Aluno: Vinícius de Oliveira Silva
Curso: Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados III

Documentação de Trabalho Prático

1. Introdução

O uso de memória virtual é hoje em dia, de fundamental importância para a Ciência da Computação. A necessidade de trabalhar com um grande volume de dados vem se tornando cada vez mais comum, e infelizmente, o desenvolvimento de memórias com capacidade compatível com a demanda atual de dados não tem obtido resultados satisfatórios em termos de custo-benefício. A solução mais simples para esse problema é virtualizar a memória, usando espaço em disco como "uma continuação" da memória principal.
Essa técnica geralmente é implementada usando paginação de memória, de forma a fazer uso de duas propriedades importantes e inerentes a computação: Localidade de referência temporal e espacial.
A virtualização de memória depende de políticas de reposição de páginas para cumprir o seu propósito. Sempre que for feito um acesso a uma posição de memória, o sistema de gerenciamento de memória virtual deve conferir se o dado requisitado encontra-se em memória principal, e caso não esteja, deve substituir uma das páginas presentes pela página que contém o dado desejado. Essa substituição deve seguir algoritmos eficientes para que se evite que substituições desnecessárias sejam feitas no futuro.
Deverão ser implementado tipos abstratos de dados (TAD's) para representar as páginas de memória virtual e assim, realizar operações sobre elas, tais como, substituir uma página pela outra, carregar uma nova página, etc.
Espera-se com isso, praticar técnicas de programação que simulem o uso de memória secundária em seu funcionamento.

2. Implementação

Estruturas de Dados:

Para a implementação desse trabalho, foram criados várias estruturas de dados, de forma a apresentar, de maneira modularizada, cada uma das políticas de paginação implementadas. As estruturas que compõem o programa são:

- Page;
- List;
- FifoMemory;
- LruMemory;
- LfuMemory;
- MruMemory;

O TAD Page representa uma página de memória virtual, e é usado para armazenar os dados que viriam do disco rígido em uma situação real.
O TAD List é uma lista encadeada simples que armazena uma coleção de páginas. Foram implementadas funções nesse TAD que fogem à implementação clássica de uma lista encadeada. Essa foi uma importante decisão de implementação no sentido de facilitar o desenvolvimento do programa como um todo.
Os TAD's "memory" implementam suas respectivas políticas de reposição de páginas, a saber: *First-in-First-Out*, *Least-Recently-Used, Least-Frequently-Used* e *Most-Recently-Used.*

Funções e Procedimentos:

TAD Page:

Page createPage(int size): Gera e retorna uma página vazia, com o tamanho máximo especificado pelo parâmetro *size.* Sua complexidade de tempo e de espaço são O(1). Obs.: A complexidade espacial é O(constante).

void loadPage(Page* page, int value): Altera o parâmetro *page,* de forma a armazenar uma página que contenha o valor *value*, juntamente com os seus "vizinhos". Sua complexidade de tempo é O(n) e sua complexidade de espaço é O(1), onde n é o tamanho máximo da página.

TAD List:

void setListEmpty(List *list): Inicializa a lista especificada, alocando espaço em memória para sua "célula-cabeça". Sua complexidade temporal e espacial são O(1). Obs.: A complexidade espacial é O(constante).

int isListEmpty(List list): Retorna 0 ou 1 para indicar se a lista está vazia ou não. Complexidade de tempo e espaço O(1).

int pageExists(List list, int pageNumber): Retorna 0 ou 1 para indicar se a pagina de número *pageNumber* se encontra na lista ou não. Complexidade de espaço O(1) e complexidade de tempo O(n), onde n é, no pior caso, o tamanho da memória física simulada.

void insertBack(List *list, Page page): Insere página especificada no final da lista. Complexidade temporal e espacial O(1). Obs.: A complexidade espacial é O(constante).

void removeFront(List *list, Page *page): remove o primeiro elemento da lista especificada e o armazena em *page.* Complexidade de tempo e de espaço O(1).

