Universidade Federal de Minas Gerais
Departamentode Ciência da Computação

TP2 – Simulação de um sistema de memória virtual

Vinícius Julião Ramos
viniciusjuliao@dcc.ufmg.br

10 de Março de 2021

Resumo

O trabalho aqui apresentado consiste na elaboração de algoritmos de reposição para tabela de páginas. Tais algoritmos funcionarão num ambiente simulado, no qual serão dados: o tamanho total da tabela, o tamanho de cada página e uma lista de endereços de que representam acessos à memória. Sendo assim, a partir de tais acessos, os algoritmos implementados terão o trabalho de validar determinado endereço encontra-se disponível. Em caso negativo, há a necessidade de tornar aquele endereço disponível e isso pode levar à exclusão de alguma outra página (alocada anteriormente), uma vez que há uma limitação de espaço. Quatro algoritmos foram desenvolvidos e tiveram seus desempenhos comparados, sendo que um, dentre estes quatro, foi proposto pelo aluno. Os outros três métodos de reposição desenvolvidos por este trabalho foram dados apresentados pela especificação.

1 Introdução

Sistemas operacionais, precisam prover acesso à memória para os diversos programas em execução. Porém, memória é um recurso limitado e que deve ser gerenciado de maneira a atender todos os processos em execução. Essa tarefa necessita de um mecanismo capaz de realizar tal controle, permitindo que programas acessem à memória e também ao disco, entretanto abstraindo do processo em execução a necessidade de identificar se determinado endereço de memória está no disco ou na memória volátil da máquina. Tal função é conhecida como virtualização de memória e nisso encontra-se a necessidade de um algoritmo que crie uma tabela na qual liste as páginas que um processo possui armazenadas em memória, ou em disco. Essa tabela possui um espaço limitado, então se um endereço (virtual) encontra-se em disco, cabe ao mecanismo de virtualização a busca desses dados e a colocação na memória física.

Quando determinado programa já ocupou todo o espaço reservado para a própria execução, é necessário iniciar o salvamento dos dados em disco. Essa ação leva à necessidade de realizar trocas entre o disco e a memória volátil; sendo que o responsável pela troca é o algoritmo de reposição. Por fim, para este trabalho, foi proposto o desenvolvimento de quatro algoritmos de reposição: *LRU*, *Second Chance*, *FIFO* e um quarto algoritmo *Custom*, que seria proposto pelo próprio aluno. Dado o ambiente virtual do trabalho, tomaremos métricas diferentes do tempo de execução para avaliar o desempenho de cada um dos métodos. O ambiente simulado não executa leitura e escrita em disco assim como acontece no ambiente real, portanto, aqui o tempo não é um grande fator de impacto. Logo, as métricas avaliadas consistem na quantidade de páginas não encontradas na memória volátil (*page-faults*) e a quantidade de páginas sujas (*dirty*) – tais métricas identificam a quantidade de leituras e escritas feitas no disco, respectivamente.

Os códigos apresentados por esse trabalho foram desenvolvidos na linguagem C e testados no sistema operacional Ubuntu 16.04 e 18.04. Uma vez que a linguagem permite a criação de tipos abstratos de dados, tal conceito fora aplicado para instanciar as entidades necessárias no controle dos métodos de substituição.

2 Modelagem e Decisões de Projeto

A principal decisão a cerca deste trabalho foi em todo da definição de uma estrutura de dados que modelasse a tabela de páginas. Dentre as opções propostas pela especificação, escolheu-se aquela que permitiu uma rápida implementação. Entretanto, um bom nível de abstração de código foi criado, logo, caso haja a necessidade de mudar a estrutura de dados utilizada, tal ação não gerará uma grande mudança no código. Para a tabela de páginas utilizamos um *array* de uma TAD do tipo `pageattr`, em que cada elemento desse tipo abstrato de dados possui uma série de atributos utilizados pelos algoritmos de substituição. Tal *array* possui n elementos, em que n é o numero máximo de páginas de tamanho k que podem ser endereçadas por números binários de 32 bits. Ou seja, imagine um exemplo em que uma página possua $k = 4$ Kilo Bytes, então, uma vez que cada byte da memória é endereçado por um número de 32 bits, temos:

$$n = 2^{32} \div 4096 = 2^{32} \div 2^{12} = 2^{32-12} = 2^{20}$$

Uma outra conta pode ser feita de forma que, dado o tamanho de página k temos n igual a:

$$n = 2^{32-\log_2 k}$$

Utilizando de tal abordagem, podemos validar se determinada página está ou não na tabela de páginas em tempo $O(1)$, uma vez que `pageattr` possui um bit identificador `is_valid` que denota a presença ou ausência daquela página na memória. Dado um endereço qualquer, o endereço de sua própria página é obtido através de uma operação de *shifting* de bits, em que $\log_2 k$ bits serão "shiftados"de forma a remover os bits menos significativos. Este valor de k é o mesmo dado pelo tamanho da página em Bytes. Além do bit identificador `is_valid`, o TAD `pageattr` possui os outro seguintes atributos:

- `dirty` – Um bit identificador de *dirty pages*, ou seja, valida se uma página já recebeu alguma operação de escrita.
- `next` e `prev` – ponteiros para referenciar elementos vizinhos. Uma vez que a maior parte dos métodos consiste na construção de sequências, esses ponteiros são necessários mara marcar os elementos vizinhos entre si e caracterizar as listas nos métodos *FIFO*, *Second Chance* e *LRU*.
- `last_operation` – Armazena um caractere com a informação da ultima operação recebida naquela página. Essa informação é usada para a impressão do relatório ao final da execução do programa.

O código principal foi organizado em quatro etapas: (1) Abertura do arquivo de entradas, (2) alocação de memória para as estruturas de dados utilizadas, (3) execução do algoritmo especificado pelos argumentos, (4) exibição do relatório de execução. Dentro da terceira etapa encontra-se a leitura do arquivo contendo os endereços de memória, mas esse fato será abordado mais adiante. Todas essas etapas estão expressas no arquivo `main.c`, sendo que cada algoritmo de substituição corresponde a uma função dentro deste mesmo arquivo. É importante ressaltar a grande semelhança no código de cada um dos métodos de substituição, e isso permite vislumbrar o uso de orientação a objetos na qual é possível definir uma *Interface* que padronize os métodos que serão chamados ao encontrar um *page fault* ou dada a necessidade de reescrever uma *dirty page* no disco. Entretanto o uso da linguagem C não permite a aplicação desses conceitos, logo a organização do código aqui proposto foi baseado em legibilidade e na separação correta de funções comuns a algum tipo de execução.

Outros dois aquivos de código foram escritos a fim de organizar e separar funcionalidades. Os aquivos `page_table.h` e `page_table.c` contém a definição das estruturas (TADs) responsáveis pela tabela de páginas, além das funções que as gerenciam. Já os arquivos `utils.h` e `utils.c` contém funções para manipulação de entrada e saída, por exemplo a exibição do relatório final de execução e também a leitura do arquivo de endereços de memória. É importante ressaltar que na função de leitura do arquivo, foi necessário tratar um possível erro de formatação; para que uma linha seja reconhecida ela deve conter um número hexadecimal, seguido de um caractere espaço e um outro caractere `R` ou `W`. Caso não haja esse padrão, a linha é desconsiderada e tenta-se realizar a leitura da próxima linha. Enquanto não houver uma linha em concordância, as linhas serão desconsideradas até encontrar o fim do arquivo.

3 Algoritmo de Substituição Próprio

O algoritmo de substituição proposto por esse trabalho, foi baseado na criação de um *hashing* de endereços. Dentre os argumentos recebidos pelo programa, temos a quantidade de memória disponível (q) e o tamanho de cada página (k). Então, sendo $l = q \div k$ a quantidade de páginas gerenciadas pelos métodos de substituição, aplicamos uma função que mapeia qualquer endereço para um número entre 0 e $l - 1$. Assim, o método de substituição criado por esse trabalho, trabalhará com a quantidade correta de páginas. Esse hashing acontece de modo a aplicar uma operação booleana *and* bit a bit, entre o endereço da página requisitada e $l - 1$. Como l será sempre um valor em potência de 2, temos que $l - 1$, quando transformado para a base 2, é um número preenchido apenas com 1.

