Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Algoritmos e Estruturas de Dados III

Trabalho Prático 1

Alexandre Torres Pimenta

Belo Horizonte, 19 de Setembro de 2014

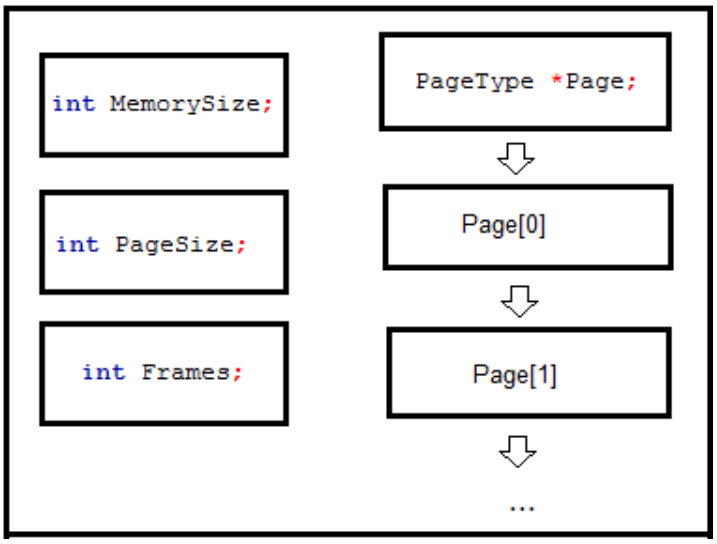
Índice

Introdução

O trabalho prático tem como objetivo implementar um simulador de sistema de memória virtual. Os sistemas de memória virtual fornecem uma abstração para a hierarquia de memória de uma dada arquitetura de modo que programas clientes enxerguem um único espaço de endereçamento sequencialmente acessível.

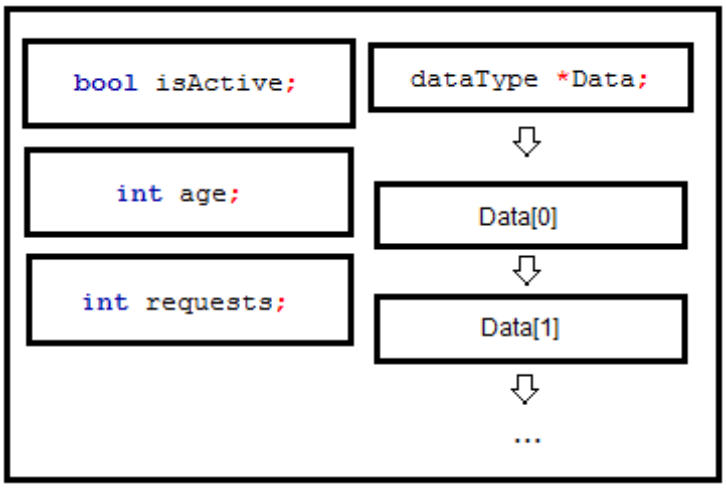
Implementação

O sistema implementado, contém uma mémória principal (RAM), que contém um número dinâmico de molduras de páginas. A representação gráfica do TAD RAM é a seguinte:



A memória contém inteiros para definir o tamanho da memória, o tamanho da página, e o número de molduras (frames) disponíveis na memória, em casa instância de execução.

Também foi utilizado o TAD PageType, que é usado para simular uma página da memória. A sua representação gráfica é a seguinte:



Funções e procedimentos

Junto com os tipos abstratos de dados, foram implementadas as seguintes operações:

int randGen(int n)

Retorna um número aleatório, entre 0 e (n-1).

void clearMemory(RAM *Memory)

A função zera todos os valores das páginas nos frames da memória.

void purgeMemory(RAM *Memory)

A função desaloca todas as páginas da memória.

void replacePage(RAM *Memory, int page, int newData)

A função substitui a página que está no frame indicado, pela que contém o registro enviado como parâmetro.

void fillMemoryFrames(RAM *Memory, int newData, int *firstEmpty)

A função insere a página que contém o registro enviado por parâmetro no primeiro frame vazio da memória.

int firstEmptyFrame(RAM *Memory)

