Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático III

BCC266 - Organização de Computadores

Vitor Oliveira Diniz
Jéssica Machado

Professor: Pedro Silva

Ouro Preto
21 de março de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

O HD é um dispositivo de armazenamento de dados que armazena e recupera dados digitais usando armazenamento magnético. Os dados são acessados de maneira aleatória, o que significa wue blocos individuais de de dados que podem ser armazenados e recuperados em qualquer ordem, além de ser um armazenamento que retém os dados mesmo quando a máquina é desligada.

1.1 Especificações do problema

Para este trabalho prático, deveríamos, a partir do TP2 que já realizado, implementar o HD se caso palavra de um bloco de memória não for encontrada na RAM, portanto a memória externa vai ser considerada como último nível. Além disso deveriamos adicionar um tratador de interrupções, que verificaria se há alguma interrupção a ser tratada e deve parar a execução do código atual para tratar a mesma.

1.2 Considerações Iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Visual Studio Code LaTeXWorkshop.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

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

1.4 Especificações da máquina

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

- Processador: Ryzen 7-5800H.
- Memória RAM: 16 Gb.
- Sistema Operacional: Arch Linux x86_64.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc main.c -c -Wall
gcc mmu.c -c -Wall
gcc memory.c -c -Wall
gcc instruction.c -c -Wall
gcc cpu.c -c -Wallnewline gcc generator.c -c -Wall
gcc main.o mmu.o memory.o instruction.o cpu.o generator.o -o exe -g
```

Usou-se para a compilação as seguintes opções:

- *-Wall*: para mostrar todos os possível *warnings* do código.
- *-c*: Para compilar o arquivo sem linkar os arquivos para obtermos um arquivo do tipo objeto.

- *-o*: Compilar para um arquivo do tipo output (saída).

Para a execução do programa basta digitar um dos exemplos:

```
./exe random [TAMANHO DA RAM] [L1] [L2] [L3] [TAMANHO DO HD]
```

2 Desenvolvimento

Seguindo as boas práticas de programação, implementamos o HD particionada em blocos e simulamos o mapeamento associativo para a troca das mesmas com a memória RAM. Usamos as políticas LFU (Last Frequently Used) e LRU (Last Recently Used) como métodos de otimização.

2.1 Operações

A seguir entraremos em detalhe sobre as principais funções utilizadas no programa e as coisas que implementamos.

2.1.1 Médodos de mapeamento

Define o tipo de método que vai ser usado para a saída dos resultados.

```
1
2       // 1 MAPEAMENTO DIRETO
3       // 2  LRU (Least Recently Used)
4       // 3  LFU (Least Frequently Used)
5
6       #define SUBSTITUTION_METHOD 3
```

Código 1: Definição do tipo de método

2.1.2 TAD File e TAD HD

Criamos um TAD File que salva o nome do arquivo na variável fileName, e uma variável file do tipo FILE. E no TAD HD criamos o int size para receber o tamanho do hd e um vetor de Files.

```
1
2  typedef struct {
3      char fileName[FILE_STR_TAM]; //salvar o nome do arquivo
4      FILE *file;
5  } File;
6
7  typedef struct {
8      int size;
9      File *files; //arquivos
10 } HD;
```

Código 2: TAD Machine

2.1.3 TAD Machine

Adicionamos o HD hd, hitRAM e o hitHd (que conta quantas vezes processador achou o dado na ram e no HD) e o missRam (que conta quantas vezes o processador precisou buscar o dado no HD). O int interruption foi criado para garantir que haja apenas uma interrupção ocorrendo, para que não tenhamos uma recursão infinita na nossa máquina.

```
1
2  typedef struct {
3      Instruction* instructions;
4      RAM ram;
5      Cache l1; // cache L1
6      Cache l2; // cache L2
7      Cache l3; // cache L3
8      HD hd;
9      int hitL1, hitL2, hitL3, hitRAM, hitHD;
10     int missL1, missL2, missL3, missRAM;
11     int totalCost;
12     int interruption;
```

```
} Machine;
```

Código 3: TAD Machine

2.1.4 startHD e stopHD

Na função startHD modificamos a variável size que está no hd->size para o size passado por parâmetro e alocamos dinamicamente o vetor de arquivos. Na estrutura de repetição for, criamos os arquivos e inicializamos eles com valores aleatórios. Na função stopHD simplesmente liberamos o vetor de arquivos que estava presente no hd.