void removeArbitrary(List* list, Pointer pointer): Remove um elemento arbitrário da lista. Tal elemento deve ser necessariamente o elemento seguinte ao indicado por *pointer.* Complexidade de tempo e espaço O(1).

int moveToLast(List *list, Page page): Pesquisa na lista o elemento correspondente a *page* e o coloca no final da lista. Retorna o número de nós pesquisados até que se encontre o nó desejado. Função usada na política LRU. Complexidade de espaço O(1) e de tempo, O(n), onde n é, no pior caso, o tamanho da memória física simulada. Obs.: A complexidade espacial é O(constante).

void moveToFront(List* list, Page page): Pesquisa na lista o elemento correspondente a *page* e o coloca no começo da lista. Procedimento usado na política MRU. Complexidade de espaço O(1) e de tempo, O(n), onde n é, no pior caso, o tamanho da memória física simulada. Obs.: A complexidade espacial é O(constante).

void insertFront(List* list, Page page): Insere a página especificada no começo da lista. Função usada na política MRU. Complexidade de tempo e de espaço são O(1). Obs.: A complexidade espacial é O(constante). Obs.: A complexidade espacial é O(constante).

Pointer searchLeastFrequency(List list): Pesquisa a página de memória menos acessada na lista e retorna um apontador que indica o elemento imediatamente anterior ao elemento pesquisado. Função útil na implementação da política LFU. Complexidade de espaço O(1) e complexidade de tempo, O(n), onde n é no pior caso, o tamanho da memória física simulada.

void incrementAccesses(List* list, int pageID): Incrementa o campo *accesses* da página cujo ID é igual ao especificado por parâmetro na função. Complexidade de espaço O(1) e de tempo O(n), onde n é no pior caso, igual ao tamanho da memória física simulada.

TAD FifoMemory:

void insertFifoPage(FifoMemory* fifo, Page page): usa a função *insertBack* para inserir a pagina especificada no final da lista, segundo a política Fifo. As complexidades temporal e espacial são ambas O(1). Obs.: Função interna ao TAD FifoMemory.
Obs.2: A complexidade espacial é O(constante).

void removeFifoPage(FifoMemory* fifo, Page page): usa a função *removeFront* para remover a primeira página da lista. Complexidade de tempo e espaço O(1). Obs.: Função interna ao TAD FifoMemory.

void createFifoMemory(FifoMemory* fifo, int memSize, int pageSize): usa a função *setListEmpty* para inicializar a lista recebida. Os demais parâmetros determinam quais valores serão usados para representar os tamanhos de memória e de página. Complexidade de tempo e espaço O(1). Obs.: A complexidade espacial é O(constante).

void freeFifoMemory(FifoMemory* fifo): libera o espaço de memória alocado durante a simulação FIFO. Complexidade de espaço O(1) e de espaço O(n), onde n é no pior caso, igual ao tamanho da memória física simulada.

int* processFifoInstructions(FifoMemory* fifo, int* instructions, int insLength, int pageSize): processa os acessos de memória provenientes da entrada e conta o número de *page-faults* gerados. Em casos onde existem *page-Faults* e a memória física está cheia, a página residente em memória há mais tempo é removida, dando lugar a uma nova página. A função retorna um array de inteiros contendo as distâncias entre as páginas acessadas. A complexidade de tempo é O(n*m), onde "n" representa o número de acessos a memoria virtual e "m" representa o tamanho da página. A complexidade de espaço é O(n*sizeof(Page)).