A função retorna a posição do primeiro frame vazio da memória.

bool checkForPage(RAM *Memory, int data, int *page, int *location)

A função verifica se a página que contém o registro enviado por parâmetro está presente na memória. Caso estiver, ela retorna true, e indica a página e em qual posição na página o registro está.

void Aging(RAM *Memory)

A função incrementa a idade de todas as páginas ativas da memória.

int oldestPage(RAM *Memory)

Retorna o frame que contém a página mais antiga da memória.

bool isMemoryEmpty(RAM *Memory)

A função determina se a memória possui frames não ocupados. Retorna true, se houver frame(s) vazio(s), e false se não houver.

int simulateSecondaryMemory(RAM *Memory, int newData)

A função simula a paginação da memória secundária. Ela retorna o primeiro registro da página onde o registro newData estaria na memória secundária.

int leastRequested(RAM *Memory)

A função retorna a posição da página com o menor número de requisições para a política LFU.

bool LFUcheck(RAM *Memory, int requests)

Verifica se existe mais de uma página com o mesmo número de requisições na memória.

int oldestLFU(RAM *Memory, int requests)

Determina a página mais antiga com um determinado número de requisições.

int spatialLocality(RAM *Memory, int newData, int lastData)

A função determina qual página cada um dos dois registros de entrada estão, e retorna a distância entre eles (localidade temporal).

Política de reposição FIFO

O funcionamento da política FIFO, foi implementado conforme o pseudo-código seguinte mostra:

```
SE A PAGINA ESTIVER NA MEMORIA
{
    PAGE HIT
}

SE NÃO ESTIVER
{
    SE AINDA HOUVER ESPAÇO NA MEMORIA
    {
        COLOCAR PAGINA NO PRIMEIRO FRAME LIVRE. INICIALIZAR IDADE E REQUISIÇÕES.
        PAGE FAULT
    }
    SE NÃO HOUVER
    {
        DETERMINAR PAGINA MAIS ANTIGA
        SUBSTITUIR PAGINA MAIS ANTIGA. INICIALIZAR IDADE E REQUISIÇÕES
        PAGE FAULT
    }
}

ENVELHECER PAGINAS
```

Política de reposição LRU

O funcionamento da política FIFO, foi implementado conforme o pseudo-código seguinte mostra:

```
SE A PAGINA ESTIVER NA MEMORIA
{
    ZERAR IDADE DA PAGINA, POIS FOI RECENTEMENTE USADA
    PAGE HIT
}

SE NÃO ESTIVER
{
    SE AINDA HOUVER ESPAÇO NA MEMORIA
    {
        COLOCAR PAGINA NO PRIMEIRO FRAME LIVRE. INICIALIZAR IDADE E REQUISIÇÕES.
        PAGE FAULT
    }
    SE NÃO HOUVER
    {
        DETERMINAR PAGINA MAIS ANTIGA
        SUBSTITUIR PAGINA MAIS ANTIGA. INICIALIZAR IDADE E REQUISIÇÕES
        PAGE FAULT
    }
}

ENVELHECER PAGINAS
```

Política de reposição LFU

O funcionamento da política FIFO, foi implementado conforme o pseudo-código seguinte mostra:

```
SE A PAGINA ESTIVER NA MEMORIA
{
    INCREMENTAR CONTADOR DE REQUISIÇÕES
    PAGE HIT
}

SE NÃO ESTIVER
{
    SE AINDA HOUVER ESPAÇO NA MEMORIA
    {
        COLOCAR PAGINA NO PRIMEIRO FRAME LIVRE. INICIALIZAR IDADE E REQUISIÇÕES.
        PAGE FAULT
    }
    SE NÃO HOUVER
    {
        DETERMINAR PAGINA MENOS REQUISITADA
        DETERMINAR SE EXISTE OUTRA PAGINA COM A MESMA QUANTIDADE DE REQUISIÇÕES.
        CASO EXISTA
        {
            DETERMINAR PAGINA MAIS VELHA COM A MESMA QUANTIDADE DE REQUISIÇÕES, E SUBSTITUI-LA
            PAGE FAULT
        {
        CASO NÃO EXISTA
        {
            SUBISTITUIR A PAGINA MENOS REQUISITADA
            PAGE FAULT
        }
    }
}

ENVELHECER PAGINAS
```

