Documentação - Trabalho Prático 2 de Sistemas Operacionais: Simulador de Memória Virtual

Arthur Vieira Silva

01 de Setembro de 2024

Sumário

1 Introdução

1.1 Memória Virtual

Nos computadores modernos, tem-se a necessidade de executar inúmeros programas que são grandes demais para se encaixar na memória e há também a necessidade de que haja sistemas capazes de dar suporte a múltiplos programas executando simultaneamente, cada um deles encaixando-se na memória, mas com todos coletivamente excedendo-a [TANENBAUM, 2010]. Sendo assim, a ideia básica da memória virtual é que cada programa possui seu próprio espaço de endereçamento, no qual é dividido em blocos chamados de páginas e, cada página, é uma série contígua de endereços. As páginas são mapeadas na memória física porém, como há uma abstração de memória, nem todas as páginas precisam estar na memória física ao mesmo tempo para que um programa seja executado.

Grande parte dos sistemas de memória virtual fazem uso de uma técnica conhecida como paginação. Nos computadores, os programas referenciam um conjunto de endereços de memória e os endereços que os computadores geram são denominados endereços virtuais, que por sua vez formam o espaço de endereçamento virtual. O espaço de endereçamento virtual consiste em unidades de tamanho fixo chamadas de páginas e a unidade correspondente na memória principal são chamadas de quadros de páginas [TANENBAUM, 2010]. Para cada processo, o sistema operacional mantém uma tabela de páginas e a sua função é justamente mapear as páginas virtuais em quadros de páginas. Uma maneira matemática de visualizar as tabelas de páginas é como uma função, com o número da página virtual como argumento e o número do quadro físico correspondente como resultado gerado.

Assim sendo, torna-se imprescindível que haja uma abstração de memória nos computadores, haja vista que a memória virtual produz na memória física um aumento efetivo, um isolamento e um gerenciamento mais eficiente. Além disso, como mencionado anteriormente, permite a execução de programas maiores que a capacidade da memória física disponível e que vários programas estejam na memória ao mesmo tempo.

1.2 Funcionamento do Programa

Diante desse contexto, o objetivo desse trabalho prático é implementar um simulador de memória virtual. Entretanto, esse trabalho não está relacionado diretamente com aspectos do sistema operacional, o propósito é implementar uma réplica das estruturas de um mecanismo de gerência de memória virtual. Dessa forma, o simulador recebe como entrada um arquivo no qual haverá uma sequência de endereços de memória que serão acessados por um programa real, tais endereços estarão em hexadecimal e são seguidos por um caractere (R ou W), indicando se o acesso é de leitura ou escrita. Na execução do programa, o tamanho da memória, o tamanho de cada página/quadro de memória, o arquivo de entrada que deverá ser lido e o algoritmo de substituição de páginas a ser adotado serão especificados.

A respeito da execução do programa, ela deverá ser especificada com 4 ou 5 argumentos e o nome do programa é tp2virtual: **./tp2virtual lru arquivo.log 4 128**, os argumentos são, respectivamente:

1. O algoritmo de substituição de páginas escolhido (lru, nru ou segunda_chance);
2. O arquivo de entrada no qual a sequência de endereços será lida (no exemplo acima, arquivo.log);
3. O tamanho de cada página/quadro de memória (em KB). Recomenda-se a utilização de valores entre 2 e 64;
4. O tamanho total da memória física disponível para o processo (em KB). Recomenda-se a utilização de valores entre 128 e 16384 (16 MB).

O quinto argumento é opcional mas, se for passado, a palavra "debug"deverá ser escrita. A sua finalidade é, a cada acesso à memória, escrever algumas linhas que descrevam o que foi feito naquela situação. Essas linhas serão escritas em um arquivo denominado depuracao.txt e serve apenas para acompanhar a operação do programa passo a passo.

Quando o simulador chegar ao fim, isto é, quando a sequência de acessos à memória terminar, o programa irá gerar um relatório final contendo algumas informações importantes:

1. Qual foi a configuração utilizada (definida pelos parâmetros utilizados);
2. Qual é o número total de acesso à memória contidos no arquivo de entrada fornecido;
3. Quantos *page faults* ocorreram durante a simulação;
4. Qual o número de páginas "sujas"(*dirty*) que tiveram que ser escritas de volta no disco (páginas sujas que restarem no final da execução não precisam ser escritas).

Como o tamanho das páginas/quadros de memória será definido na execução do programa, é necessário a implementação de um mecanismo capaz de descartar os s *bits* menos significativos de endereço para um tamanho de página variável. Desse modo, s deve ser calculado a cada execução. A função que realiza esse cálculo será mostrada a seguir e, além disso, a página é determinada simplesmente realizando a operação $addr >> s$, onde $addr$ é um determinado endereço lido do arquivo de entrada.