TAD LruMemory:

void insertLruPage(LruMemory* lru, Page page): usa a função *insertBack* para inserir a pagina especificada no final da lista, segundo a política lru. As complexidades temporal e espacial são ambas O(1). Obs.: Função interna ao TAD LruMemory.

void removeLruPage(LruMemory* lru, Page page): usa a função *removeFront* para remover a primeira página da lista. Complexidade de tempo e espaço O(1). Obs.: Função interna ao TAD LruMemory.

void createLruMemory(LruMemory* lru, int memSize, int pageSize): usa a função *setListEmpty* para inicializar a lista recebida. Os demais parâmetros determinam quais valores serão usados para representar os tamanhos de memória e de página. Complexidade de tempo e espaço O(1). Obs.: A complexidade espacial é O(constante).

void freeLruMemory(LruMemory* lru): libera o espaço de memória alocado durante a simulação LRU. Complexidade de espaço O(1) e de espaço O(n), onde n é no pior caso, igual ao tamanho da memória física simulada.

int* processLruInstructions(LruMemory* lru, int* instructions, int insLength, int pageSize): processa os acessos de memória provenientes da entrada e conta o número de *page-faults* gerados. Em casos onde existem *page-Faults* e a memória física está cheia, a página utilizada há mais tempo é removida, dando lugar a uma nova página. A função retorna um array de inteiros contendo as distâncias de pilha entre as páginas acessadas durante a simulação. A complexidade de tempo é O(n*m), onde "n" representa o número de acessos a memoria virtual e "m" representa o tamanho da página. A complexidade de espaço é O(n*sizeof(Page)).

TAD LfuMemory:

void insertLfuPage(LfuMemory* lfu, Page page): usa a função *insertBack* para inserir a pagina especificada no final da lista, segundo a política LFU. As complexidades temporal e espacial são ambas O(1). Obs.: Função interna ao TAD LfuMemory. Obs.: A complexidade espacial é O(constante).

void removeLfuPage(LfuMemory* lfu, Pointer pointer): usa a função *removeArbitrary* para remover a página apontada por "pointer" (a menos acessada da lista). Complexidade de tempo e espaço O(1). Obs.: Função interna ao TAD LfuMemory.

void createLfuMemory(LfuMemory* lfu, int memSize, int pageSize): usa a função *setListEmpty* para inicializar a lista recebida. Os demais parâmetros determinam quais valores serão usados para representar os tamanhos de memória e de página. Complexidade de tempo e espaço O(1). Obs.: A complexidade espacial é O(constante).

void freeLfuMemory(LfuMemory* lfu): libera o espaço de memória alocado durante a simulação LFU. Complexidade de espaço O(1) e de espaço O(n), onde n é no pior caso, igual ao tamanho da memória física simulada.

void processLfuInstructions(LfuMemory* lfu, int* instructions, int insLength, int pageSize): processa os acessos de memória provenientes da entrada e conta o número de *page-faults* gerados. Em casos onde existem *page-Faults* e a memória física está cheia, a página menos utilizada da memória é removida, dando lugar a uma nova página. A complexidade de tempo é O(n*m), onde "n" representa o número de acessos a memoria virtual e "m" representa o tamanho da página. A complexidade de espaço é O(n*sizeof(Page)).

TAD MruMemory:

Esse TAD implementa uma política de reposição de páginas não especificada no enunciado no trabalho e por isso, é conveniente fornecer maiores detalhes sobre o algoritmo implementado.
Basicamente, a política MRU funciona de forma inversa a LRU, eliminando da memória primária, a página mais recentemente utilizada. Esse algoritmo parte do princípio que se uma página foi recentemente lida, existe probabilidade baixa de ser lida logo em seguida, uma vez que a mesma já forneceu seus dados para o processo que a leu.
Segue abaixo um pseudo-código explicando o funcionamento do algoritmo:

```
Algoritmo MRU {
        Leia a posição de memória a ser acessada;
        Procure na memoria principal a página que contem aquela posição;
        Se a pagina existir em memoria {
                Armazene numero da pagina acessada;
                Mova a página para a primeira posição da memória;
                Retorne o dado desejado ao processo chamador;
        }Senão{
                Gere um page-Fault;
                Se existem posições vazias na memória principal{
                    Carregue a página desejada para a primeira posição da memória;
                }Senão{
                     Remova a primeira página da memória;
                     Carregue a página desejada para a primeira posição da memória;
                }
                Retorne o dado desejado ao processo chamador;
        }
Fim_Algoritmo
```

