Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Trabalho Prático 3

BCC266 - Organização de Computadores

Caio Lucas Pereira da Silva, Vinicius Nunes dos Anjos, Thalles Felipe Rodrigues de Almeida Santos

Professor: Pedro Henrique Lopes Silva

Ouro Preto
21 de março de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Este trabalho prático tem como objetivo uma simulação de um funcionamento de um UCM ou MMU, no qual ele gerenciará as memórias tanto a externa, principal e as caches, as quais possuem um nivelamento de importância e sua forma de construção diferentes, umas por linhas e outras por blocos. Além disso, a máquina a qual está construida é capaz de ocasionar uma interrupção durante o processo, salvando o contexto e inicializando o processo de tratamento de interrupções e após isso é voltado para o processo anterior a interrupção, sendo possível não ser só uma única interrupção

1.1 Especificações do problema

O problema consiste em prosseguir com a construção de um programa em C que já fora feito anteriormente do TP2 de mesmo curso, só que agora foi-se apresentado os sistemas de memórias, no qual é de necessidade a implementação de memória externa e tratamento de interrupções. O código-fonte deve ser modularizado em vários arquivos, e o programa não pode ter memory leaks.

1.2 Pontos fortes

- O código é bem modularizado, com a divisão de funções em arquivos separados para lidar com objetos, pontos e ordenação.
- As funções possuem nomes claros e intuitivos, que facilitam a leitura e compreensão do código.
- As funções são bem definidas e focadas em apenas uma tarefa, o que as torna reutilizáveis em outros contextos.
- Há comentários que ajudam a entender o objetivo de cada função.
- O código faz uso de ponteiros para evitar a cópia desnecessária de estruturas e dados.

1.3 Pontos fracos

- Não há tratamento de erros em relação à alocação dinâmica de memória, o que pode resultar em erros de execução caso a alocação falhe.

1.4 Ferramentas utilizadas

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

[1] VScode está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

1.5 Ferramentas adicionais

Algumas ferramentas foram utilizadas para auxiliar no desenvolvimento, como:

- *Live Share*: ferramenta usada para *pair programming* à distância.

- *Valgrind*: ferramentas de análise dinâmica do código.

1.6 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: AMD Ryzen 5-5500U.

- Memória RAM: 8GB.

- Sistema Operacional: Linux Pop_OS.

1.7 Instruções de compilação e execução

Para a compilação do projeto, basta digitar o seguinte comando para rodar o arquivo *Makefile* disponível:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-o*: para definir o arquivo de saída.

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

- *-Wall*: para mostrar todos os possível *warnings* do código.

- *-c*: para compilação do código e geração dos arquivos objetos.

Para a execução do programa basta digitar :

```
./exe < arquivo.in
```

Onde o arquivo de entrada contém o tamanho da matriz e seus valores.

2 Desenvolvimento

O desenvolvimento foi realizado utilizando da técnica de pair programming, onde todos os integrantes do grupo programaram e participaram ativamente do código ao mesmo tempo. O uso das ferramentas Live Share para o compartilhamento de código e do Discord para a comunicação em equipe foi o que viabilizou o uso da técnica citada.

2.1 Codificação

Para a codificação foram utilizados sete arquivos (8 ".c"e 7 ".h"), os quais foram preechidos corretamente para o funcionamento.

