Simulador de memória virtual - Relatório

- Caio Teles Cunha - caiotelescunha2001@gmail.com - 2020006434
- Ivan Vilaça de Assis - ivan.assis07@gmail.com - 2021421931

1. Resumo do projeto

Na nossa implementação do simulador temos duas estruturas fundamentais que são a tabela de páginas (“tabelaDePaginas”) e a struct *pagina* para representar a página. A tabela é um vetor de structs *pagina* criado dinamicamente no início da execução com o maior número de entradas que foi solicitado na especificação do trabalho.

Na struct inserimos as variáveis

- “modificada”: é setada para 0 e 1 para indicar, respectivamente, quando uma página sofreu acesso de escrita e de leitura;
- “valida”: setada para 1 para indicar que uma página já foi carregada na memória virtual.
- “referenciada”: é utilizada pelo algoritmo de segunda chance, podendo assumir valores 0 ou 1. Sendo setada para 1 sempre que a página é acessada.
- Ponteiro para *pagina* *pred e *next: foram usados para simular a memória física, participando da elaboração das estruturas usadas nos algoritmos de substituição: listas simples e duplamente encadeadas , lista circular e pilha.

Além destas estruturas, utilizamos algumas variáveis de controle para saber se a nossa “memória física" já estava totalmente ocupada e para coletar a quantidade de páginas sujas e de páginas falhas. Também criamos 3 ponteiros globais (“head”, “tail”, “watcher”) para ajudar na manipulação das nossas estruturas.

Em relação às funções, criamos uma para algoritmo de substituição pedido, sendo que para o LRU criamos uma função auxiliar responsável por atualizar a pilha de páginas de forma a sempre deixar a mais recente no topo quando ela é acessada e já está na “memória física”. Além dessas, temos a *main* que é responsável por ler os argumentos de entradas e por ler as sequências de acesso a memória do arquivo fornecido.

2. Decisões de projeto

Optamos por utilizar os ponteiros nas páginas porque assim teríamos a base de todas estruturas necessárias para controlar o que está na “memória física”. No FIFO uma fila, no random uma lista duplamente encadeada e no LRU optamos por uma fila duplamente encadeada, mas alteramos seu funcionamento para que funcionasse como uma pilha.

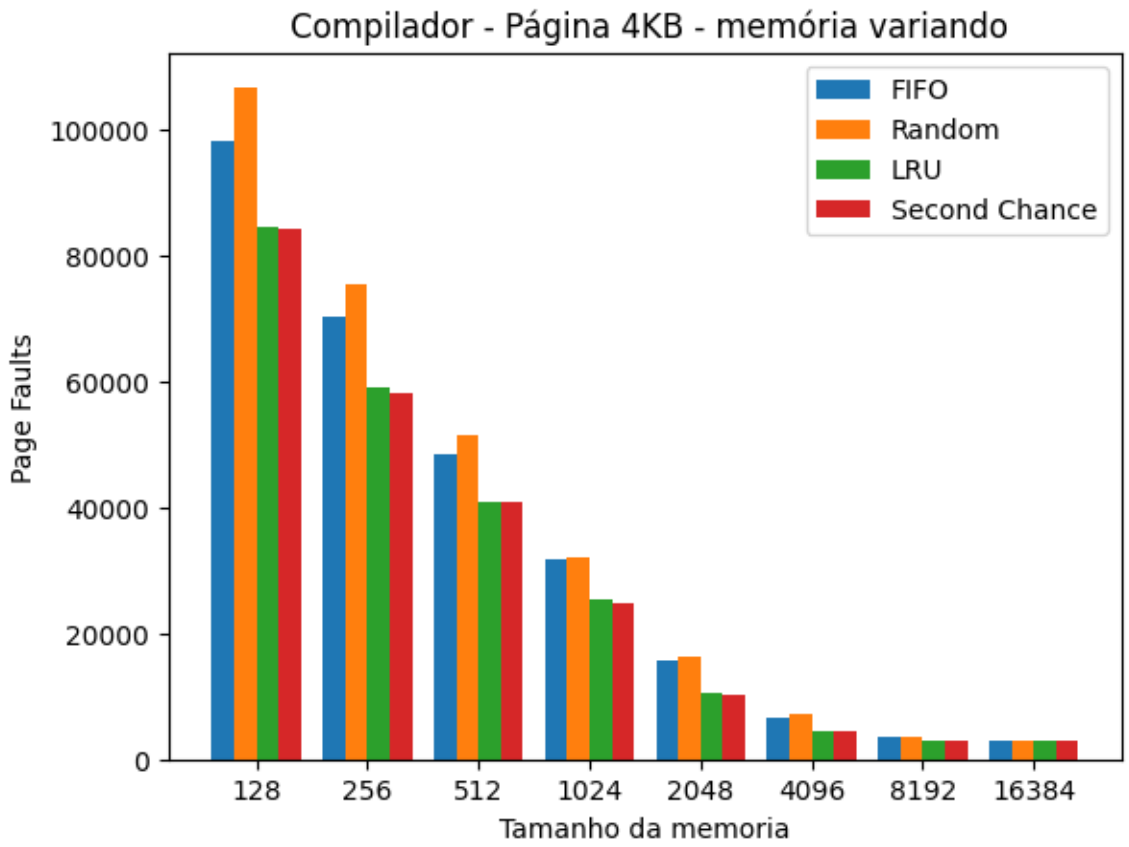
No segunda chance, usamos uma lista duplamente encadeada circular com base no que foi apresentado em aula, sendo que o ponteiro global “watcher” foi usado para ter controle de onde começar a buscar a página a ser tirada da fila circular em cada chamada do algoritmo.

