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 I

Felipe Braz Marques
Matheus Peixoto Ribeiro Vieira
Pedro Henrique Rabelo Leão de Oliveira

Professor: Pedro Henrique Lopes Silva

Ouro Preto
18 de março de 2023

Sumário

1 Introdução **1**
1.1 Considerações iniciais . . . 1
1.2 Ferramentas utilizadas . . . 1
1.3 Especificações da máquina . . . 1
1.4 Instruções de compilação e execução . . . 1

2 Desenvolvimento **2**
2.1 constants.h . . . 2
2.2 LFU . . . 2
2.3 FIFO . . . 2
2.4 RANDOM . . . 2
2.5 Interrupções . . . 2
2.6 Implementação do Disco . . . 3

3 Análise dos resultados **6**

4 Impressões Gerais e Considerações Finais **8**

Lista de Códigos Fonte

1 Código no OPCODE 0 . . . 2
2 Gerar uma probabilidade de ocorrer uma interrupção . . . 2
3 Tratador de interrupções . . . 3
4 Leitura dos dados para salvar na RAM . . . 3
5 memoryRAMMApping . . . 4
6 Procurando um dado na RAM ou no disco . . . 4
7 Cálculo para a porcentagem de cache hit . . . 6

1 Introdução

Para este trabalho será entregue um código em C, no qual estará simulado a memória externa, disco, de um computador, e o tratamento de interrupções, além do funcionamento da memória cache de um computador, acessando os níveis L1, L2, L3 e RAM.

1.1 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: Overleaf LaTeX. [1]

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *GCC*: versão 11.3.0.
- *Valgrind*: versão 3.18.1.

1.3 Especificações da máquina

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

- Processador: Intel i5-9300H.
- Memória RAM: 16Gb.
- Sistema Operacional: Ubuntu 22.04.1 LTS.

1.4 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc -c cpu.c -Wall;
gcc -c generator.c -Wall;
gcc -c instruction.c -Wall;
gcc -c main.c -Wall;
gcc -c memory.c -Wall;
gcc -c mmu.c;
gcc cpu.o generator.o instruction.o main.o memory.o mmu.o -o exe -lm
```

Usou-se para a compilação as seguintes opções:

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

Para a execução do programa basta digitar:

```
./exe instrucao-desejada tam-RAM tam-L1 tam-L2 tam-L3
Exemplo: ./exe random 32 4 8 10
```

[1] Disponível em https://www.overleaf.com/

2 Desenvolvimento

2.1 constants.h

Este aquivo .h, contém tem todos os defines necessários. Como por exemplo o custo de acesso das caches, e da RAM. Por meio dele também é possível escolher o tipo de mapeamento que vai executar. Se for selecionado o mapeamento associativo ou associativo por conjunto, deve selecionar também o método que a máquina irá usar para substituir as informações nas caches.

2.2 LFU

O LFU utiliza em seu funcionamento na cache um contador na struct Line que começa como 0 e tem o seu valor atualizado sempre que ocorrer um cache hit na sua linhacontando, assim, quantas vezes ele foi utilizado no sistema. Sendo removido, então, aquele bloco que tem o menor valor.

Para o funcionamento na RAM, como ela agora pode ter cache miss, também terão valores que deverão ser trocados para um do disco. Assim, o seu contador foi definido na struct MemoryBlock, pois a RAM tem o seu tamanho e os blocos.

Dessa forma, sempre que um bloco for utilizado, ele terá o seu contador incrementado, e, para remover, ele procurará o menor valor. Assim, o seu funcionamento ficou muito parecido com o LFU para a Cache, tanto que todo o seu funcionamento ocorre da mesma forma, somente em funções adaptadas para a RAM, como a adicionaMaisUmNoContadorRAM() e reiniciaContadorRam().

2.3 FIFO

No método FIFO (First in First Out), é utilizada a mesma variável de contador, entretanto, partindo do conceito de que o primeiro a entrar na RAM, será o primeiro a sair, todos os contadores são incrementados com mais 1 sempre que houver uma pesquisa de memória. Dessa forma, quando é necessário remover um valor substituí-lo por outro utilizando a chamada da função blockFromRAMWillBeRemoved(), o item mais antigo que está ali, terá o maior valor em seu contador, representando ser o primeiro a ter entrado, e , assim, ele será o primeiro da "fila"para ser removido.

Assim como no LFU, o seu funcionamento é o mesmo do que foi implementado para a cache.

2.4 RANDOM

Para o método random, o valor escolhido para ser removido é aleatório, assim, quando é procurado qual o bloco que irá sair, tanto para as caches quanto para a RAM, é gerado um valor aleatório para tal.

2.5 Interrupções

Para gerar as interrupções, foi adotada a seguinte estratégia: Sempre que o opcode da instrução atual for 0, será chamada uma função que calculará a probabilidade de gerar uma interrupção. Por padrão, a chance é de 5%, porém pode ser alterada no arquivo constants.h ao modificar o define INTERRUPTION_PROBABILITY.

