AEDS3 - TP1

Nome: João Pedro Samarino

1. Introdução

O trabalho teve como objetivo principal a implementação de um programa que simula o funcionamento de uma memória virtual genérica, com o intuito de reforçar alguns conceitos já passados em sala de aula como o gerenciamento de memória e localidade de referencia. A memória virtual é a memória endereçada pelo computador, porem não existe fisicamente, ou seja, para usar uma quantidade de memória superior a memória física o sistema tem que gerenciar as paginas que serão realocadas para a memória secundaria para depois inserir novas paginas na memória principal.

Os métodos utilizados para a remoções de paginas têm como principal objetivo minimizar de maneira geral a quantidade de falhas (troca de informação entre dois níveis de memórias), neste trabalho foram implementados três modelos já existentes, são eles: LRU (retira a pagina menos utilizada), FIFO (retira a pagina que entrou primeiro na memória) e LFU (retira a pagina com menor numero de acessos). Neste trabalho também foi proposta a implementação de um novo método, este foi desenvolvido durante o trabalho, funciona sempre retirando a segunda pagina menos usada dando a oportunidade das paginas menos usadas serem ainda usadas é não prejudicando a pagina mais usada.

2. Solução Proposta

A solução proposta constitui de simples algoritmos para a simulação dos métodos de substituição de paginas e calculo das localidades de referencias, espacial e temporal. Para facilitar o trabalho foram criados três tipos abstratos de dados, e o programa foi separado em funções (modulos) elementares independentes, desta maneira foi possível criar um programa simples é menor.

Apesar dos métodos implementados serem simples, para se ter um bom aproveitamento dos recursos da linguagem, primeiro foi necessário pensar em estruturas de dados que iria atender, pois era necessário simular uma memória principal de um computador e às vezes trocar a posição das paginas na memória, o que levou a usar dentro da estrutura de dados principal outra estrutura de lista duplamente encadeada dinâmica, facilitando assim boa parte das operações realizadas nas funções principais.

2.1. Estrutura de Dados:

Para entendermos o funcionamento do programa primeiro temos que entender as estruturas de dados abstratas criadas nesta solução, para então explorarmos os algoritmos de maneira completa. Foram criadas cinco estruturas básicas para armazenamento de registros de maneira mais abstrata, ou seja, para facilitar a manipulação pelas funções dos registros lá armazenados. As estruturas de dados estão contidas nos arquivos (lista.h) e (memoria.h),

separadas por finalidades, no arquivo (lista.h) existem somente estruturas relacionadas a lista implementada, já em (memoria.h) existe a estrutura principal de memória .

As estruturas de dados abstratos do arquivo (lista.h):

- typedef struct _Celula { int chave; int lfu; struct _Celula * proximo; struct _Celula * anterior; } Celula;

A estrutura (Celula) é a estrutura básica do programa, todas as outras estruturas a contem internamente, está é responsável por armazenar os itens para o funcionamento correto das funções, estes itens são, (chave) responsável pro guardar a chave indicador da célul , (lfu) contador responsável pela método de reposição de paginas LFU, (*proximo) ponteiro responsável por apontar para a célula seguinte da lista, (*anterior) ponteiro responsável por apontar para a célula anterior.

- typedef Celula* Apontador;

Esta estrutura feita para abstrair o apontamento da estrutura (celula), ou seja, simplificando o uso do ponteiro quando necessário.

- typedef int TipoChave;

Esta estrutura define o indicador da célula como um inteiro, foi escolhido esse tipo de variável, pois ela e a menor estrutura que atendia os requisitos impostos pelos algoritmos.

- typedef struct {Apontador primeiro; Apontador fim;} Lista;

A estrutura (Lista) é responsável por armazenar dois ponteiro do tipo (Apontado), a estrutura foi constituída desta maneira porque a alocação das células e feita de maneira dinâmica, assim os ponteiros (primeiro) e (fim) definem só as células do inicio e do fim da lista, desta maneira podemos começar a percorrer a lista do inicio ou do fim, melhorando o tempo de acesso dos dados da estrutura.