A respeito da implementação da tabela de páginas, como os endereços nos arquivos fornecidos serão de 32 *bits*, para o tamanho mínimo recomendado de cada página (2 KB), a tabela de páginas necessita de dois milhões de entradas tornado-se, então, inviável de ser empregada. Com isso, uma tabela de páginas invertida será utilizada.

A tabela de páginas invertida é uma única tabela que o sistema operacional mantém para todos os processos, nela há apenas uma entrada por quadro de página na memória real, em vez de uma entrada por página de espaço de endereço virtual. O número de entradas em uma tabela de páginas invertida e o número de quadros são iguais na memória principal. Embora o desperdício de memória seja reduzido, a pesquisa em uma tabela de páginas invertida não é tão eficiente. Para isso, utiliza-se uma tabela *hash* nos endereços virtuais. Todas as páginas virtuais atualmente na memória que têm o mesmo valor *hash* são encadeadas juntas. A Figura 1 mostra uma tabela de páginas tradicional do lado esquerdo e uma tabela de páginas invertida do lado direito.

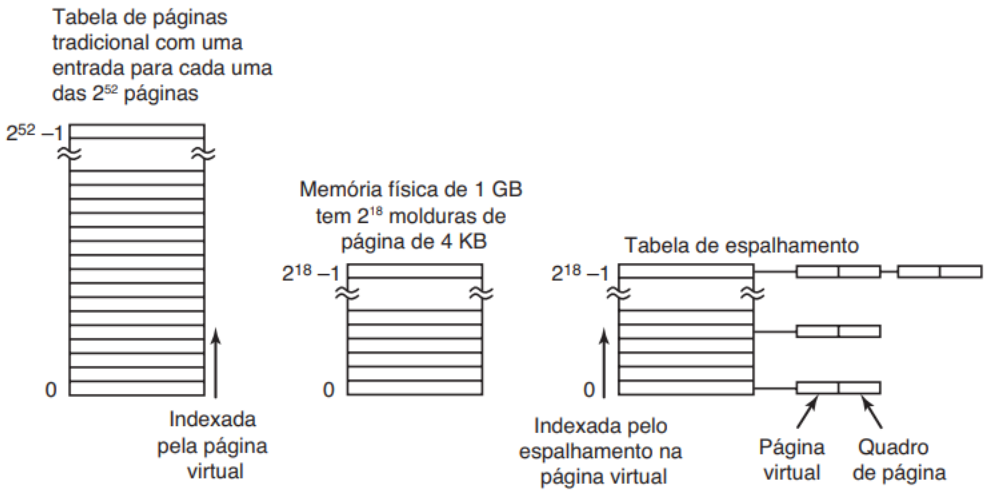


Figura 1: Tabela de páginas tradicional x Tabela de páginas invertida.

2 Implementação

2.1 Estruturas de Dados

Na implementação do simulador, foram utilizadas algumas estruturas de dados fundamentais para o seu funcionamento. Assim sendo, o arquivo *dataStructures.h* possui as seguintes estruturas de dados:

- **Page**: Estrutura utilizada para representar cada entrada na tabela de páginas. Possui campos como *page_number* (número da página gerado a partir do seu endereço e dos *s bits* menos significativos), *referenced* (*bit Referenciada* configurado sempre que uma página é referenciada, seja para leitura ou para escrita), *modified* (*bit Modificada* é configurado ao escrever na página), *counter* (contador utilizado pelo algoritmo de substituição de páginas LRU) e *next* que aponta para a próxima página que está na mesma posição dessa página na tabela *hash*. É claro que as entradas de uma tabela de páginas possui outros campos porém, para esse trabalho, apenas esses campos são suficientes;

- **ReadingData**: Essa estrutura é utilizada apenas para ler as linhas do arquivo de entrada fornecido. Há dois campos, *addr* que irá armazenar o endereço e *rw* que irá armazenar o caractere (R ou W) representando o modo de acesso (leitura ou escrita) desse endereço;

- **OutputData**: Armazena alguns dados que serão atualizados durante a execução do simulador e mostrados no relatório gerado no final do simulador. Há um campo *read_pages* que armazena o número de páginas que foram lidas, um campo *page_faults* responsável por armazenar o número de faltas de página ocorridos e um campo *written_pages* que irá contar o número de páginas "sujas", ou seja, que tiveram que ser escritas de volta no disco;

- **Node**: Essa estrutura representa um nó de uma lista simplesmente encadeada que será utilizada no algortimo de substituição de página segunda chance. Há um campo *page* que é do tipo **Page** e um ponteiro *next* que aponta para o próximo nó da lista;

- **List**: Por fim, essa estrutura é a lista propriamente dita do algoritmo de substituição de página segunda chance. Há dois campos do tipo **Node**, *beginning* que é um ponteiro para o início da lista e *end* que é um ponteiro para o fim da lista.