```
if(verificarGeracaoInterrupcao())
    tratador_de_interrupcoes(machine);
```

Código 1: Código no OPCODE 0

```
int verificarGeracaoInterrupcao(){
    //Cria uma probabilidade da interrupcao ocorrer
    int valor = (rand() % 100) + 1; //Gera um aleatorio entre 1 e 100
    if(valor <= INTERRUPTION_PROBABILITY) return 1;
    else  return 0;
}
```

Código 2: Gerar uma probabilidade de ocorrer uma interrupção

Quando é chamado o tratador de interrupções, a máquina será passada por referência para a função. Pois, todas as suas instruções serão copiadas para um vetor auxiliar. Em seguida, serão geradas instruções aleatórias para a interrupção, porém, elas serão somente para soma ou subtração, evitando que ocorra uma interrupção dentro de outra.

A geração das instruções de interrupção serão feitas no arquivo generator.c, onde o número de instruções será dado pelo define NUM_INTERRUPTIONS.

Após as novas instruções serem geradas, elas irão para as instruções da machine, e depois será levado para a função run, criando um tipo de recursividade. Nesse momento, as instruções dessa interrupção serão executadas.

Com o fim da execução das interrupções, o vetor de instruções originais voltará para a machine e todo o funcionamento continuará como estava anteriormente.

```
void tratador_de_interrupcoes(Machine *machine){
    int qtdInstrucoes; //armazena a quantidade de instrucoes do vetor de
        instrucoes da maquina
    for(qtdInstrucoes = 0; machine->instructions[qtdInstrucoes].opcode != -1;
        qtdInstrucoes++);
    qtdInstrucoes++;//incrementando para as alocacoes
    //Salvando as instrucoes originais em um vetor separado
    Instruction * instrucoesOriginais = malloc(qtdInstrucoes * sizeof(
        Instruction));
    for(int i = 0; i<qtdInstrucoes; i++)
        instrucoesOriginais[i] = machine->instructions[i];
    //passando para maquina as instrucoes da interrupcao, geradas
        randomicamente
    free(machine->instructions);
    machine->instructions = gerarInstrucoesInterrupcoes();
    //Executar as instrucoes de interrupcoes
    run(machine);
    //Voltar com as instrucoes originais para a machine
    free(machine->instructions);
    machine->instructions = malloc(qtdInstrucoes *sizeof(Instruction));
    for(int i = 0; i < qtdInstrucoes; i++)
        machine->instructions[i] = instrucoesOriginais[i];
    free(instrucoesOriginais);
}
```

Código 3: Tratador de interrupções

2.6 Implementação do Disco

Assim que o programa iniciar, é chamado a função inicaDisco() que irá salvar n valores definidos pelo DISK_SIZE, onde os dados serão salvos no arquivo disk.dat.

Em seguida, quando for chamada a função start(), a RAM será incializada com os arquivos sendo puxados do arquivo do disco, porém como a RAM possui um tamanho menor que o disco, serão lidos somente o suficiente para preencher a RAM.

