Relatório de Infraestrutura de Software

Simulação de Memória Virtual de um Sistema Operacional

Autor
Lucas Borges Dalcin

O código foi organizado de maneira a tal que visa deixar o mais conciso e intuitivo o possível para que a pessoa que venha a observar o código no futuro, desde que tenha conhecimento da teoria de como funciona uma virtualização de memória, possa de maneira intuitiva e fácil entender o que se passa ao longo do código sem que tenha de perder tempo procurando entender cada linha separada o que faz e o que envia.

Para tentar ao máximo possível facilitar a leitura do código eu usei a lógica que fazia mais sentido para mim e que se assemelha com a nomenclatura utilizada nos métodos de JAVA, ou seja, ao invés de se utilizar de um corpo de código muito longo, as operações mais complexas foram quebradas em várias funções separadas as quais em seu nome já dizem o que fazem e o que retornam de sua operação de maneira intuitiva, o que acarreta no fácil acompanhamento e entendimento do código como um todo, isto é, assumindo que a pessoa já tenha ciência de como o processo de virtualização de memória funciona junto da ordem em que se deve realizar as operações.

1. Structs:

Para começar, visto que teríamos de utilizar ou um array ou uma lista encadeada para realizar as operações ao longo do código com a tabela de página junto da tradução do endereço, pensei que a melhor solução seria me utilizar de um array de structs, onde eu teria uma maleabilidade maior e um entendimento mais fácil sobre como o programa estaria ocorrendo. As structs estão sendo criadas num arquivo separado dentro da pasta “utils”, o arquivo sendo: “structs.h”

2. Funções:

Sabendo que o código seria enorme caso eu o fizesse somente em um único arquivo, pensei que o melhor método de abordagem seria quebrar o código inteiro em pequenas funções, as quais os próprios nomes ja dizem o que realizam, e as chamar toda vez que necessário. Desse jeito eu poderia automatizar o processo inteiro de escrita do código junto de poupar muito tempo, já que tanto o FIFO e o LRU se utilizam de lógicas similares de implementação, tanto na atualização da tabela de página em relação a referenciação da memória física (Memória Principal). Assim, tudo que eu precisava fazer era chamar as funções quebradas para cada operação necessária.

Utilizar esse método se demonstrou extremamente útil e um gigantesco facilitador ao longo da compreensão do código, já que eu não precisava procurar revisar o que estava acontecendo em determinada parte do código já que a própria função já me dizia a operação realizada. Isto fez com que o arquivo principal se tornasse extremamente limpo e nada poluído.

As funções estão localizadas na pasta “utils”, suas assinaturas no arquivo “functions.h”, e cada header ja tem em seu nome quais as funções que la se situam para deixar mais organizado e de mais fácil leitura e procura. Junto disso, dentro desse arquivo do “functions.h”, as structs estão sendo inicializadas no tamanho das tabelas de páginas, onde a struct da tabela de página possui o tamanho 256, pois representa os 256 bytes. Enquanto, a struct da memória física possui 256 Bytes para cada página, multiplicado pela quantidade de páginas utilizadas no programa, que é de 128, logo a memória física tem tamanho de 256 * 128.

3. Função principal:

A função principal do código, ou seja onde o código de fato ocorre, está localizado no arquivo main.c , nele está contida toda a lógica operacional para a simulação da virtualização de memória.

Primeiramente é chamada as funções que serão utilizadas no código, então incluímos as bibliotecas de C junto dos headers criados dentro da pasta Utils.

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#include "utils/structs.h"
#include "utils/functions.h"
#include "utils/save_file.h"
#include "utils/fifo.h"
#include "utils/lru.h"
```

Logo em seguida é declarado as variáveis globais que serão utilizadas ao longo do código, essas variáveis foram escolhidas para serem globais devido ao fato de não precisarem ser atualizadas constantemente ao longo do código, visto que elas teriam apenas uma função única.

```c
int translated_addresses = 0;
int page_fault_total = 0;
float page_fault_rate= 0.0;

int endereco_fisico = -1;
int frame_substituicao = 0;
int ciclo = 0;
```

Agora nós criamos a nossa função main, onde ocorrerá a toda a lógica operacional por trás do código. Junto dela nós passamos como parâmetro o “argc” e o “argv” que serão utilizados para pegar os comandos do usuário pelo terminal.

Após isso eu crio as variáveis que serão utilizadas ao longo do código e terão suas modificações feitas.

