Trabalho Prático II - Memória Virtual

Sistemas Operacionais

Gabriel Victor Carvalho Rocha

gabrielcarvalho@dcc.ufmg.br - 2018054907

1. Resumo

O projeto se resume à simulação de uma memória virtual, recebendo endereços e os tipos de operações de um arquivo *.log* para acesso à memória, podendo as operações serem de leitura (*R - Read*) ou de escrita (*W - Write*). Com isso, é necessário criar um mapeamento das páginas (sendo calculadas realizando uma operação de shift sobre o endereço recebido) para um endereço na memória física.

Inicialmente o programa consiste no cálculo de alguns valores, como: *Offset*, o número máximo de páginas na memória física e o número máximo de páginas na tabela de páginas. A tabela de páginas é um vetor de *struct Pagina*, possuindo atributos fundamentais para a realização da simulação: dois *booleanos* que definem se a página é válida (alocada na memória física) e se a página está "suja", e dois *unsigned* que armazenam o endereço em que a página está alocada (caso seja válida) e um contador utilizado no algoritmo *Least Recently Used*.

Sua função mais importante é *leitura_acessos*, aqui ocorre todo o funcionamento principal do programa, lendo os endereços e operações do arquivo e adicionando a página no mapeamento. É preciso lidar com diversas situações, por exemplo, se a página é válida, basta atribuí-la uma segunda chance, caso contrário, há duas possibilidades:

a) A memória possui espaço disponível, precisando apenas buscá-la e aloca-la na memória, a tornando válida;

b) A memória está cheia, precisando então realizar a substituição pelo algoritmo escolhido, removendo a página selecionada e colocando a nova no seu lugar.

Para determinar se a página é "suja", antes que se verifiquem as condições anteriores, é feita a verificação se a operação atual é de escrita. Além disso, atribui-se à

página o valor do contador que será usado no algoritmo *Least Recently Used*.

2. Algoritmos de substituição

Na simulação da memória virtual, todos algoritmos realizam a mesma lógica, diferenciando apenas o método de encontrar a página para substituir, ou seja, para a memória com espaço disponível ou para a página já alocada (válida), sempre ocorrem as mesmas operações. Em seguida será explicado como cada algoritmo se comporta para realizar a substituição:

a. First in, first out (FIFO)

Para implementar o algoritmo FIFO, foi necessário criar uma TAD (Tipo Abstrato de Dados) de fila. A execução desse algoritmo é bem simples, pois enquanto há memória disponível, basta ir inserindo as páginas na fila. Quando a memória acaba, se retira da fila a primeira página, removendo-a da memória física e alocando a nova página.

b. Least recently used (LRU)

Já para o LRU, foi necessário ter um contador em cada página e um contador geral da função. Esse contador geral é atribuído à página sempre que houver acesso, e após isso, é acrescentado o valor 1 no contador geral para o próximo acesso ser "mais recente". No caso de realizar a substituição, a função *pos_pag_menos_recente* percorre todas páginas alocadas na memória física e retorna a posição que possui o menor contador (menos recente), removendo a página dessa posição e alocando uma nova.

c. Segunda chance (2A)

No algoritmo de substituição Segunda chance foi necessária a criação de um vetor secundário do tipo *bool*, que guarda a informação se uma posição da memória física possui ou não segunda chance. Além disso, também há um "apontador", sendo esse o responsável em encontrar a página na substituição.

Toda vez que uma página for acessada e ela já estiver na memória, ela recebe a segunda chance, atribuindo *true* no vetor de acordo com sua posição. Quando realizada a substituição, a função *pos_pag_seg_chance* irá percorrer todo o vetor de forma circular. Caso encontre uma posição com segunda chance, remove essa chance colocando *false* no

vetor. Esse processo ocorre até que se encontre alguma página que não possua segunda chance, retornando ela e a removendo da memória para a alocação da nova página.

d. Aleatório (**ALE**)

O algoritmo Aleatório foi o escolhido para o projeto, sendo este o mais simples entre todos, pois não precisa de nada adicional. Quando realizada a substituição, esse algoritmo simplesmente retorna uma posição aleatória da memória física, removendo esta página escolhida e alocando a nova página na memória física. Para ser utilizada na execução, é necessário informar como primeiro argumento o comando **ale**.

3. Instruções

Para compilar, basta digitar o comando “make”, gerando o executável “tp2virtual” que pode ser executado da seguinte forma:

./tp2virtual <alg> <arq> <tam_pag> <tam_mem>, onde: alg é o algoritmo que se deseja utilizar (fifo, lru, 2a, ale); arq é o arquivo contendo os acessos à memória; tam_pag é o tamanho da página em KB e em potência de 2; e tam_mem é o tamanho da memória física em KB e em potência de 2.

