Projeto de Sistemas de Programação

Nome: Raphael Chypriades Junqueira Amarante
nUSP: 9348856

Sistemas de Programação - PCS 3216
Professor: João José Neto

Escola Politécnica da USP

São Paulo
2018

Índice

1. Introdução

O objetivo deste projeto é elaborar e produzir um sistema para o desenvolvimento de programas em linguagem simbólica absoluta para uma máquina virtual, isto é, em uma máquina hospedeira deve-se criar um ambiente que simule os comportamentos primitivos de outra, como carregar um loader na memória e, a partir deste, carregar e executar demais programas. Isso será feito através de um interpretador de comandos em linguagem de alto nível. Além disso, para que se possa utilizar programas-fonte em linguagem simbólica, também será desenvolvido um montador em linguagem de alto nível.

2. Formato das instruções e pseudo instruções

Para a descrição de códigos, tanto em linguagem de máquina, quanto em linguagem simbólica, é necessário um conjunto de instruções e pseudo-instruções.

Segue abaixo cada instrução em sua forma mnemônica, hexadecimal e uma breve descrição de sua funcionalidade:

Desvio incondicional: JP (hex: 0xxx)

Transfere o fluxo de execução do programa em andamento para o endereço efetivo determinado por seu operando (xxx) e pelo modo (direto ou indireto) de endereçamento, alterando o contador de instruções.

Desvio se zero: JZ (hex: 1xxx)

Testa o acumulador, e no caso de conter o valor nulo, modifica o contador de instrução para transferir o fluxo de execução do programa em andamento para o endereço efetivo determinado por seu operando (xxx) e pelo modo (direto ou indireto) de endereçamento. Caso contrário, executa a instrução seguinte.

Desvio se negativo: JZ (hex: 2xxx)

Similar ao desvio se zero, porém o testa se o acumulador é negativo.

Controle: CN (hex: 3x)

Comando de controle, onde cada operando (x) determina uma instrução:

Halt machine: HM (x = 0)

Paralisa e termina a execução do programa em curso.

Indirect: IN (x = 2)

Coloca a máquina em modo indireto. Caso a instrução seguinte referencie a memória, a referência será indireta, e a máquina retornará ao modo normal. Caso a instrução seguinte não seja de referência à memória, nada acontece, e a máquina também retornará ao modo normal.

Soma: + (hex: 4xxx)

Adiciona ao número inteiro contido no acumulador, o conteúdo numérico (inteiro) obtido no endereço efetivo determinado por seu operando (xxx) e pelo modo (direto ou indireto) de endereçamento, e guarda o resultado no acumulador.

Subtração: - (hex: 5xxx)

Similar à soma, porém subtrai do número inteiro contido no acumulador, o conteúdo numérico (inteiro) obtido no endereço efetivo.

Multiplicação: * (hex: 6xxx)

Similar à soma, porém multiplica número inteiro contido no acumulador com o conteúdo numérico (inteiro) obtido no endereço efetivo.

Divisão: / (hex: 7xxx)

Similar à soma, porém obtem a parte inteira da divisão do número inteiro contido no acumulador pelo conteúdo numérico (inteiro) obtido no endereço efetivo.

Load: LD (hex: 8xxx)

Copia para o acumulador o conteúdo obtido no endereço efetivo determinado por seu operando (xxx) e pelo modo (direto ou indireto) de endereçamento.

Store: MM (hex: 9xxx)

Copia o conteúdo do acumulador para o endereço efetivo determinado por seu operando (xxx) e pelo modo (direto ou indireto) de endereçamento.

Chamada de subrotina: SC (hex: Axxx)

Guarda no endereço efetivo, determinado por seu operando (xxx) e pelo modo (direto ou indireto) de endereçamento, e na posição seguinte, dois bytes contendo um ponteiro para o endereço de retorno, e desvia para a posição seguinte de memória.

Chamada de sistema operacional: OS (hex: Bx)

As 16 combinações do operando (x) ainda estão sem uso, aguardando definição do sistema operacional (fora do escopo do projeto), portanto neste projeto sua função é terminar a execução do programa, promovendo a devolução do controle ao sistema, no caso representado pelo interpretador.

Entrada/saída: IO (hex: Cx)

Cada operando (x) determina uma instrução a ser aplicada no sistema:

Get data: GD (x = 0)

Copia no acumulador o dado lido em arquivo externo e passa a execução à instrução seguinte.

Put data: PD (x = 4)

Copia em arquivo externo o conteúdo do acumulador e passa o controle à instrução seguinte.

Segue abaixo cada pseudo-instrução em sua forma mnemônica, seu operando, quando houver, e uma breve descrição de sua funcionalidade:

Origem: @

Define através de seu operando (hex: xxxx) o endereço inicial do código a ser carregado na memória.

End: #

Demarca o final físico do programa. Não possui operando.

Constante: K

Define através de seu operando (hex: xx) o valor de uma constante.

3. Formato do programa-fonte em linguagem simbólica e do código-objeto absoluto

Para que seja possível a elaboração de um montador e de uma máquina virtual funcionais, é preciso ter um formato padrão tanto do programa-fonte simbólico, quanto do código-objeto absoluto.

Programa-fonte em linguagem simbólica

O programa-fonte sempre deve começar com a pseudo-instrução @ e acabar com a pseudo-instrução #. Entre estes pode vir qualquer sequência de instruções e definições de constantes. Segue abaixo uma amostra genérica que explora o formato de cada tipo de instrução e pseudo-instrução; um exemplo real, no caso um programa que calcula n^2; e na última coluna, as posições de memória referentes ao n^2 (nota-se que labels e pseudo-instruções não ocupam a memória):