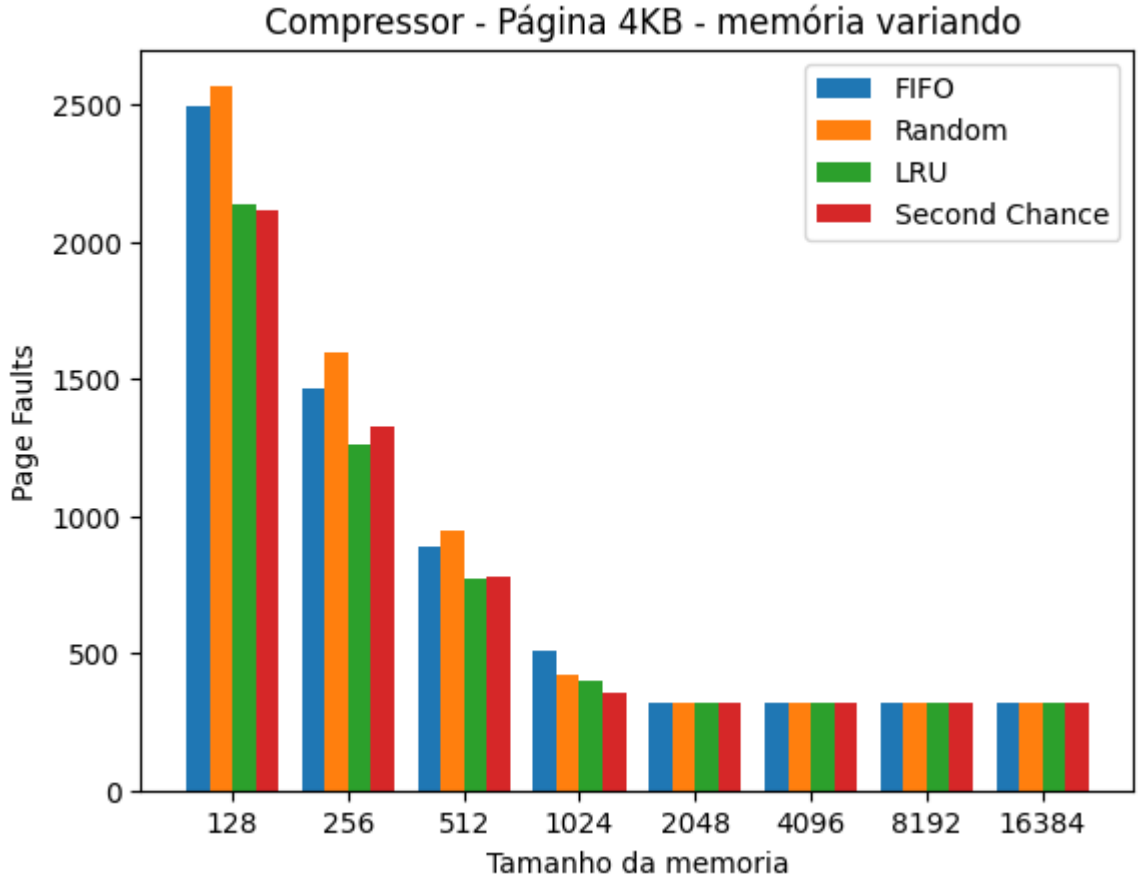
No LRU, optamos por colocar uma página no final da fila que representa a memória física toda vez que ela é acessada ao invés de inserir na struct *pagina* uma variável para armazenar os momentos em que uma página foi acessada. Essa decisão foi tomada devido à necessidade de percorrer todas as páginas em memória para encontrar a que foi utilizada há mais tempo e substituí-la na segunda abordagem. Logo, pensando em evitar este custo da busca, optamos por usar a fila de forma similar a uma pilha que sempre coloca no topo a página que foi acessada, deixando na base a variável utilizada há mais tempo e que, por isso, será substituída. A opção pela fila duplamente encadeada foi pela facilidade de desacoplar um nó da fila, sem necessariamente ter que percorrer a fila em busca do nó anterior ao que está sendo desacoplado.