2.2 Main code

```
1
2  #include "cpu.h"
3  #include "generator.h"
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <time.h>
8
9  int main(int argc, char **argv)
10 {
11
12     srand(1507); // Inicializacao da semente para os numeros aleatorios.
13
14     int memoriesSize[5];
15     Machine machine;
16     Instruction *instructions;
17
18     // mudar para fazer funcionar a leitura de arquivos
19     if (strcmp(argv[1], "random") == 0)
20     {
21         if (argc != 7)
22         {
23             printf("Numero de argumentos invalidos! Sao 6.\n");
24             printf("Linha de execucao: ./exe TIPO_INSTRUCAO [TAMANHO_MEMORIA|
                   ARQUIVO_DE_INSTRUCOES] TAMANHO_RAM TAMANHO_L1 TAMANHO_L2
                   TAMANHO_L3\n");
25             printf("\tExemplo 1 de execucao: ./exe random 20 10 2 4 6\n");
26             printf("\tExemplo 2 de execucao: ./exe file arquivo_de_instrucoes.
                   txt\n");
27             return 0;
28         }
29
30         memoriesSize[0] = atoi(argv[2]);
31         memoriesSize[1] = atoi(argv[3]);
32         instructions = generateRandomInstructions(memoriesSize[1]);
33         memoriesSize[2] = atoi(argv[4]);
34         memoriesSize[3] = atoi(argv[5]);
35         memoriesSize[4] = atoi(argv[6]);
36     }
37     else if (strcmp(argv[1], "file") == 0)
38     {
39         instructions = readInstructions(argv[2], memoriesSize);
40     }
41     else
```

```
42      {
43          printf("Invalid option.\n");
44          return 0;
45      }
46
47      printf("Starting machine...\n");
48      start(&machine, instructions, memoriesSize);
49      // if (memoriesSize[0] < 20)
50      //     printMemories(&machine);
51      run(&machine);
52      // if (memoriesSize[0] < 20)
53          // printMemories(&machine);
54      stop(&machine);
55      // printf("Stopping machine...\n");
56      return 0;
57  }
```

Código 1: Main

Este código é um programa em linguagem C que implementa a execução de instruções em uma máquina virtual. O programa pode ser executado a partir da linha de comando, com opções de entrada para indicar se as instruções serão geradas aleatoriamente ou lidas a partir de um arquivo de entrada, bem como o tamanho de três níveis de cache e da memória principal.

Na função main(), as opções de entrada são processadas para determinar como as instruções e tamanhos de memória serão obtidos. Se a opção de entrada for "random", as instruções são geradas aleatoriamente e os tamanhos de memória são lidos a partir dos argumentos da linha de comando. Caso contrário, a opção de entrada é "file"e as instruções e tamanhos de memória são lidos a partir de um arquivo.

Em seguida, o programa inicializa a máquina virtual, executa as instruções, imprime o conteúdo da memória se seu tamanho for menor que 20 e finaliza a máquina virtual. O programa então termina sua execução.

2.3 TADs utilizada

```
typedef struct {
    Instruction* instructions;
    Storage storage;
    RAM ram;
    Cache l1; // cache L1
    Cache l2; // cache L2
    Cache l3; // cache L3
    int hitL1, hitL2, hitL3, hitRAM, hitStorage;
    int missL1, missL2, missL3, missRAM;
    int interruptions;
    int totalCost;
} Machine;

typedef struct {
    int block;
    int word;
} Address;

typedef struct {
    Address add1;
    Address add2;
    Address add3;
    int opcode;
} Instruction;

typedef struct
{
    int words[WORDS_SIZE];
    int tag;
    bool updated;
    int cost;
    int hit;
    int timesUsed;
    int cacheTime;
} MemoryBlock;

typedef struct
{
    MemoryBlock *blocks;
    int size;
} Storage;

typedef struct
{
    MemoryBlock *blocks;
    int size;
} RAM;

typedef struct
{
    MemoryBlock block;
    int tag; /* Address of the block in memory RAM */
    bool updated;
    int cost;
    int cacheHit;
    int timesUsed;
    int cacheTime;
} Line;
```

```

typedef struct
{
    Line *lines;
    int size;
} Cache;
```

Código 2: TADs

Para a implementação das interrupções e também da memória externa foi criada uma nova TAD sendo essa a 'Storage' e ademais foi modificada uma TAD já existente para o acrescimo de informações providas da implementação dos requerintes sendo colocado 'Storage storage' e 'int interruptions' no TAD 'Machine'