Programa genérico:			Programa de n^2:			
@	<endereço inicial>		@	0030		0030
<label 1>			INIC			0030
	JP	<label 1>		LD	UM	0030
	JZ	<label1>		MM	CONT	0032
	JN	<label2>		MM	ÍMPAR	0034
	CN	HM		MM	N2	0036
	CN	IN				
	+	<símbolo 1>	LOOP			0038
	-	<símbolo 1>		LD	CONT	0038
				-	N	003A
<label 2>				JZ	FORA	003C
	*	<símbolo 2>		LD	CONT	003E
	/	<símbolo 2>		+	UM	0040
	LD	<símbolo 3>		MM	CONT	0042
	MM	<símbolo 3>		LD	ÍMPAR	0044
	SC	<label 2>		+	DOIS	0046
	OS	<vazio>		MM	ÍMPAR	0048
	IO	GD		+	N2	004A
	IO	PD		MM	N2	004C
				JP	LOOP	004E
<símbolo 1>	K	<constante 1>	FORA			0050
<símbolo 2>	K	<constante 2>		OS		0050
<símbolo 3>	K	<constante 3>				
			UM	K	01	0051
	#		DOIS	K	02	0052
			ÍMPAR	K	00	0053
			N	K	04	0054
			N2	K	00	0055
			CONT	K	00	0056
				#		

Código-objeto absoluto

O código-objeto absoluto deve ser escrito byte a byte, em hexadecimal, onde os primeiros dois valores informam o endereço inicial e o terceiro informa a quantidade total de bytes, excetuando-se esses 3 primeiros. Segue abaixo programa de n^2 (o arquivo seria apenas uma coluna, mas aqui prezou-se a boa formatação da página):

00	90	54	90	53	B0
30	53	10	56	40	01
27	90	50	80	55	02
80	55	80	53	90	00
51	80	56	40	55	04
90	56	40	52	00	00
56	50	51	90	38	00

4. Funcionalidades do interpretador

No enunciado do projeto, são previstos 4 comandos básicos do interpretador ($DIR, $DEL <nome>, $RUN <nome>, $END). Apenas para fins de se melhor acompanhar o passo-a-passo do sistema foram elaborados mais dois comandos ($MTD <nome>, $MAP). Segue abaixo a explicação de cada comando:

$DIR

Este comando lista todos os arquivos do sistema que estão na pasta do usuário, tanto os que estão disponíveis para execução, quanto aqueles marcados pelo usuário para serem deletados.

$DEL <nome>

Este comando modifica o nome do programa escolhido, adicionando-se o caracter “0” em seu início, para que seja posteriormente deletado quando o sistema for encerrado.

Obs.: este comando não tira diretamente o acesso de tal arquivo do usuário, porém em seu acesso, que demanda o input do nome do arquivo com o “0” o precedendo, o usuário está ciente que esse arquivo está para ser removido.

$RUN <nome>

Este comando primeiramente aciona o loop do motor de eventos da máquina de van Neumann para carregar o **código-objeto absoluto** na memória

através do loader (este, como decisão de projeto, já carregado na memória, simulando o efeito de circuitos de hardware); em seguida, findado o carregamento, força o desvio do contador de instruções para o início de tal programa; e, finalmente, aciona novamente o loop para a execução do programa.
Obs.: Sempre que o loop for acionado ele gera (ou renova) um arquivo nomeado “saida.txt”. Isso se dá para, caso o programa em execução for um dumper, este arquivo ser o recipiente final do conteúdo desejado da memória.

$END

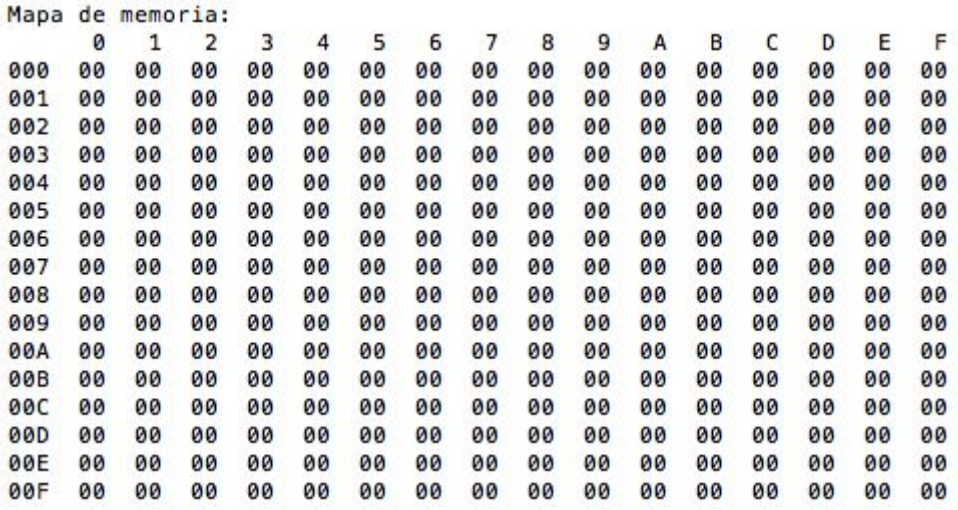
Este comando encerra o sistema e apaga definitivamente os arquivos previamente escolhidos da pasta do usuário.

$MTD <nome>

Este comando poderia ser acoplado ao comando $RUN, mas foi desvinculado deste com o intuito, tanto de uma melhor manutenção do código em linguagem de alto nível, quanto para o melhor acompanhamento e controle dos passos do sistema por parte do usuário. Seu propósito é acionar o montador de 2 passos para traduzir um **programa-fonte em linguagem simbólica**, gerando outro arquivo com o código-objeto absoluto. Este segundo arquivo tem o mesmo nome do que contém o programa-fonte, porém com o apêndice “_hex” (exemplo: n2.txt e n2_hex.txt”).

$MAP

Este comando foi introduzido com o intuito de melhor se visualizar o que está ocorrendo na memória da máquina virtual. Para isso ele exibe na caixa de diálogo com o usuário o conteúdo desta memória virtual no seguinte formato:



5. Lógica do código

Está parte do relatório será dedicada a explicar a implementação do código, desde a definição de bibliotecas e variáveis utilizadas, até a lógica de cada função.