Dessa maneira o hashing obtém os bits menos significativos do endereço de uma página e assim, pode-se consultar nesse array de l endereços, se determinada página requisitada já está na memória ou não. É importante informar que essa operação booleana é aplicada sobre o endereço da página, ou seja, o endereço dado no arquivo de entrada passa por uma transformação de shifting e assim é obtido o endereço da página, e só depois disso aplica-se a operação *and* bit a bit com $l - 1$. O hashing se dá da seguinte maneira:

Algorithm 1 Hashing Transformation

1: **function** HASHING TRANSFORMATION(*Page Address* $PAddr$, *Pages Quantity* l)
    return $PAddr$ && $(l - 1)$
2: **end function**

Para melhor esclarecer, observe o seguinte exemplo: Dada um programa com uma quantidade de memória disponível igual a 128 Kilo Bytes, em que cada página possui 8 Kilo Bytes de memória disponível. Nesse caso, teremos então $128 \div 8 = 16$ elementos na tabela de páginas, logo $l = 16$. Então suponha que o programa deseje acessar uma página x em que o valor decimal do endereço será 550. Portanto, dado um array A que armazena os endereços das páginas disponíveis na memória, em que o tamanho de A é igual a l, poderemos mapear qualquer endereço de página para uma posição em A através da operação booleana previamente citada. Observe o mapeamento da página x:

$$\langle and \rangle \frac{x \quad \rightarrow 550_{10} \; = 1000100110_2}{l - 1 \rightarrow 15_{10} = 0000001111_2} = 0000000110_2 = 12_{10}$$

Por fim, a página de que tem o endereço igual a 550 será armazenada no índice 12 do vetor A, sendo que este vetor é indexado com o primeiro elemento na posição 0. Para executar o algoritmo criado por este trabalho, o argumento de execução que contem o nome do método de reposição a ser executado dever ser `custom`.

4 Análise Experimental

Antes de demonstrar os comparativos entre os gráficos e avaliar qual dos algoritmos possuiu melhor desempenho, vamos basear a questão do desempenho pela quantidade de *page faults* reportados pela execução. Como não é possível obter métricas reais do tempo de execução da escrita e leitura do disco, o número de páginas não encontradas na memória será o norteador. A justificativa vem do motivo de que: uma vez que uma página não é encontrada na memória é necessário acessar o disco para carregar as informações daquela página. Além disso, quando um *pag fault* é reportado, e o programa já faz uso de toda a memória disponível para sua própria execução, são necessárias duas operações: A possível escrita de uma *dirty page* no disco e a leitura da página solicitada. Portanto, o uso da quantidade de paginas não encontradas na memória possui uma forte justificativa para ser utilizada como métrica nesse ambiente simulado que fora apresentado por este trabalho.

4.1 Algoritmo customizado

Nessa análise, foi executado todas as combinações disponíveis entre *Tamanho da Página*, *Memória Disponível* e *Arquivo de Entrada*. Após obter os resultados, para cada valor de *Memória disponível*, calculou-se

a média da quantidade de *page faults* entre todos os pares de *Tamanho da página* e *Arquivo de entrada*. Ou seja, um ponto no gráfico da Figura 1 corresponde a um valor fixo do *Algoritmo* e da *Memoria Disponível* em contraste com a média das execuções que variaram o valor dos outros dois argumentos.