Nova política de reposição: RAND

A nova política de reposição proposta, foi apelidada de RAND. O seu funcionamento é simples: Caso a memória esteja cheia, e o registro requisitado não esteja em memória principal, uma página é sorteada aleatoriamente para ser reposta. O funcionamento da política é descrita pelo pseudo-código:

```
SE A PAGINA ESTIVER NA MEMORIA
{
    PAGE HIT
}

SE NÃO ESTIVER
{
    SE AINDA HOUVER ESPAÇO NA MEMORIA
    {
        COLOCAR PAGINA NO PRIMEIRO FRAME LIVRE.
        PAGE FAULT
    }
    SE NÃO HOUVER
    {
        DETERMINAR ALEATORIAMENTE PAGINA PARA SUBSTITUIR
        SUBSTITUIR PAGINA DETERMINADA.
        PAGE FAULT
    }
}
```

Organização do código e detalhes técnicos

O código está separado em três arquivos. Os arquivos header.h e functions.c são usados para implementar os tipos abstratos de dados e suas operações, e o arquivo main.c implementa o programa principal. O compilador utilizado foi o GNU CCC COMPILER no sistema operacional Microsoft Windows 8.1

Análise de complexidade

int randGen(int n)

A função realiza apenas uma operação. Então, é O(1).

void clearMemory(RAM *Memory)

A função percorre todas as páginas, e todas as posições da páginas. Então, é O(PxT), onde P é a quantidade de molduras de página, e T o tamanho de cada página. Também pode ser notada como O(n), onde n é o número de registros na memória.

void purgeMemory(RAM *Memory)

A função percorre todas as molduras de páginas da memória. Então, é O(P), onde P é a quantidade de molduras de páginas.

void replacePage(RAM *Memory, int page, int newData)

A função percorre todas as posições da página referenciada. Então, é O(T), onde T é o tamanho da página.

void fillMemoryFrames(RAM *Memory, int newData, int *firstEmpty)

O procedimento determina a primeira página vazia usando a função *firstEmptyFrame*, que no pior caso, é O(P). Então, a função percorre todas as posições da página determinada. No pior caso, ela tem complexidade O(PT).

int firstEmptyFrame(RAM *Memory)

A função percorre todos os frames da memória até achar um frame vazio. No melhor caso, é O(1), no caso médio é O(n/2), e no pior caso é O(n).

bool checkForPage(RAM *Memory, int data, int *page, int *location)

A função percorre todos os registros da memória, até encontrar o registro referenciado. No melhor caso, é O(1), no caso médio é O((PT)/2), e no pior casó é O(PT).

void Aging(RAM *Memory)

A função percorre todos os frames da memória. Então, é O(P).

int oldestPage(RAM *Memory)

A função percorre todos os frames da memória. Então, é O(P).

bool isMemoryEmpty(RAM *Memory)

A função percorre todos os frames de memória até encontrar um frame vazio. No melhor caso, é O(1), no caso médio O(P/2), e no pior caso é O(P)

int simulateSecondaryMemory(RAM *Memory, int newData)

A função realiza apenas uma operação. Então, é O(1)

int leastRequested(RAM *Memory)

A função percorre todos os frames da memória. Então, é O(P).

bool LFUcheck(RAM *Memory, int requests)

A função percorre todos os frames até achar uma página com o mesmo número de requisições. No melhor caso é O(1), no caso médio é O(P/2), e no pior caso O(P).

int oldestLFU(RAM *Memory, int requests)

A função percorre todos os frames da memória. Então, é O(P).

int spatialLocality(RAM *Memory, int newData, int lastData)

A função realiza apenas duas operações. Então é O(2)

Complexidade do programa principal

Somando as complexidades das funções, no melhor, médio e pior caso, verificou-se que:

No melhor caso: $O(8P + 4T + 9)$

No caso médio: $O(2PT + 10P + 4T)$

No pior caso: $O(4PT + 13P + 4T)$

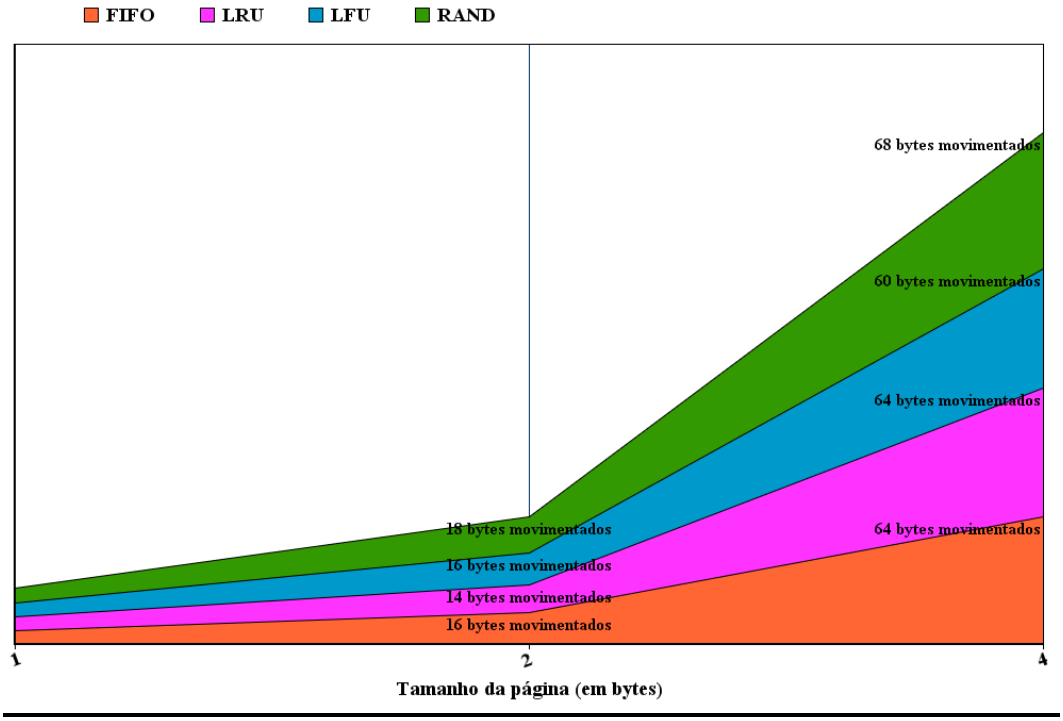
Onde P representa o número de molduras de página (frames), e T o tamanho da página.