As estruturas de dados abstratos do arquivo (memoria.h):

- typedef struct { int tamanho_total; int tamanho_pagina; int qt_paginas_atual; Lista pagina; } Memoria_principal;

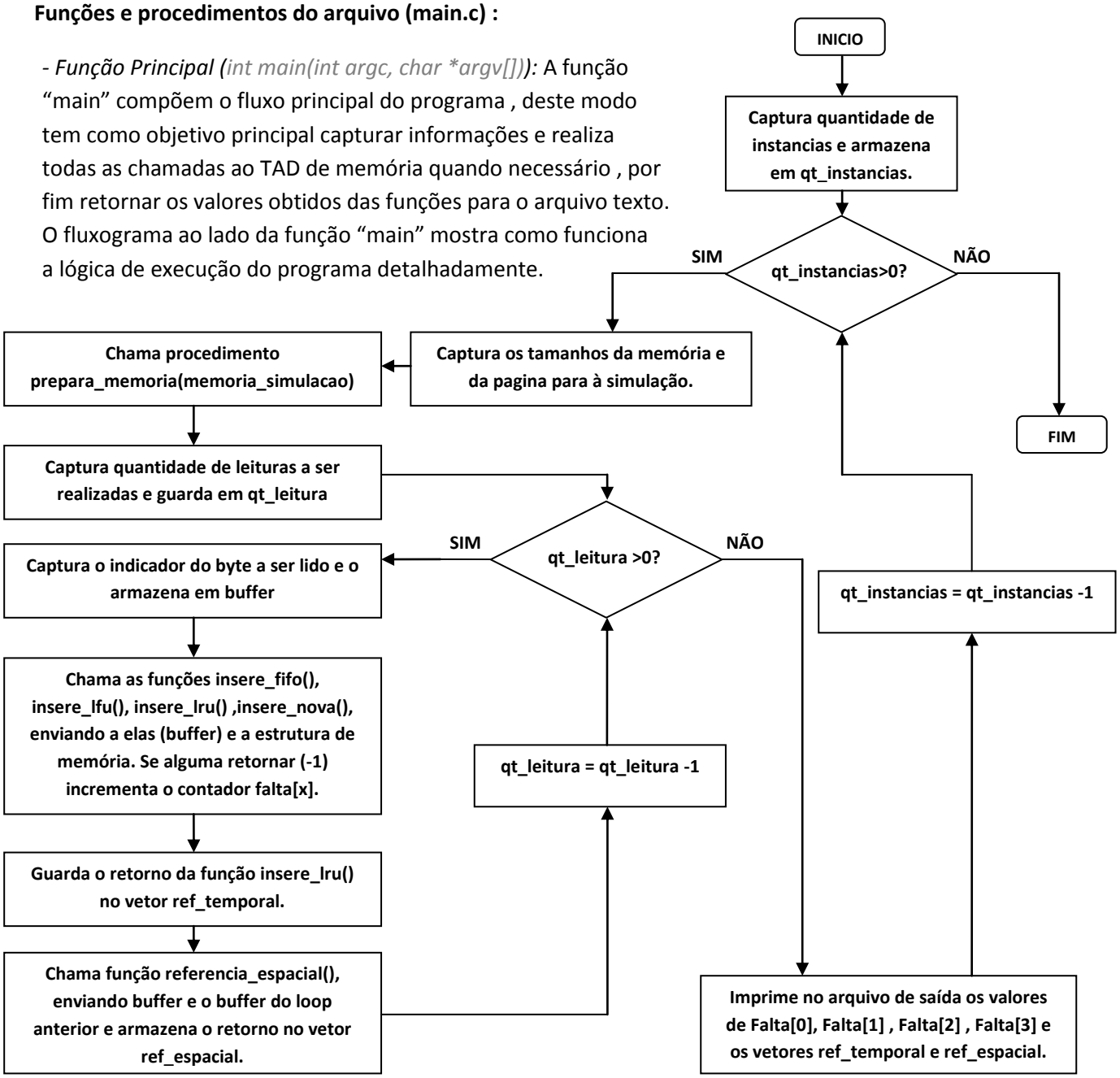
A estrutura (Memoria_principal) é a mais externa do programa , esta é a principal estrutura utilizada nas funções, a mesma simula a memória principal do computador, contendo as seguintes informações, (tamanho_total) tamanho em bytes da memória, (tamanho_pagina) tamanho em bytes de cada pagina, (qt_paginas_atual) a quantidade de paginas que estão sendo armazenadas no momento na estrutura, (pagina) uma lista que simula as paginas de memória na estrutura, vale ressaltar que por se tratar de uma simulação os bytes dentro de cada pagina não são realmente alocados, o objetivo do programa e apenas simular o funcionamento de métodos de substituição de paginas, o que não necessita da alocação destes bytes.