2.2 Arquivo tp2virtual.c

Esse arquivo, além de ser o responsável por iniciar a simulação, possui algumas funções primordiais para o funcionamento do simulador. Primeiramente, a função chamada é **check_arguments()**.

A função **check_arguments()** é usada para verificar se os argumentos passados na execução do programa estão corretos. Então, verifica-se se o número de argumentos está correto (se é 5 ou 6 - um argumento a mais porque ./tp2virtual também entra na contagem). A seguir, verifica-se se o algoritmo digitado foi lru, nru ou segunda_chance, se o tamanho de cada página/quadro não é menor ou igual a 0, se o tamanho total da memória física disponível não é menor ou igual 0 ou se não é menor do que o tamanho de cada página/quadro e, se o parâmetro opcional foi passado, verifica-se se a palavra "debug"foi digitada.

Se os argumentos passados estão corretos, isto é, a função não retornou -1, é declarada uma variável *TABLE_SIZE* que irá representar o tamanho da tabela *hash* e que é definida como sendo o tamanho total de memória física disponível dividido pelo tamanho de cada página/quadro. Sendo assim, uma variável *hash_table* do tipo **Page** (ponteiro para ponteiro) é utilizada para representar a tabela de páginas invertida. Há também uma variável *debug_mode* inicializada como o valor 0 e que receberá o valor 1 caso o argumento opcional tenha sido passado. Após isso, a função **read_file()** é chamada.

A função **read_file()** será utilizada para ler o arquivo de entrada fornecido que deverá estar dentro de uma pasta chamada *logs*. Algumas variáveis são importantes nessa função, são elas: *page_number*, *page_index*, *page_found* e *clock_count* (iniciada com o valor 0). Além disso, as estruturas de dados **ReadingData**, **OutputData** e **List** são utilizadas. A variável do tipo **OutputData** é inicializada com todos os campos com o valor 0. Dessa maneira, há um laço *while* que irá ler linha por linha do arquivo de entrada e armazenar os dados na variável do tipo **ReadingData**. Primeiramente, se *clock_count* for maior do que a constante *CLOCK_INTERRUPT* (interrupção de relógio - definida com o valor 10), as funções **update_pages_ages()** (se o algoritmo de substituição for o LRU) **update_referenced_bit()** (se o algoritmo de substituição for o NRU) serão chamadas e *clock_count* será reiniciada com o valor 0, essas funções serão explicadas a seguir.

Ainda dentro do laço *while*, *page_number* e *page_index* são obtidas utilizando as funções **determine_page()** e **hash_function()**, respectivamente, tais funções também serão mostradas a seguir. Uma decisão de projeto tomada é que se em uma determinada linha do arquivo não há a letra R ou W após o endereço (ou há uma letra diferente), o *while* irá para a próxima iteração, ou seja, aquele endereço é completamente ignorado. Com isso, verifica-se se o número da página está na posição *page_index* na tabela *hash*. Se ocorreu *page fault* (*page_found* = 0), Uma das funções **lru()**, **nru()** ou **second_chance()** (mostradas posteriormente) é chamada e a nova página é criada, inicialmente com R = 1. Se a nova página for somente de leitura, M = 0 e *read_pages* é incrementado, se ela for de escrita, M = 1 e *written_pages* é incrementado, em ambos os casos, *page_faults* é incrementado e se o modo de depuração tiver sido selecionado, uma função **save_test_file()**, também explicada a seguir, é chamada. Já se não ocorreu *page fault* (*page_found* = 1), a única diferença é que não há necessidade de colocar uma nova página na tabela *hash* e nenhum algoritmo de substituição de páginas será chamado. Além disso, *clock_count* é incrementada a cada iteração e, no final da função, os dados de saída são retornados.

A função **determine_page()** é utilizada para obter os *s bits* menos significativos de um endereço, utilizando o tamanho de cada página/quadro e, assim, retornar o número de página associado a um determinado endereço.

Já a função **hash_function()** retorna um determinado índice na tabela *hash* por meio da operação *page_number* % *TABLE_SIZE*.