4. Decisões de projeto

A principal decisão tomada foi a criação de uma única função que processa os endereços e operações vindo do arquivo. Ou seja, para qualquer algoritmo que se escolha, a fila sempre estará alocada, mas só irá inserir/remover caso FIFO seja o escolhido. O mesmo vale para o 2A, onde o vetor secundário é alocado mas só são atribuídos valores quando for o algoritmo selecionado. E por último, o contador utilizado pelo LRU permanece em todos os algoritmos, pois no final da execução, o contador geral é utilizado como o número de acessos que houve na memória.

Quando há alguma linha no arquivo onde não há operação definida, ou seja, não há *R* ou *W*, o algoritmo segue para a próxima linha imediatamente, não contabilizando como acesso à memória.

A impressão da tabela exigida no enunciado do trabalho foi realizada somente no final do programa, já que imprimir a tabela inicial onde não teria nenhuma página

alocada parecia não ser tão necessário. O modo *debug* não foi implementado, pois considerei apenas como uma sugestão e foi dispensável para o desenvolvimento.

5. Análise

A fim de melhorar a legibilidade da documentação e não poluí-la com muitos gráficos, serão analisadas aqui principalmente as execuções do arquivo *compilador.log*, os demais gráficos se encontram nas pastas "./graficos/mem_const" e ./graficos/pag_const", enviadas em anexo.

Através da análise dos gráficos 1 e 2, onde o tamanho da página é constante, é perceptível que os algoritmos 2A e LRU possuem um desempenho bem similar (2A se sai melhor por muito pouco), sendo ambos superiores ao FIFO, que por sua vez, é superior ao ALE. Além disso, pode-se notar que quanto mais o valor da memória aumenta, todos os algoritmos convergem para o valor mínimo possível de falhas do determinado programa e zerando o número de escrita de páginas sujas, ou seja, a quantidade de memória é maior ou igual ao número de páginas que se precisa alocar.

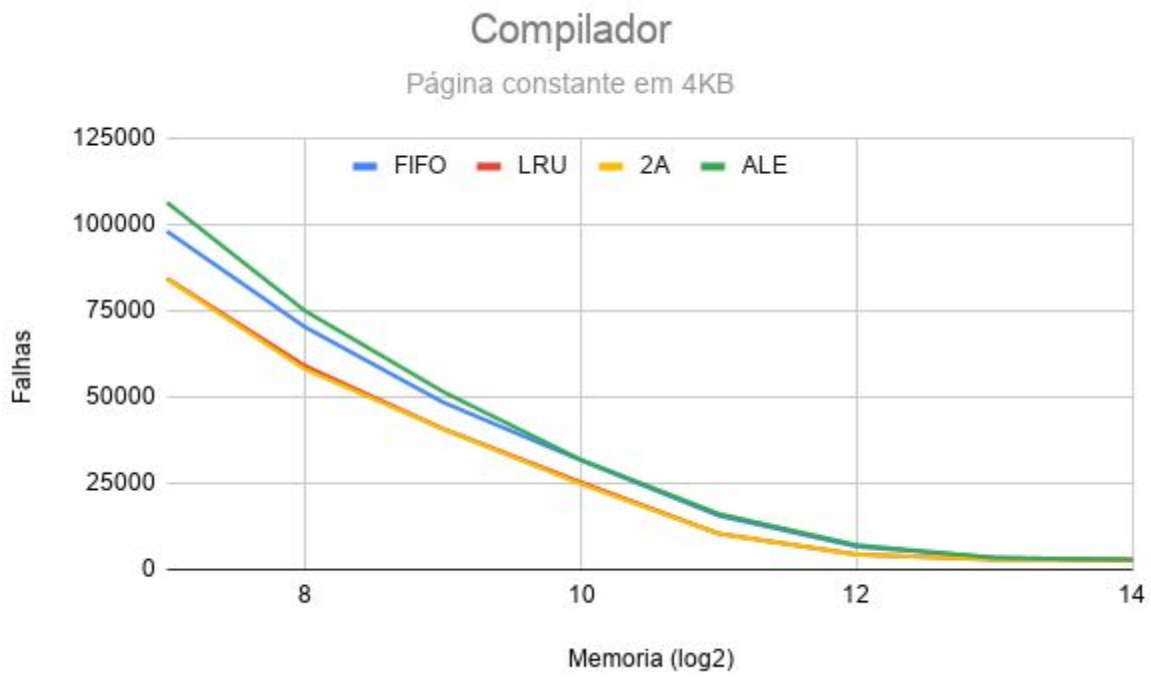

Gráfico 1: Falhas de páginas com tamanho de página constante.