2.2.Funções e Procedimentos

No programa foram criadas funções é procedimentos para manipulação de dados é execução da proposta de solução que tem como objetivo gerar o arquivo solução. O arquivo (main.c) contem procedimentos do fluxo principal e o (memoria.c) possui procedimentos relacionados ao TAD da manipulação da memória, contendo os procedimentos e funções de reposição de paginas necessários. No arquivo (lista.c) existem os procedimentos de manipulação da estrutura da lista encadeada que são usados continuamente pelas funções da TAD de memória.

Funções e procedimentos do arquivo (main.c) :

- *Função Principal (int main(int argc, char *argv[])):* A função "main" compõem o fluxo principal do programa , deste modo tem como objetivo principal capturar informações e realiza todas as chamadas ao TAD de memória quando necessário , por fim retornar os valores obtidos das funções para o arquivo texto. O fluxograma ao lado da função "main" mostra como funciona a lógica de execução do programa detalhadamente.



Funções e procedimentos do arquivo (lista.c) :

- void FLVazia(Lista *l): Procedimento responsável por alocar na lista uma célula cabeça, este procedimento tem como finalidade de evitar erros de apontamento caso a lista esteja vazia, depois de alocar a célula cabeça ele define os apontadores da lista (l) para esta célula.

- void Insere_l(TipoChave x, Apontador posicao,Lista *l, int con_lfu): Procedimento responsável por inserir uma célula nova na lista (l) , nesta célula e atribuído os valores da chave (x) e do contador (con_lfu). A inserção e feita na frente da célula enviada (posicao), o procedimento e feito através da troca de apontamentos das células da lista, caso exista a célula cabeça na lista(l) ela e deletada para não afetar a lógica do programa.

- void limpa_l(Lista *l): Procedimento responsável por limpar a lista (l) , para fazer isso o procedimento percorre a lista e vai excluindo as células em que passa até o final da lista. No final do processo ele chama a função (FLVazia) para criar uma nova célula cabeça na lista (l).

-void busca_l_lfu(Lista *l, Apontador *celula_v): Este procedimento tem o objetivo de percorrer a lista do final em direção ao inicio, procurando a primeira célula da seqüência desta lista que possui o menor valor da variável da célula (lfu). Quando e percorrida toda a lista então a célula procurada já foi definida, o apontador para a mesma é copiado em (*celula_v).

- int Busca_l(TipoChave chave, Lista *l, Apontador *celula): Função responsável por buscar a célula que possui o valor de sua chave igual (chave) na lista (l) , a função percorre do inicio da lista em direção ao final, a cada célula que a mesma pula e incrementado um valor em uma variável genérica (i) , quando a célula procurada é achada o apontador da mesma é copiado para (*celula) é a função retorna o valor de (i) . O valor desta variável é muito importante, pois a mesma e utilizada para calcular a referencia temporal na função (insere_lru()). Quando o valor de (chave) não é encontrado a função retorna (-1).

-int Remove_l(Apontador posicao, Lista *l):Função responsável por deletar a célula (posicao) da lista (l) , este procedimento e realizado mudando quatro ponteiros das células para não se perder a seqüência da lista em questão. Quando os ponteiros são mudados o espaço que a célula apontada pelo ponteiro (posicao) é desalocado da memória. Retorna (1) se for possível remover o item.

Funções e procedimentos do arquivo (memoria.c) :

-void limpa_memoria(Memoria_principal* mp): Procedimento responsável por excluir todas as paginas da memória (mp) , como foi descrito nas estruturas de dados as paginas são representadas por uma lista dentro da estrutura (Memoria_principal), então para limpar a memória a função zera as variáveis de (mp) e manda a lista interna para o procedimento (limpa_l()) assim a estrutura (mp) é zerada é pode receber novos dados.

- void prepara_memoria(Memoria_principal* mp, int tamanho_p, int tamanho_t): Este procedimento é muito simples , tem como finalidade preparar a estrutura de memória (mp) para receber inserções variadas, para isto o procedimento atribui os valores de (tamanho_p), (tamanho_t) as variáveis internas (tamanho_pagina) e (tamanho_total) consecutivamente. Em

(mp) e aloca uma célula cabeça na estrutura de lista interna (pagina) , para alocar a célula cabeça se envia a lista (pagina) mp para o procedimento (FLVazia()).