Testes

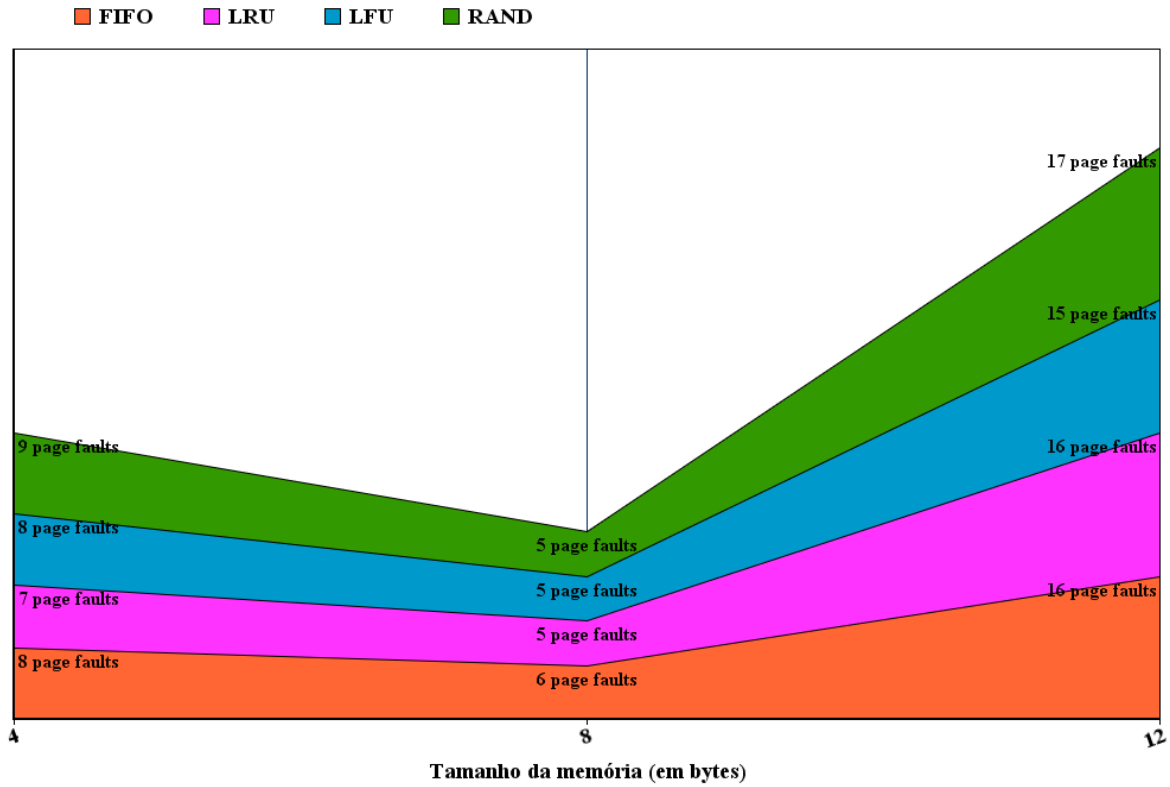
Os testes foram realizados usando um AMD Phenom II X4 840, com 8 GB de memória RAM, no sistema operacional Microsoft Windows 8.1.

Os gráficos a seguir foram gerados utilizando o arquivo de teste com 5 instâncias disponibilizado pelos monitores no moodle