```c
int main (int argc, char *argv[]) {

    // Argumentos da linha de comando do terminal
    char *command_line_file = argv[1];
    char *command_line_table = argv[2];
    char *command_line_tlb = argv[3];

    // Vaiaveis
    int endereco_logico, valor = 0;
    int page_number = 0, offset = 0;
    int frame_atual = 0;

    // String Linha do arquivo addresses.txt
    char linha[100];

    // Arquivos
    FILE *addresses_file;
    FILE *backing_store_file;
```

Agora que nós recebemos os parâmetros do usuário através da linha de comando do terminal, daremos início aos tratamentos de erro, primeiramente devemos saber se a quantidade de argumentos passadas pelo usuário está de acordo com o proposto.

```c
    if(argc < 3) {
        printf("ERROR! YOU HAVE NOT GIVEN ENOUGH ARGUMENTS! PLEASE ENTER 4 ARGUMENTS.");
        return 0;
    }

    if (argc >= 4) {
        printf("ERROR! YOU HAVE GIVEN MORE ARGUMENTS THAN NECESSARY! PLEASE ENTER 4 ARGUMENTS.");
        return 0;
    }
```

Caso o usuário tenha passado nos testes nós iremos criar o espaço que será utilizado para a memória física, nós iremos passar por um loop, 256 vezes, onde chamaremos a struct da memória física e a variável que contém a página, dentro dessa página iremos alocar na memória um inteiro de 8 bits, 256 vezes, pois as páginas irão conter a instrução e dentro de cada bloco de página da memória física, nós temos 256 bytes, ou seja cada espaço dos 256 bytes da memória física terão uma linha de 1 byte.

```c
    // Cria o espaço da memória física, cada bloco de página receber um espaço de 8Bits 256 vezes.
    for(int index = 0; index < 256; index++) {
        memoria_fisica[index].pagina = (int8_t*)malloc(sizeof(int8_t) * 256);
    }
```

Após fazer essa alocação de memória, eu vou checar se os arquivos passados conseguem ser abertos ou se os mesmos existem.

```
53      // Tenta abrir o arquivo addresses.txt e trata o erro caso não consigo encontrar o arquivo ou abrir ele
54      addresses_file = fopen(command_line_file, "r");
55      if(check_adresses_file(command_line_file)) {
56          return 0;
57      }
58
59      // Tenta abrir o arquivo BACKING_STORE.txt e trata o erro caso não consigo encontrar o arquivo ou abrir ele
60      backing_store_file = fopen("BACKING_STORE.bin", "r");
61      if (check_backing_store_file()) {
62          return 0;
63      }
64      fclose(backing_store_file);
```

Essas funções utilizadas para fazer a chegam dos arquivos estão dentro da pasta “utils” e no arquivo “functions.h”

```
40  /*################################ FUNCTIONS ################################*/
41  int check_adresses_file(char* file) {
42      FILE * addresses_file;
43      addresses_file = fopen(file, "r");
44
45      if(addresses_file == NULL) {
46          printf("ERROR! ADDRESS FILE COULD NOT BE FOUND OR COULD NOT BE OPENED");
47          fclose(addresses_file);
48          return 1;
49      }
50
51      return 0;
52  }
53
54  int check_backing_store_file() {
55      FILE * backing_store_file;
56      backing_store_file = fopen("BACKING_STORE.bin", "r");
57
58      if(backing_store_file == NULL) {
59          printf("ERROR! BACKING_STORE FILE COULD NOT BE FOUND OR COULD NOT BE OPENED");
60          fclose(backing_store_file);
61          return 1;
62      }
63
64      return 0;
65  }
```

Agora que foi checado que de fato os arquivos conseguiram ser abertos ou eles existem, nós iremos começar a ler o arquivo de endereços lógicos, usando o a função de C fgets() e salvando a linha no array “linha”, logo em seguida convertendo essa String em um inteiro e salvando na variável, “endereco_logico”, a partir daqui todas as operações ocorreram dentro de um loop while com o “fgets()” até que o programa chegue ao fim do arquivo.

```
65
66      while (fgets(linha, 100, addresses_file) != NULL) {
67
68          /* Pega a linha retirada do addresses.txt e converte a String para um Inteiro
69          Referência: https://www.tutorialspoint.com/c_standard_library/c_function_atoi.htm*/
70          endereco_logico = atoi(linha);
```

Em seguida, nós incrementamos a variável “translated_addresses”, para contabilizar a quantidade de vezes que um endereço lógico foi traduzido. Agora nós iremos traduzir o endereço lógico utilizando a operação de bit masking (referências nos comentários do código, tanto de funcionamento quanto de onde fora tirado a ideia).