-int insere_fifo(Memoria_principal* mp, int indicador): Função responsável por inserir o (indicador) na estrutura de memória (mp) , a função retorna (-1) se o indicador não estiver contido em nenhuma pagina da estrutura (mp) ,ou seja, quando se tem uma falta de pagina. Esta funciona da seguinte maneira, o primeiro passo e transformar o indicador em um indicador principal, ou seja, o indicador do primeiro byte da pagina, para realizar este procedimento é feita a seguinte operação matemática, (piso((indicador)/(total de bytes de uma pagina)) * (total de bytes de uma pagina)) , esta operação gera o indicador principal o qual agora e utilizado em todas as operações da função. Com este indicador principal a função procura se existe alguma pagina que o contem, enviando o indicador principal é a lista interna (pagina) para a função (Busca_I()), se esta retornar (-1) indica que o indicador principal não esta na pagina, então a função verifica se a lista (pagina) não esta cheia , se a mesma não estiver, adiciona o indicador valido no inicio da lista através da função (insere_I()). Se a mesma estiver cheia a função exclui o ultimo item da lista (pagina) através da função (Remove_I()) para depois adicionar o indicador valido no inicio da lista(pagina). A função retorna o valor de retorno da função (Busca_I()), que retorna (-1) quando não acha o indicador valido.

-int insere_lru(Memoria_principal* mp, int indicador): Função responsável por inserir o (indicador) na estrutura de memória (mp), respeitando os critérios do método LRU, a função retorna (-1) se o indicador não estiver contido em nenhuma pagina da estrutura (mp) ,ou seja, quando se tem uma falta de pagina. Esta funciona da seguinte maneira, o primeiro passo e transformar o indicador em um indicador principal, ou seja, o indicador do primeiro byte da pagina, para realizar este procedimento é feita a seguinte operação matemática, (piso((indicador)/(total de bytes de uma pagina)) * (total de bytes de uma pagina)) , esta operação gera o indicador principal o qual agora e utilizado em todas as operações da função. Com este indicador principal a função procura se existe alguma pagina que o contem, enviando o indicador principal é a lista interna (pagina) para a função (Busca_I()), se esta retornar (-1) indica que o indicador principal não esta na pagina, então a função verifica se a lista (pagina) não esta cheia , se a mesma não estiver, adiciona o indicador valido no inicio da lista através da função (insere_I()). Se a mesma estiver cheia a função exclui o ultimo item da lista (pagina) através da função (Remove_I()) para depois adicionar o indicador valido no inicio da lista(pagina). Agora se a função (Busca_I()) retornar um valor diferente de (-1) esta função pega a célula referente ao retorno da função de busca é passa esta célula para o inicio da lista(pagina). A função retorna o valor de retorno da função (Busca_I()), que retorna (-1) quando não acha o indicador valido ou o valor da distancia de pilha ate achar o indicador, este retorno já fornece a referencia temporal para o programa.

-int insere_lfu(Memoria_principal* mp, int indicador): Função que insere o (indicador) na memória principal (mp), como nas outras funções ele gera um indicador valido para realizar as operações internas da função. Com o indicador valido a função busca através da a função (Busca_I()) se o indicador principal já esta contido na lista (pagina), se o mesmo já esta na sub-estrutura a função incrementa (1) no contador (lfu) presente na célula é passa ele para a primeira posição da lista(pagina). Se o indicador valido não estiver presente na lista, verifica se a lista não esta cheia, se não estiver insere no inicio da lista, porem se a lista estiver cheia

envia a lista(pagina) para o procedimento (void busca_l_lfu()) o qual de da uma copia do apontador da célula que possui o menor contador (lfu), então essa célula e excluída através da função (Remove_l()) é a nova célula e inserida no inicio da fila. A função retorna o valor de retorno da função (Busca_l()), que retorna (-1) quando não acha o indicador valido.

-int insere_nova(Memoria_principal* mp, int indicador): Esta função teve como objetivo criar uma nova maneira de retirar paginas da memória principal, funcionando da seguinte maneira , primeiro como as outras funções , transforma o indicador em um indicador valido, então com este indicador valido verifica se o mesmo já esta contido na sub-estrutura (pagina) se já estiver, o passa para o inicio da lista, como a função (insere_lru()), porem se o indicador não estiver na sub-estrutura (pagina), será verificado se a mesma esta cheia, se estiver, então é removido o segundo item da fila, ou seja, o segundo item mais utilizado , dando a oportunidade dos outros que foram menos utilizados serem ainda utilizados é preservando a ultima pagina que foi utilizada. Se ainda houver espaço na lista (pagina) a célula e incluída no início da lista. O retorno da função e o mesmo das outras funções, retorna (-1) se houver uma falha de pagina, pois o retorno também e dado pela função (Busca_l()).