Definições iniciais

```c
// bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

// definicao de constantes
#define MAX 150
#define TAMANHO 1024

// definicao de tipos

// programas a serem deletados
typedef struct deletado {
   char nome[MAX];
   struct deletado* proximo;
} deletado_t;

// simbolos do montador
typedef struct simbolo {
   char nome[10];
   uint16_t valor;
   struct simbolo* proximo;
} simbolo_t;

// prototipos
void show_map (uint8_t *memoria);
void loop_instrucao (uint16_t CI, uint8_t *memoria, char *nome_arquivo, char *diretorio, uint16_t *numero_bytes);
void push_del(deletado_t* *head, char *nome);
void push_simbolo(simbolo_t* *head, char *nome);
int busca_simbolo(simbolo_t* *head, char *nome);
uint16_t get_valor(simbolo_t* *head, char *leitor);
void montador_1(char *nome_arquivo, uint16_t *inicio, uint8_t *numero_bytes);
void montador_2(char *nome_arquivo, uint16_t *inicio, uint8_t *numero_bytes, simbolo_t* *head);
void carga_loader(char *nome_arquivo, uint8_t *memoria);
```

São incluídas algumas bibliotecas padrão. Em particular, a biblioteca <dirent.h> serve para se navegar em um diretório escolhido.

Define-se duas constantes: “MAX” é o tamanho máximo que uma string pode ter e “TAMANHO” é o número de posições da memória virtual.

Define-se dois novos tipos de variável usando struct, para que tais variáveis possam ter múltiplos atributos e, também, formar uma lista ligada: “deletado_t” representa os programas a serem deletados; “simbolo_t” representa os símbolos de um programa-fonte em linguagem simbólica, associando um nome a um valor.

Define-se os protótipos das funções.

Funções

```c
// primeiro passo do montador
void montador_1(char *nome_arquivo, uint16_t *inicio, uint8_t *numero_bytes) {

    FILE *programa;
    char leitor[10];
    int aux = 0;
    uint16_t posicao;

    // iniciando lista vazia
    simbolo_t* head = NULL;

    // abre o o arquivo de entrada para leitura
    programa = fopen(nome_arquivo, "r");
    if (programa == NULL) {
        printf("Erro na abertura do arquivo\n");
        return;
    }

    while (1) {

        aux = fscanf(programa, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // sai do loop quando acha simbolo de fim de programa
        if(!strcmp(leitor, "#")) break;


        if(!strcmp(leitor, "@")) { // ORIGIN: define endereco inicial
            aux = fscanf(programa, "%04X", inicio);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            posicao = *inicio;
        }

        else if(!strcmp(leitor, "JP")) { // jump incondicional
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "JZ")) { // jump if zero
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "JN")) { // jump if negative
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "CN")) { // instrucao de controle
            // nao faz nada no primeiro passo, apenas avanca na leitura e incrementa posicao
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            posicao += 0x1;
        }

        else if(!strcmp(leitor, "+")) { // soma
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "-")) { // subtracao
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "*")) { // multiplicacao
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "/")) { // divisao
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }
```

```
        else if(!strcmp(leitor, "LD")) { // carrega valor da memoria
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "MM")) { // guarda valor na memoria
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "SC")) { // chamda de subrotina
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            push_simbolo(&head, leitor); // add simbolo ao qual a operacao se refere na lista
            posicao += 0x2;
        }

        else if(!strcmp(leitor, "OS")) { // chamada de sistema operacional
            // nao faz nada no primeiro passo, apenas incrementa posicao
            posicao += 0x1;
        }

        else if(!strcmp(leitor, "IO")) { // input/output/interrupcao
            // nao faz nada no primeiro passo, apenas avanca na leitura e incrementa posicao
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            posicao += 0x1;
        }

        else if(!strcmp(leitor, "K")) { // constante
            // nao faz nada no primeiro passo, apenas avanca na leitura e incrementa posicao
            aux = fscanf(programa, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");
            posicao += 0x1;
        }

        else { // simbolos ou labels
            push_simbolo(&head, leitor);
            // associa endereco ao simbolo correto
            simbolo_t* atual = head;
            while(strcmp(atual->nome, leitor)) atual = atual->proximo;
            atual->valor = posicao;
        }
    }

    *numero_bytes = posicao - *inicio;

    // fecha programa
    fclose(programa);

    // chamada do passo 2
    montador_2(nome_arquivo, inicio, numero_bytes, &head);
}
```

Esta função representa o primeiro passo de um montador, ou seja, tem como objetivo criar uma lista ligada dos símbolos e seus respectivos valores. Como parâmetros ela recebe o nome completo do arquivo (diretório + nome + extensão) que contém o programa-fonte, um ponteiro do endereço inicial, e outro do número de bytes.

Inicia-se criando as variáveis necessárias, iniciando a lista ligada de símbolos, e abrindo o arquivo do programa para leitura.

Em seguida, entra-se em um loop que lê um elemento do arquivo e o trata de acordo: caso o elemento seja “#”, termina-se o loop; caso o elemento seja “@”, lê-se o próximo elemento, que representa o endereço inicial, e o guardamos tanto no ponteiro “início”, quanto na variável “posição”; caso o elemento seja o mnemônico de alguma instrução de referência a memória, lê-se o próximo elemento, que

representa um label ou símbolo, aciona-se a função “push_simbolo” (explicada mais adiante), e incrementa-se a variável “posição” de acordo; caso o elemento seja algum outro mnemônico, apenas incrementa-se a posição de leitura e a variável “posição” de acordo; finalmente, caso o elemento seja um label ou símbolo não imediatamente precedidos por mnemônico, aciona-se a função “push_simbolo”, busca-se na lista o símbolo em questão, e associa-se a ele o valor “posição”. (Vale ressaltar que a variável “posição” progride de acordo com a primeira tabela do capítulo 3 deste relatório).