2.4 Executar e interromper

```
void executeInstruction(Machine *machine, int PC, bool interrupted)
{
  Instruction instruction = machine->instructions[PC];
  // Registers
  int word1, word2;

  // Addresses to be consulted
  Address add1 = instruction.add1;
  Address add2 = instruction.add2;
  Address add3 = instruction.add3;

  // Line find in memory
  Line *line;

  switch (instruction.opcode)
  {
  case -1:
    printf("  > Ending execution.\n");
    break;
  case 0:
    if (interrupted)
    {
      // Taking information to RAM
      line = MMUSearchOnMemories(add1, machine); /* Searching block on
          memories */
      word1 = line->block.words[add1.word];
#ifdef PRINT_LOG
      printf("  > MOV BLOCK[%d.%d.%d](%4d) > \n", line->cacheHit, add1.block,
          add1.word, line->block.words[add1.word]);
#endif

      line = MMUSearchOnMemories(add2, machine); /* Searching block on
          memories */
#ifdef PRINT_LOG
      printf("BLOCK[%d.%d.%d](%4d| \n", line->cacheHit, add2.block, add2.word,
           line->block.words[add2.word]);
#endif

      line->block.words[add2.word] = word1;
      line->updated = true;
#ifdef PRINT_LOG
      printf("%4d).\n", line->block.words[add2.word]);
#endif
      break;
    }
```

```
42
43      int nInterruptions = rand() % 10;
44      machine->interruptions = nInterruptions;
45
46  #ifdef PRINT_INTERRUPTIONS
47      printf("  > %d interruptions!\n", nInterruptions);
48  #endif
49
50      for (int i = 0; i < nInterruptions; i++)
51      {
52        instruction.opcode = rand() % 3;
53      }
54
55    case 1:                                         // Sum
56      line = MMUSearchOnMemories(add1, machine); /* Searching block on memories
          */
57      word1 = line->block.words[add1.word];
58  #ifdef PRINT_LOG
59      printf("  > SUM BLOCK[%d.%d.%d](%4d)\n", line->cacheHit, add1.block, add1.
          word, line->block.words[add1.word]);
60  #endif
61
62      line = MMUSearchOnMemories(add2, machine); /* Searching block on memories
          */
63      word2 = line->block.words[add2.word];
64  #ifdef PRINT_LOG
65      printf(" + BLOCK[%d.%d.%d](%4d)\n", line->cacheHit, add2.block, add2.word,
           line->block.words[add2.word]);
66  #endif
67
68      line = MMUSearchOnMemories(add3, machine); /* Searching block on memories
          */
69  #ifdef PRINT_LOG
70      printf(" > BLOCK[%d.%d.%d](%4d|\n", line->cacheHit, add3.block, add3.word,
           line->block.words[add3.word]);
71  #endif
72
73      line->updated = true;
74      line->block.words[add3.word] = word2 + word1;
75  #ifdef PRINT_LOG
76      printf("%4d).\n", line->block.words[add3.word]);
77  #endif
78      break;
79    case 2:                                         // Subtract
80      line = MMUSearchOnMemories(add1, machine); /* Searching block on memories
          */
81      word1 = line->block.words[add1.word];
82  #ifdef PRINT_LOG
83      printf("  > SUB BLOCK[%d.%d.%d](%4d)\n", line->cacheHit, add1.block, add1.
          word, line->block.words[add1.word]);
84  #endif
85
86      line = MMUSearchOnMemories(add2, machine); /* Searching block on memories
          */
87      word2 = line->block.words[add2.word];
88  #ifdef PRINT_LOG
89      printf(" - BLOCK[%d.%d.%d](%4d)\n", line->cacheHit, add2.block, add2.word,
           line->block.words[add2.word]);
90  #endif
91
92      line = MMUSearchOnMemories(add3, machine); /* Searching block on memories
          */
```

```
93  #ifdef PRINT_LOG
94      printf(" > BLOCK[%d.%d.%d](%4d|\n", line->cacheHit, add3.block, add3.word,
            line->block.words[add3.word]);
95  #endif
96
97      line->updated = true;
98      line->block.words[add3.word] = word2 - word1;
99
100 #ifdef PRINT_LOG
101     printf("%4d).\n", line->block.words[add3.word]);
102 #endif
103
104     break;
105
106   default:
107     printf("Invalid instruction.\n");
108   }
109 #ifdef PRINT_INTERMEDIATE_RAM
110   printMemories(machine);
111 #endif
112
113   if (interrupted == true)
114     machine->interruptions -= 1;
115
116   while (machine->interruptions > 0)
117   {
118 #ifdef PRINT_INTERRUPTIONS
119     printf("> Fix interruption %d\n", machine->interruptions);
120 #endif
121     executeInstruction(machine, machine->interruptions - 1, true); // avoid
          generate infinite interruptions
122   }
123 }
```