Funções usadas na implementação:

void insertMruPage(MruMemory* mru, Page page): usa a função *insertFront* para inserir a pagina especificada no início da lista, segundo a política MRU. As complexidades temporal e espacial são ambas O(1). Obs.: Função interna ao TAD LfuMemory. Obs.2: A complexidade espacial é O(constante).

void removeMruPage(MruMemory* mru, Page page): usa a função *removeFront* para remover a primeira página da lista. Complexidade de tempo e espaço O(1).
Obs.: Função interna ao TAD MruMemory.

void createMruMemory(MruMemory* mru, int memSize, int pageSize): usa a função *setListEmpty* para inicializar a lista recebida. Os demais parâmetros determinam quais valores serão usados para representar os tamanhos de memória e de página. Complexidade de tempo e espaço O(1). Obs.: A complexidade espacial é O(constante).

void freeMruMemory(MruMemory* mru): libera o espaço de memória alocado durante a simulação MRU. Complexidade de espaço O(1) e de espaço O(n), onde n é no pior caso, igual ao tamanho da memória física simulada.

void processMruInstructions(MruMemory* mru, int* instructions, int insLength, int pageSize): processa os acessos de memória provenientes da entrada e conta o número de *page-faults* gerados. Em casos onde existem *page-Faults* e a memória física está cheia, a página mais recentemente utilizada da memória é removida, dando lugar a uma nova página. A complexidade de tempo é O(n*m), onde "n" representa o número de acessos a memoria virtual e "m" representa o tamanho da página. A complexidade de espaço é O(n*sizeof(Page)).

Programa Principal:

O programa principal consiste basicamente de um loop que a cada iteração simula um sistema de memória virtual usando todas as políticas implementadas. Em cada iteração do loop, um conjunto de instruções de acesso à memória é lido do arquivo de entrada juntamente com as configurações do sistema a ser simulado. A partir desses dados, uma simulação é feita usando cada política. O número de *page-faults* de cada política é gravado no arquivo de saída, juntamente com as distâncias de página e de pilha. Ao final de cada simulação, a memória utilizada é liberada.

Organização do Código, Decisões de Implementação e Detalhes Técnicos:

O código está dividido em 13 arquivos principais. Os arquivos "<XXX>Memory.c" e "<XXX>Memory.h" implementam seus respectivos TAD's, que foram assim criados para melhorar a modularização do código.
Uma importante decisão de implementação que foi tomada durante o desenvolvimento desse trabalho, foi a máxima modularização possível de cada tarefa, especialmente, dividindo cada política de reposição de páginas em TAD's diferentes. A implementação foi inspirada no paradigma Orientado a Objetos.
O compilador utilizado foi o MacOSX GCC, no sistema operacional Mac OSX 10.9.2 (Mavericks) através da IDE Eclipse Kepler.

3. Testes

Diversos testes foram realizados com o programa no intuito de avaliar seu comportamento em diversas configurações de memória virtual, tais como páginas grandes ou pequenas demais, memória principal de tamanho muito reduzido, etc.
Os testes foram executados em um computador equipado com processador Intel Core 2 Duo @2.4GHz e 4GB de memória. A saída abaixo mostra uma execução típica do programa:

Console do IDE Eclipse executando o trabalho prático:

```
TP1 [C/C++ Application] /Users/viniciusoliveira/Escolar/Graduacao/UFMG/3oPeriodo/AEDS3/TPs/TP1/Debug/TP1 (19/09/14 10:12)
3
8 4 10
0 2 4 2 10 1 0 0 6 8
6 5 5 5
0 1 -1 2 -2 0 0 1 1
-1 0 -1 1 -1 1 0 0 -1 -1
8 1 10
0 2 4 2 10 1 0 0 6 8
7 7 7 7
2 2 -2 8 -9 -1 0 6 2
-1 -1 -1 1 -1 -1 4 0 -1 -1
10 5 28
1 2 5 10 20 1 2 4 8 17 1 3 7 14 1 2 5 11 23 1 3 6 12 25 1 2 4 9
20 20 15 16
0 1 1 2 -4 0 0 1 2 -3 0 1 1 -2 0 1 1 2 -4 0 1 1 3 -5 0 0 1
-1 0 -1 -1 -1 -1 0 0 -1 -1 -1 0 -1 -1 -1 0 -1 -1 -1 0 -1 -1 -1 -1 0 0 -1
```