-int referencia_espacial(int tamanho_p,int inicio , int fim): Esta função funciona de uma maneira muito simples, ela retorna a referencia espacial do passo (inicio) para o (fim), para executar essa operação e simplesmente feita a seguinte conta, (piso((fim)/(tamanho_p))-(piso(inicio)/(tamanho_p))) , então a função retorna este valor que é a referencia espacial deste passo.

3.Análise de Complexidade

Procedimentos e funções, FLVazia , Insere_l , Remove_l, referencia_espacial e prepara_memoria: Estes tem uma complexidade de tempo O(1). Esta complexidade de tempo e devido às atribuições que estes procedimentos fazem, todos fazem operações ou chamadas de custo fixo, como atribuição de valores ou mudança em ponteiros. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis buffers é no caso de (insere_l) a mesma cria de uma célula a cada chama, continuando sendo sua complexidade de espaço O(1).

Procedimento limpa_memoria , limpa_l e busca_l_lfu: São procedimento tem uma complexidade de tempo O(n), onde n e a quantidades de células na estrutura, esta complexidade se da devido ao loop internos que percorrem a lista da estrutura que trabalham. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Função Busca_l : Esta função é um pouco diferente das outras , pois ela nem sempre percorre toda a lista , ela percorre somente ate achar o valor procurado ,se não achar ela chega ao final, desta maneira no pior caso tem uma complexidade de tempo O(n) ,onde n e a quantidades de células na estrutura. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Funções insere_nova, insere_lfu e insere_fifo : Estas funções realizam operações O(1) e uma chamada a função Busca_l que é O(n) no pior caso , desta maneira elas herdam a

complexidade de Busca_l, ou seja, O(n) no pior caso. Vale lembrar que esta complexidade é referente a simulação, pois quando uma pagina é deslocada realmente para a memória secundaria o custo de tempo e relativo a velocidade de deslocamento. Já a complexidade de espaço é O(1) pois elas alocam no máximo uma célula por chamada.

Função insere_lfu: A diferença de complexidade desta função em relação as outras funções de inserção esta nas chamadas que está realiza , diferente das outras esta função realiza duas chamada, uma a Busca_l que é no pior caso O(n) e outra a busca_l_lfu que é sempre O(n) , desta maneira a complexidade é O(n). E a complexidade de espaço e O(1) igual das outras funções de inserção.

Função main – função principal: a função principal faz uma varreduras que podem varia de 0 a X , onde X é a quantidade de instancias informada no arquivo de entrada, no final de cada ciclo, o programa chama as funções da TAD , porem ele pode chamar ate C vezes quatro funções O(n) alem de executar comandos O(1) ,onde C é a quantidade de itens a serem lidos, em cada instancia, considerando que se pode ser feitos indefinidas instancias. Temos O(x)* K*O(n*C) +Z*O(1)=O(n*x*C) ,onde K é Z são constantes. Porem essa analise só pode ser feita considerando que todas as memórias da instancia vão ter o mesmo tamanho, mantendo assim a relação “n” sempre igual, se o tamanho das memórias varia a complexidade de tempo pode variar também. A complexidade de espaço é O(C) para cada instancia, lembrando que o C pode variar em cada instancia, o programa pode chamar C vezes uma função de complexidade O(1), porem a cada novo ciclo o espaço alocado e limpado para poder ser novamente alocado com um novo tamanho.

4. Experimentos e Análises

Foram feitos experimentos para realizar as análises pedidas no trabalho, o arquivo de entrada possui sete instâncias, para elas foi feito as seguintes análises, referencia temporal, referencia espacial e histograma das referências obtidas. Também foi necessário fazer dois gráficos (tamanho da pagina X bytes movimentados) e (tamanho memória X falhas).

4.1. Maquina utilizada

Os testes foram realizadas em um Pentium core 2 duo, com 3 Gb de memória principal, o sistema operacional utilizado foi o Ubuntu Linux para arquitetura AMD64 (64 bits por ciclo de maquina) .