A função **save_test_file()** é chamada caso o modo de depuração tenha sido selecionado. Ela irá criar um arquivo chamado "depuracao.txt"que descreverá, em algumas linhas, o que é feito em cada acesso à memória. Por exemplo, é salvo o número de página acessado, se ocorreu ou não *page fault*, se é necessário atualizar a cópia que está no disco ou se a página apenas é lida.

Por fim, ao final no final do simulador, a função **display_results()** é chamada para mostrar os dados armazenados na estrutura **OutputData**. Além disso, há uma função chamada **free_hash_table()** que é utilizada para liberar cada página na tabela de páginas e, em seguida, liberar a tabela de páginas como um todo.

2.3 Arquivo algorithms.c

Nesse arquivo temos as funções necessárias para o funcionamento dos algoritmos de substituição de páginas. A função **lru()** é utilizada para simular a execução do algoritmo de substituição de páginas LRU (*Least Recently Used*). Para isso, uma pequena modificação no algoritmo precisou ser realizada para simular o algoritmo LRU em software por meio do **algoritmo de envelhecimento**. Após cada interrupção de relógio, o *counter* de cada página é atualizado mediante a função **update_pages_ages()**, mostrada logo abaixo. O LRU irá selecionar, dentre as páginas presentes na memória, aquela que possui o menor contador para removê-la por meio da função **remove_page()**, também explicada a seguir.

Sobre a função **update_pages_ages()**, o *counter* de cada página é um número binário de 8 *bits*, com isso, para atualizá-lo, eles são deslocados um *bit* à direita e o *bit* R é adicionado ao *bit* mais à esquerda. Além disso, o *bit* R recebe o valor 0.

A função **nru()** é utilizada para simular a execução do algoritmo de substituição de páginas NRU (Not Recently Used). Nesse algoritmo, todas as páginas são classificadas em uma das 4 classes abaixo:

1. Classe 0: não referenciada, não modificada;
2. Classe 1: não referenciada, modificada;
3. Classe 2: referenciada, não modificada;
4. Classe 3: referenciada, modificada.

Dessa forma, o algoritmo irá remover uma página ao acaso de sua classe de ordem mais baixa que não esteja vazia, utilizando a função **remove_page()**. Como mencionado anteriormente, a função **update_referenced_bit()** é utilizada para limpar, a cada interrupção de relógio, o *bit* R de todas as páginas.

A função **second_chance()** é utilizada para simular a execução do algoritmo de substituição de páginas segunda chance. Nesse algoritmo, há uma lista de todas as páginas atualmente na memória, com a chegada mais recente no fim e a mais antiga na frente. Remove-se a página mais antiga, isto é, a primeira página da lista. Se o *bit* R da página for 0 a página é removida da lista utilizando a função **remove_firs_element()**, que é uma função utilizada apenas para remover o primeiro elemento de uma lista. No entanto, se o *bit* R for 1, o bit R é zerado, a página é removida da lista utilizando essa mesma função porém, ela é adicionada novamente no final da lista utilizando a função **insert_end()**, que é uma função utilizada apenas para inserir um elemento no final de uma lista. Por fim, a página mais antiga que foi selecionada é removida utilizando a função **remove_page()** descrita logo abaixo.

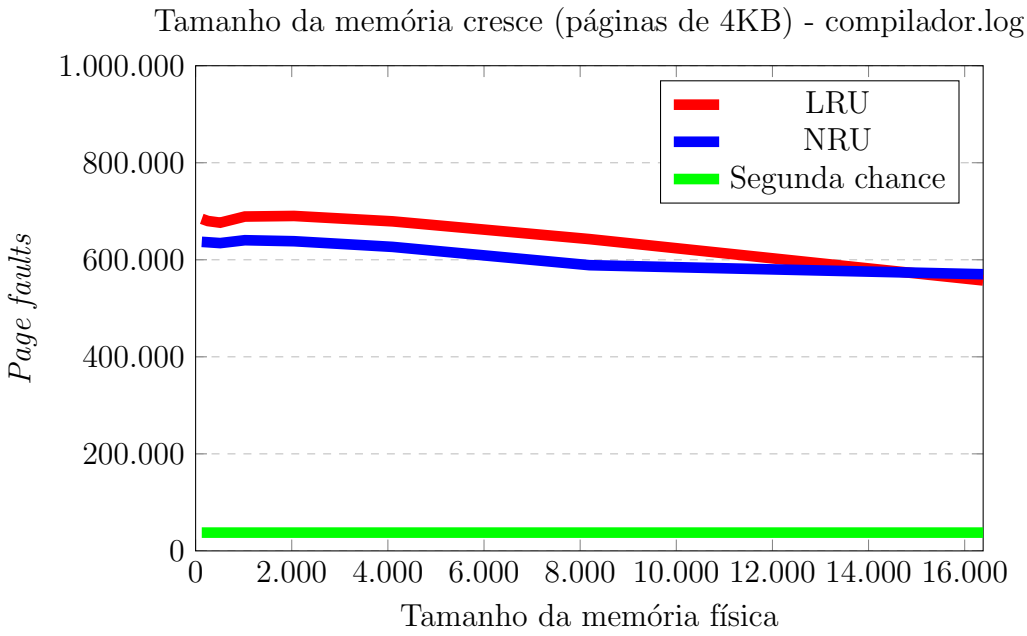
Finalmente, a função **remove_page()** é utilizada para remover uma página da memória escolhida por um dos algoritmos de substituição quando ocorrer uma falta de página. Nessa função, a partir do índice da página que será removida, procura-se por essa página na lista de páginas que estão na tabela *hash* na posição desse índice e, ao encontrar, a página é removida e a memória alocada para ela é desalocada.