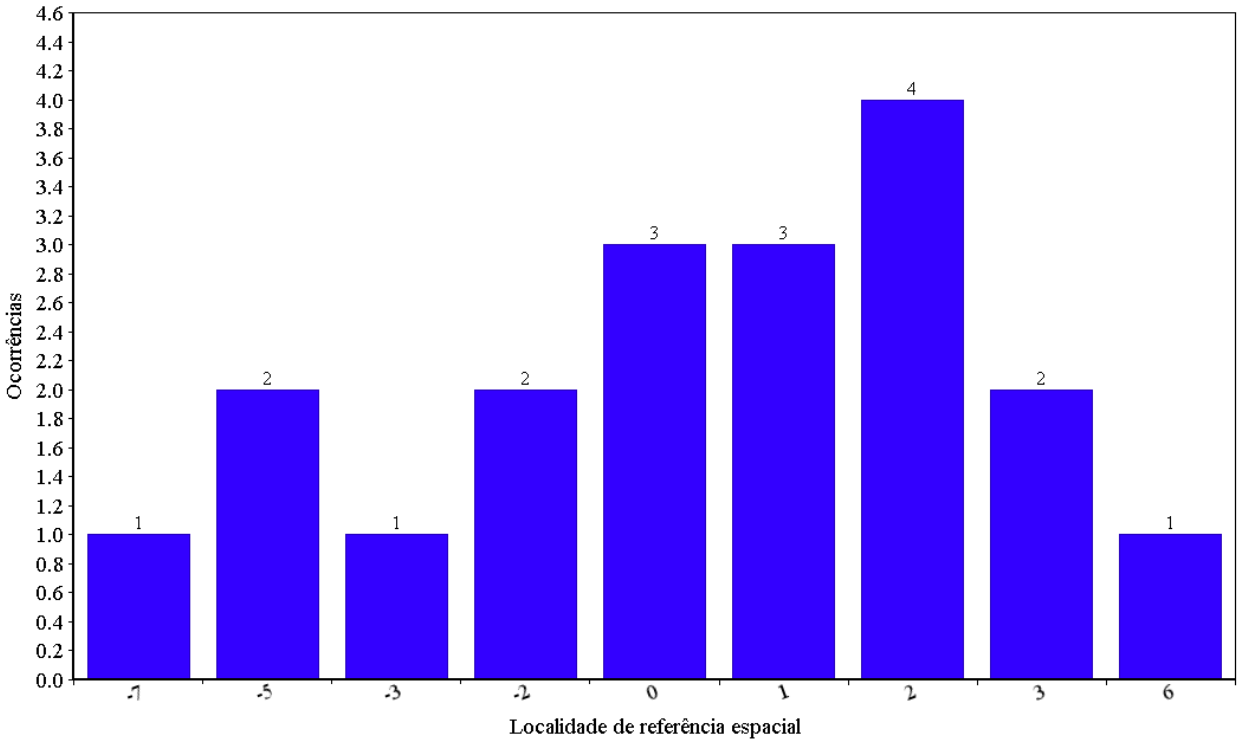
Tamano da página x Bytes Movimentados

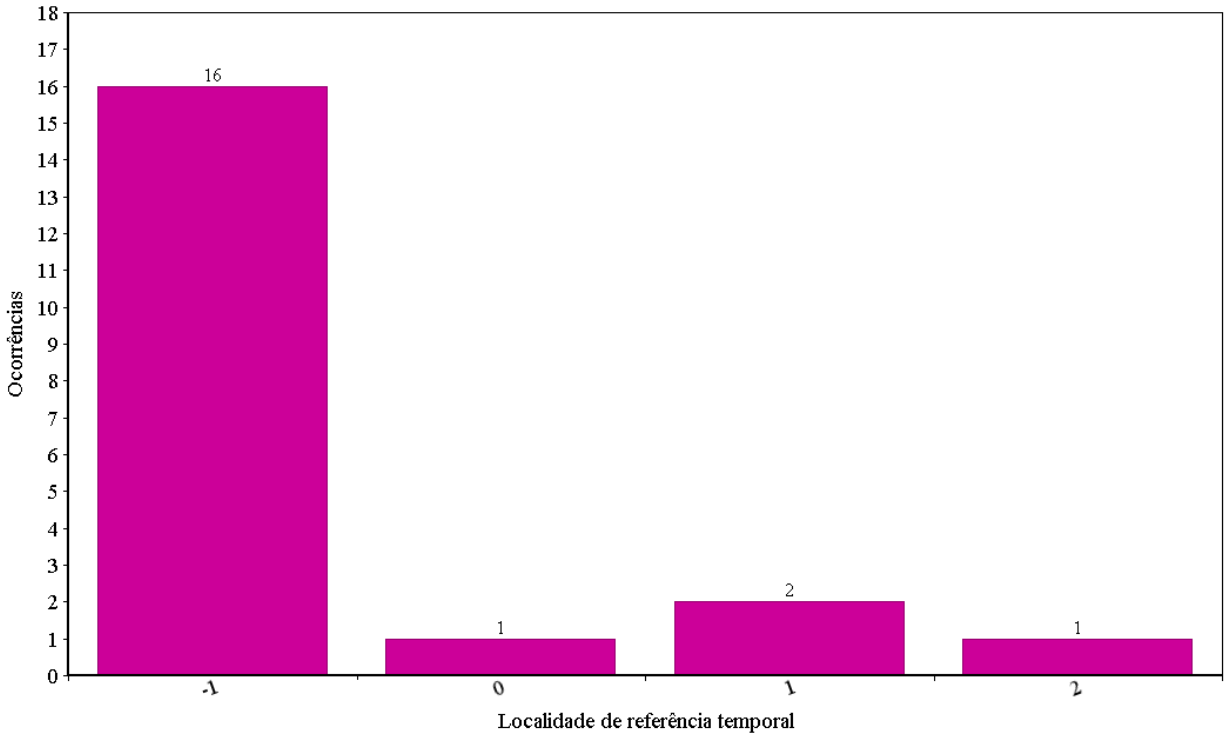


Tamanho da memória x Page Faults