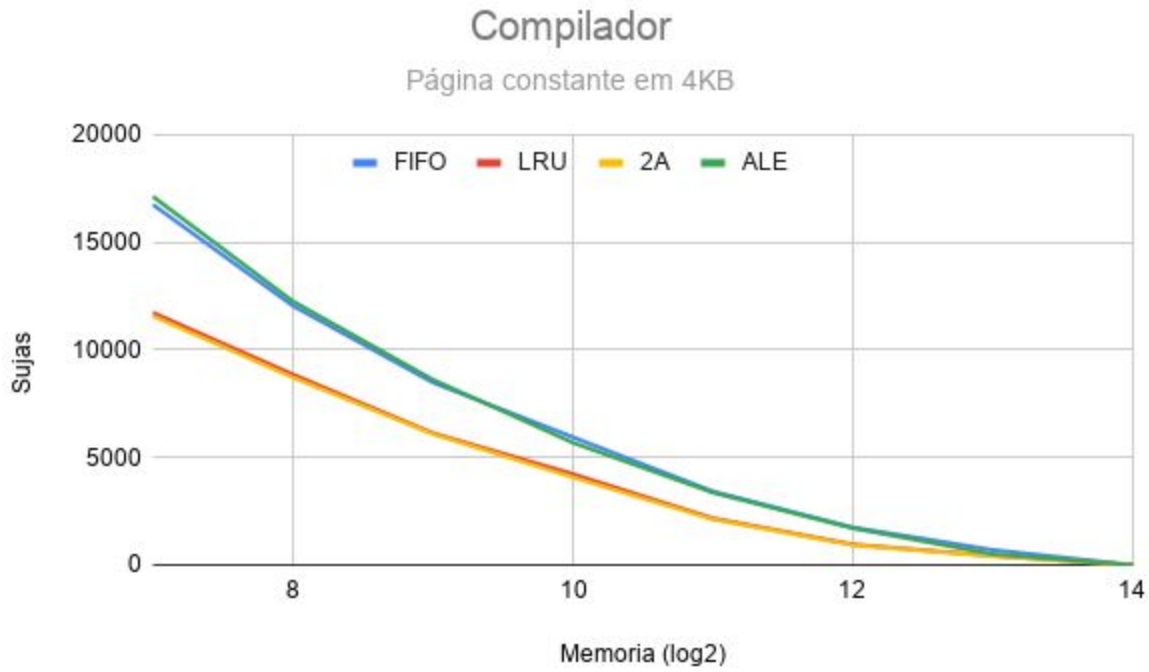

Gráfico 2: Páginas sujas com tamanho de página constante.

Mantendo a memória constante, é possível verificar pelos gráficos 3 e 4 que quanto mais se aumenta o tamanho da página, mais falhas de páginas e mais escritas de páginas sujas ocorrem. Novamente é visível como 2A e LRU se destacam entre todos os outros, com 2A sendo um pouco melhor do que o LRU. Os outros dois (FIFO e ALE) ficam com um desempenho bem similar, porém ALE sai com um desempenho pior do que o FIFO.

Esse desempenho se repete para todos os outros arquivos de entrada fornecidos, ou seja, o algoritmo 2A sempre se sai melhor que todos, LRU possuindo um desempenho próximo ao 2A mas com o valor minimamente pior, enquanto FIFO e ALE possuem desempenhos notavelmente piores, com ALE tendo o pior desempenho entre todos.

O tempo de execução de todos os algoritmos foram bem similares, ficando todos em um intervalo de 0.10 a 0.12 segundos nos testes.