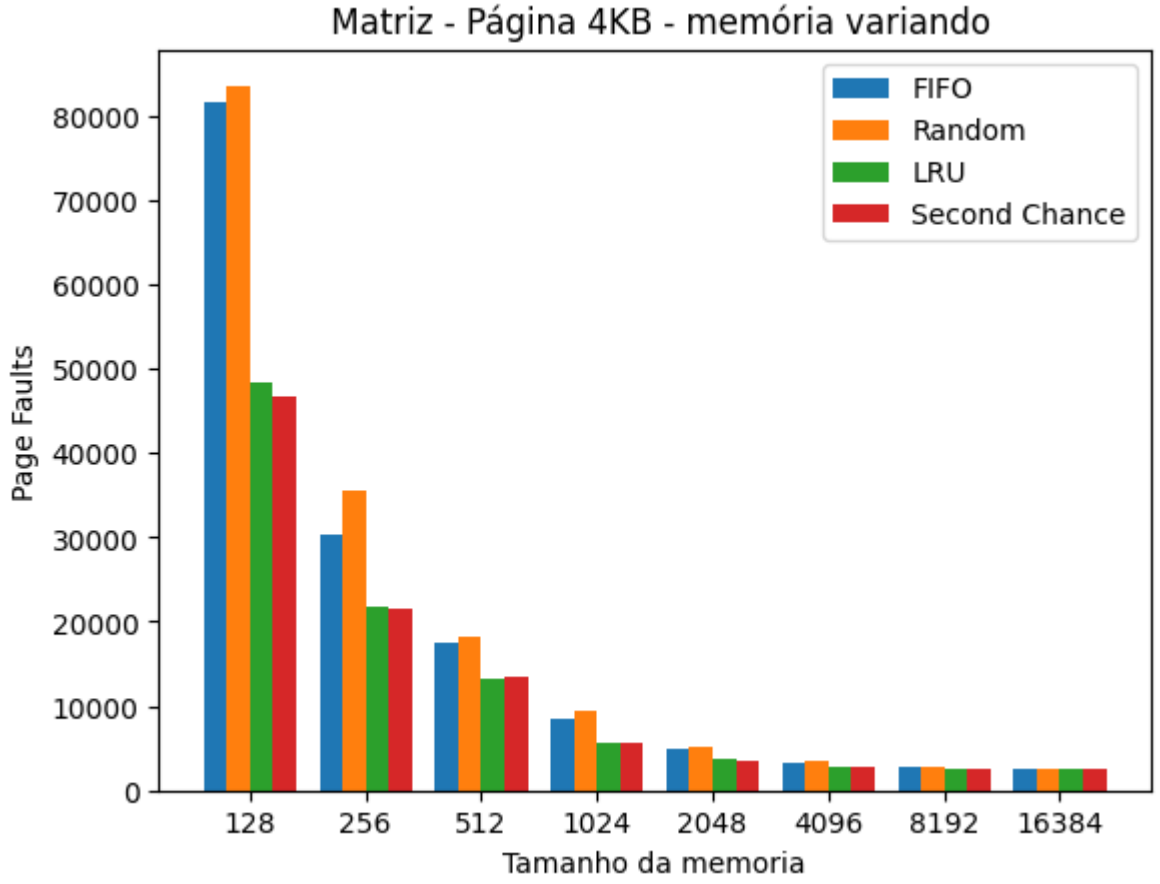
3. Análise do desempenho

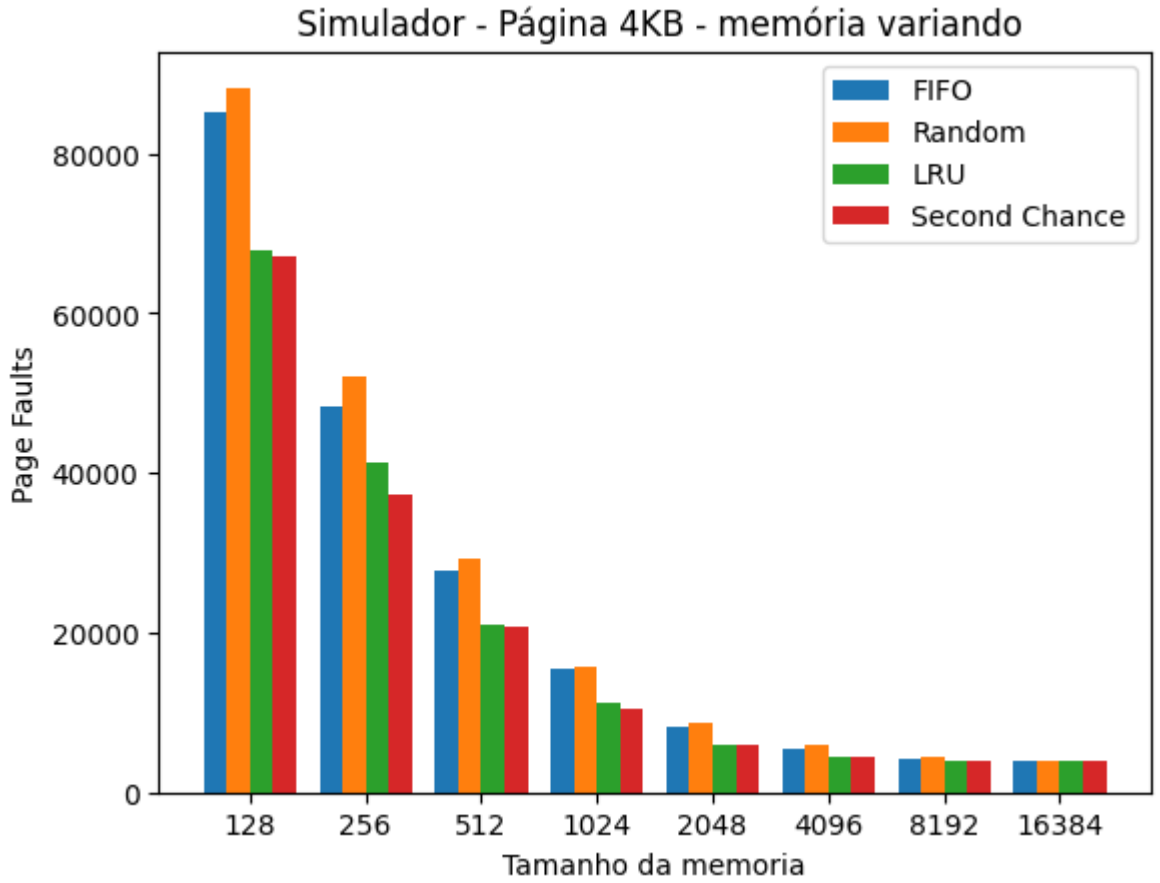
Para a análise de desempenho testarmos cada um dos programas disponibilizados em duas situações: Mantendo o tamanho da página em 4KB e variando o tamanho da memória de 128 KB a 16384 KB, mantendo o tamanho da memória física 8192 KB e variando o tamanho da página de 2KB a 64 KB.

Para memória variando:


Compilador - Página 4KB - memória variando
FIFO
Random
LRU
Second Chance
Page Faults
0
20000
40000
60000
80000
100000
128
256
512
1024
2048
4096
8192
16384
Tamanho da memoria


Compressor - Página 4KB - memória variando
FIFO
Random
LRU
Second Chance
Page Faults
0
500
1000
1500
2000
2500
128
256
512
1024
2048
4096
8192
16384
Tamanho da memoria


Matriz - Página 4KB - memória variando
FIFO
Random
LRU
Second Chance
Page Faults
0
10000
20000
30000
40000
50000
60000
70000
80000
128
256
512
1024
2048
4096
8192
16384
Tamanho da memoria


Simulador - Página 4KB - memória variando
FIFO
Random
LRU
Second Chance
Page Faults
0
20000
40000
60000
80000
128
256
512
1024
2048
4096
8192
16384
Tamanho da memoria

Como esperado, em todos os programas o aumento na quantidade de memória possível diminui a quantidade de *Page Faults*,pois a medida que aumenta a quantidade de páginas que se pode manter na memória ao mesmo tempo aumenta a probabilidade de *Hits*.

