PROJETO 2017

(PCS3446 – Sistemas Operacionais)

Professor: João José Neto

Autor: Victor Funabashi

NUSP: 8992221

Sumário

[1. Introdução 2](#_Toc500891336)

[2. Inclusão do ponteiro de 16 bits 2](#_Toc500891337)

[3. Modificação das instruções para abrigar endereçamento indireto 3](#_Toc500891338)

[4. Multiprogramação e Extensão da Memória Física 5](#_Toc500891339)

# Introdução

O objetivo deste projeto é a elaboração de uma simulação da operação de um sistema operacional multiprogramado em uma versão modificada da Máquina de von Neumann (MVN).

Para isso, utilizou-se a MVN projetada na disciplina PCS3216 – Sistemas de Programação na linguagem Java, modificando-a para aceitar as necessidades.

# Inclusão do ponteiro de 16 bits

Para a inclusão do conceito do ponteiro de 16 bits, criou-se uma classe em Java que recebe os dois byes e já os separa nas duas características:

* Processo: Os 4 bits mais significativos indicam um dos 16 possíveis processos do programa
* Endereço: Os outros 12 bits (os menos significativos) representam a posição de memória em que se encontra o processo indicado

Realizou-se os seguintes testes para verificar o funcionamento correto do ponteiro:

* Passagem correta: um número hexadecimal de 4 algarismos
* Erro tipo 1: Processo inválido – o primeiro caractere não corresponde a um valor hexadecimal
* Erro tipo 2: Endereço inválido – os 3 últimos caracteres não correspondem a um número hexadecimal
* Erro tipo 3: Passa-se um valor para o ponteiro que não contenha 2 bytes

A Figura 1 contém os valores testados e os resultados. Ressalta-se que os erros aqui considerados muito provavelmente não ocorrerão, devido ao uso deste ponteiro. Contudo foram formas defensivas planejadas, principalmente pelo fato de se utilizar Strings para o número em hexadecimal (sendo simples separar em/concatenar bytes).

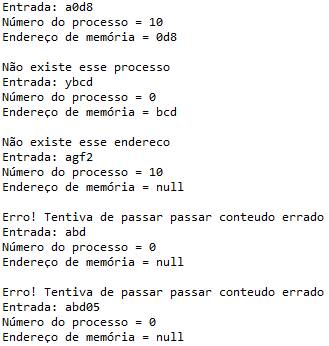


Figura 1 - Resultado do teste do ponteiro de 16 bits

# Modificação das instruções para abrigar endereçamento indireto

Como recomendado no enunciado do projeto, eliminou-se a instrução RS (código de instrução B), substituindo-a por IND.

Essa nova instrução transforma a instrução seguinte em endereçamento indireto, por exemplo:

|  |  |  |
| --- | --- | --- |
| Endereço de memória | Instrução (memoria[endereço]+memoria[endereço+1]) | PC |
| 010 | D000 | 012 |
| 012 | B000 | 014 |
| 014 | 43A2 | 3A2 |
| 016 | 3002 |  |
| 3A2 | 9066 | 3A4 |

A Figura 2 apresenta o resultado dessa implementação. O programa (“sqrtN.txt”) se inicia no endereço 00E. Lá se tem a instrução B000. Isso implica que a instrução seguinte (contida no endereço 010) será agora um ponteiro de 16 bits para o operando. Exatamente o que ocorre, dado que ao fim do que se tem na figura, o endereço atual (PC) é justamente o conteúdo dos 12 bits menos significativos do conteúdo de memória em 010 (região destacada).

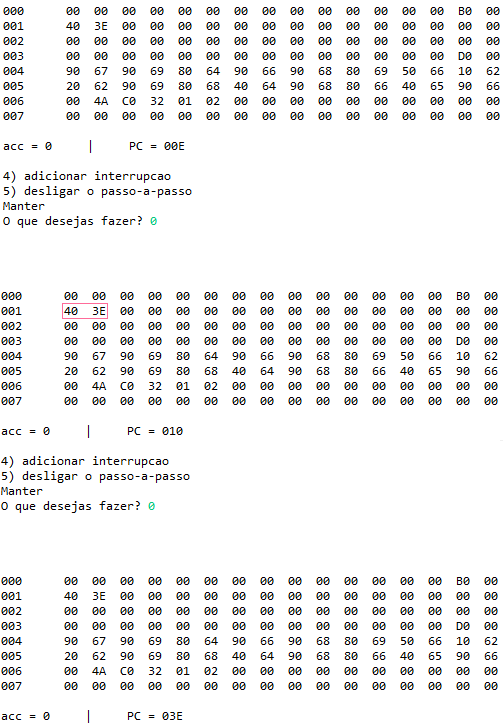


Figura 2 - Resultado do teste de instrução de endereçamento indireto

Para se verificar isso, carregou-se a lista inicial de eventos (opção 0), ligou-se o modo passo-a-passo (opção 2), iniciou-se a simulação (opção 4). Em seguida foi-se seguindo os passos (digitando qualquer número que não fosse 4 ou 5), carregou-se, no meio do processo, o programa “sqrtN.txt” (digitou-se o conteúdo entre as aspas) até chegar no instante da Figura 2.