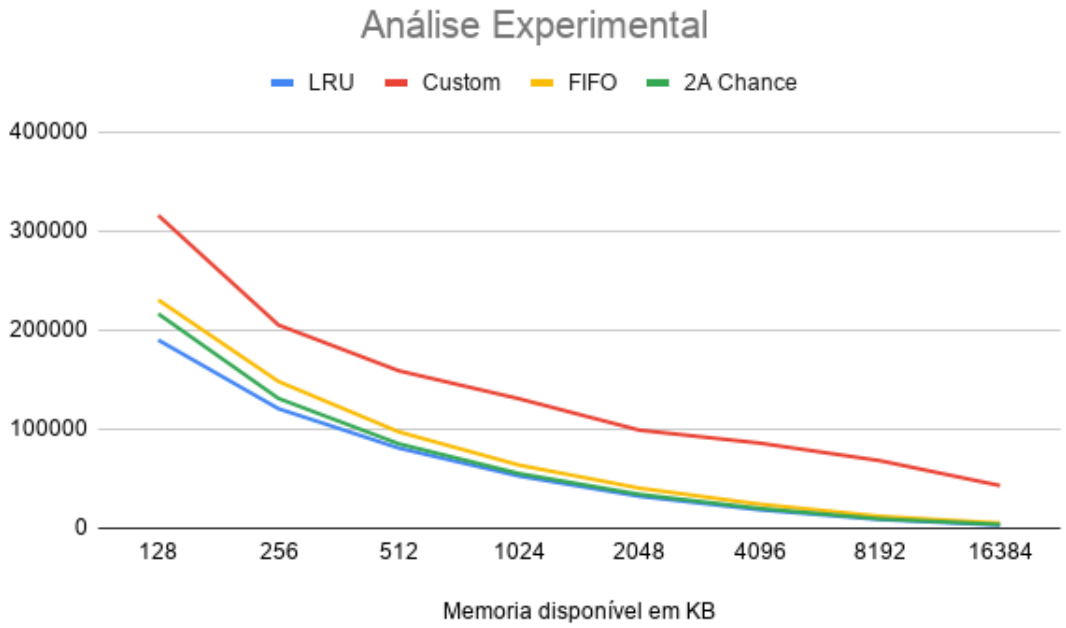

Figura 1

No gráfico da Figura 1, observamos que o algoritmo customizado obteve uma quantidade de *page faults* muito maior que os demais. Podemos observar então, que esse tipo de hashing não é eficiente para essa aplicação uma vez que muitos endereços possuem o conjunto dos bits menos significativos muito próximos. Possivelmente, uma abordagem melhor, seria criar tal hashing pelo bits mais significativos, uma vez que valores de endereços com bits mais significativos muito diferentes, identificam que tais endereços estão dispersos. Entretanto, foi solicitado pelo trabalho um algoritmo de reposição ainda não difundido, e o algoritmo que utiliza um tipo de hashing com os bits mais significativos é aplicado em hardware para a transição entre memória RAM e caches do processador.

4.2 Tamanho de página constante e variação da memória disponível

Para tal abordagem, mantivemos constantes o tamanho de cada página, em 4 Kilo Bytes. Entretanto, variamos o tamanho da memória disponível entre o intervalo de 128 a 1638 Kilo Bytes. Além disso, como são 4 arquivos de entrada, também variamos tais arquivos, porém realizamos uma análise da média de execução entre estes quatro arquivos. A inclusão do algoritmo `custom` nessa análise, interferiria na qualidade do gráfico, uma vez que os valore dos resultados do método customizado são muito grandes. Isso acarretaria numa aproximação entre as linhas dos outros três métodos de reposição, fazendo com que eles ficassem muito semelhantes a "olho nu". Além disso, na especificação desse trabalho, solicitou-se esse comparativo apenas para os *FIFO*, *LRU* e *2A Chance*. O gráfico dessa extração de dados está contido na Figura 2.

A análise visual, permite identificar uma aproximação de desempenho entre os métodos. Porém é possível notar que *FIFO* performou pior que os demais, uma vez que para uma mesma quantidade de memória disponível e uma mesmo tamanho de página, esse método possui um número maior de *page faults*. Porém, com o crescimento da quantidade de memória disponível, observou-se uma forte aproximação do *FIFO* com o método *2A Chance* e há uma razão por trás desse fato. Ambos os algoritmos de reposição utilizam uma lista circular para apontar a próxima pagina que será reposta no disco em caso de necessidade. Entretanto,