Finalmente, atualiza-se o ponteiro do número de bytes, fecha-se o arquivo, e chama-se a função “montador_2”.

```c
// segundo passo do montador
void montador_2(char *nome_arquivo, uint16_t *inicio, uint8_t *numero_bytes, simbolo_t* *head) {

    FILE *programa_hex;
    FILE *programa_simb;
    char leitor[10];
    char novo_nome[MAX];
    int aux = 0;

    // abre o o arquivo de entrada para leitura
    programa_simb = fopen(nome_arquivo, "r");
    if (programa_simb == NULL) {
        printf("Erro na abertura do arquivo\n");
        return;
    }

    // monta nome correto do arquivo de escrita
    strncpy(novo_nome, nome_arquivo, strlen(nome_arquivo) - 4);
    strcat(novo_nome, "_hex.txt");

    // abre arquivo para escrita
    programa_hex = fopen(novo_nome, "w");
    if (programa_hex == NULL) {
        printf("Erro na abertura do arquivo\n");
        return;
    }

    // escreve endereco inicial e numero de bytes
    fprintf(programa_hex, "%02X\n", (*inicio / 0x100));
    fprintf(programa_hex, "%02X\n", (*inicio % 0x100));
    fprintf(programa_hex, "%02X\n", *numero_bytes);

    // descarta primeiras leituras, referentes ao simbolo @ e ao endereco inicial
    aux = fscanf(programa_simb, "%s", leitor);
    if (aux != 1) printf("Erro na leitura do arquivo\n");
    aux = fscanf(programa_simb, "%s", leitor);
    if (aux != 1) printf("Erro na leitura do arquivo\n");
```

```
while (1) {

    aux = fscanf(programa_simb, "%s", leitor);
    if (aux != 1) printf("Erro na leitura do arquivo\n");

    // sai do loop quando acha simbolo de fim de programa
    if(!strcmp(leitor, "#")) break;


    if(!strcmp(leitor, "JP")) { // jump incondicional
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime os dois bytes da operacao
        fprintf(programa_hex, "%02X\n", 0x00 + (get_valor(head, leitor) / 0x100));
        fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
    }

    else if(!strcmp(leitor, "JZ")) { // jump if zero
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime os dois bytes da operacao
        fprintf(programa_hex, "%02X\n", 0x10 + (get_valor(head, leitor) / 0x100));
        fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
    }

    else if(!strcmp(leitor, "JN")) { // jump if negative
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime os dois bytes da operacao
        fprintf(programa_hex, "%02X\n", 0x20 + (get_valor(head, leitor) / 0x100));
        fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
    }

    else if(!strcmp(leitor, "CN")) { // instrucao de controle
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime o byte da operacao
        if(!strcmp(leitor, "HM")) fprintf(programa_hex, "%02X\n", 0x30);
        else if(!strcmp(leitor, "IN")) fprintf(programa_hex, "%02X\n", 0x32);
    }

    else if(!strcmp(leitor, "+")) { // soma
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime os dois bytes da operacao
        fprintf(programa_hex, "%02X\n", 0x40 + (get_valor(head, leitor) / 0x100));
        fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
    }

    else if(!strcmp(leitor, "-")) { // subtracao
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime os dois bytes da operacao
        fprintf(programa_hex, "%02X\n", 0x50 + (get_valor(head, leitor) / 0x100));
        fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
    }

    else if(!strcmp(leitor, "*")) { // multiplicacao
        // le proximo simbolo
        aux = fscanf(programa_simb, "%s", leitor);
        if (aux != 1) printf("Erro na leitura do arquivo\n");

        // imprime os dois bytes da operacao
        fprintf(programa_hex, "%02X\n", 0x60 + (get_valor(head, leitor) / 0x100));
        fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
    }
```

```
        else if(!strcmp(leitor, "/")) { // divisao
            // le proximo simbolo
            aux = fscanf(programa_simb, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");

            // imprime os dois bytes da operacao
            fprintf(programa_hex, "%02X\n", 0x70 + (get_valor(head, leitor) / 0x100));
            fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
        }

        else if(!strcmp(leitor, "LD")) { // carrega valor da memoria
            // le proximo simbolo
            aux = fscanf(programa_simb, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");

            // imprime os dois bytes da operacao
            fprintf(programa_hex, "%02X\n", 0x80 + (get_valor(head, leitor) / 0x100));
            fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
        }

        else if(!strcmp(leitor, "MM")) { // guarda valor na memoria
            // le proximo simbolo
            aux = fscanf(programa_simb, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");

            // imprime os dois bytes da operacao
            fprintf(programa_hex, "%02X\n", 0x90 + (get_valor(head, leitor) / 0x100));
            fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
        }

        else if(!strcmp(leitor, "SC")) { // chamda de subrotina
            // le proximo simbolo
            aux = fscanf(programa_simb, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");

            // imprime os dois bytes da operacao
            fprintf(programa_hex, "%02X\n", 0xA0 + (get_valor(head, leitor) / 0x100));
            fprintf(programa_hex, "%02X\n", (get_valor(head, leitor) % 0x100));
        }

        else if(!strcmp(leitor, "OS")) { // chamada de sistema operacional
            // imprime o byte da operacao
            fprintf(programa_hex, "%02X\n", 0xB0);
        }

        else if(!strcmp(leitor, "IO")) { // input/output
            // le proximo simbolo
            aux = fscanf(programa_simb, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");

            // imprime o byte da operacao
            if(!strcmp(leitor, "GD")) fprintf(programa_hex, "%02X\n", 0xC0);
            else if(!strcmp(leitor, "PD")) fprintf(programa_hex, "%02X\n", 0xC4);
        }

        else if(!strcmp(leitor, "K")) { // constante
            // le proximo simbolo
            aux = fscanf(programa_simb, "%s", leitor);
            if (aux != 1) printf("Erro na leitura do arquivo\n");

            // imprime o byte da constante
            fprintf(programa_hex, "%s\n", leitor);
        }

        // else { } -> ignora labels e simbolos quando nao precedidos por mneumonico
    }

    // limpa lista ligda de simbolos para nao conflitar com outros programas
    simbolo_t* atual = *head;
    simbolo_t* proximo;
    while (atual != NULL) {
        proximo = atual->proximo;
        free(atual);
        atual = proximo;
    }
    *head = NULL;

    // fecha programas
    fclose(programa_hex);
    fclose(programa_simb);
}
```

Esta função representa o segundo passo de um montador, ou seja, tem como objetivo criar um arquivo contendo o código-objeto absoluto. Como parâmetros ela recebe o nome completo do arquivo (diretório + nome + extensão) que contém o programa-fonte, um ponteiro do endereço inicial, outro do número de bytes, e ainda um terceiro que aponta para o início da lista de símbolos.

Inicia-se criando as variáveis necessárias, abrindo o arquivo do programa-fonte para leitura, e criando-se um arquivo para escrita do código-objeto absoluto em hexadecimal, cujo nome tem um apêndice “_hex”.