# Multiprogramação e Extensão da Memória Física

Para essa seção faz-se duas ações: partição (lógica) da memória em 16 bancos (cada banco com 256 espaços cada); tabelas, uma para cada programa alocado, que realizam o mapeamento entre a posição lógica (definida no programa) com a física (posição absoluta) de memória de cada instrução.

Para tal, para cada programa, tem-se um endereço base a partir do qual ele será alocado (000, 100, 200, assim por diante). Assim, caso um programa ultrapasse 256 instruções ele alocará outro banco.

Uma desvantagem desse método é que, caso ele não tenha um número de instruções superior a 256 e próximo de um múltiplo (por exemplo, 512) ter-se-á desperdício de espaço de memória. Em contrapartida, é possível alocar mais de um programa que estivesse com os mesmos endereços lógicos, tendo-se assim, uma memória física aparente maior.

Para se testar este módulo carregou-se dois programas (apresentados na Figura 3).

|  |  |
| --- | --- |
| a - Raíz de n | b - n² |

Figura 3 - Códigos-fonte carregados na MVN

O resultado da carga desses programas na memória deu nas tabelas de mapeamento mostradas na Figura 4.

|  |  |
| --- | --- |
| a - mapa de raíz de N | b - mapa de n² |

Figura 4 - Mapas de memória (por programa)

Para se testar esse mapeamento, executou-se os dois programas. Dessa forma já se implementou uma forma de *Multithreading*: *Multithreading* com Granularidade Grossa (CGMT), que opera da forma apresentada na Figura 5.



Figura 5 - CGMT (Fonte: Slide de aula de PCS3422 - Organização e Arquitetura de Computadores II)

A Figura 6 apresenta a memória com os dois programas devidamente carregados, um em cada banco. Na Figura 6.a tem-se apenas os dois programas carregados.

Por outro lado, a Figura 6.b tem-se após a execução, a qual foi devidamente encerrada. Os dois bytes destacados em (**–**) tem-se o resultado de “raíz de n”, sendo que o byte à esquerda contém o dado fornecido e o à direita, o resultado.

Já os bytes destacados em (**–**), o byte à direita contem o dado fornecido e o à esquerda, o resultado (n² em hexadecimal).

A visibilidade, tanto do mapa quanto da memória ocorreu por carregar a lista inicial de eventos (0), iniciar a simulação (4), carregar os dois programas (“sqrtN.txt” e “n2.txt”) e colocar os dados, no caso “9” para o primeiro programa e “4” para o segundo.

|  |
| --- |
| a. Memória pré-execução |
| b. Memória pós-execução |

Figura 6 - Memória com os dois programas testados

## Alocação paginada

Para melhor testar-se essa função de extensão da memória, utilizou-se dos mesmos programas, contudo com algumas modificações, de modo que o programa de cálculo de ocupe duas páginas de memória lógica e n², 3 páginas (Figura 7).

|  |  |
| --- | --- |
| c - Raíz de n | d - n² |

Figura 7 - Códigos-fonte carregados na MVN

A verificação da carga desses programas foi feita analisando suas tabelas de mapeamento (Figura 8). Essas, a menos dos endereços lógicos, são idênticas às contidas na Figura 4. Esse fato era esperado pois a mudança neste teste é para o comportamento de diversas páginas em um mesmo programa.

|  |  |
| --- | --- |
| c - mapa de raíz de N | d - mapa de n² |

Figura 8 - Mapas de memória (por programa)

Por fim, averiguou-se o resultado obtido da execução desses programas, do mesmo modo descrito anteriormente, obtendo-se o resultado da Figura 9. Como pode-se reparar, não apenas teve-se os resultados corretos ( e ) como a carga foi feita de modo sequencial, como implementado, independente da página em que a instrução se encontra. Isso implica que a alocação foi correta, bem como sua tradução de endereços.

