Essa implementação simula uma memória virtual. O código consiste em pegar instruções que estão dentro de um arquivo de texto, achar elas na memória virtual (arquivo chamado BACKING STORE), e a partir da memória virtual escrever em um segundo arquivo de texto chamado "correct.txt" as informações contidas nas instruções.

Foram criadas 3 estruturas para o funcionamento do sistema, sendo elas a memória física que é a “memoryStack” , a tabela de páginas que é a “page_table_args” e a TLB. Cada uma dessas estruturas será um array e por isso a criação de 3 Structs, pois a page table terá que guardar a página que será lida na instrução, um bit de sinal para representar se aquela página está alocada na memória e um "relógio" que irá marcar em qual tempo a página foi alocada na memória(implementação para o LRU), a “memoryStack” por outro lado foi implementada de forma diferente já que essencialmente ela é uma matriz de 128 por 256, entretanto eu preferi implementar ela como uma Struct e trabalhar com essa estrutura como linhas em um array e por fim a TLB que guarda o número da página e um outro “relógio”. Antes de começar o loop principal foi necessário passar em todas as posições da TLB para colocarmos o valor inicial como -1 pois as posições começam com o valor padrão que é zero, e isso gera erros ao longo da execução do programa.

No início da função main é feito um tratamento de erro para verificar se as entradas do argc e argv correspondem com as requisitadas, a partir desse ponto que o código começa de fato a trabalhar, primeiramente é usada uma função chamada “fgetpos()” que serve para salvar a posição atual do ponteiro para leitura do arquivo de texto que contém as instruções para o funcionamento do código. Um loop “while” é aberto e só irá terminar quando ele encontrar o fim do arquivo, para isso foi utilizada uma macro chamada de EOF(End Of File) assim que ele abrir o arquivo de texto ele irá salvar aquelas informações em uma variável chamada decimal que irá passar por 2 outras funções para podermos extrair o “offset” e a “page” desse valor, as funções são chamadas de “getPage” e “getOffset” ambas funcionam de forma semelhante utilizando de “shift operators”, que nada mais é que um deslocamento síncrono binário, já que o “offset” e a “page” são respectivamente os 8 bits menos significativos e os 8 bits mais significativos do valor da instrução.

Após recebermos os valores extraídos da instrução começamos a procurar na TLB se a “page” já foi alocada previamente a leitura da instrução atual, caso tenha sido alocada foi implementada uma chave que tranca o resto do código, até ocorrer uma nova leitura de instrução, e nos leva diretamente para as linhas 157 onde fica a parte de escrever a posição da memória virtual, física e o valor da instrução; Porém se a “page” não tiver sido encontrada na TLB a procura será efetuada no array chamado de “pageTable” e iremos varrer em busca desta página. Anteriormente foi dito que a estrutura “page_table_args” deveria guardar o número da página porém não foi implementado nenhum argumento contendo essa informação pois foi utilizado o número posicional do array como número da página por essa razão a não será necessário percorrer todo o array e sim apenas irmos direto para a posição de número igual a “page” que foi extraída da variável “decimal”, assim que chegamos nessa posição iremos olhar se o bit de sinal, chamado de “isCreated” está como “NOT_IN_MEMORY” que é uma macro que significa que a página não foi alocada a memória. Considerando que o bit de sinal está

invalido ou seja a página não foi carregada, page fault, temos que alocar um espaço na memória física e em seguida colocarmos toda a página correspondente la, a página está contida num arquivo binário chamado “BACKING_STORE.bin”, cada “page” contém 256 valores, por essa razão para podermos achar a página que nós procuramos teremos que multiplicar o valor da página por 256 e salvar na memória os próximos 256 valores, esses valores serão salvos na Struct de memória física

“memoryStack[posição_na_sequência].bin"

já que nossa posição na sequência foi salva na nossa tabela de páginas no atributo “indexMemory” temos que a nossa posição ficará como:

“**pageTable[page].indexMemory**”

logo teremos que salvar na memória dessa forma:

“**memoryStack[pageTable[page].indexMemory].bin**”