```
void startHD(HD* hd, int size){

    hd->size = size;
    hd->files = malloc (size * sizeof(File));

    for(int i = 0; i < size; i++){
        sprintf(hd->files[i].fileName, "0x0%d.txt", i);
        // strcpy(, nomeArquivo);
        hd->files[i].file = fopen(hd->files[i].fileName, "w");

        fprintf(hd->files[i].file, "%d\n", WORDS_SIZE);


        for (int j=0;j<WORDS_SIZE;j++){

            fprintf(hd->files[i].file, "%d\n", rand() % 100);

        }

        fclose(hd->files[i].file);
    }

}

void stopHD(HD* hd){

    free(hd->files);

}
```

Código 4: Funçao run

2.1.5 start

Nessa função, iniciamos o valor do hitRam, hitHD e missRAM como 0. Além de adicionarmos o a função startHD.

```
    void start(Machine* machine, Instruction* instructions, int* memoriesSize)
        {
        startRAM(&machine->ram, memoriesSize[0]);
        startCache(&machine->l1, memoriesSize[1]);
        startCache(&machine->l2, memoriesSize[2]);
        startCache(&machine->l3, memoriesSize[3]);
        startHD(&machine->hd, memoriesSize[4]);
        machine->interruption = 0;

        machine->instructions = instructions;
        machine->hitL1 = 0;
        machine->hitL2 = 0;
        machine->hitL3 = 0;
        machine->hitRAM = 0;
```

```
        machine->hitHD = 0;

        machine->missL1 = 0;
        machine->missL2 = 0;
        machine->missL3 = 0;
        machine->hitRAM = 0;
        machine->totalCost = 0;
    }
```

Código 5: Funçao run

2.1.6 run

Inicializa o contador PC com valor 0 e, enquanto o opcode de machine->instructions[PC] for diferente de -1 (condição de parada), printamos a quantidade de vezes que o processador achou (hit) o dado na L1, L2, L3, RAM e no HD, a quantidade de vezes que o processador precisou buscar o dado no HD (miss), e o custo total.

```

void run(Machine* machine) {
    int PC = 0; // Program Counter
    while(machine->instructions[PC].opcode != -1) {
        executeInstruction(machine, PC++);
        printf("\tL1:(%6d, %6d) | L2:(%6d, %6d) | L3:(%6d, %6d) | RAM:(%6d, %6d) | HD:(%6d) | COST: %d\n",
            machine->hitL1, machine->missL1,
            machine->hitL2, machine->missL2,
            machine->hitL3, machine->missL3,
            machine->hitRAM, machine->missRAM,
            machine->hitHD,
             machine->totalCost);
    }
}
```

Código 6: Funçao run

2.1.7 printMemories

Aqui adicionamos uma linha a mais na tabela para representar cada bloco presente no HD. Printa as informações da RAM, se está atualizado é a cor verde e, se não, cor vermelha.

```
    void printMemories(Machine* machine) {
    printf("\x1b[0;30;47m     ");
    printc("HD", WORDS_SIZE * 8 - 1);
    printc("RAM", WORDS_SIZE * 8 - 1);
    printc("Cache L3", WORDS_SIZE * 8 + 6);
    printc("Cache L2", WORDS_SIZE * 8 + 6);
    printc("Cache L1", WORDS_SIZE * 8 + 6);
    printf("\x1b[0m\n");

    MemoryBlock hd_block;

    for (int i=0;i<machine->hd.size;i++) {
        printf("\x1b[0;30;47m%5d|\x1b[0m", i);
        hd_block = readHDblocks(&machine->hd.files[i]);

        for (int j=0;j<WORDS_SIZE;j++)
            printf(" %5d |", hd_block.words[j]);


        if(i < machine->ram.size){
            printf("|");
```

```
            for (int j=0;j<WORDS_SIZE;j++)
                printf(" %5d |", machine->ram.blocks[i].words[j]);

            if (i < machine->l3.size) {
                printf("|");
                printcolored(machine->l3.lines[i].tag, machine->l3.lines[i].
                    updated);
                for (int j=0;j<WORDS_SIZE;j++)
                        printf(" %5d |", machine->l3.lines[i].block.words[j]);

                if (i < machine->l2.size) {
                    printf("|");
                    printcolored(machine->l2.lines[i].tag, machine->l2.lines[i
                        ].updated);
                    for (int j=0;j<WORDS_SIZE;j++)
                        printf(" %5d |", machine->l2.lines[i].block.words[j]);
                    if (i < machine->l1.size) {
                        printf("|");
                        printcolored(machine->l1.lines[i].tag, machine->l1.
                            lines[i].updated);
                        for (int j=0;j<WORDS_SIZE;j++)
                            printf(" %5d |", machine->l1.lines[i].block.words[
                                j]);
                    }
                }
            }