```
72          translated_addresses++;
73
74          /* Faz a operação de shift binário pra esquerda e compara com uma porta AND
75          Apagando todos os bits que forem inúteis, mantendo todos os 1 e 0 do necessário
76          Referência: Operating System Concepts, Silberchatz, P-51; Programming Projects - Specific
77          Referência: https://www.geeksforgeeks.org/left-shift-right-shift-operators-c-cpp/
78          Referência: https://www.tutorialspoint.com/cprogramming/c_operators.htm             */
79          offset = (((1 << 8) - 1) & endereco_logico);
80          page_number = (((1 << 8) - 1) & (endereco_logico >> 8));
```

Após a tradução do endereço lógico, como mostrado acima, nós salvamos os bits necessários nas variáveis do “offset” e “page_number”.
Agora nesta etapa nós iremos checar se aconteceu um “Page Fault” ou seja, se está página a qual foi traduzida ja está carregada na tabela de páginas. Utilizando a função “page_fault_check”, onde passamos como parametro a página que queremos checar, e a função irá retornar verdadeiro se há de fato um “page fault” e falso se não houver.

```
82
83          /*################################ PAGE FAULT ################################*/
84          if (page_fault_check(page_number)) {
```

```
67  int page_fault_check(int page_number) {
68      if (tabela_pagina[page_number].bit_validacao == false) {
69          return 1;
70      }
71
72      return 0;
73  }
```

Caso aconteça um “page fault”, nós teremos de carregar a página na tabela de página, então nós entramos dentro da condicional, incrementamos a variável “page_fault_total” que será usada para escrever no documento gerado ao final da realização do código.
Agora nós iremos abrir o arquivo “BACKING_STORE.bin” que é onde estão as páginas e suas instruções em binário.
Após abrir o arquivo nós iremos passar o ponteiro deste arquivo para a variável “backing_store_file”, em seguida daremos início ao procedimento de carregamento de página.

Utilzamos a função de C “fseek()” (Referências dentro do comentário do código, tanto de funcionamento quanto de onde fora tirado a ideia), para mover o ponteiro do “backing_store_file” na posição da página atual, vezes 256 que é a quantidade de instruções contidas em cada página.

```
86          page_fault_total++;
87          backing_store_file = fopen("BACKING_STORE.bin", "rb");
88
89          /* Move o ponteiro para o BACKING_STORE.bin na posição especificada pelo número da página
90          Para escrevermos na memória física a instrução dessa posição do BACKING_STORE.bin
91          Referência: Operating System Concepts, Silberchatz, P-53; Programming Projects - Handling Page Faults
92          Referência: https://acervolima.com/fseek-em-c-c-com-exemplo/                    */
93          fseek(backing_store_file, page_number * 256, 0);
```

Após mover o ponteiro, precisamos carregar a página na tabela de páginas, porém antes devemos checar se a tabela de páginas já está cheia ou não, usando a função “tabela_pagina_cheia”, passando como parametro o frame atual da memória física, dentro de uma condicional, caso a tabela não estiver cheia, nós prosseguimos.

```
94
95              if(!(tabela_pagina_cheia(frame_atual))) {
96
```

```
74
75  int tabela_pagina_cheia(int frame_atual) {
76
77      if(frame_atual < 128) {
78          return 0;
79      }
80
81      return 1;
82  }
```

Se a tabela de páginas não estiver cheia nós começaremos a popular a memória principal, usando a função "colca_instrucao_memoria_fisica", passando de parâmetro o frame atual e o ponteiro para o arquivo “.bin”.

Dentro desta função nós remos utilizar a função da biblioteca de C “fread()” (Referências dentro do comentário do código, tanto de funcionamento quanto de onde fora tirado a ideia), através dela nós iremos dentro do arquivo “.bin” pegar a instrução (lembre-se que antes no código nós movemos o ponteiro do arquivo para a posição que queríamos) da página, no tamanho de 1 byte, e iremos alocar 256 vezes na struct da memória física, naquele frame atual (como essa seria a primeira interação, será no frame 0, naquela página do frame.

```
97              /* Armazena na memória física, naquele frame atual e seu respectivo numero de página,
98              A Instrução do arquivo BACKING_STORE.bin. Usando o fread() dentro da função. (veja o functions.h)
99              O qual acima na função, fseek(), teve seu ponteiro mudado para apontar para a posição que queríamos.
00              1 Byte (8Bits), 256 vezes. Isso no caso é o tamanho que temos da página do arquivo BACKING_STORE.bin
01              Referência: Operating System Concepts, Silberchatz, P-53; Programming Projects - Handling Page Faults
02              Referência: https://terminalroot.com.br/2014/10/exemplos-de-funcoes-fread-fwrite-remove.html
03              coloca_instrucao_memoria_fisica(frame_atual, backing_store_file);
```