Em um caso de um programa que usa estruturas simples como o compressor, pelo gráfico podemos perceber que com o tamanho da memória em 2048 KB ele consegue colocar todas as estruturas que precisa dentro da memória principal, então a taxa de faults em todos os algoritmos é igual.

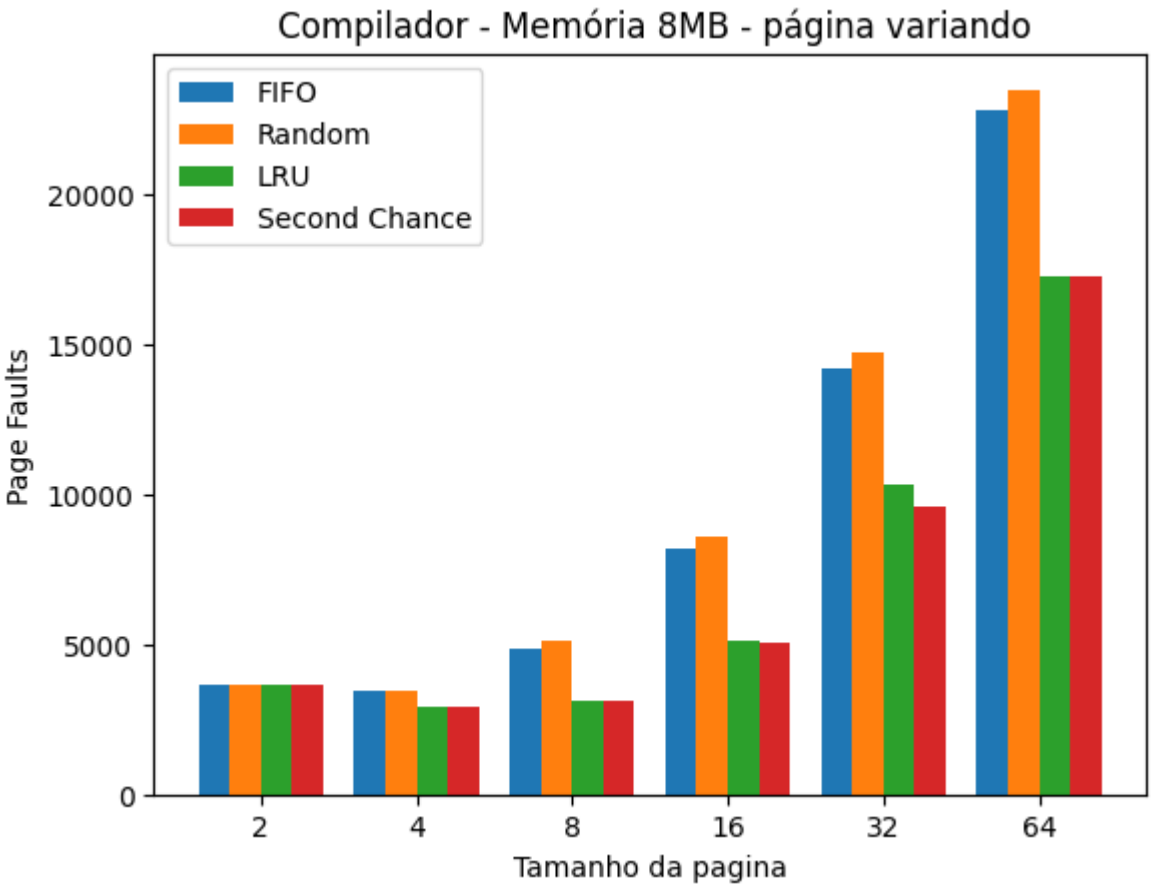
Logo, isso nos leva a concluir que quanto mais complexo as estruturas de dados maior deve ser o tamanho da memória para evitar que os dados necessários pela estrutura para executar corretamente não estejam na memória, gerando uma grande quantidade de page faults.