        }

        printf("\n");
    }


}
```

Código 7: Função printMemories

2.1.8 readHDblocks

Lê os blocos de HD que estão salvos em um arquivo.txt e retorna uma palavra do tamanho definido no programa.

```
1
2  MemoryBlock readHDblocks(File *hd){
3
4      MemoryBlock block;
5
6      hd->file = fopen(hd->fileName, "r");
7
8      int words_size;
9      fscanf(hd->file, "%d", &words_size);
10
11     for(int i = 0; i < WORDS_SIZE; i++){
12         fscanf(hd->file, "%d", &block.words[i]);
13     }
14
15     fclose(hd->file);
16     return block;
17
18 }
```

Código 8: startCache

2.1.9 memoryCacheMapping

Ela é responsável por checar qual tipo de mapeamento estamos utilizando e, dependendo do mapeamento, iremos aplicar uma política diferente. No caso do mapeamento direto, retornamos o possível endeeço da cache e a verificação de sua existência é feita no MMUSearchOnMemorys. No caso da LRU, percorremos o vetor sempre checando qual endereço está a mais tempo na cache e salvando seu índice, caso encontremos o endereço que procuramos retornamos seu índice, se a substituição vai acontecer ou não depende do MMUSearchOnMemorys. No LFU acontece a mesma coisa que o LRU, apenas mudando a condição de checagem para o bloco menos usado.

```
1
2      int memoryCacheMapping(int address, Cache* cache) {
3
4      int index = 0;
5
6      switch(SUBSTITUTION_METHOD){
7          //DIRECT MAPPING
8          case 1:
9              return address % cache->size;
10             break;
11         //LRU METHOD (Least Recently Used)
12         case 2:
13             cache->lines[0].timeOnCache++;
14             for( int i = 0; i < cache->size; i++){
15
16                 cache->lines[i].timeOnCache++;
17
18                 if(cache->lines[i].timeOnCache > cache->lines[index].
                       timeOnCache)
19                     index = i;
20
21                 if(cache->lines[i].tag == address)
22                     return i;
23
24             }
25             return index;
```

```
            break;

        //LFU METHOD (Least Frequently Used)
        case 3:
            for( int i = 0; i < cache->size; i++){
                if(cache->lines[i].timesUsed < cache->lines[index].timesUsed)
                    index = i;

                if(cache->lines[i].tag == address)
                    return i;
            }

            return index;
            break;
    }

    return address % cache->size;
}
```

Código 9: Função memoryCacheMapping

2.1.10 updateMachineInfos

Modificamos o valor do hitHD e do missRAM. Fazemos a alteração do custo total da máquina, tempo na cache e quantas vezes que foi usada.

```

void updateMachineInfos(Machine* machine, Line* line) {

    switch (line->cacheHit) {
        case 1:
            machine->hitL1 += 1;
            break;

        case 2:
            machine->hitL2 += 1;
            machine->missL1 += 1;
            break;

        case 3:
            machine->hitL3 += 1;
            machine->missL1 += 1;
            machine->missL2 += 1;
            break;

        case 4:
            machine->hitRAM += 1;
            machine->missL1 += 1;
            machine->missL2 += 1;
            machine->missL3 += 1;
            break;

        case 5:
            machine->hitHD += 1;
            machine->missRAM += 1;
            machine->missL1 += 1;
            machine->missL2 += 1;
            machine->missL3 += 1;
            break;
    }