Em seguida, escreve-se no arquivo do código-objeto as 3 primeiras linhas obrigatórias em seu formato, duas contendo o endereço inicial e outra contendo o número de bytes.

Em seguida, entra-se em um loop que lê um elemento do arquivo e o trata de acordo: caso o elemento seja “#”, termina-se o loop; caso o elemento seja o mnemônico de alguma instrução de referência a memória, lê-se o próximo elemento, que representa um label ou símbolo, busca-se seu valor correspondente acionando-se a função “get_valor” (explicada mais adiante), e escreve-se no arquivo os dois bytes que correspondem à instrução; caso o elemento seja algum outro mnemônico, excetuando-se “K”, lê-se o próximo elemento (apenas não ocorre na instrução “OS”), que representa uma especificação da instrução, e escreve-se no arquivo o byte correspondente; caso o elemento seja “K”, lê-se o próximo elemento, que representa o valor de uma constante, e escreve-se no arquivo o byte correspondente; finalmente, caso o elemento seja um label ou símbolo não imediatamente precedidos por mneumônico, não se faz nada.

Finalmente, limpa-se a lista ligada de símbolos, para que seus valores não sejam confundidos com os de programas futuros, e fecha-se os arquivos de leitura e escrita.

```
// retorna valor associado ao simbolo
uint16_t get_valor(simbolo_t* *head, char *leitor) {
    simbolo_t* atual = *head;
    while(strcmp(atual->nome, leitor)) atual = atual->proximo;
    return atual->valor;
}
```

Esta função tem como objetivo retornar o valor associado ao símbolo desejado. Para isso, recebe como parâmetros o início da lista ligada e o nome do símbolo desejado, e executa a busca.

```c
// adiciona simbolo no final da lista
void push_simbolo(simbolo_t* *head, char *nome) {

    // primeiro da lista
    if(*head == NULL) {
        simbolo_t* novo = (simbolo_t*)malloc(sizeof(simbolo_t));
        if(novo == NULL){
            printf("Nao foi possivel alocar memoria\n");
            exit(-1);
        }
        strcpy(novo->nome, nome);
        novo->proximo = NULL;
        *head = novo;
        return;
    }

    // criando novo item somente se ele ainda nao existir na lista
    if (busca_simbolo(head, nome)) return;

    simbolo_t* novo = (simbolo_t*)malloc(sizeof(simbolo_t));
    if(novo == NULL){
        printf("Nao foi possivel alocar memoria\n");
        exit(-1);
    }

    // definindo novo item
    strcpy(novo->nome, nome);
    novo->proximo = NULL;

    // adicionando no final da lista
    simbolo_t* atual = *head;
    while (atual->proximo != NULL) atual = atual->proximo;
    atual->proximo = novo;
}
```

Esta função tem como objetivo adicionar um símbolo na lista. Como parâmetros ela recebe o início da lista ligada e o nome do símbolo desejado.

Caso seja a lista esteja vazia, adiciona-se o símbolo e a função encerra.

Caso já tenha algo na lista, primeiro faz-se uma busca na lista pelo nome do símbolo através da função “busca_símbolo” (explicado mais adiante). Se já existir, a função encerra; caso contrário, adiciona-se o símbolo.

```c
// busca simbolo, retorna 1 se ja existe e 0 caso contrario
int busca_simbolo(simbolo_t* *head, char *nome) {
    simbolo_t* atual = *head;
    while (atual != NULL) {
        if(!strcmp(atual->nome, nome)) return 1;
        atual = atual->proximo;
    }
    return 0;
}
```

Esta função tem como objetivo verificar a existência de um símbolo na lista. Recebe como parâmetros o início da lista ligada e o nome do símbolo desejado. Retorna 1 caso já exista e 0 caso contrário.

```c
// exibe conteudo da memoria em hexadecimal
void show_map (uint8_t *memoria) {

    printf ("Mapa de memoria: \n");
    printf ("      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F\n");
    int i = 0;
    for (int linha = 0; i < TAMANHO; linha++) {
        printf ("%03X  ", linha);
        for (int coluna = 0; coluna < 16; coluna++, i++) {
            printf ("%02X  ", memoria[i]);
        }
        printf ("\n");
    }
}
```