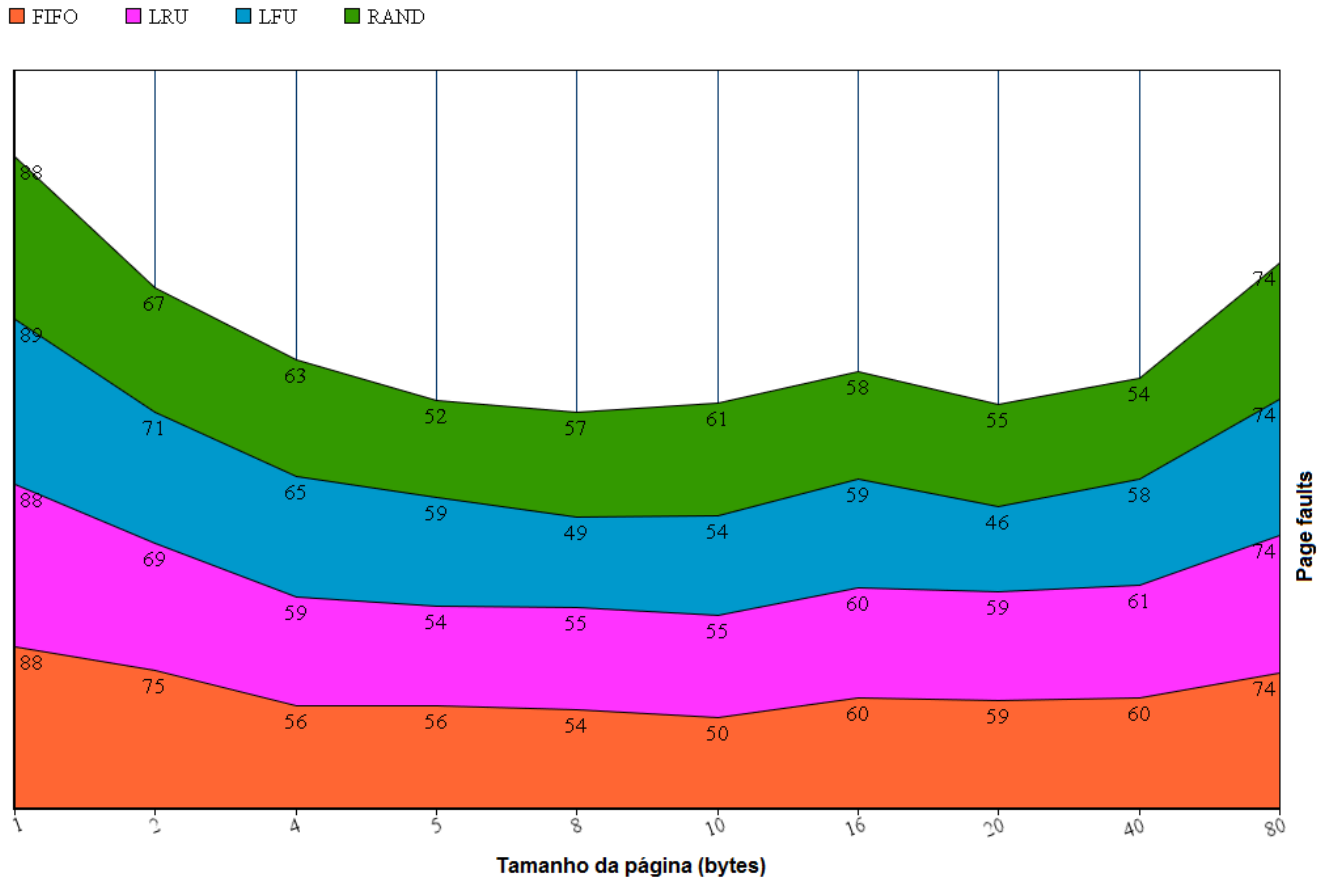
Histogramas de localidade temporal e espacial (feitos com a última instância do arquivo)