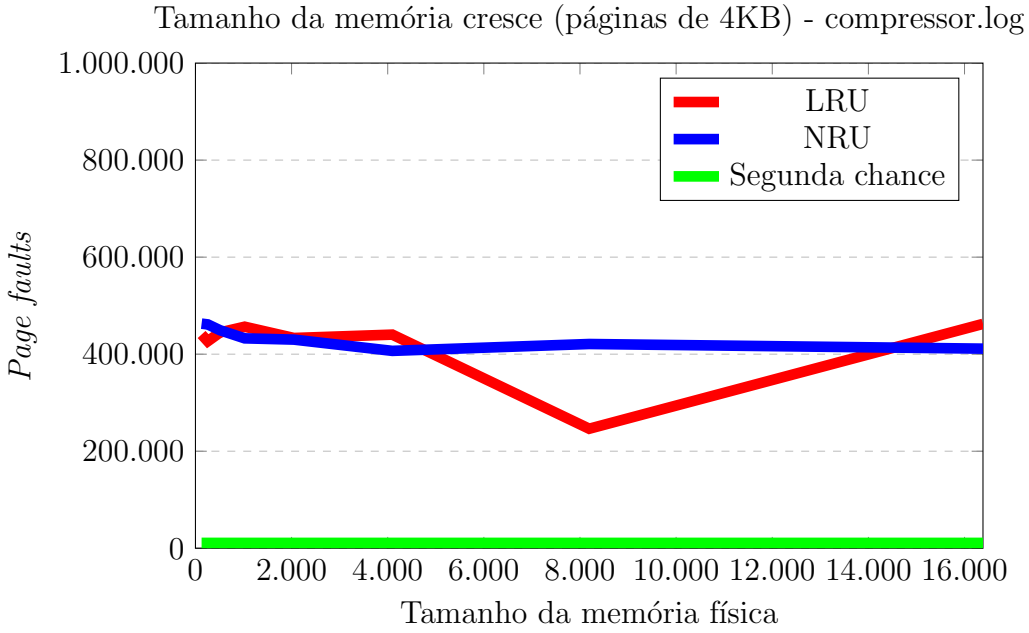
3 Resultados

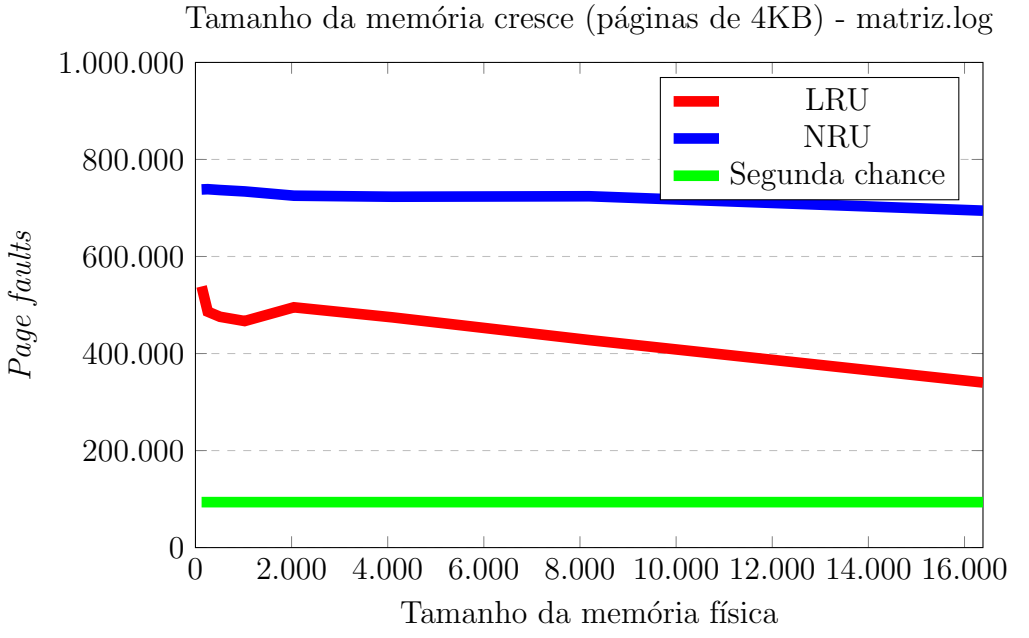
A seguir será apresentada uma análise de desempenho dos algoritmos de substituição de páginas para alguns arquivos de testes utilizados. Dessa forma, cada gráfico mostrará como o número de *page faults* se comporta quando o tamanho da memória física disponível varia ou o tamanho de cada página/quadro varia. Com isso, os 4 primeiros gráficos mostram, para cada arquivo de teste, o que ocorreu com a variação de *page faults* quando o tamanho da memória física cresce, mas as páginas possuem um tamanho fixo de 4 KB. Já os 4 últimos gráficos mostram, também para cada arquivo de teste, o que ocorreu com a variação de *page faults* quando o tamanho da memória fica constante (4096 KB), mas o tamanho das páginas varia de 2 KB a 64 KB (em potências de 2).