4.2.1 Arquivo de entrada

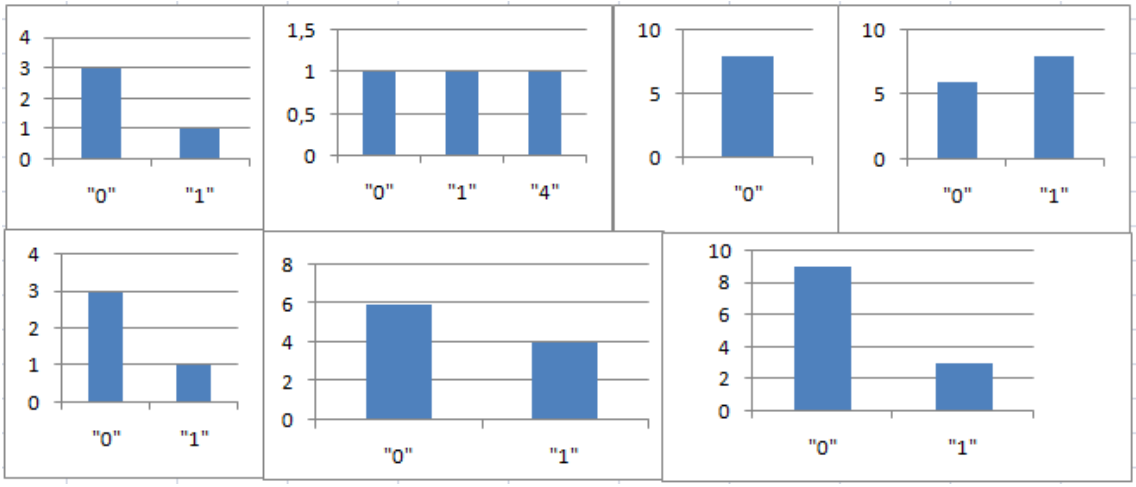
Abaixo podemos ver a formatação de entrada do arquivo fornecido para análise

```
7
8 4 10
0 2 4 2 10 1 0 0 6 8
8 1 10
0 2 4 2 10 1 0 0 6 8
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
4 2 20
1 2 3 1 2 3 4 1 2 3 1 2 1 2 3 1 2 3 4 5
6 3 24
5 1 6 0 2 4 3 7 6 0 3 5 4 6 1 3 0 5 3 7 2 6 1 4
4 2 20
1 2 3 4 5 6 7 6 5 4 3 4 5 5 6 5 4 3 2 2
12 4 20
25 5 13 14 20 31 28 1 7 6 19 29 22 2 24 30 16 9 18 23
```

4.2.2 Análises localidades de referências temporais

Instancia 1: -1 0 -1 1 -1 1 0 0 -1 -1
Instancia 2: -1 -1 -1 1 -1 -1 4 0 -1 -1
Instancia 3: -1 0 -1 -1 -1 -1 0 0 -1 -1 -1 0 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 0 -1
Instancia 4: -1 -1 0 1 1 0 -1 -1 -1 0 1 1 1 1 0 1 1 0 -1 0
Instancia 5: -1 -1 -1 1 0 -1 0 -1 0 -1 -1 0 0 -1 -1 -1 1 1 0 -1 -1 1 1 -1
Instancia 6: -1 -1 0 -1 0 -1 0 0 1 0 -1 1 0 0 -1 1 0 -1 0 0
Instancia 7: -1 -1 -1 0 -1 -1 0 -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 1 -1

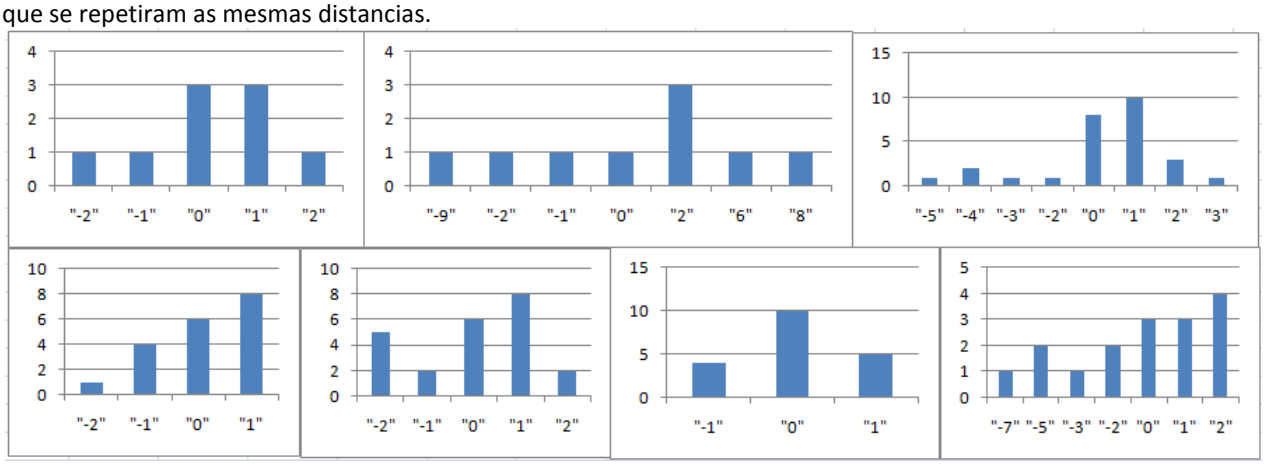
Abaixo podemos ver o histograma da instância 1 a 9 consecutivamente, desta maneira fica fácil para se perceber a distancia de pilha que cada instância gera. O numero entre “ ” se refere a distancia de pilha e as linhas representam a quantidade de vezes que a distância ocorreu naquela instância.