Código 3: executeInstruction

O código apresenta uma interrupção aleatória que pode ocorrer na instrução de soma (opcode 1) e subtração (opcode 2). Se uma interrupção ocorrer durante a execução da instrução de movimentação (opcode 0), a instrução é interrompida e o valor é salvo na RAM.

Ao receber uma interrupção, o programa gera um número aleatório de interrupções adicionais (até 10) e executa uma instrução aleatória para cada uma delas. Isso é feito através de um loop for que percorre a quantidade de interrupções geradas.

Após a execução da instrução, o código verifica se ocorreu alguma interrupção e decrementa a quantidade de interrupções restantes. Se ainda houver interrupções pendentes, o código executa a instrução na posição de memória correspondente à quantidade de interrupções pendentes. Isso é feito através de um loop while que executa a instrução para cada interrupção restante.

Em resumo, o código implementa uma funcionalidade de interrupção aleatória que pode ocorrer durante a execução de algumas instruções e pode gerar interrupções adicionais que devem ser tratadas antes de prosseguir com a execução do programa.

2.5 Memória Externa

Para a implementação correta pedida pelo professor, seguindo os seus critérios, foram feitas a 'file.c' e 'file.h'.

```h
#pragma once
#include "memory.h"

void saveStorage(Storage);

void readBinaryFile(Storage *);

MemoryBlock *getBlockFromFile(int);

int appendBlockToBinary(Storage *);
```

Código 4: file.h

```c
#include "file.h"
#include <stdio.h>

void saveStorage(Storage storage)
{
    FILE *fp = fopen("storage.bin", "wb");

    // check if file was opened successfully
    if (fp == NULL)
    {
        printf("Error opening file\n");
        return;
    }

    // write the size of the storage to the file
    fwrite(&storage.size, sizeof(int), 1, fp);

    // write each memory block to the file

    fwrite(storage.blocks, sizeof(MemoryBlock) * storage.size, 1, fp);

    // close file
    fclose(fp);
}

void readBinaryFile(Storage *storage)
{
    FILE *fp = fopen("storage.bin", "rb"); // Open file in "read binary" mode
    if (fp == NULL)
    {
        perror("Error opening file");
    }
    else
    {

        // Get the file size
        fseek(fp, 0L, SEEK_END);
        fseek(fp, 0L, SEEK_SET);

        // Calculate the number of blocks in the file
        fread(&storage->size, sizeof(int), 1, fp);
        printf("%d", storage->size);

        // Allocate memory for the Storage struct and its blocks
```

```
45          storage->blocks = malloc(sizeof(MemoryBlock) * storage->size);
46
47          // Read the blocks from the file into the Storage struct
48
49          fread(storage->blocks, sizeof(MemoryBlock) * storage->size, 1, fp);
50
51          // Close the file
52          fclose(fp);
53      }
54  }
55
56  MemoryBlock *getBlockFromFile(int blockIndex)
57  {
58      // open file for reading in binary mode
59      FILE *fp = fopen("storage.bin", "rb");
60
61      // check if file was opened successfully
62      if (fp == NULL)
63      {
64          printf("Error opening file\n");
65          return NULL;
66      }
67
68      // seek to the position in the file where the block is stored
69      fseek(fp, sizeof(int) + blockIndex * sizeof(MemoryBlock), SEEK_SET);
70
71      // allocate memory for the block
72      MemoryBlock *block = malloc(sizeof(MemoryBlock));
73
74      // read the block from the file
75      fread(block->words, sizeof(int), WORDS_SIZE, fp);
76
77      // close file
78      fclose(fp);
79
80      return block;
81  }
82
83  int appendBlockToBinary(Storage *storage)
84  {
85      FILE *fp = fopen("storage.bin", "ab"); // Open file in "append binary" mode
86      if (fp == NULL)
87      {
88          perror("Error opening file");
89          return -1;
90      }
91
92      // Write the new block to the end of the file
93      fwrite(storage->blocks[storage->size - 1].words, sizeof(int), 4, fp);
94
95      // Close the file
96      fclose(fp);
97
98      return 0;
99  }
```