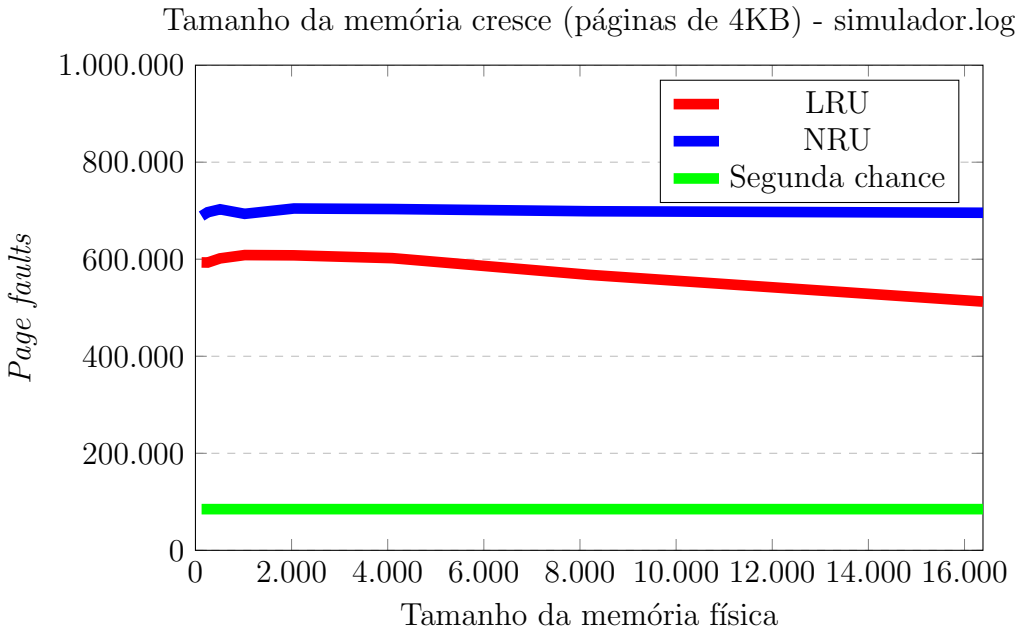
Verifica-se, portanto, que nos casos em que o tamanho da memória cresce e as páginas são de 4KB, o número de *page faults*, apesar de diminuir, é bastante elevado. Note também que o LRU obteve um desempenho melhor do que o NRU porém, ambos muito abaixo do segundo chance. No entanto, o tempo de execução do segunda chance foi maior do que o tempo de execução do LRU e do NRU. Em relação ao tamanho da memória constante (4096 KB) e as páginas variando de 2 KB a 64 KB (em potências de 2), o número de *page faults* decaiu consideravelmente, sendo que o desempenho do LRU continua ligeiramente melhor do que o do NRU e ambos são superados pelo segundo chance. Além disso, o tempo de execução do segundo chance continua sendo mais elevado em relação aos outros dois algoritmos utilizados.