    line->timeOnCache = 0;
```

```
37      line->timesUsed++;
38
39      machine->totalCost += line->cost;
40  }
```

Código 10: Função updateMachineInfos

2.1.11 MMUSearchOnMemorys

Inicialmente, essa função pega a possível posição do bloco que queremos na CACHE e conferimos se o que queremos está realmente na RAM. Se estiver, atualizamos o cache hit referente aonde ele se encontra, o custo de acesso da memória, a nossa máquina e retornamos a linha que o endereço se encontra. Se não encontrarmos o endereço na cache, fazemos as movimentações necessárias entre as memórias, levando em consideração o mapeamento escolhido e, caso precise, levamos o endereço da cache para RAM. Como proposto para o trabalho prático deveriamos ter adicionado uma verificação a mais. Caso o bloco desejado não esteja presente na RAM, deveriamos pegar essa informação do HD, que no nosso caso é simulado por diferentes arquivos do tipo txt, cada um com o nome referente ao bloco. Depois de pegarmos o valor no HD, deveriamos salvar ele na cache, sempre obedecendo o mapeamento escolhido.

```
1
2       Line* MMUSearchOnMemorys(Address add, Machine* machine) {
3           // Strategy => write back
4
5           int l1pos = memoryCacheMapping(add.block, &machine->l1);
6           int l2pos = memoryCacheMapping(add.block, &machine->l2);
7           int l3pos = memoryCacheMapping(add.block, &machine->l3);
8
9
10          Line* cache1 = machine->l1.lines;
11          Line* cache2 = machine->l2.lines;
12          Line* cache3 = machine->l3.lines;
13
14          MemoryBlock* RAM = machine->ram.blocks;
15
16          if (cache1[l1pos].tag == add.block) {
17              cache1[l1pos].cost = COST_ACCESS_L1;
18              cache1[l1pos].cacheHit = 1;
19          } else if (cache2[l2pos].tag == add.block) {
20              /* Block is in memory cache L2 */
21              cache2[l2pos].tag = add.block;
22              cache2[l2pos].updated = false;
23              cache2[l2pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2;
24              cache2[l2pos].cacheHit = 2;
25
26              updateMachineInfos(machine, &(cache2[l2pos]));
27
28              return &(cache2[l2pos]);
29          } else if (cache3[l3pos].tag == add.block){
30              /* Block is in memory cache L3 */
31              cache3[l3pos].tag = add.block;
32              cache3[l3pos].updated = false;
33              cache3[l3pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 +
                    COST_ACCESS_L3;
34              cache3[l3pos].cacheHit = 3;
35
36              updateMachineInfos(machine, &(cache3[l3pos]));
37              return &(cache3[l3pos]);
38
39          } else {
40              /* Block only in memory RAM, need to bring it to cache and
                    manipulate the blocks */
41              //fazer o mapeamento para decidir quem tirar da ram.
42
43              if (!canOnlyReplaceBlock(cache1[l1pos])) {
44                  /* The block on cache L1 cannot only be replaced, the memories
                        must be updated */
45                  if (!canOnlyReplaceBlock(cache2[l2pos])){
```

```
46                      /* The block on cache L2 cannot only be replaced, the
                            memories must be updated */
47                      if(!canOnlyReplaceBlock(cache3[l3pos])){
48                          /* The block on cache L2 cannot only be replaced, the
                                memories must be updated */
49                          RAM[cache3[l3pos].tag] = cache3[l3pos].block;
50                      }
51                      cache3[l3pos] = cache2[l2pos];
52                      cache3[l3pos].timeOnCache = 0;
53
54
55                  }
56
57                  cache2[l2pos] = cache1[l1pos];
58                  cache2[l2pos].timeOnCache = 0;
59
60              }
61
62
63              cache1[l1pos].block = RAM[add.block];
64              cache1[l1pos].timesUsed = 1;
65              cache1[l1pos].tag = add.block;
66              cache1[l1pos].updated = false;
67              cache1[l1pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 +
                    COST_ACCESS_L3 + COST_ACCESS_RAM;
68              cache1[l1pos].timeOnCache = 0;
69              cache1[l1pos].cacheHit = 4;
70
71
72
73          }
74          updateMachineInfos(machine, &(cache1[l1pos]));
75          return &(cache1[l1pos]);
76      }
```

Código 11: Função MMUSearchOnMemorys

2.2 Função Main

Na função main adicionamos mais um espaço a memoriesSize para o HD

```
int main(int argc, char**argv) {

    srand(1507);   // Inicializacao da semente para os numeros aleatorios.