```
83
84  void coloca_instrucao_memoria_fisica(int frame_atual, FILE* backing_store_file) {
85      fread(memoria_fisica[frame_atual].pagina, 1, 256, backing_store_file);
86  }
```

Agora a struct da memória física, naquele frame atual, terá a sua variável de número de página alterada para o número da página que que traduzimos do endereço lógico.

Depois disso iremos atualizar a tabela de páginas, usando a função “atualiza_tabela_pagina”, passando como parâmetros o número da página atual, o frame atual e o ciclo que essa página se encontra, iremos atualizar a tabela, a struct da tabela de páginas naquela página atual, na variável “numero_frame” vai receber o frame atual (o frame que a página referência), logo após nós dizemos que o bit daquela página está válido, ou seja ela foi carregada na memória, e por fim salvamos o ciclo em que esta operação aconteceu na variavel “quando_usado”.

E por fim, incrementamos a variável “frame_atual”, para dizer que agora iremos para o próximo frame.

```
105             memoria_fisica[frame_atual].page_number = page_number;
106             atualiza_tabela_pagina(page_number, frame_atual, ciclo);
107             frame_atual++;
108         }
```

```
88  void atualiza_tabela_pagina(int page_number, int frame_atual, int ciclo) {
89      tabela_pagina[page_number].numero_frame = frame_atual;
90      tabela_pagina[page_number].bit_validacao = true;
91      tabela_pagina[page_number].quando_usado = ciclo;
92  }
```

Após isso nós fechamos o arquivo do “BACKING_STORE.bin”

```
127
128             fclose(backing_store_file);
129         }
```

Agora com a operação feita, nós iremos criar o endereço físico, usando a função “cria_endereco_fisico”, passando de parametro o número da página atual, o offset e a variavel global “endereco_fisico).

Dentro da função nós realizamos o algoritmos de criação de endereço físico, onde ele irá receber o número do frame da página atual da tabela de páginas, e logo em seguida o binário dele será multiplicado por 256 que é o tamanho do frame, e iremos somar isso com o offset.

```
135
136             cria_endereco_fisico(page_number, offset, &endereco_fisico);
```

```
93
94  void cria_endereco_fisico(int page_number, int offset, int *endereco_fisico) {
95      *endereco_fisico = tabela_pagina[page_number].numero_frame;
96      *endereco_fisico = (((*endereco_fisico) * 256) + offset);
97  }
```

Agora iremos pegar o valor, que no caso consideremos ser a instrução, para realizar essa operação, basta irmos na memória física, dentro da tabela de páginas, naquele número de página atual, e pegar o número do frame que ele referência, após isso, vamos na página disso tudo, com o index do offset, que é onde estará guardado a instrução da página, e agora salvamos isso dentro da variável “valor”.

```
137         valor = memoria_fisica[tabela_pagina[page_number].numero_frame].pagina[offset];
```

Agora iremos escrever essas variáveis no arquivo que iremos gerar, e o chamaremos de “correct.txt”, usando a função “save_addresses_value”, passando como parâmetro o endereço lógico traduzido, o endereço físico e o valor.

Dentro dessa função nós abriremos o arquivo e iremos escrever nele os parâmetros.

```
138         save_addresses_value(endereco_logico, endereco_fisico, valor);
```

```
5   // Salva o endereço virtual, fisico e instrução no correct.txt
6   void save_addresses_value(int endereco_virtual, int endereco_fisico, int valor) {
7
8       FILE *file;
9       file = fopen("correct.txt", "a");
10      fprintf(file,"Virtual address: %d Physical address: %d Value: %d\n", endereco_virtual, endereco_fisico, valor);
11      fclose(file);
12  }
```

Agora que acabamos as operações principais, nós incrementamos a variável “ciclos” que será utilizada para o método do LRU (Least Recently Used).

```
139         ciclo++;
```

Pronto, essa operação irá se repetir até que determinadas condições aconteçam, então vamos por ordem agora de cada condição.

Ainda dentro do loop “while()” onde estamos pegando a linha do arquivo de endereços lógicos, vamos para a condição onde ocorre o “page fault”, caso em algum momento aquela página se repita e já tenha sido carregada na tabela de páginas, então deveremos ir para a parte do código que lida com essa condição onde não ocorre um “page fault”.

Dentro dela nós iremos apenas atualizar a struct da tabela de páginas, naquele número de página atual, na variável “quando_usado”, e atualizar o ciclo a qual ele foi realizado.

```
131         /*############################### SEM PAGE FAULT ################################*/
132         else {
133             tabela_pagina[page_number].quando_usado = ciclo;
134         }
```

Agora vamos para os métodos de organização, dentro da condicional do “page fault” nós fazemos uma checagem se a tabela de páginas está cheia, caso a tabela ja tenha sido completamente carregada, então nós iremos realizar a organização da memória física baseado no comando dado pelo usuário na linha de comando do terminal, nós temos duas opções, ou o FIFO (First In First Out) ou o LRU (Least Recently Used).

Caso o usuário tenha escolhido o método fifo nos entraremos na condicional dele
E chamaremos a função “fifo”, que tem como parâmetros, o ponteiro do arquivo “BACKING_STORE.bin”, o frame que será substituido, o número da página atual e o ciclo que a operação se encontra.