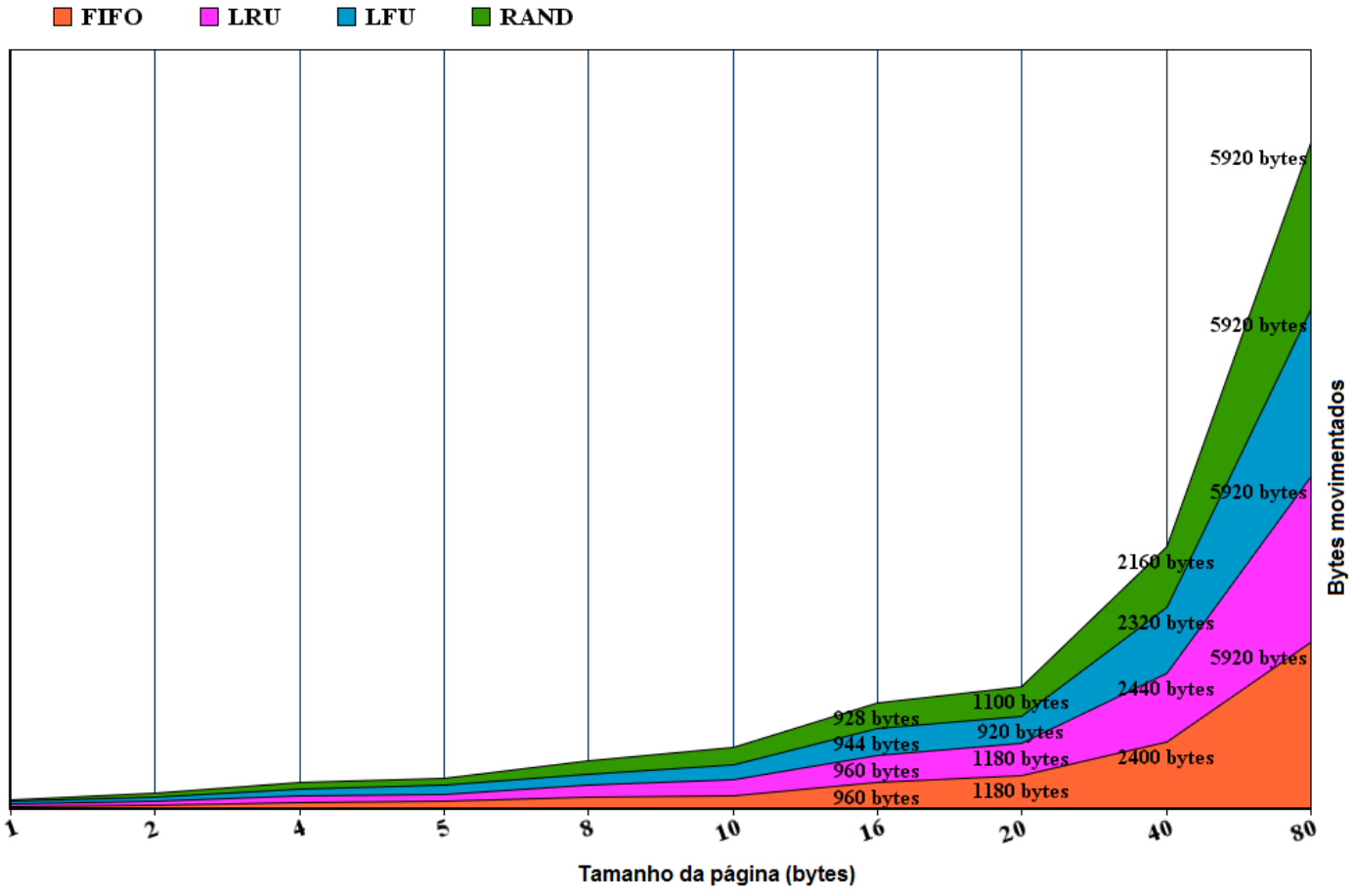


Os gráficos a seguir foram feitos à partir de um arquivo de testes com 150 acessos gerados aleatoriamente, com o tamanho da memória fixado em 80 bytes.

Tamanho da página (em bytes) x Page Faults



Tamanho da página x Bytes movimentados



Conclusão

O trabalho foi essencial para exercitar e desenvolver o conceito de sistemas de memória virtual.

Referências

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

http://en.wikipedia.org/wiki/Cache_algorithms

http://en.wikipedia.org/wiki/FIFO

Anexos

Listagem dos códigos:

- main.c
- procedimentos.c
- tad.h
- geradorTestes.c (Algoritmo utilizado para gerar os testes)