```
void startRAM(RAM * ram, int size){
    FILE * arquivo = fopen("disk.dat", "rb");
    if(arquivo == NULL){
        printf("Erro ao ler o arquivo\n");
        exit(1);
    }
    ram->blocks = (MemoryBlock *)calloc(size, sizeof(MemoryBlock));
    ram->size = size;
    for (int i = 0; i < size; i++){
        fread(&ram->blocks[i].enderecoEmDisco, sizeof(int), 1, arquivo);
        for (int j = 0; j < WORDS_SIZE; j++)
            fread(&ram->blocks[i].words[j], sizeof(int), 1, arquivo);
```

```
    }
    fclose(arquivo);
}
```

Código 4: Leitura dos dados para salvar na RAM

Em seguida, na função run e execute instructions, ocorrerá a chamada à MMUSearchOnMemorys(), que é nela onde ocorrerá a procura de dados nas caches e na RAM, a partir da memoryCacheMapping e da memoryRAMMapping, que possuem um funcionamento similar. Ou seja, ocorrerá um mapeamento associativo varrendo toda a RAM à procura de uma chave que indica o valor procurado, sendo, neste caso, a enderecoEmDisco.

```
int memoryRAMMapping(int address, RAM *ram){
    for(int i = 0; i < ram->size; i++){
        if(address == ram->blocks[i].enderecoEmDisco) //caso o endereco
            passado seja igual a tag do bloco da ram, retorna a posicao no
            vetor de memoryblocks
            return i;
    }
    return 0;
}
```

Código 5: memoryRAMMApping

Com os valores de endereço, o programa segue à procura de onde pode encontrar o item, verificando em todas as caches e, depois, na RAM, a partir do endereco do bloco. Caso encontre na RAM, esse bloco será, então, movido para a cache L1 e será enviado um hit para a updateMachineInfos. Porém, em um eventual momento onde nenhum dado seja encontrado, a partir da função verificaRamDisco(), caso não esteja na RAM também, o item é procurado no disco, e transferido para RAM, para que assim, o item se faça presente nela a partir de então.

```
int verificaRamDisco(MemoryBlock *ram, Address address, int *ramPos, RAM *
    ramMachine){
    //Caso em que o endereco esta na ram
    if(ramMachine->blocks[*ramPos].enderecoEmDisco == address.block){
        ramMachine->blocks[*ramPos].count++; //Acresecentando mais um no
            contador sempre que o valor for utilizado
        return 4; //Informa que teve de acessar a ram somente
    }
    //Caso em que o endereco esta no disco
    *ramPos =  blocoSairDaRam(ramMachine); //Posicao do bloco que saira da ram

    //Armazenando bloco que saira da ram
    MemoryBlock aux;
    aux.count = ramMachine->blocks[*ramPos].count;
    aux.enderecoEmDisco = ramMachine->blocks[*ramPos].enderecoEmDisco;
    for (int j = 0; j < WORDS_SIZE; j++)
        aux.words[j] = ramMachine->blocks[*ramPos].words[j];

    //Obtendo endereco do disco
    FILE *arq = fopen("disk.dat", "rb");
    int id;
    int v[WORDS_SIZE];
    for(int i = 0; i <= address.block; i++){
        fread(&id, sizeof(int), 1, arq);
        fread(v, sizeof(int), WORDS_SIZE, arq);
    }
    fclose(arq);

    //Passando do disco para ram
    ram[*ramPos].enderecoEmDisco = address.block;
    for (int i = 0; i < WORDS_SIZE; i++)
```