```
110             else {
111                 /*################################ FIFO ################################*/
112                 if(!(strcmp(command_line_table, "fifo"))) {
113                     fifo(backing_store_file, frame_substituicao, page_number, ciclo);
```

Agora vamos explicar como a função do “fifo” funciona dentro do código, já que o fifo é literalmente, o primeiro que entra é o primeiro que sai, tudo que precisamos fazer é com que ele passe pela próxima página dentro da tabela de página, e substituir a mesma.

Primeiro criamos um loop que percorrerá a tabela de páginas inteira, ou seja um loop de 256 interações, agora checamos se aquele página está de fato carregada na tabela de páginas para tratamento de erro, utilizando a função “page_fault_check”, e negando o resultado da mesma, caso a página esteja carregada, precisamos saber se naquela interação do loop, o número da página que estamos é o mesmo do frame que queremos substituir, usando a função “frame_number_same_as_substitute”, passando como parametro o frame a ser substituido e o número da página atual.

Dentro da função iremos comparar o número do frame naquele número de página dentro da tabela de página com o frame a ser substituido, caso a condição seja verdadeira a função retorna 1, caso seja falsa ela retorna 0.

Agora que todas as condições para realizar o “fifo” foram checadas, vamos de fato fazer o “fifo” em si. Usando a função “fifo_change_table”, passando os parâmetros do ponteiro do arquivo “BACKING_STORE.bin”, o número da página que estamos atualmente no programa principal, o index da interação do loop dentro da função do “fifo”, o frame a ser substituido e o ciclo.

```c
void fifo(FILE* backing_store_file, int frame_substituicao, int page_number_fifo, int ciclo) {

    for (int page_number = 0; page_number < 256; page_number++) {
        if (!(page_fault_check(page_number))) {
            if (frame_number_same_as_substitute(frame_substituicao, page_number)) {
                fifo_change_page_table(backing_store_file, page_number_fifo, page_number, frame_substituicao, ciclo);
                break;
            }
        }
    }
}
```

```c

int frame_number_same_as_substitute(int frame_substituicao, int page_number) {
    if (tabela_pagina[page_number].numero_frame == frame_substituicao) {
        return 1;
    }

    return 0;
}
```

Agora para mudar a tabela de páginas usando o “fifo” nós iremos trocar o bit de validação, da tabela de página, no index do loop, para falso (já que iremos trocar a página precisamos dizer que aquela página irá sair, logo, não estará mais carregada na tabela de páginas).

Após isso usamos a função “fread()” na struct da memória física com o frame de substituição como index.

E por fim chamamos a função de atualizar a tabela de página, “atualiza_tabela_pagina”, passando como parâmetros o número da página na função principal, o frame a ser substituído, e o ciclo.

```
20
21  void fifo_change_page_table(FILE* backing_store_file, int page_number_fifo, int page_number, int frame_substituicao, int ciclo) {
22      tabela_pagina[page_number].bit_validacao = false;
23      fread(memoria_fisica[frame_substituicao].pagina, 1, 256, backing_store_file);
24      atualiza_tabela_pagina(page_number_fifo, frame_substituicao, ciclo);
25  }
```

Após realizar esta operação do “fifo”, nós iremos checar se ele já foi realizado 128 vezes, que é a quantidade de páginas na tabela de páginas, caso já tenha sido preenchido, nós iremos resetar a variável, “frame_substituicao” para que o “fifo” possa continuar a seguir sua lógica.

```
115
116                     // Reseta o fifo quando estiver cheio
117                     if (frame_substituicao == 128) {
118                         frame_substituicao = 0;
119                     }
120                 }
```

Agora, caso o usuário tenha escolhido o método do LRU (Least Recently Used) ao invés do FIFO (First In First Out), nós iremos entrar na condição do LRU e realizar a sua operação.

Chamando a função do “lru()”, passando como parâmetros, a variável “backing_store_file”, o número da página atual e o ciclo que nós estamos.