Esta função tem como objetivo exibir ao usuário o mapa de memória da máquina virtual simulada. Para isso, recebe como parâmetro o vetor de memória e imprime na caixa de diálogo o mapa, como ilustrado no capítulo 4 deste relatório.

```c
// executa instrucao (de 1 ou 2 bytes) de acordo com seu tipo e seu operando
void loop_instrucao (uint16_t CI, uint8_t *memoria, char *nome_arquivo, char *diretorio, uint16_t *numero_bytes) {

    FILE *programa;
    FILE *saida;
    int8_t operacao;
    int16_t operando;
    int8_t acumulador = 0x00; // inicialmente contem zero
    int aux;
    char auxiliar[10];
    int HALT = 0;
    int modo_indireto = 0;
    char arquivo_saida[MAX];

     // abre o arquivo de entrada para leitura
    programa = fopen(nome_arquivo, "r");
    if (programa == NULL) {
        printf("Erro na abertura do arquivo\n");
        return;
    }

    // monta nome correto do arquivo de escrita
    strcpy(arquivo_saida, diretorio);
    strcat(arquivo_saida, "saida.txt");

    // abre arquivo para escrita
    saida = fopen(arquivo_saida, "w");
    if (saida == NULL) {
        printf("Erro na abertura do arquivo\n");
        return;
    }

    // descarta primeiras leituras, referentes ao endereco inicial e pega numero de bytes
    aux = fscanf(programa, "%02X", numero_bytes);
    aux = fscanf(programa, "%02X", numero_bytes);
    aux = fscanf(programa, "%02X", numero_bytes);
    if (aux != 1) printf("Erro na leitura do arquivo\n");
    rewind(programa);
```

```
while(!HALT) {

    operacao = (memoria[CI] / 0x10); // identifica o tipo da instrucao
    // identifica o operando para instrucoes de 1 byte
    if (operacao == 0x3 || operacao == 0xB || operacao == 0xC)
        operando = (memoria[CI] % 0x10);
    // identifica o operando para instrucoes de 2 bytes
    else operando = (memoria[CI] % 0x10)*0x100 + memoria[CI + 0x1];

    switch (operacao) {

    case 0x0: // jump incondicional
        if(!modo_indireto) CI = operando;
        else CI = memoria[operando]*0x100 + memoria[operando + 0x1];
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x1: // jump se acumulador for 0
        if (acumulador == 0 && !modo_indireto) CI = operando;
        else if(acumulador == 0) CI = memoria[operando]*0x100 + memoria[operando + 0x1];
        else CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x2: // jump se acumulador for negativo
        if (acumulador < 0 && !modo_indireto) CI = operando;
        else if(acumulador < 0) CI = memoria[operando]*0x100 + memoria[operando + 0x1];
        else CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x3: // instrucao de controle
        switch (operando) {

        case 0x0: // halt machine
            printf("Halt Machine\n\n");
            HALT = 1;
            break;

        case 0x2: // indirect
            modo_indireto = 1;
            CI += 1;
            break;
        }
        break;

    case 0x4: // soma
        if(!modo_indireto) acumulador += memoria[operando];
        else acumulador += memoria[memoria[operando]*0x100 + memoria[operando + 0x1]];
        CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x5: // subtracao
        if(!modo_indireto) acumulador -= memoria[operando];
        else acumulador -= memoria[memoria[operando]*0x100 + memoria[operando + 0x1]];
        CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x6: // multiplicacao
        if(!modo_indireto) acumulador *= memoria[operando];
        else acumulador *= memoria[memoria[operando]*0x100 + memoria[operando + 0x1]];
        CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x7: // divisao
        if(!modo_indireto) acumulador /= memoria[operando];
        else acumulador /= memoria[memoria[operando]*0x100 + memoria[operando + 0x1]];
        CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x8: // load
        if(!modo_indireto) acumulador = memoria[operando];
        else acumulador = memoria[memoria[operando]*0x100 + memoria[operando + 0x1]];
        CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;

    case 0x9: // store
        if(!modo_indireto) memoria[operando] = acumulador;
        else memoria[memoria[operando]*0x100 + memoria[operando + 0x1]] = acumulador;
        CI += 2;
        modo_indireto = 0; // sempre retorna ao enderecamento direto
        break;
```

```
        case 0xA: // chamada de sub-rotina
            if(!modo_indireto) {
                memoria[operando] = (CI + 0x2) / 0x100;
                memoria[operando + 0x1] = (CI + 0x2) % 0x100;
                CI = operando + 0x2;
            }
            else {
                memoria[memoria[operando]*0x100 + memoria[operando + 0x1]] = (CI + 0x2) / 0x100;
                memoria[(memoria[operando]*0x100 + memoria[operando + 0x1]) + 0x1] = (CI + 0x2) % 0x100;
                CI = (memoria[operando]*0x100 + memoria[operando + 0x1]) + 0x2;
            }
            modo_indireto = 0; // sempre retorna ao enderecamento direto
            break;

        case 0xB: // chamada do sistema operacional
            HALT = 1; // devolve controle de volta ao interpretador
            break;

        case 0xC: // entrada, saida e interrupcao
            switch (operando) {

            case 0x0: // get data (do arquivo.txt para o acumulador)
                fgets(auxiliar, sizeof(auxiliar)-1, programa);
                sscanf(auxiliar, "%02X", &acumulador);
                CI += 1;
                break;

            case 0x4: // put data (do acumulador para o arquivo.txt)
                fprintf(saida, "%02X\n", acumulador);
                CI += 1;
                break;
            }
            break;
        }
    }

    // fecha programas
    fclose(programa);
    fclose(saida);
}
```

Esta função representa o loop de um motor de eventos de uma máquina de van Neumann, ou seja, a partir do contador de instruções, extrai-se da memória o código de operação da instrução e seu operando. Como parâmetros ela recebe o contador de instruções, o vetor que representa a memória, o nome completo do arquivo (diretório + nome + extensão) que contém o código-objeto absoluto, o nome apenas do diretório, e um ponteiro do número de bytes do programa.

Inicia-se criando as variáveis necessárias, abrindo o arquivo do código-objeto para leitura, criando-se um arquivo “saida.txt” para escrita de código-objeto absoluto em hexadecimal, caso o programa em execução seja um dumper, e atualizando o ponteiro do número de bytes.

Em seguida, entra-se em um loop que extrai da memória o código de operação da instrução e seu operando, executa a instrução conforme previsto pelo capítulo 2 deste relatório, e incrementa o contador de instruções de acordo. Vale observar que este loop prevê o tipo de endereçamento (direto o indireto), e termina somente quando se alcança uma instrução de Halt Machine ou chamada de Sistema Operacional.

Finalmente, após sair do loop, fecha os arquivos de escrita e leitura.

```c
// adiciona programa a ser deletado no fim da lista
void push_del(deletado_t* *head, char *nome) {

    // primeiro da lista
    if(*head == NULL) {
        deletado_t* novo = (deletado_t*)malloc(sizeof(deletado_t));
        if(novo == NULL){
            printf("Nao foi possivel alocar memoria\n");
            exit(-1);
        }
        strcpy(novo->nome, nome);
        novo->proximo = NULL;
        *head = novo;
        return;
    }

    // criando novo item
    deletado_t* novo = (deletado_t*)malloc(sizeof(deletado_t));
    if(novo == NULL){
        printf("Nao foi possivel alocar memoria\n");
        exit(-1);
    }

    // definindo novo item
    strcpy(novo->nome, nome);
    novo->proximo = NULL;

    // adicionando no final da lista
    deletado_t* atual = *head;
    while (atual->proximo != NULL) atual = atual->proximo;
    atual->proximo = novo;
}
```

Esta função tem como objetivo adicionar um programa na lista de arquivos a serem deletados. Como parâmetros ela recebe o início da lista ligada e o nome do arquivo desejado.