```
30          ram[*ramPos].words[i] = v[i];
31
32      //Passando o bloco que saira da RAM para o disco
33      FILE *arq2 = fopen("disk.dat", "ab+");
34      int idAux;
35      int vAux[WORDS_SIZE];
36      for (int i = 0; i < address.block; i++){
37          fread(&idAux, sizeof(int), 1, arq2);
38          fread(vAux, sizeof(int), WORDS_SIZE, arq2);
39      }
40      fwrite(&aux.enderecoEmDisco, sizeof(int), 1, arq2);
41      fwrite(aux.words, sizeof(int), WORDS_SIZE, arq2);
42      fclose(arq2);
43
44      return 5; //Informa que teve de acessar o disco
45  }
```

Código 6: Procurando um dado na RAM ou no disco

Em seguida, sairá um valor da RAM que irá receber um que está na L3, a partir do blocoSairDaRam(), que decidirá a partir da política adotada (LFU, FIFO ou RANDOM). Por fim, os valores sairão da L3 e irão para a RAM e, por fim, o contador do bloco da RAM será zerado, pois agora os valores serão outros. funcionando da mesma forma como está implementado para a Cache.

3 Análise dos resultados

Para a análise dos dados, o código foi rodado diversas vezes com os valores solicitados para as caches. Dessa forma, atribuimos que a RAM seria o dobro do L3 e o disco seria o dobro da RAM. Assim, copiamos todos os valores da última linha da execução e executamos o código a seguir para realizar o cáclulo das informações, gerando, como saída, os valores necessários para inserir na tabela do LaTex.