    if (argc != 6) {
        printf("Numero de argumentos invalidos! Sao 6.\n");
        printf("Linha de execucao: ./exe TIPO_INSTRUCAO [TAMANHO_RAM|
            ARQUIVO_DE_INSTRUCOES] TAMANHO_L1 TAMANHO_L2 TAMANHO_L3\n");
        printf("\tExemplo 1 de execucao: ./exe random 10 2 4 6\n");
        printf("\tExemplo 2 de execucao: ./exe file arquivo_de_instrucoes.
            txt\n");
        return 0;
    }

    int memoriesSize[4];
    Machine machine;
    Instruction *instructions;

    memoriesSize[1] = atoi(argv[3]);
    memoriesSize[2] = atoi(argv[4]);
    memoriesSize[3] = atoi(argv[5]);


    if (strcmp(argv[1], "random") == 0) {
        memoriesSize[0] = atoi(argv[2]);
        instructions = generateRandomInstructions(memoriesSize[0]);
    } else if (strcmp(argv[1], "file") == 0) {
        instructions = readInstructions(argv[2], memoriesSize);
    }
    else {
        printf("Invalid option.\n");
        return 0;
    }

    printf("Starting machine...\n");
    start(&machine, instructions, memoriesSize);
    if (memoriesSize[0] < 10)
        printMemories(&machine);
    run(&machine);
    if (memoriesSize[0] < 10)
        printMemories(&machine);
    stop(&machine);
    printf("Stopping machine...\n");
    return 0;
}
```

Código 12: Main

3 Impressões Gerais

Primeiramente, nos reunimos no discord para a leitura e compreensão do documento disponibilizado para a realização do trabalho. Foi bem difícil compreender como era pra ser feito esse trabalho, a parte de realizar a interrupção foi ainda mais díficil pois não tivemos conteúdo o suficiente para nos basear e "ter um norte"para começar o trabalho. A simulação de um HD também trouxe dificuldades, faltou uma ideia boa de como ocorreria essa implementação, e acabamos ficando perdidos sobre a implementação. Vale lembrar que infelizmente faltou um pouco de tempo, considerando que esse periodo da UFOP foi mais corrido que o normal, havendo uma sobrecarga de atividades. Por isso devido a complexidade das funções a serem implementadas e pela falta de material para pesquisa acabamos deixando a desejar em alguns aspectos como a implementação do MMUSearchOnMemorys com o HD.

4 Análise

Tivemos uma ideia boa de como deveria ocorrer a implementação do HD, chegamos a realizar a montagem inicial do TAD e dos parâmetros necessários para a implementação do HD. Além disso criamos os arquivos para simular o acesso em memória externa, assim como ter uma ideia da lentidão da memória externa(comumente conhecida como SWAP em sistemas UNIX), juntamente com um tratador de interrupções, que se prova extremamente util, especialmente em sistemas operacionais, porque sempre haverá algumas tarefas com prioridades maiores a serem executadas.

4.1 Análise de Complexidade

Na função memoryCacheMapping, a sua complexidade vai depender do método de mapeamento escolhido. Para o mapeamento direto, como não percorremos o vetor, já que a posição na cache depende unicamente do final de seu endereço, sua complexidade será $\mathcal{O}(1)$.

Na política LRU, na função memoryCacheMapping, percorremos o vetor para encontrar o bloco que está a mais tempo sem ser utilizado, por isso teremos uma complexidade de $\mathcal{O}(n)$.

Na política LFU, na memoryCacheMapping, também percorremos o vetor para encontrar o bloco que está a mais tempo sem ser utilizado, por isso teremos uma complexidade de $\mathcal{O}(n)$.

5 Conclusão

Dentro do possível o trabalho apresentou uma didática interessantes, nos fornecendo um jeito mais palpável de aprender conceitos intrisicos a um sistema operacional. Porem algumas coisas deixaram a desejar, sendo a principal o material fornecido para podermos realizar o trabalho e o prazo, que entendemos que não seja culpa apenas do professor, mas também do calendário acadêmico. Com isso a parte que conseguimos finalizar do trabalho nos deu uma excelente ideia do funcionamento e como computadores são organizados desde o conceito mais básico, como uma simples instrução de soma, até ao conceito de interupções e armazenamento, tanto interno quanto externo. Sendo uma disciplina extremamente proveitosa e que irá agregar para a nossa bagagem como profissionais.