```
121
122                 /*################################ LRU ################################*/
123                 else if(!(strcmp(command_line_table, "lru"))) {
124                     lru(backing_store_file, page_number, ciclo);
125                 }
126             }
```

Dentro da função do “lru”, nós criaremos 3 variáveis locais, “page_number_index”, “tempo_atual”, “frame”. As quais serão utilizadas exclusivamente dentro da função do “lru”.

Já que a lógica que o “LRU” segue é que devemos atualizar a página que está a mais tempo sem ser usada, criei a variável “ciclo” que conta qual o ciclo que a página foi carregada na memória principal, desse jeito basta eu pegar a página que contém o menor ciclo (ou seja a que ficou mais tempo sem ser usada) e através disso eu já sei qual página precisa ser atualizada.

Eu crio um loop de 256 interações, para passar em todas as páginas da tabela de páginas, igualmente ao “fifo” checo se o bit daquela página está valido usando a negação da função “page_fault_check”, e após isso crio uma condicional, para saber se a página da tabela de página naquela página atual (página do programa principal) na variável que tem o ciclo que ela foi utilizada guardada, é menor que o ciclo que estamos agora.

Por exemplo digamos que não existe nenhuma página repetida que foi carregada na tabela de páginas, podemos então assumir que todas as páginas terão um ciclo sequencial, ou seja, da primeira página à última, 1, 2, 3, 4, 5…128

Por essa lógica quando carregarmos toda a tabela de página iremos incrementar a variável do ciclo novamente, tendo como valor 129, então todas as páginas a seguir terão um ciclo menor que 129.

Caso a condição do loop seja verdadeira, eu armazeno o ciclo daquela página na variável “tempo_atual”, e o index da interação do loop na variável “page_number_index”

```c
void lru(FILE* backing_store_file, int page_number_lru, int ciclo) {
    int page_number_index = 0;
    int tempo_atual = ciclo;
    int frame = 0;

    for (int page_number = 0; page_number < 256; page_number++) {
        if (!(page_fault_check(page_number))) {
            if(tempo_menor_que_ciclo(page_number, ciclo)) {
                tempo_atual = tabela_pagina[page_number].quando_usado;
                page_number_index = page_number;
            }
        }
    }
```

```
19  int tempo_menor_que_ciclo(int page_number, int ciclo) {
20      if(tabela_pagina[page_number].quando_usado < ciclo) {
21          return 1;
22      }
23
24      return 0;
25  }
```

Quando o loop acabar teremos pego a página com o menor ciclo, agora iremos trocar a tabela de páginas seguindo a mesma lógica na função do “fifo”. Usando a função “lru_change_table”, passando como parametros o “backing_store_file”, a variável “frame”, o número da página atual no programa principal, o ciclo do programa principal, e o index da interação do loop do “lru”.

```
15      lru_change_table(backing_store_file, frame, page_number_lru, ciclo, page_number_index);
```

Dentro dessa função nós atualizamos a variável do frame para receber o número do frame da tabela de páginas naquela página da interação do loop do “lru”. Depois nós trocamos o bit de validação daquela página do loop para falso. Em seguida chamamos a função “coloca_instrucao_memoria_fisica”, passando como parâmetros, a variável “frame”, e o “backing_store_file”. E por último nós atualizamos a referência na tabela de página chamando a função “atualiza_tabela_pagina”, passando como parâmetro, o número da página no programa principal, o frame, e o ciclo do programa principal.

```
26
27  void lru_change_table(FILE* backing_store_file, int frame, int page_number_lru, int ciclo, int page_number_index) {
28      frame = tabela_pagina[page_number_index].numero_frame;
29      tabela_pagina[page_number_index].bit_validacao = false;
30      coloca_instrucao_memoria_fisica(frame, backing_store_file);
31
32      atualiza_tabela_pagina(page_number_lru, frame, ciclo);
33  }
```

E pronto, essas são as operações realizadas caso as condições ditas acima sejam verdadeiras.

Agora para finalizar o programa, após todas essas operações serem feitas, ou seja até terminarmos de ler o arquivo contendo os endereços lógicos, nós iremos escrever no arquivo “correct.txt” a quantidade de endereços traduzidos, a quantidade de “page faults” que ocorreram ao longo do programa, e por fim, iremos fechar o arquivo que contém os endereços lógicos e encerrar o programa.

```
141
142         save_translated_addresses(translated_addresses);
143         save_page_faults(page_fault_total);
144         fclose(addresses_file);
145
146         return 0;
147     }
```