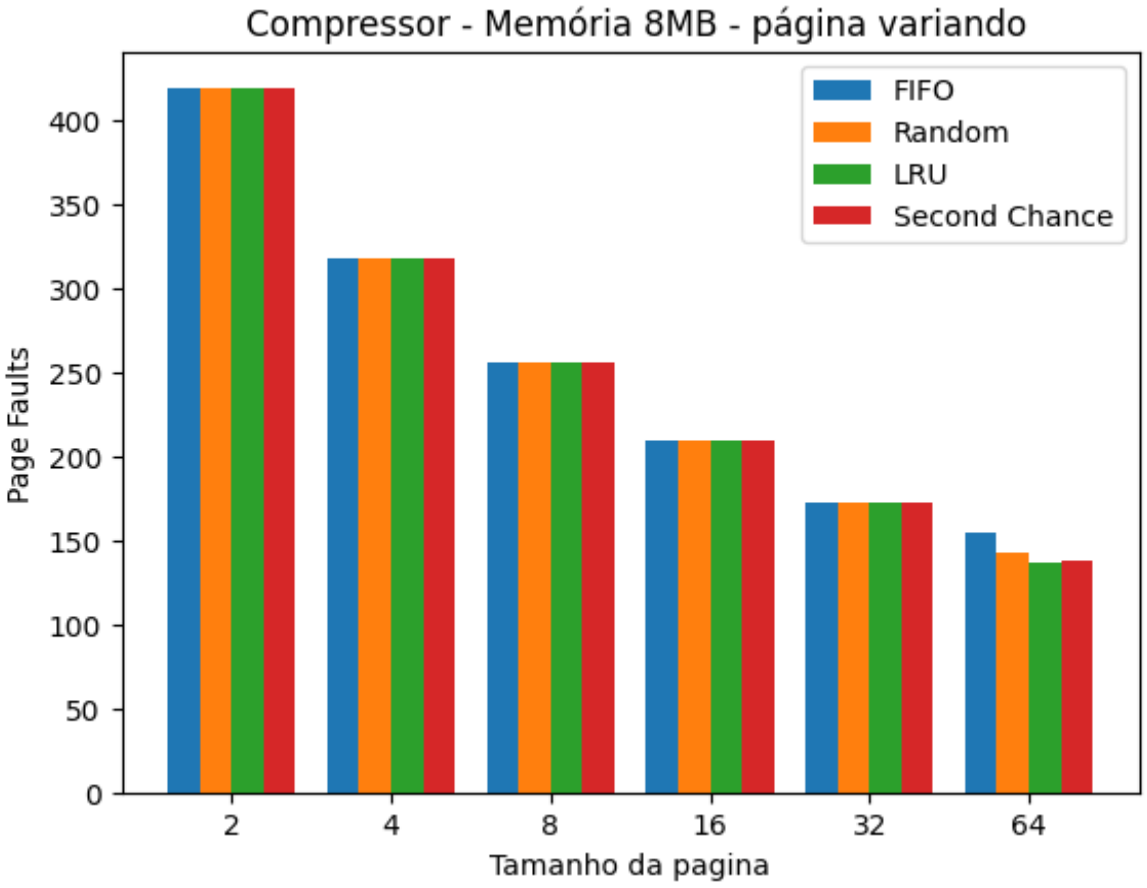
Podemos notar também que, no caso do programa da matriz, o método FIFO é muito pior do que os outros métodos. Podemos supor que ele utiliza várias vezes a mesma estrutura, suponho que uma operação matricial, mas com diferentes dados. Dessa forma, o FIFO retiraria frequentemente essa operação da memória principal levando a um maior número de *Page Faults*. Um algoritmo como o *Second Chance* manteria a operação na memória.