Após salvarmos a página na memória o programa irá utilizar outra função de manipulação de ponteiro, ”fsetpos()”, para fazer com que o ponteiro de leitura retorne para a posição que foi salva na função “fgetpos()” fazendo com que o programa leia novamente a instrução, vasculhe na TLB(não irá achar pois a página só é salva após a sua utilização) e procure na Page Table assim ele irá encontrar e irá escrever no arquivo de texto “correct.txt” as informações da instrução. Contudo a memória física não é grande o suficiente para suportar todas as páginas que o BACKING_STORE possui por isso é necessário utilizar métodos de substituição, FIFO (First In First Out) ou LRU(Least Recently Used), a forma que esses métodos foram implementados foi bastante simples, primeiramente o FIFO. Esses métodos são

```
else if(!strcmp(argv[2], "fifo")){
    lock = 1;
    if (count == SIZE_MEMORY){
        count = 0;
    }
    for(int i = 0; i < SIZE_PAGE_TABLE; i++){
        if(pageTable[i].isCreated == IN_MEMORY && pageTable[i].indexMemory == count){
            pageTable[i].isCreated = NOT_IN_MEMORY; //na pageTable mas não aponta para memoria
            pageTable[i].indexMemory = -1;
        }
    }
    pageTable[page].indexMemory = count;
    count++;
}
```

apenas utilizados quando a memória física não possui nenhum espaço livre para guardar nenhuma nova página.

Primeiramente é feita uma verificação se o argumento escrito durante a chamada do programa é igual a “fifo”, caso seja o programa irá travar a parte do código que efetuava a alocação antes de encher a memória, fazendo com que esse bloco FIFO seja o único utilizado pois iremos utilizar uma variável chamada “count” que era utilizada no bloco anterior. Já que esse método consiste em remover o primeiro que foi colocado apenas iremos igualar a zero o contador sempre que ele chegar ao valor máximo da memória em seguida iremos varrer a tabela de páginas em busca da página que será removida, por exemplo no primeiro caso que a memória estourar iremos varrer a tabela de páginas em busca da página que está alocada na posição 0. Quando essa página for encontrada iremos dizer que a mesma não é mais válida (NOT_IN_MEMORY) e dizer que seu index não aponta para nenhum local de memória válido, nesse caso estamos usando “-1”. Com a antiga página removida iremos agora apenas alocar o espaço que agora está vazio para a nova página.

```
else if(!strcmp(argv[2], "lru")){
    lock = 1;
    int lower = time;
    int value = page;

    for(int i = 0; i < SIZE_PAGE_TABLE; i++){
        if(pageTable[i].isCreated == IN_MEMORY && pageTable[i].uses < lower){
            lower = pageTable[i].uses;
            value = i;
        }
    }
    pageTable[page].indexMemory = pageTable[value].indexMemory;
    pageTable[value].indexMemory = -1;
    pageTable[value].isCreated = NOT_IN_MEMORY;
}
```

O LRU por outro lado remove o item que foi usado por último, o que está sem ser usado por mais tempo, por isso foi implementado um relógio que marca o tempo que a página foi usada. Da mesma forma que foi utilizado uma chave para o FIFO a mesma chave foi utilizada para o LRU, porém foram criadas 2 novas variáveis neste bloco para auxiliarem a substituição das páginas, são elas “lower” e “value”, a variável lower é usada para achar a página que possui o valor de “uses” mais baixo, pois esse atributo é o que salva em qual momento essa pagina foi usada, ao encontrar a menos usada seu valor de posição é salvo na variável ” value” para podermos fazer a substituição de forma adequada, primeiramente igualando os frames(posição no array da memória física) e em seguida colocando dizendo que ela não possui nenhuma alocação na memória física e dizendo que seu bit é invalido.

Após a adição da página na memória ela é adicionada na TLB, que nada mais é que uma forma mais rápida de procurar a página pois ela possui um tamanho muito reduzido, e por possuir um tamanho reduzido a mesma também precisa de métodos de substituição, que são os mesmo que foram citados anteriormente.

```
}else{
    if(!strcmp(argv[3], "fifo")){
        if(countTLB == SIZE_TLB){
            countTLB = 0;
        }
    }
    if(!strcmp(argv[3], "lru")){
        if(countTLB == SIZE_TLB){
            lock2 = 1;
            int lowerTLB = time;
            int position = 0;

            for(int i = 0; i < SIZE_TLB; i++){
                if(tlb[i].uses < lowerTLB){
                    lowerTLB = tlb[i].uses;
                    position = i;
                }
            }
            tlb[position].page = page;
            tlb[position].uses = time;
        }
    }
    if(lock2 == 0){
        tlb[countTLB].page = page;
        tlb[countTLB].uses = time;
        countTLB++;
    }
    pageTable[page].uses = time;
}
```

Para o FIFO a implementação da TLB é extremamente simples já que apenas iremos sobrescrever o conteúdo que estava escrito já que não temos que fazer nenhum tipo de remoção para desligar a página que está no local que irá receber a nova página. Já o RLU é um pouco mais complexo, porém não muito, iremos implementar também um relógio para a TLB, iremos vasculhar o menor tempo possível e iremos sobrescrever o conteúdo que está lá e assim como nos métodos de substituição da memória física iremos utilizar uma chave para bloquearmos o bloco de código que efetua a adição de um item na lista de forma sequenciada.