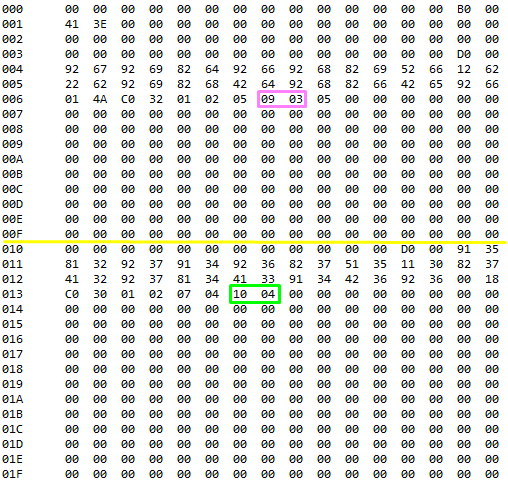


Figura 9 - Memória pós-execução dos programas

# Memória Virtual e suas Interrupções

O primeiro passo para se realizar a memória virtual foi implementar uma estrutura de dados que corresponda ao disco. Segundo o enunciado seu tamanho deve ser tal que comporte 16 programas (16 bancos físicos disponíveis), cada um podendo endereçar 16 bancos lógicos de 4kB. Assim que a memória deve ter um tamanho de 1MB, que, para se assimilar à memória, simulou-a como sendo uma matriz de 64kB x 16B



O passo seguinte foi simular uma memória parcialmente cheia, isto é, um caso em que determinada página do programa não pode ser alocada. Isto foi necessário pois até o momento a carga de programa era feita completa. E a simulação se deve por estar carregando poucos programas de uma memória recém iniciada, logo vazia.

Para um dos programas anteriores (“sqrtN.txt”) foi-se testada essa nova funcionalidade, o resultado (Figura 10) está conforme esperado, o qual, se comparado com o obtido na seção 4, possui diferenças nas instruções contidas em diferentes páginas (do endereço lógico).

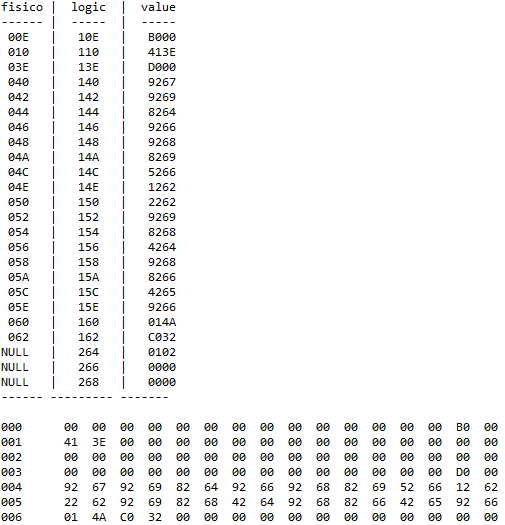


Figura 10 - Carga de apenas uma página de um programa

Conforme informado no enunciado do projeto, a memória virtual (o disco) opera sobre blocos de 4kB, ou seja, o tamanho equivalente a uma memória. Logo, caso uma página não esteja presente, será necessário substituir todo o conteúdo da memória para executar a instrução.

Para isso, o primeiro passo é realizar a carga, no disco, de diferentes páginas do programa em diferentes blocos do disco. A fim de se identificar o bloco que aquela instrução se encontra, será adicionada essa característica à PMT (Page Map Table).

A verificação foi feita em duas etapas: escrita de um programa e escrita de mais de um programa (no caso dois programas).

Para ambos os casos, visando ter melhor visibilidade, apresenta-se apenas os blocos dos discos que são preenchidos e as posições vazias possuem “..”, dado que podem haver blocos preenchidos apenas com “00” (o que impossibilitaria distinguir).

Também para simplificação fez-se que o bloco do disco ao qual a página é alocada é definido pelo algarismo hexadecimal (4 bits) mais significativo da posição lógica de memória.

No primeiro caso, o resultado obtido é apresentado na Figura 11.

|  |
| --- |
| a - PMT |
| b - Memória Principal |
| c - Memória Virtual (Disco) |

Figura 11 - Carga de um programa em memória e disco

|  |
| --- |
| a - PMT (sqrtN à esquerda | n² à direita) |
| b - Memória Principal |
| c - Memória virtual (disco) |

Figura 12 - Carga de dois programas em memória e disco

Já o segundo caso, foi feito para verificar a integridade do que estava feito, ou seja, sem sobrescrever conteúdo de memória que poderiam se sobrepor e manter mais de um programa na memória (e no disco). O resultado, como esperado e desejado, pode ser visto na Figura 12.

O passo seguinte foi criar um tratamento para o caso de o bloco necessário não ser o contido na memória. Para isso criou-se uma variável “blocoAtual” que indica a qual bloco do disco corresponde o conteúdo da memória.