```
14  // Salva a quantidade de arquivos traduzidos no correct.txt
15  void save_translated_addresses(int translated_addresses) {
16
17      FILE *file;
18      file = fopen("correct.txt", "a");
19      fprintf(file, "Number of Translated Addresses  = %d\n", translated_addresses);
20      fclose(file);
21  }
22
23
24  // Salva a quantidaded de page faults e a taxa de page faults no correct.txt
25  void save_page_faults(int page_fault_total) {
26
27      FILE *file;
28      file = fopen("correct.txt", "a");
29
30      fprintf(file, "Page Faults = %d\n", page_fault_total);
31
32      double page_fault_rate = (double)page_fault_total / 1000.0;
33      fprintf(file, "Page Fault Rate = %.3f", page_fault_rate);
34      fclose(file);
35  }
```

Perceba que a operação do “page fault rate” é feita dentro da função que escreve a quantidade de “page faults”.

4. Dificuldades:

4.1 LRU

Algumas das dificuldades que tive ao longo do programa foram, principalmente e acima de tudo, o LRU, não me entenda errado, a teoria por trás do LRU e do FIFO são extremamente simplórias e muito fáceis de se entender, mas o programa em si é tão grande e com tantas operações que precisam ser feitas e checadas ao mesmo tempo, que em certo momento se você não organizar direto o código, você se perde muto facilmente, por isso fiz o código movido a funções que já tem no nome o que fazem.

A dificuldade em si que tive com o LRU foi mais devido ao fato de que eu tentei durante umas 5h implementar algum método que tomasse controle da ordem em que as coisas ocorriam na tabela de páginas e que eu pudesse usar como método de checagem, e por mais que eu tenha conseguido fazer com que a função me entregasse o valor, no caso a instrução, correto, ainda assim não consegui fazer com que o programa desse a quantidade de page faults correto usando o LRU, e não consegui descobrir o porquê.

4.2 Tempo

Seguinte, vamos deixar uma coisa muito bem clara aqui, eu irei comentar mais isso na parte do feedback mas por agora irei apenas passar a dificuldade, após estudar muito sobre a teoria por trás da virtualização de memória, seja no livro do “silberchartz” ou em vídeos de youtube ou perguntando aos meus colegas de classe, consegui perceber que na realidade a atividade não é tão difícil assim, muito pelo contrário, a implementação é fácil, porém longa e demorada, demorada por ser longa e quanto maior o tamanho mais testes a se fazer e mais vezes tem de se checar o que esta de errado, e longa porque várias coisas ocorrem ao longo do código.

Meu problema em relação ao tempo se deu ao fato de que no momento da atividade eu estava sobrecarregado de atividades **EXTRACURRICULARES,** no dado momento eu estava atuando como monitor do NExT até o dia 06/06, junto de estar estagiando dentro do **CESAR,** e ainda participar do **BOOTCAMP**, as tasks que tinha do estágio me custavam normalmente de 1h a 2h diárias, a monitoria me consumia 3h a noite, e o BOOTCAMP foi algo periódico que no total me consumiu apenas 5h.

Tendo isso em mente que tinhamos 3 semanas para a entrega eu dei prioridade maior a esses 3 fatores junto das outras atividades que vieram das outras cadeiras que tinham prazo menor de entrega. Eu pessoalmente acabei começando a realizar a implementação no dia 02/06 e **FOI POR ESCOLHA MINHA,** deixo bem claro aqui que eu tinha plena noção das consequências que trariam começar tão tarde e aceito as mesmas sem problema algum. Agora mais uma vez gostaria de deixar claro que, durante esses dias que eu fiz a implementação eu percebi que é sim viável fazer em 3 semanas talvez até seja uma tempo leniente demais, se você for um aluno que põem a matéria em dia e esta constantemente estudando pelo livro dado no primeiro dia de aula, você consegue facilmente acompanhar tanto a matéria quanto essa implementação. Então eu vejo que foi sim de fato passado tempo mais que o suficiente para realizar a implementação. Ponto final sem discussões. Se alguém reclamar que não houve tempo suficiente para fazer a atividade tendo dito menos ou mais responsabilidades as quais eu tive neste dado momento, eu pessoalmente diria que é incompetência e falta de disciplina e organização da pessoa em si.

4.3 TLB

Como já dito em cima eu tive problemas com o tempo, e por isso acabei conseguindo fazer apenas em torno de 70% do código (como eu fiz o código movido a função eu considero que a TLB ja seria uma pequena parte do código já que bastaria eu passar as funções do FIFO e do LRU para dentro da TLB e o resto seria automático).

Eu ainda assim tentei fazer a TLB porém sem sucesso ja que eu só consegui tentar a fazer a TLB 2 dias antes da entrega. (ainda por cima tive alguns outros problemas em relação a decisões que tomei nos arquivos, mais sobre isso na aba de feedback)

5 FEEDBACK:

Em relação ao feedback tem algumas coisas que eu queria passar tanto da atividade como um todo, de como eu me senti realizando a mesma, e de alguns erros que cometi ao longo do caminha que acho importante compartilhar com o senhor para que possa avisar a próxima leva de alunos e evitar que os mesmos cometam os erros que eu tive.