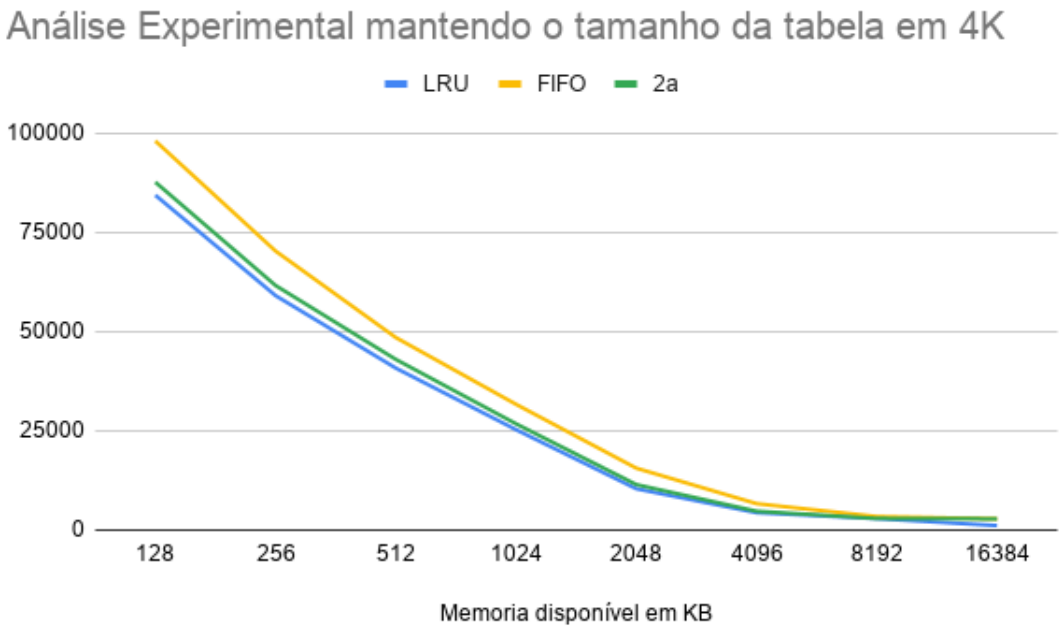

Figura 2

com diz o nome, *2A Chance* aplica uma revalidação que dá a uma pagina uma chance a mais de permanecer na lista. Com uma memória disponível grande, a nova chance concedida às páginas não terá impacto se comparado ao *FIFO* tradicional. Já o método *LRU* possui um melhor desempenho que todos os algoritmos independentemente do cenário, entretanto o à medida que a memória disponível cresce o desempenho do *LRU* também converge, juntamente com os outros dois métodos de substituição de páginas.

4.3 Variação do tamanho da página vs. Memória Constante

Neste ultimo caso, apresentado na Figura 3 mantivemos a quantidade de memória constante em 512 Kilo Bytes e variamos o tamanho das páginas. Assim como nos demais casos, das seções anteriores, os valore de *page faults* são referentes à media dentre os quatro arquivos de entrada fornecidos. Comparando as Figura 2 e 3 o algoritmo *LRU* também possui uma performance melhor do que os demais, sendo que *2A Chance* tem comportamento semelhante, porém com um pequeno gap que a faz desse método menos eficaz.

Na Figura 3 também não incluímos o desempenho de `custom`, pois esse método se apresentou ineficaz e assim como na seção 4.1, a inclusão das métricas desse gráfico atrapalharia a visualização das demais informações. Outro fato que merece antenção é de que, em todos os gráficos observamos um aumento na quantidade de *page faults* à medida em que o tamanho da página aumenta. Também há uma melhora de desempenho à medida em que a memória disponível aumenta. O que demonstra uma dependência forte do desempenho dos algoritmos com o espaço disponível. Um possível algoritmo que conseguiria reduzir ou ter um impacto diferente destes apresentados aqui, seria algum algoritmo preditivo, o qual alocaria as páginas na memória baseado em fatores futuros, mas ainda assim haveria tal dependência.

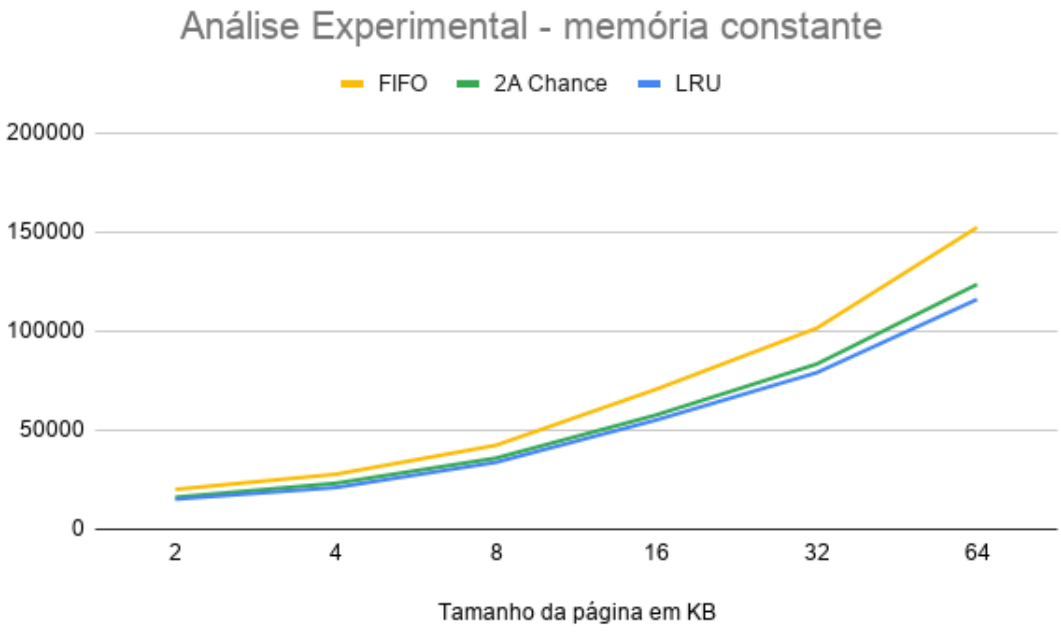
Análise Experimental - memória constante
FIFO
2A Chance
LRU
200000
150000
100000
50000
0
2
4
8
16
32
64
Tamanho da página em KB

Figura 3