Então fez-se os seguintes passos:

* Armazenar o conteúdo da memória no bloco correspondente do disco;
* Puxar o conteúdo do bloco desejado do disco para a memória;

Como o objetivo desta etapa era apenas o tratamento, fez-se como condições iniciais (anteriores à substituição dos blocos):

* Conteúdo da memória vazio (todas as posições com “00”);
* Conteúdo do disco vazio, para visualização se fez
  + Bloco ‘1’ com “\*\*”
  + Blocos restantes com “..” (mas mostrar-se-á somente o bloco ‘0’ pelo uso no teste)
* blocoAtual = 0 e ‘blocoDesejado’ = 1

Ao executar o tratamento, repara-se que foi executado corretamente, pois o conteúdo da memória foi devidamente armazenado no disco, enquanto o conteúdo desejado do disco foi colocado na memória.

|  |
| --- |
| a – Memória |
|  |
| b – Disco |

Figura 13 - Estado das Estruturas de Armazenamento priori à substituição de conteúdo

|  |
| --- |
| a - Disco |
|  |
| b – Memória |

Figura 14 - Estado das Estruturas de Armazenamento posteriori à substituição de conteúdo

Então, para utilizar essa troca de blocos criou-se um evento de código “C”. Para testá-lo fez-se que a lista inicial de eventos teria três eventos: “Início”; “Troca”; “Fim”. Ao executar a simulação obteve-se o resultado da Figura 16. Se comparado com o da Figura 14, pode-se reparar que são idênticos, o que já era esperado.

|  |
| --- |
| c – Memória |
|  |
| d – Disco |

Figura 15 - Estado das Estruturas de Armazenamento priori à substituição de conteúdo

|  |
| --- |
| c - Disco |
|  |
| d – Memória |

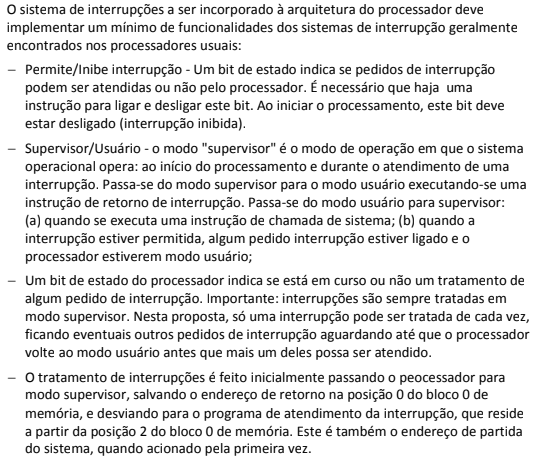
Figura 16 - Estado das Estruturas de Armazenamento posteriori à substituição de conteúdo

Com o sistema de execução multipaginado foi necessário salvar em uma variável qual a instrução atual, pois para casos de instrução de acesso à memória em que o endereço a ser acessado está em bloco diferente da instrução lida, ainda é necessário saber a ação. Assim colocou-se também uma variável booleana que indica esse estado, para que não se leia uma outra instrução (totalmente não relacionada, lixo).

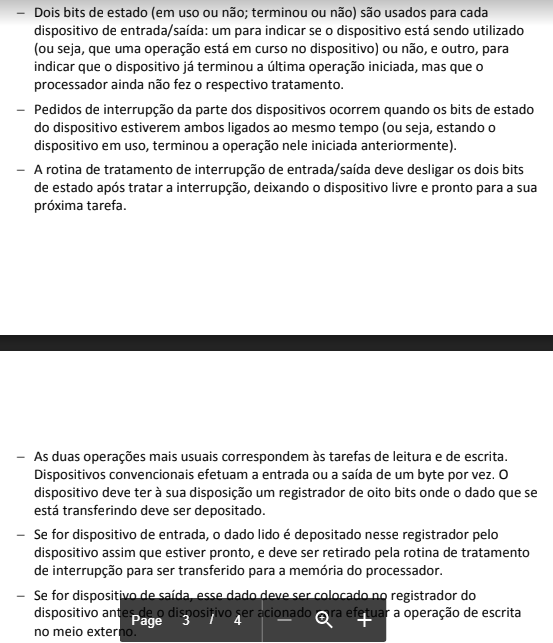
|  |
| --- |
| - PMT (sqrtN à esquerda | n² à direita) |
| - Memória Principal |
| - Memória virtual (disco) |

Figura 17 - Execução de dois programas em memória e disco

# Sistema de Interrupção



# Entrada e Saída e Suas Interrupções



# Chamada de Supervisor e a Interrupção Correspondente