Obs.: Os caracteres em verde são valores de entrada, enquanto os pretos são de saída

4. Análises

Algumas análises de caso foram realizadas com o intuito de compreender o comportamento do programa em diversas configurações diferentes. Gráficos foram gerados para melhor compreensão dos resultados. Seguem-se as análises feitas:

1ª Instância:

Tamanho da Memória Física: 8 bytes
Tamanho da página: 4 bytes
Conjunto de acessos: 0 2 4 2 10 1 0 0 6 8
Localidade de Referência temporal: (0+1+1+0+0)/5 = 2/5
Localidade de Referência Espacial: (3x0+4x1+2x2)/9 = 8/9

2ª Instância:

Tamanho da Memória Física: 8 bytes
Tamanho da página: 1 byte
Conjunto de acessos: 0 2 4 2 10 1 0 0 6 8
Localidade de Referência temporal: (0+1+4)/3 = 5/3
Localidade de Referência Espacial: (1x0+1x1+4x2+1x8+1x9)/9 = 26/9

3ª Instância:

Tamanho da Memória Física: 10 bytes
Tamanho da página: 5 bytes
Conjunto de acessos: 1 2 5 10 20 1 2 4 8 17 1 3 7 14 1 2 5 11 23 1 3 6 12 25 1 2 4 9
Localidade de Referência temporal: (8x0)/8 = 0
Localidade de Referência Espacial: (8x0+10x1+4x2+2x3+2x4+1x5)/27 = 37/27

4ª Instância:

Tamanho da Memória Física: 4 bytes
Tamanho da página: 2 bytes
Conjunto de acessos: 1 2 3 1 2 3 4 1 2 3 1 2 1 2 3 1 2 3 4 5
Localidade de Referência temporal: (6x0+8x1)/14 = 8/14
Localidade de Referência Espacial: (6x0+12x1+1x2)/19 = 14/19

5ª Instância:

Tamanho da Memória Física: 6 bytes
Tamanho da página: 3 bytes
Conjunto de acessos: 5 1 6 0 2 4 3 7 6 0 3 5 4 6 1 3 0 5 3 7 2 6 1 4
Localidade de Referência temporal: (18x1+6x0)/11 = 18/11
Localidade de Referência Espacial: (6x0+10x1+7x2)/23 = 24/23

6ª Instância:

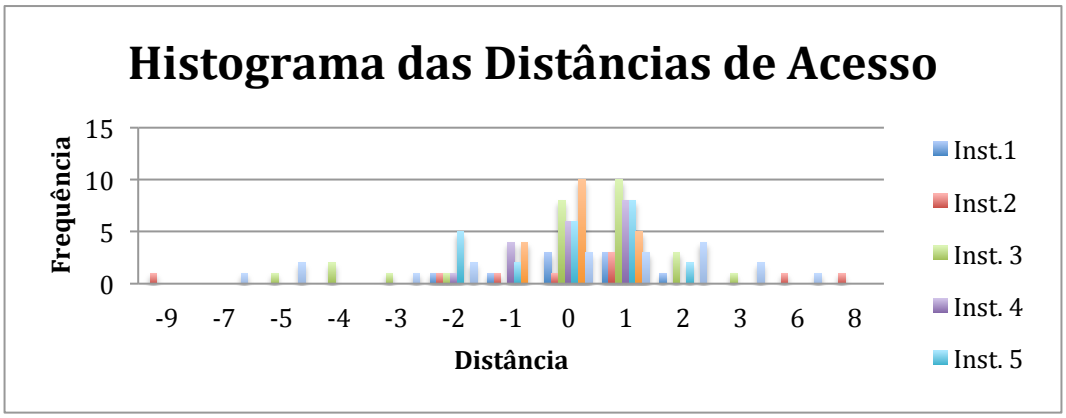
Tamanho da Memória Física: 4 bytes
Tamanho da página: 2 bytes
Conjunto de acessos: 1 2 3 4 5 6 7 6 5 4 3 4 5 5 6 5 4 3 2 2
Localidade de Referência temporal: (10x0+5x1)/13 = 5/13
Localidade de Referência Espacial: (10x0+3x1)/19 = 3/19