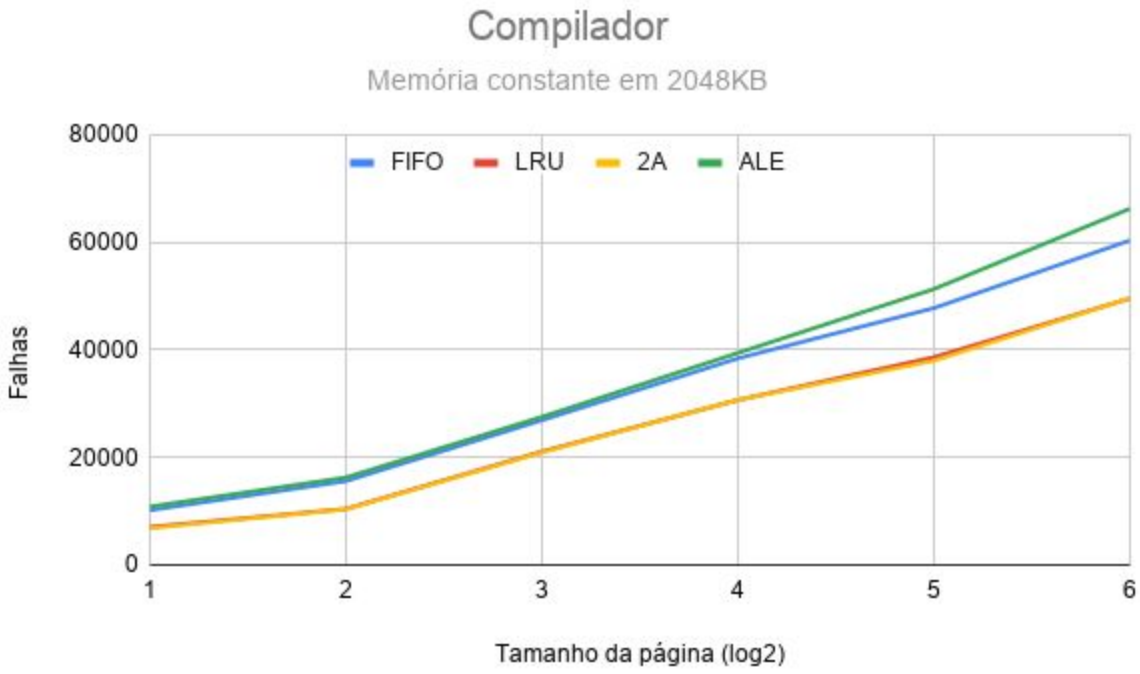

Gráfico 3: Falhas de páginas com tamanho de memória constante.

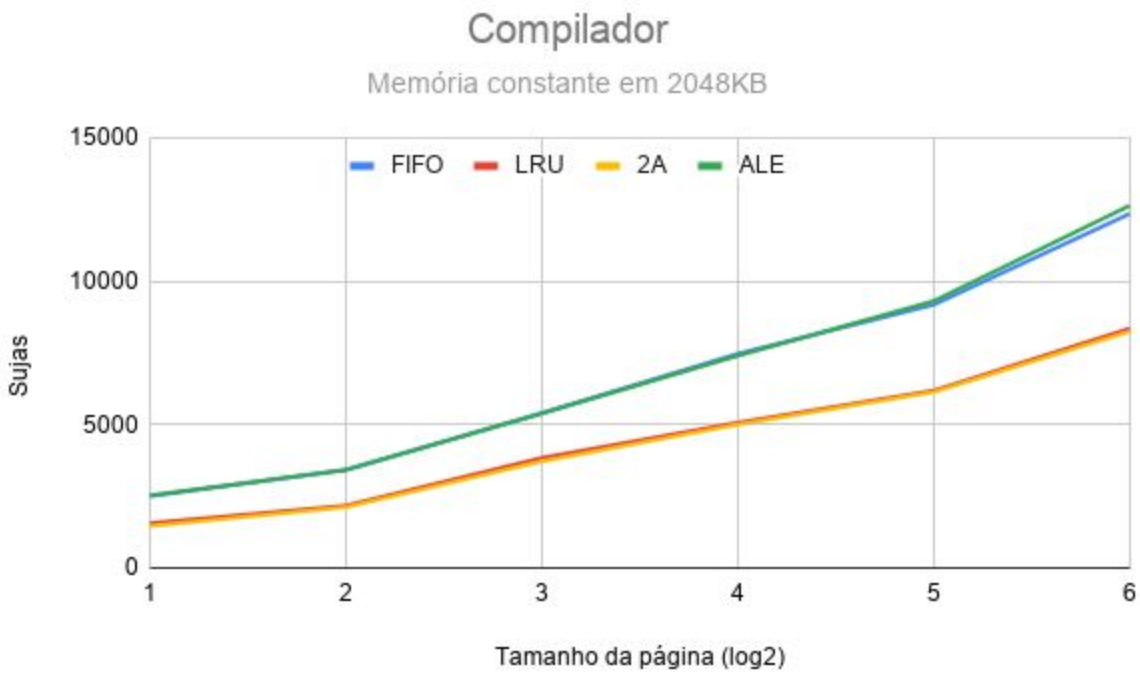

Gráfico 4: Páginas sujas com tamanho de memória constante