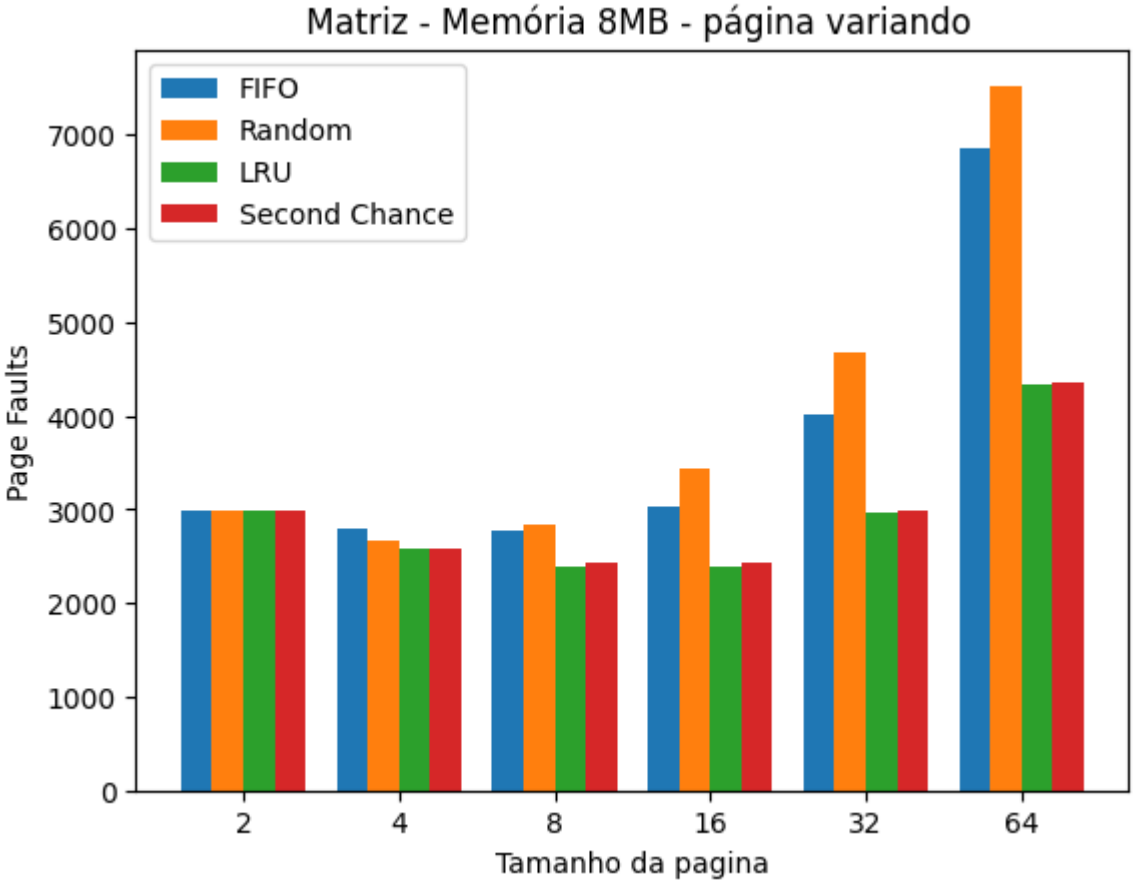
Ainda analisando o log de matriz, é relevante ver como seus *Page Faults* possuem valores bem mais altos que o compressor e pensar que isso pode ser uma reflexo da forma que estamos acessando as matrizes nos cálculos, visto que o acesso a uma coluna gera uma quantidade de *Page Faults* bem maiores, já que que elemento da coluna está em uma página diferente.

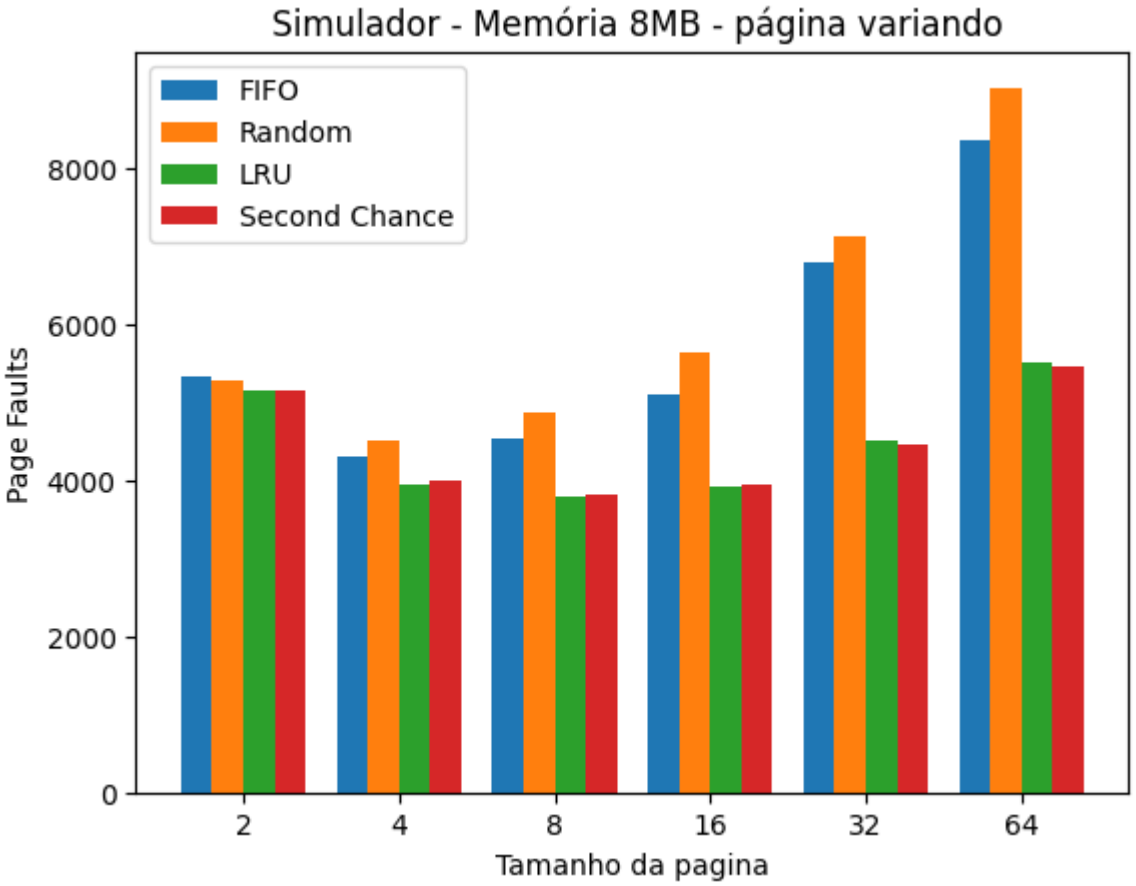
Já no caso do compilador, o programa utiliza uma grande quantidade de estruturas internas complexas e diferentes, então a diferença entre o *fifo* e o algoritmo de *Second Chance* não é tão drástica quanto no programa da matriz.