Diferentemente da função “push_simbolo”, esta não verifica a existência do arquivo na lista, pois não há necessidade.

```c
// faz a carga do loader na memoria
void carga_loader(char *nome_arquivo, uint8_t *memoria) {
    FILE *programa;

    programa = fopen(nome_arquivo, "r");
    if (programa == NULL) {
        printf("Erro na abertura do arquivo\n");
        return;
    }

    // le arquivo e coloca na memoria
    for (int i = 0; i < 44; i++) fscanf(programa, "%02X", &memoria[0x0 + i]);

    // fecha programa
    fclose(programa) ;
}
```

Esta função tem como objetivo fazer a carga do loader na memória. Para isso, recebe como parâmetros o nome completo do arquivo (diretório + nome + extensão) que contém o código-objeto absoluto do loader em hexadecimal, e o vetor que representa a memória.

Main

```c
int main()
{
    // criando variaveis necessarias
    uint8_t numero_bytes = 0x00;
    uint16_t inicio = 0x0000;

    char comando[MAX];
    char nome_arquivo[MAX];
    char diretorio[MAX];

    uint8_t memoria[TAMANHO] = {0x00}; // cada posicao de memoria representa um byte e todas sao iniciadas com 0
    uint16_t CI = 0x0000; // contador de instrucoes (16 bits); inicialmente aponta pra 0

    // iniciando lista vazia
    deletado_t* head = NULL;

    printf("Identifique o usuario (nome do diretorio completo onde estao os programas): \n");
    printf("***Ex.: /Users/raphael-amarante/Desktop/a1/*** \n");

    fgets(diretorio, MAX, stdin);
    if(diretorio[strlen(diretorio) - 1] == '\n')
        diretorio[strlen(diretorio) - 1] = NULL; // necessario retirar newline

    // inicializacao: carregando o loader na memoria
    strcpy(nome_arquivo, diretorio);
    strcat(nome_arquivo, "loader_hex.txt");
    carga_loader(nome_arquivo, memoria);

    while (1) {

        printf("\nDigite um dos seguintes comandos:\n");
        printf("$DIR - lista programas disponiveis no sistema\n");
        printf("$DEL <nome> - remove programa do sistema\n");
        printf("$RUN <nome> - carrega na memoria e executa programa hexadecimal\n");
        printf("$MTD <nome> - montador: gera programa hexadecimal a partir do simbolico\n");
        printf("$MAP - exibe mapa de memoria\n");
        printf("$END - encerra sistema\n");
        printf("***Ex. de <nome>: programa.txt*** \n\n");

        fgets(comando, MAX, stdin);
        if(comando[strlen(comando) - 1] == '\n')
            comando[strlen(comando) - 1] = NULL; // necessario retirar newline
```

```c
        if (!strncmp(comando, "$DIR", 4)) {

            printf("Arquivos iniciados com 0 sao os que serao apagados quando o sistema for finalizado\n");

            struct dirent *de;  // ponteiro de entrada do diretorio

            DIR *dr = opendir(diretorio);

            if (dr == NULL) {
                printf("Nao foi possivel abrir o diretorio\n" );
                return 0;
            }

            while ((de = readdir(dr)) != NULL) {
                if ( !strcmp(de->d_name, ".") || !strcmp(de->d_name, "..") || !strcmp(de->d_name, ".DS_Store")) {
                    /*NOP*/
                }
                // "." representa o diretorio atual
                //".." representa o diretorio "pai"
                // ".DS_Store" representa um arquivo gerado pelo OS da maquina hospedeira
                else printf("%s\n", de->d_name);
            }

            closedir(dr);
        }

        else if (!strncmp(comando, "$DEL", 4)) {

            // prepara nome completo do arquivo + diretorio
            strcpy(nome_arquivo, diretorio);
            strcat(nome_arquivo, comando + 5);

            // prepara novo nome completo (adiciona-se "0" antes do nome do arquivo)
            char novo_nome[MAX];
            strcpy(novo_nome, diretorio);
            strcat(novo_nome, "0");
            strcat(novo_nome, comando + 5);

            // rename para posterior remocao definitiva do programa
            if(!rename(nome_arquivo, novo_nome)) {
                printf("Nome do arquivo a ser deletado foi modificado com sucesso\n");
                push_del(&head, novo_nome);
            }
            else printf("Erro na operacao\n");
        }
```

```
        else if (!strncmp(comando, "$RUN", 4)) {

            // prepara nome completo do arquivo + diretorio
            strcpy(nome_arquivo, diretorio);
            strcat(nome_arquivo, comando + 5);

            // executa loader para carga do programa na memoria
            loop_instrucao (CI, memoria, nome_arquivo, diretorio, &numero_bytes);

            // seta CI para inicio do programa a executar
            CI = memoria[0x0028]*0x100 + memoria[0x0029] - numero_bytes;

            // executa programa
            loop_instrucao (CI, memoria, nome_arquivo, diretorio, &numero_bytes);
        }

        else if (!strncmp(comando, "$MTD", 4)) {
            // prepara nome completo do arquivo + diretorio
            strcpy(nome_arquivo, diretorio);
            strcat(nome_arquivo, comando + 5);
            montador_1(nome_arquivo, &inicio, &numero_bytes);
        }

        else if (!strncmp(comando, "$MAP", 4)) {
            show_map(memoria);
        }

        else if (!strncmp(comando, "$END", 4)) {
            deletado_t* atual = head;
            while (atual != NULL) {
                if (!remove(atual->nome))
                    printf("Programa removido com sucesso\n");
                else printf("Nao foi possivel deletar o arquivo\n");

                atual = atual->proximo;
            }
            break;
        }
    }
    return 0;
}
```

Primeiramente, inicia-se as variáveis importantes, e a lista ligada dos programas a serem deletados. Vale ressaltar que a memória da máquina virtual é aqui representada por um vetor, onde cada elemento é um byte sem sinal, e o contador de instruções é um elemento de dois bytes, também sem sinal.

Em seguida, para a correta inicialização do sistema, a caixa de diálogo com o usuário exige a identificação deste (como decisão de projeto, esta identificação consiste no diretório da pasta do usuário na máquina hospedeira); e aciona-se a função “carga_loader”, carregando previamente o loader na memória para imitar na máquina virtual o que circuitos de hardware fariam.

Finalmente, entra-se em um loop infinito que exibe na caixa de diálogo os possíveis comandos do interpretador e, após sua escolha, o executa de acordo com o especificado no capítulo 4 deste relatório. O loop apenas é quebrado com o comando “$END”.