4.2.3 Análises localidades de referências espaciais

Instancia 1: 0 1 -1 2 -2 0 0 1 1
Instancia 2: 2 2 -2 8 -9 -1 0 6 2
Instancia 3: 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
Instancia 4: 1 0 -1 1 0 1 -2 1 0 -1 1 -1 1 0 -1 1 0 1 0
Instancia 5: -1 2 -2 0 1 0 1 0 -2 1 0 0 1 -2 1 -1 1 0 1 -2 2 -2 1
Instancia 6: 1 0 1 0 1 0 0 -1 0 -1 1 0 0 1 -1 0 -1 0 0
Instancia 7: -5 2 0 2 2 0 -7 1 0 3 3 -2 -5 6 1 -3 -2 2 1

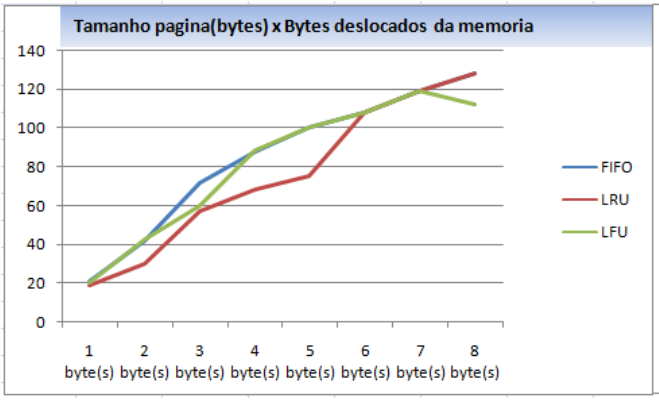
Abaixo podemos ver os histogramas das distancias de acesso das instâncias, começando de 1 e indo até 9. Os números entre “ ” se refere a distancia de acesso e as linhas representam a quantidade de vezes que se repetiram as mesmas distancias.

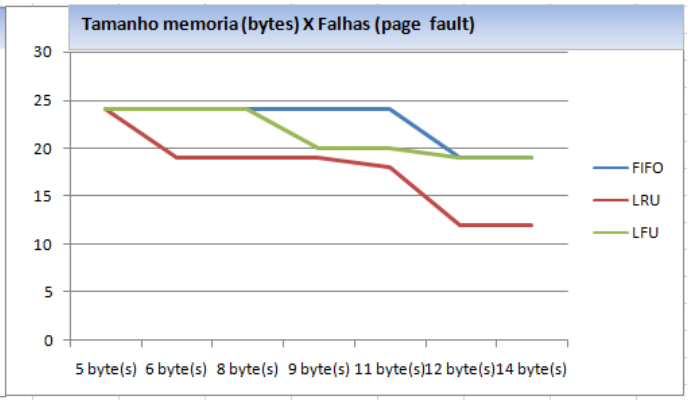


4.2.4 Análises dos Gráficos

Para gerar os gráficos foi utilizada a maior instância das fornecidas no arquivo disponibilizado, no primeiro gráfico o valor de X foi deixado como (10), sendo que X representa a memória principal e o de Y que representa o tamanho da pagina que variou. No segundo gráfico o X variou é o Y foi deixado como (3), abaixo podemos ver a instancia utilizada e os gráficos.

```
1  X Y 28
2  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
```