7ª Instância:

Tamanho da Memória Física: 12 bytes
Tamanho da página: 4 bytes
Conjunto de acessos: 25 5 13 14 20 31 28 1 7 6 19 29 22 2 24 30 16 9 18 23
Localidade de Referência temporal: (10x0+5x1)/13 = 5/13
Localidade de Referência Espacial: (3x0+3x1+4x2+3x3+2x5+1x6+1x7)/19 = 43/19

Gráficos:

Histogramas:



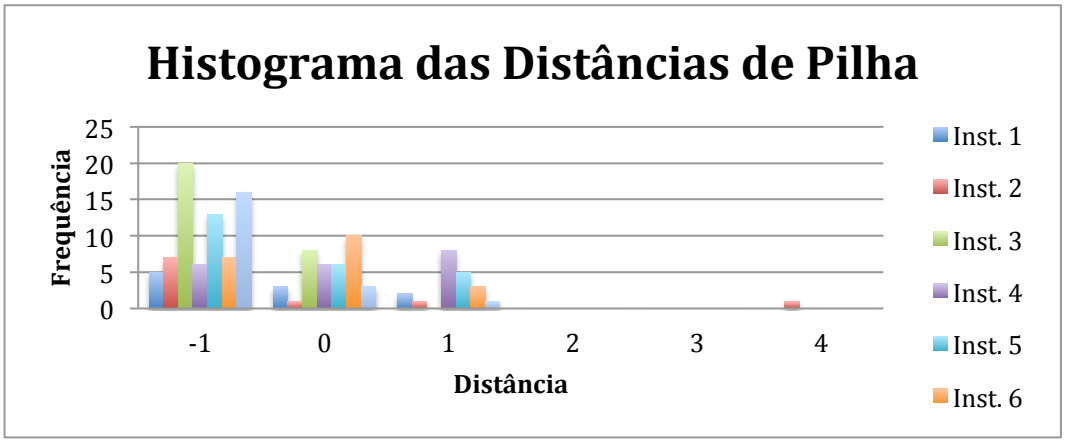


Gráfico Tamanho da página x Bytes Movimentados:

Sequência de Acessos: 25 5 13 14 20 31 28 1 7 6 19 29 22 2 24 30 16 9 18 23
Tamanho da Memória: 12 bytes

