temporal

5.3. Localidade de referência espacial

A Tabela 2 mostra os resultados que obtivemos para o cálculo da localidade de referência espacial. É gritante a diferença entre o resultado da instância 2 pras outras, mas isso ocorreu pois ela tem um padrão bem peculiar. A grande maioria dos acessos são sequenciais na memória (posição 8, depois 7, depois 6, depois 5), isso caracteriza uma boa localidade de referência espacial.

Instância	Espacial
1	16.69
2	2.55
3	10.46
4	19.71

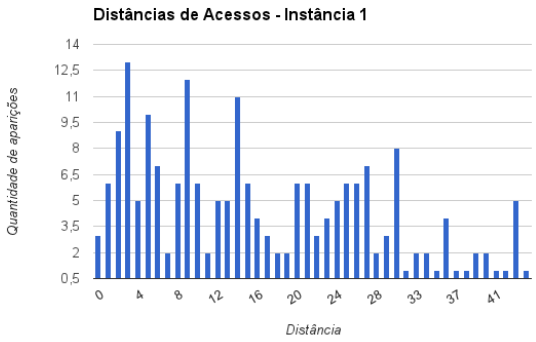
Tabela 2. Localidade de referência temporal

5.4. Histogramas de Distância de Acessos

Na Figura 1 agrupamos os quatro histogramas que foram pedidos. Logo abaixo é possível ver o resultado para cada uma das instâncias.

Na instância 1 e 4 ocorre uma distribuição maior dos acessos. Na primeira os ápices se encontram nas menores distâcias, já na outra o ápice ocorre na distância 32. As distâncais de acesso da instância 1 se encontram muito melhor distribuídas que a instância 4.

Já as instâncias 2 e 3 possuem uma simularidade: quase a totalidade dos acessos se encontram à distância 1. Isso indica que a localidade espacial deles é melhor que a das outras instâncias, por não terem que caminhar tanto na memória para acessar o próximo valor.



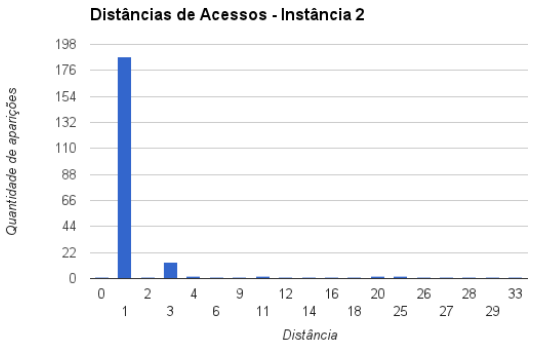
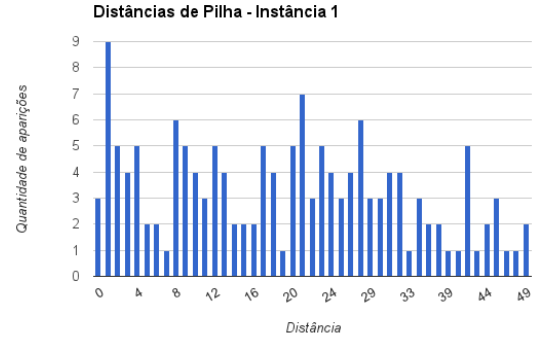
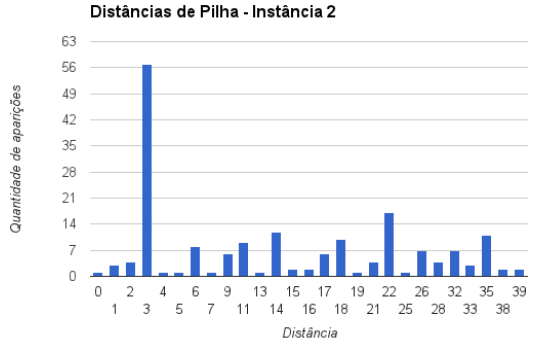
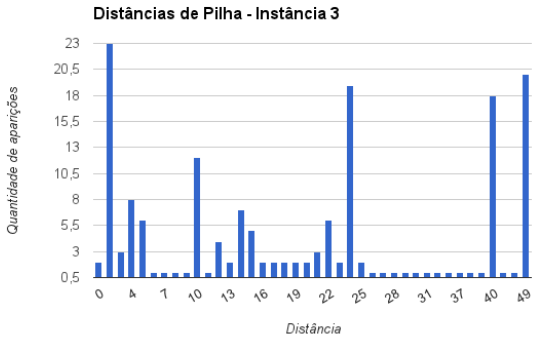


Figura 1. Distâncias de Acessos de todas as instâncias







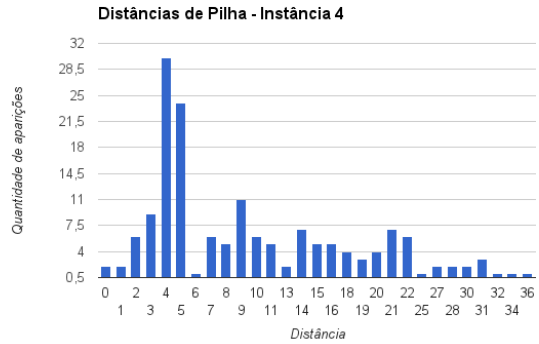


Figura 2. Distâncias de Pilha de todas as instâncias

5.5. Histogramas de Distância de Pilha

Na Figura 2 foi agrupado os histogramas referentes ao calculo de distância de pilha de cada instância.

5.6. 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 **??**, e de 5 palavras **??**, 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 **??**, 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 **??**, 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 **??**, 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`.