5.1 Diversão & Evolução

No sábado dia 03/06, eu estava completamente exaurido mentalmente das atividades que foram passadas pelas outras cadeiras junto das minhas atividades complementares. Porém fiquei o dia todo fazendo a implementação, e teve um momento, das 22h às 01:30h, em que eu estava fazendo o FIFO e comparando o que eu fazia com o livro do “Silberchartz” e em dado momento meu foco era tão extremo que eu estava 100% centralizado na função do FIFO, e tive uma certa realização, já que a TLB vai usar o FIFO e o LRU, porque não criar funções separadas para cada um e daí eu só chamaria eles dentro da TLB? Logo em seguida me veio um flash de inspiração, já que o código inteiro vai ter que ser várias etapas similares tanto no FIFO quanto no LRU dentro e fora da TLB, porque só não quebrar o código inteiro com essas repetições dentro de funções e deixar os nomes intuitivos? Dessa maneira que poderia simplesmente bater o olho no código e saber tudo que acontecia.

Após eu ter feito a refatoração do código ele se tornou muito mais legível, e organizado, então eu queria deixar claro aqui que por conta da sua atividade eu vi uma maneira completamente diferente de fazer as coisas programando, pensando em escalabilidade, coisa que eu nunca tinha sentido ou visto antes nas cadeiras anteriores até então. Obrigado por me fazer evoluir como programador nessa atividade e além disso no mesmo momento eu me sentia extremamente alegre, eu estava gostando de cada momento que fazia a atividade, por mais que tiveram horas que eu queria arrancar meus cabelos e tentar entender o porque tal coisa não funcionava, eu me diverti muito fazendo essa implementação, e genuinamente gostei muito dela e achei desafiadora, foi um desafio que até então não tínhamos dito nos períodos passados.

Por conta de nas cadeiras dos períodos anteriores nós alunos estarmos focados demais na linguagem, na sintaxe, no processo mecânico nos exercícios passados a chama que eu tinha estava lentamente se apagando pois era tudo muito mecânico, cade o desafio, cade a teoria, cade a experiência de ficar horas intermináveis martelando a cabeça na parede até resolver um problema? Foi justamente por isso que eu escolhi a área de computação, não pra ficar repetindo a mesma coisa sempre de maneira automática, mas sim pra ser desafiado com uma lógica que eu não conheço e tentar botar aquilo em prática.

Então por isso, eu agradeço por ter passado essa atividade pra gente, eu voltei a pegar fogo como quando eu tinha entrado na faculdade, e me lembrei por que eu gosto tanto da área e dos desafios que ela trazem. Se possível, continue no mesmo esquema de aula com os próximos alunos, por mais exigente que seja a cadeira, é inegável que ela traz muito evolução para nós como programadores e como pessoas.

5.2 Erros que cometi

Alguns erros que cometi ao longo da implementação que gostaria de repassar ao senhor e já lhe atentar, e se possível, avisar a próxima turma de alunos a não cometerem os mesmos erros que eu cometi.

Então, por mais óbvio que seja para algumas pessoas, ainda mais considerando que todas as tutorias que fizemos ao longo do semestre foram postadas no GITHUB, eu cometi a gafe de não ficar comitando as alterações que fazia no código, nem de criar um repositório para guardar os meus arquivos, todos eles estavam locais.

E em dado momento, quando decidi mover alguns headers e organizar melhor o código, de alguma forma eu quebrei a função do LRU e não sabia o porque ele tinha quebrado, e não havia maneira de eu voltar atrás no código e ver que alterações foram feitas. Então peço encarecidamente, se possivel avise os alunos ou até mesmo escreva no PDF da atividade, para eles criarem um repositório e a cada alteração no código feita com sucesso, eles comitem isso no repositório para não perder as alterações. E claro, se o senhor quiser usar de argumento também, você poderia cobrar dos alunos esses comitts frequentes e checar se a pessoa de fato estava realizando a atividade ou não ao longo dos dias, assim você teria um controle maior de como as pessoas estavam andando com a atividade e ver se de fato estavam falando a verdade, baseado nos commits do repositório, enfim, só uma ideia fica a sua decisão.

E como segundo erro, Subestimar a atividade e o tempo passado, ja tinha em mente que ela seria longa e mais uma vez repito, eu tinha ciência das consequências que trariam eu dar prioridade aos meus afazeres extracurriculares ao invés da atividade.

Já sabendo disso, eu acabei não conseguindo terminar tudo a tempo infelizmente, mas é o que é, foi uma escolha minha e cabe a eu lidar com isso. Eu não me arrependo de não ter começado antes, eu penso que a decisão que tomei naquele dado momento era a mais apropriada para o contexto, e no fim paguei o esperado por isso.

Se servir de algo mesmo não terminando tudo, só a diversão que tive fazendo a atividade mais a evolução que tive estudando sobre a mesma e a implementando, ja foram o suficiente para mim, então de toda forma estou feliz com o resultado. =)