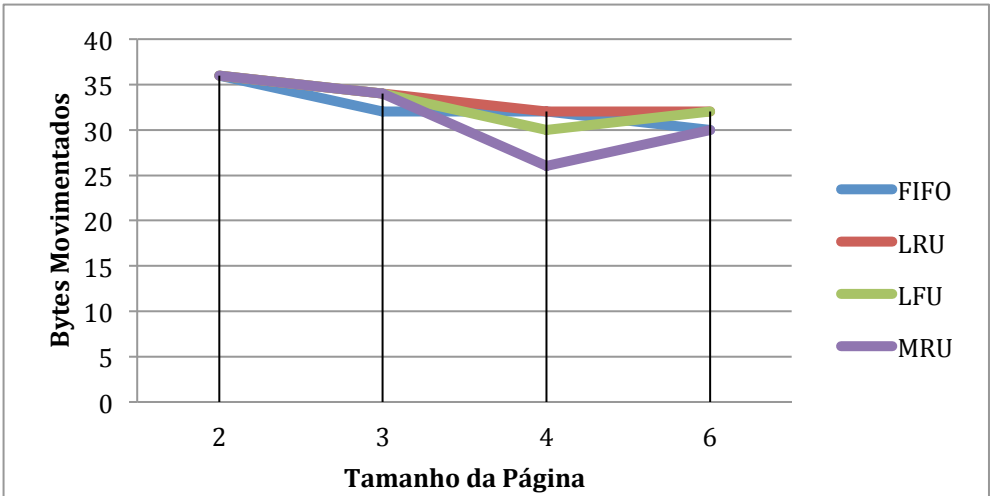
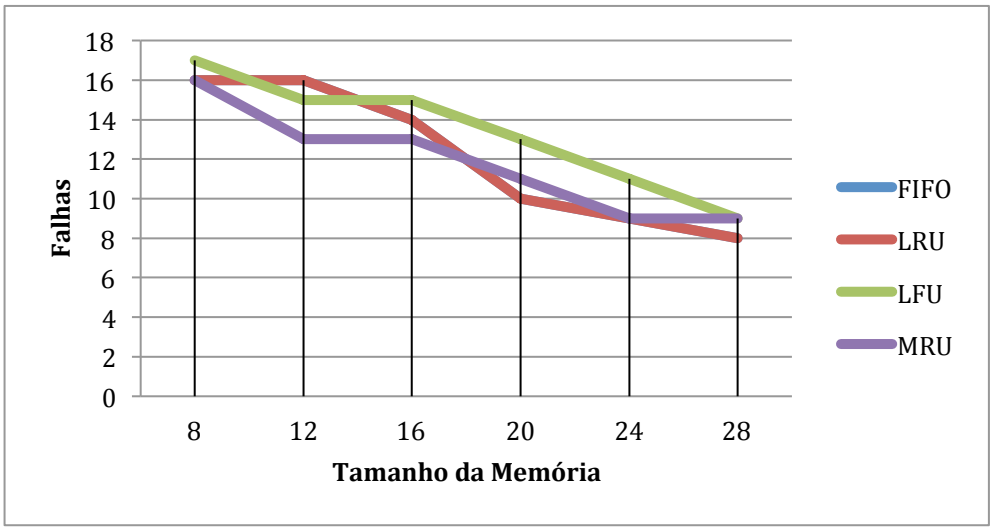


Gráfico Tamanho da memória x Falhas

Sequência de Acessos: 25 5 13 14 20 31 28 1 7 6 19 29 22 2 24 30 16 9 18 23
Tamanho da Página: 4 bytes



Analisando os gráficos elaborados, podemos estimar que para a entrada dada, considerando o custo benefício, o ideal é existirem 20 bytes de memória com páginas de 4 bytes.
Esses valores foram estimados observando que a medida que as configurações se aproximam das ideais, existem bruscas quedas no número de bytes movimentados.

4. Conclusão

A implementação desse trabalho ocorreu como o esperado, porém como não houve avaliação em tempo real (como nos casos onde o TP é corrigido pelo sistema "PRATICO"), muito tempo foi gasto realizando testes exaustivos.
É interessante ressaltar também que o maior desafio na implementação deste trabalho foi a elaboração de sua documentação, que exige uma abordagem analítica bastante detalhada.
Obs.: Apesar do trabalho ter sido compilado usando o comando UNIX *make*, e assim, o uso de IDE's ser dispensável, é importante salientar que o desenvolvimento do código fonte foi feito no ambiente eclipse, uma vez que tal IDE oferece inúmeras facilidades na edição de códigos-fonte. O *makefile* utilizado, no entanto, foi criado a partir do ambiente gedit, e não gerado automaticamente pelo IDE.

5. Referências

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3a Edição, Cengage Learning.