3.1 Tamanho da memória cresce e páginas de 4KB

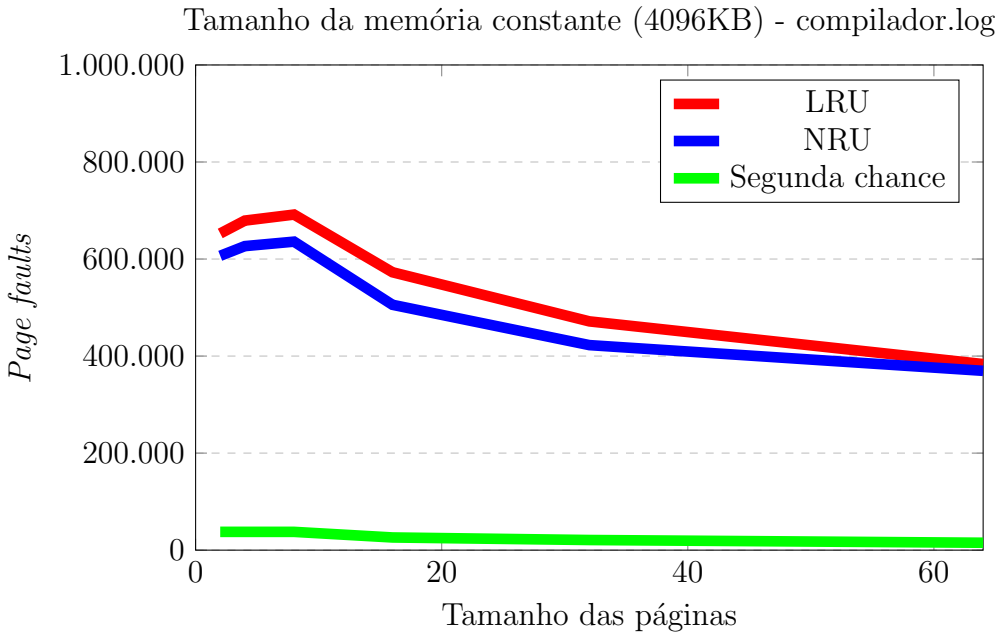


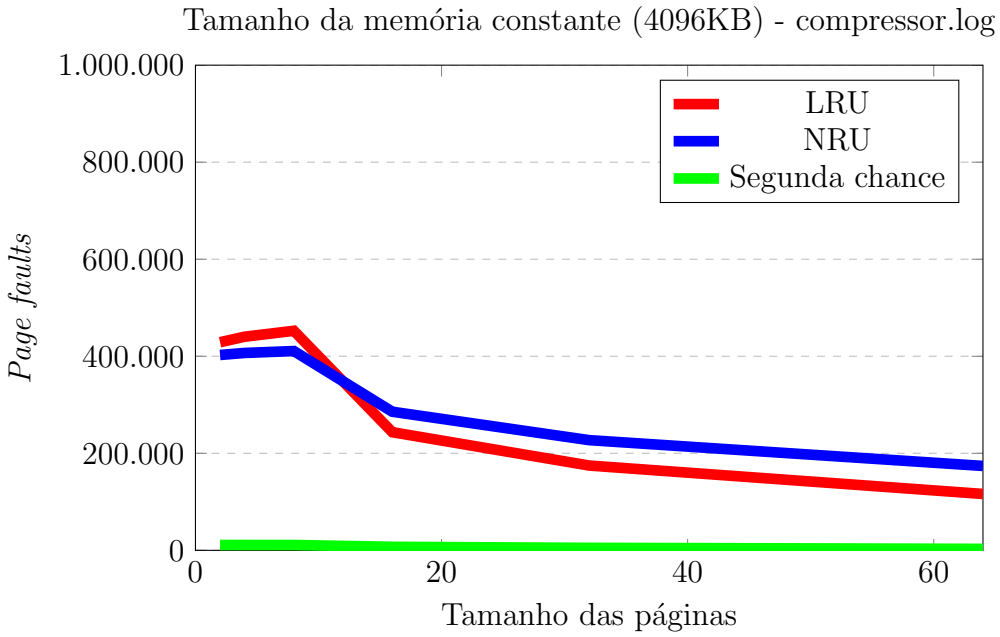


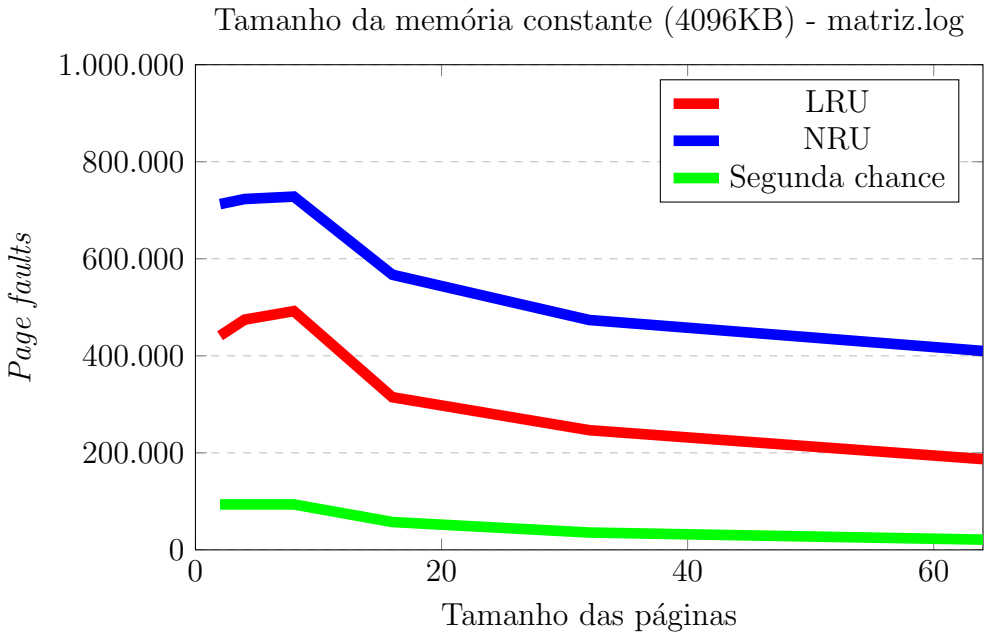


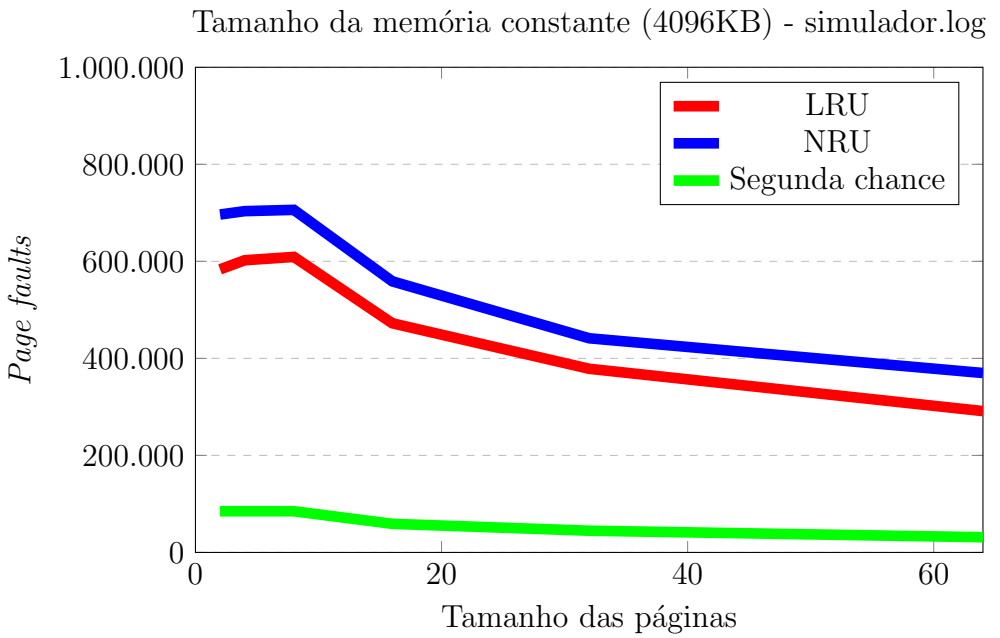


3.2 Tamanho da memória constante e as páginas variam de 2KB a 64KB




Tamanho da memória constante (4096KB) - compressor.log
1.000.000
800.000
600.000
400.000
200.000
0
Page faults
LRU
NRU
Segunda chance
0
20
40
60
Tamanho das páginas


Tamanho da memória constante (4096KB) - matriz.log
1.000.000
800.000
600.000
400.000
200.000
0
Page faults
LRU
NRU
Segunda chance
0
20
40
60
Tamanho das páginas


Tamanho da memória constante (4096KB) - simulador.log
1.000.000
800.000
600.000
400.000
200.000
0
Page faults
LRU
NRU
Segunda chance
0
20
40
60
Tamanho das páginas

4 Referências

TANENBAUM, A. S. **Sistemas Operacionais Modernos** 3 ed. São Paulo: Pearson, 2010.