```
typedef struct{
    char metodo[20];
    int hit[5];//L1, L2, L3, RAM, Disco respectivamente
    int miss[4];//L1, L2, L3, RAM respectivamente
    int cost;
}Info;
int main(){
    int n;
    scanf("%d", &n);
    Info * infos = malloc(n * sizeof(Info));
    for(int i = 0; i < n; i++){
        scanf("%s", infos[i].metodo);
        //Hits e Miss  L1, L2, L3, RAM
        for(int j = 0; j < 4; j++)
            scanf("%d%d", &infos[i].hit[j], &infos[i].miss[j]);
        //Hit no disco
        scanf("%d", &infos[i].hit[4]);
        scanf("%d", &infos[i].cost);
    }
    for(int i = 0; i < n; i++){
        double somaHit[] = {0,0,0,0};
        for(int j = 0; j < 4; j++)
            somaHit[j] = ((double) infos[i].hit[j] / (infos[i].hit[j] + infos[
                i].miss[j])) * 100;
        printf("%s & %.2f\\%% & %.2f\\%% & %.2f\\%% & %.2f\\%% & %.2d & %d
            \\\\ \\hline", infos[i].metodo, somaHit[0], somaHit[1], somaHit
            [2], somaHit[3], infos[i].hit[4] ,infos[i].cost);
        printf("\n");
    }
    return 0;
}
```

Código 7: Cálculo para a porcentagem de cache hit

Utilizando os valores 64, 8, 16, 32 para RAM, L1, L2 e L3 respectivamente e tamanho do disco de 128, com 10000 instruções aleátorias, 10 interrupções e a probabilidade de 5% de ser gerada.

Método	Cache Hit L1	Cache Hit L2	Cache Hit L3	Hit RAM	Acesso Disco	Custo
LFU	6.26%	13.37%	30.62%	43.37%	9981	120298688
FIFO	6.56%	13.12%	30.78%	85.80%	2553	46439598
RANDOM	6.43%	13.25%	30.35%	64.51%	6414	85140198

Utilizando os valores 256, 32, 64, 128 para RAM, L1, L2 e L3 respectivamente e tamanho do disco de 512, com 10000 instruções aleátorias, 10 interrupções e a probabilidade de 5% de ser gerada.

Método	Cache Hit L1	Cache Hit L2	Cache Hit L3	Hit RAM	Acesso Disco	Custo
LFU	6.00%	13.33%	30.80%	43.71%	10078	121600600
FIFO	6.20%	13.45%	30.59%	86.27%	2469	45593840
RANDOM	6.16%	13.17%	30.40%	63.97%	6484	85749870

Utilizando os valores 512, 16, 64, 256 para RAM, L1, L2 e L3 respectivamente e tamanho do disco de 1024, com 10000 instruções aleátorias, 10 interrupções e a probabilidade de 5% de ser gerada.

Método	Cache Hit L1	Cache Hit L2	Cache Hit L3	Hit RAM	Acesso Disco	Custo
LFU	1.65%	6.29%	26.95%	36.47%	13407	158400413
FIFO	1.54%	6.12%	26.42%	72.20%	5977	84535963
RANDOM	1.63%	6.07%	26.82%	61.88%	8094	105416923

Utilizando os valores 256, 8, 32, 128 para RAM, L1, L2 e L3 respectivamente e tamanho do disco de 512, com 10000 instruções aleátorias, 10 interrupções e a probabilidade de 5% de ser gerada.

Método	Cache Hit L1	Cache Hit L2	Cache Hit L3	Hit RAM	Acesso Disco	Custo
LFU	1.55%	6.42%	27.27%	37.24%	13583	160799000
FIFO	1.61%	6.03%	27.12%	72.71%	5868	83475960
RANDOM	1.47%	6.29%	26.86%	62.64%	7997	104648830

Utilizando os valores 128, 16, 32, 64 para RAM, L1, L2 e L3 respectivamente e tamanho do disco de 256, com 10000 instruções aleátorias, 10 interrupções e a probabilidade de 5% de ser gerada.

Método	Cache Hit L1	Cache Hit L2	Cache Hit L3	Hit RAM	Acesso Disco	Custo
LFU	6.14%	13.14%	30.10%	44.77%	10088	122093347
FIFO	6.22%	13.19%	31.03%	86.78%	2375	44652857
RANDOM	6.47%	13.50%	30.92%	64.65%	6292	83624317

A partir dos resultados obtidos, percebe-se o crescimento das porcentagens de cache hit com o distanciamento do processador. O que faz sentido a partir da análise do tamanho dos mesmos itens, pois, como a cache L1 é menor que a cache L2 e assim por diante, a chance de um item estar presente na cache vai aumentando proporcionalmente ao tamanho da mesma.

Na primeira execução, é perceptível que o LFU teve o pior desempenho a partir da análise do seu custo, pois ele é o que teve o menor número de hit na RAM, ocasionando um maior custo, posto que o acesso ao disco é o mais caro. Nota-se também o belo desempenho do FIFO, que teve um menor custo e um desempenho similar aos outros métodos para as caches, mas teve um hit muito grande na RAM, o que melhorou muito o seu custo final.

A partir da análise do primeiro item, percebe-se a repetição do mesmo tipo de resultado, com o FIFO tendo sempre o menor custo entre o LFU e o random.

Percebe-se, também, que no terceiro e quarto caso, os valores de hit e miss foram os mais discrepantes dos outros, mas obtiveram valores próximos, assim como os testes um, dois e cinco também obtiveram valores próximos.

4 Impressões Gerais e Considerações Finais

Realizando o trabalho, foi possível entender mais sobre o uso de uma memória do disco para ter um valor "extra"para armazenar informações voláteis. Já para as interrupções, foi possível entender melhor como elas funcionam ao chamar mais instruções e depois voltar para o estado original.

Em um primeiro momento, o grupo teve dificuldades com o entendimento de como funcionaria o funcionamento da memória externa utilizando um arquivo. Todavia, após tirar dúvidas com o professor e com outros grupos, foi possível desenvolver o funcionamento dessa simulação.

Já na parte de interrupções, elas foram mais simples de serem implementadas, uma vez que o seus funcionamento é mais simples, rápido e intuitivo, já que a principal dificuldade foi pensar em uma lógica para salvar as instruções originais.

Em geral, pode-se dizer que a maior dificuldade do código foi o entendimento inicial do trabalho e a adaptação do que já foi escrito para suportar as novas funcionalidades.