6. Modo de uso

Para o correto funcionamento do sistema alguns quesitos devem ser sanados:

- Quando o usuário se identificar com o nome do diretório de sua pasta, esta já deve existir na máquina hospedeira. O formato a ser seguido seria, por exemplo, “/Users/raphael-amarante/Desktop/projeto/”;
- É necessário que já exista nessa pasta o programa “loader_hex.txt” (detalhado no apêndice deste relatório), que contém o código-objeto absoluto em hexadecimal do loader;
- É necessário que todos os demais arquivos, seja de programa-fonte em linguagem simbólica ou código-objeto absoluto, também estejam nessa pasta;
- Tais programas, quando referidos por comandos do interpretador, devem seguir, por exemplo, o seguinte formato “programa.txt”.

7. Simulação

Nesta seção será explorada a simulação de todos os comandos do interpretador, através da execução do programa que calcula o valor de “n” ao quadrado, como exposto no capítulo 3 deste relatório.

Primeiramente, foi criada uma pasta na máquina hospedeira com o diretório “/Users/raphael-amarante/Desktop/projeto/”, que contém os arquivos “loader_hex.txt” (código-objeto absoluto do loader) e “n2.txt” (programa-fonte em linguagem simbólica).

À seguir, está o passo-a-passo da execução do programa:

Output de identificação do usuário:

```
Identifique o usuario (nome do diretorio completo onde estao os programas):
***Ex.: /Users/raphael-amarante/Desktop/projeto/***
```

Input do usuário: /Users/raphael-amarante/Desktop/projeto/

Output de opções de comando do interpretador:

```
Digite um dos seguintes comandos:
$DIR - lista programas disponiveis no sistema
$DEL <nome> - remove programa do sistema
$RUN <nome> - carrega na memoria e executa programa hexadecimal
$MTD <nome> - montador: gera programa hexadecimal a partir do simbolico
$MAP - exibe mapa de memoria
$END - encerra sistema
***Ex. de <nome>: programa.txt***
```

Input do usuário: $MAP

Output:

```
Mapa de memoria:
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
000  C0  90  28  C0  90  29  C0  90  2A  C0  32  90  28  80  29  40
001  2B  90  29  10  1F  80  2A  50  2B  90  2A  10  27  00  09  80
002  28  40  2B  90  28  00  15  B0  00  00  00  01  00  00  00  00
003  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
005  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
006  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
007  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
008  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
009  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00A  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00B  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00C  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00D  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00E  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00F  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
```

Obs.: o mapa de memória é mais longo, mas aqui exibe-se apenas a parte que contém algo relevante para a simulação. No caso, as primeiras posições foram ocupadas pelo loader.

Input do usuário: $MTD n2.txt
Obs.: arquivo “n2_hex.txt” criado na pasta do usuário.

Input do usuário: $DIR

Output:

```
Arquivos iniciados com 0 sao os que serao apagados quando o sistema for finaliza
do
loader_hex.txt
n2.txt
n2_hex.txt
```

Input do usuário: $RUN n2_hex.txt
Obs.: arquivo “saida.txt” criado na pasta do usuário.

Input do usuário: $MAP

```
Mapa de memoria:
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
000  C0  90  28  C0  90  29  C0  90  2A  C0  32  90  28  80  29  40
001  2B  90  29  10  1F  80  2A  50  2B  90  2A  10  27  00  09  80
002  28  40  2B  90  28  00  15  B0  00  57  00  01  00  00  00  00
003  80  51  90  56  90  53  90  55  80  56  50  54  10  50  80  56
004  40  51  90  56  80  53  40  52  90  53  40  55  90  55  00  38
005  B0  01  02  07  04  10  04  00  00  00  00  00  00  00  00  00
006  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
007  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
008  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
009  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00A  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00B  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00C  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00D  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00E  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00F  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
```

Obs.: vale ressaltar que este mapa representa a memória **após** a execução do programa. No caso, a posição 0054 contém “n” (= 4), e a posição seguinte contém “n” ao quadrado (10 hex = 16).

Input do usuário: $DEL n2_hex.txt

Output:

```
Nome do arquivo a ser deletado foi modificado com sucesso
```

Input do usuário: $DIR

Output:

```
Arquivos iniciados com 0 sao os que serao apagados quando o sistema for finaliza
do
0n2_hex.txt
loader_hex.txt
n2.txt
saida.txt
```

Input do usuário: $END

Output:

```
Programa removido com sucesso
```

Obs.: o arquivo “0n2_hex” foi definitivamente removido da pasta do usuário.

8. Apêndice

Nesta seção, expõe-se o algoritmo usado para implementar o loader. Segue abaixo sua versão hexadecimal (a utilizada pelo sistema no arquivo “loader_hex.txt”), e sua versão simbólica comentada:

```
C0      GD              # le byte do arquivo e põe no acc
90      MM    EI1       # coloca na posicao de end_in_1
28      GD              # le byte do arquivo e põe no acc
C0      MM    EI2       # coloca na posicao de end_in_2
90      GD              # le byte do arquivo e põe no acc
29      MM    NB        # coloca na posição de num_bytes
C0
90      LOOP
2A      GD              # le byte no arquivo e põe no acc
C0      IN              # ativa endereçamento indireto
32      MM    EI1       # coloca tal byte no end. a partir
90                      # do endereço inicial de carga
28      LD    EI2       # carrega valor da posição no acc
80      +     UM        # aumenta
29      MM    EI2       # guarda acc na memória
40      JZ    VAIUM     # tratamento de vai um
2B
90      VOLTA
29      LD    NB        # carrega no acc
10      -     UM        # subtrai 1 do número de bytes a ler
1F      MM    NB        # guarda acc na memória
80      JZ    FIM       # termina rotina
2A      JP    LOOP      # volta ao loop
50
2B      VAIUM
90      LD    EI1       # carrega valor da posição no acc
2A      +     UM        # aumenta
10      MM    EI1       # guarda acc na memória
27      JP    VOLTA     # volta do tratamento do vai um
00
09      FIM
80      OS              # termina execução do loader
28
40
2B      EI1   K   00     # parte do endereço inicial
90      EI2   K   00     # parte do endereço inicial
28      NB    K   00     # número de bytes (em hex)
00      UM    K   01     # constante 1
15
B0
00
00
00
01
```