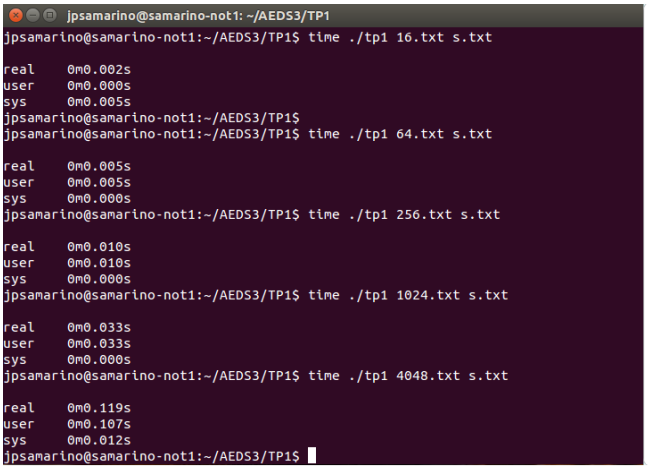


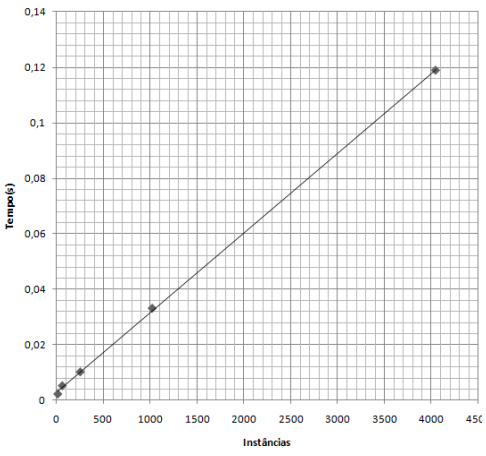


Podemos ver no primeiro gráfico que em geral com o aumento das paginas a quantidade de dados transferidos aumenta, existem alguns picos neste, mas no geral o tamanho da pagina ideal depende do seu tipo de uso. No segundo gráfico podemos ver que existem quedas repentinas de falhas, o aumento de certo valor de memória nem sempre resulta em uma queda proporcional nas faltas, com esse tipo de gráfico e possível determinar o número mínimo de memória para uma quantidade de faltas mínimas em uma determinada instância.

4.3. Teste de desempenho

Foi realizado um teste de desempenho para determinar o comportamento do programa para um numero maior de instancias é ver se a ordem de complexidade corresponde aos dados práticos, abaixo podemos ver o tempo que o programa demorou em processar as instancias é o gráfico da relação tempo por instancias, para este teste o tamanho da memória, pagina é numero de leituras são constantes para todos os arquivos testados.





Podemos visualizar através gráfico que a relação entre as entradas é linear, ou seja, ocorreu a saída esperada, pois na ordem de complexidade calculada da (main()) chegamos a está analise.

5.Implementação

Para desenvolver este trabalho, o mesmo foi dividido em cinco arquivos com o intuito de melhorar a organização, normalizar as funções e estruturas de acordo com os seus objetivos.

5.1.Arquivos utilizados

(main.c): Arquivo principal, responsável pelo fluxo do programa, relacionado a entrada e saída de dados dos arquivos.
(memoria.c): Arquivo responsável pelos procedimentos e funções relacionadas ao TAD de memória, ou seja, o TAD que é usado pela função (main()).
(memoria.h): Este arquivo contem o cabeçalho das funções relacionadas ao TAD de memória, também possui as estruturas de dados ligadas ao programa e suas definições.
(lista.c): Arquivo responsável pelos procedimentos e funções relacionadas ao TAD da lista duplamente encadeada.
(lista.h): Este arquivo contem o cabeçalho das funções relacionadas ao TAD da lista, também possui as estruturas de dados ligadas a lista.

5.2. Compilação

O programa deve ser compilado no compilador GCC através de um makefile ou do seguinte comando no terminal:
gcc main.c memoria.c lista.c -o tp1

5.3. Execução

A execução do programa tem como parâmetros:
• Arquivo de entrada.
• Arquivo de saída.
O comando no terminal para a execução do programa e da forma:
./tp1 < arquivo de entrada > < Arquivo de saída >

5. Conclusão

Foi possível com este trabalho aprender mais sobre política de reposição de memórias e o tratamento de paginas que um sistema de memória virtual realiza. Indiretamente houve a necessidade de revisar conceitos aprendidos em outros cursos, como o funcionamento de uma lista encadeada, pois a mesma foi implementada na solução deste problema. A maior dificuldade durante a execução do trabalho foi à falta de informações disponíveis sobre as análises que deveriam ser realizadas, algumas considerações tiveram que ser feitas para que as análises fossem concluídas.
Em relação às políticas de reposição programadas conclui que cada uma dessas possui vantagens e desvantagem uma em relação à outra, no entanto a política LRU possui uma boa eficiência é tem uma fácil implementação, já a LFU possui uma boa eficiência, porem se tem a necessidade de ter um contador de acesso o que a torna mais pesada que as outras estudadas. Todas essas inclusive a nova política desenvolvida seguem a linha de análise do passado para prever o futuro e obter o menor numero de faltas possíveis, esta idéia de previsão faz com que se possa obter uma eficiência maior nestas políticas.