Código 5: file.c

A função addToStorage recebe dois parâmetros: o nome do arquivo em que os dados serão escritos e um ponteiro para uma estrutura MemoryBlock, que contém um array de 'words size' inteiros. A função abre o arquivo em modo de "append"(adicionar ao final do arquivo) utilizando a função fopen, e verifica se houve algum erro durante a abertura do arquivo

utilizando a função perror. Em seguida, a função percorre o array de inteiros contido na estrutura MemoryBlock e utiliza a função fprintf para escrever cada um dos inteiros no arquivo. Finalmente, a função fecha o arquivo utilizando a função fclose.

A função readStorage recebe como parâmetro o nome do arquivo a ser lido. A função abre o arquivo em modo de "leitura"utilizando a função fopen, e verifica se houve algum erro durante a abertura do arquivo utilizando a função perror. Em seguida, a função percorre o arquivo caractere por caractere utilizando a função fgetc, imprimindo cada caractere na tela. Finalmente, a função fecha o arquivo utilizando a função fclose.

2.6 Novas funções

Ademais foram criadas novas funções para o gerenciamento correto da memória, tanto da RAM quanto da Externa (Storage).

```
int ramMapping(int address, int word, RAM *RAM)
{
    int pos = 0;
    for (int i = 0; i < RAM->size; i++)
    {
        if (RAM->blocks[i].tag == (address % RAM->size))
        {
            for (int j = 0; j < WORDS_SIZE; j++)
            {
                if (RAM->blocks[i].words[j] == word)
                    pos = i;
            }
        }
    }

    return pos;
}
```

Código 6: ramMapping

Em resumo, a função está mapeando o endereço de memória para o bloco correspondente na estrutura "RAM"e procurando a palavra de dados nesse bloco.

```
void storageToRAM(RAM *RAM, Storage *storage, int tag)
{
    int ramPos, storagePos;

    printf("1231");
    for (int i = 0; i < storage->size; i++)
    {
        if (storage->blocks[i].tag == tag)
        {
            storagePos = i;
            break;
        }
    }

    ramPos = tag % RAM->size;
```

```
16      MemoryBlock aux;
17      aux = storage->blocks[storagePos];
18      storage->blocks[storagePos] = RAM->blocks[ramPos];
19      RAM->blocks[ramPos] = aux;
20
21      saveStorage(*storage);
22  }
```

Código 7: storageToRAM

Em resumo, a função está movendo um bloco de dados da estrutura "Storage"para a estrutura "RAM"com base na tag fornecida como parâmetro.

3 Conclusão

3.1 Considerações finais

Diante da análise do PDF e da implementação das funções nos arquivos .c correspondentes, o grupo conseguiu desenvolver as funcionalidades requeridas no projeto. Iniciando pelas funções mais simples e seguindo para as mais complexas, foi possível consolidar a compreensão do problema e garantir a correta integração das funcionalidades no código final. Além disso, a organização dos arquivos permitiu uma melhor estruturação do projeto e uma leitura mais clara do código. Ao final do projeto, os resultados esperados foram alcançados e o grupo pôde comprovar que a solução desenvolvida é capaz de simular uma MMU, com gerenciamento de memórias RAM, cache e externa, além de um sistema funcional de interrupções. Dessa forma, concluímos que o projeto foi bem sucedido e que a colaboração do grupo foi fundamental para atingirmos os objetivos propostos.