Para tamanho de página variando:


Compilador - Memória 8MB - página variando
FIFO
Random
LRU
Second Chance
Page Faults
0
5000
10000
15000
20000
2
4
8
16
32
64
Tamanho da pagina


Compressor - Memória 8MB - página variando
FIFO
Random
LRU
Second Chance
Page Faults
0
50
100
150
200
250
300
350
400
2
4
8
16
32
64
Tamanho da pagina


Matriz - Memória 8MB - página variando
FIFO
Random
LRU
Second Chance
Page Faults
0
1000
2000
3000
4000
5000
6000
7000
2
4
8
16
32
64
Tamanho da pagina


Simulador - Memória 8MB - página variando
FIFO
Random
LRU
Second Chance
Page Faults
0
2000
4000
6000
8000
2
4
8
16
32
64
Tamanho da pagina

Nestes gráficos o que fizemos foi analisar o aumento do tamanho da página/quadro na memória física. Sendo que o esperado seria que com o aumento do tamanho da página, a quantidade de fragmentações internas aumentaria, o que ocasionaria um aumento na quantidade de *Page Fault* por estar utilizando de maneira não ótima a memória. Com a diminuição do número de quadros na memória, a demanda total de quadros dos processos ultrapassa a quantidade de quadros disponíveis, fazendo eles pegarem dados do disco e trazer para memória gerando mais *Page Faults*.

No caso do compilador, podemos perceber que o aumento no tamanho das páginas aumentou o número de *Page Faults*. Esse programa utiliza uma grande quantidade de estruturas complexas em sua execução, então podemos supor que, por necessitar de mais estruturas na memória para executar, diminuir o número de quadros (mesmo que aumentando o seu tamanho) atrapalha a execução do algoritmo fazendo com que ele faça mais acessos ao disco.

No caso do compressor, a memória escolhida para o teste deixa o número de *Page Faults* constantes para todos os tamanhos de página menos o de 64 KB. Também podemos ver que o número de *Page Faults* diminui à medida que aumentamos o tamanho das páginas. Podemos supor que o programa do compressor precisa trazer para a memória arquivos inteiros para comprimi-los, então páginas maiores podemos colocar arquivos inteiros em uma página.

No caso da matriz, o aumento do tamanho das páginas/quadros amplificou mais os *Page Faults* após 16 KB. Isto, muito provavelmente, aconteceu porque as linhas das matrizes envolvidas no processo devem ter um tamanho próximo a 16 KB que era atendido pelo número de quadros, sendo que com o aumento do tamanho das páginas/quadros a demanda passou a quantidade de quadros na memória não se mostrou suficiente fazendo com que fosse necessário carregar os dados da memória virtual, gerando *Page Faults.*

No caso do simulador percebemos que ele se comporta de maneira bem semelhante para todos os tamanhos de página, atingindo o menor número de *Page Faults* em tamanhos de página medianos. Podemos supor

que as estruturas do programa tem tamanho por volta de 8KB e por isso a fragmentação interna diminui gerando menos *Page Faults*.

4. **Referências bibliográficas**

Slides e vídeos gravados da disciplina no moodle

https://www.youtube.com/watch?v=pJ6qrCB8pDw

Silberschatz, A., Galvin, P. B., & Gagne, G. (2022). Memória Virtual. In Silberschatz, A., Galvin, P. B., & Gagne, G.,Fundamentos de Sistemas Operacionais (9ª, 449-524). Local de publicação: Grupo Nacional Editora.