Trabalho Prático 2 - SO

Júlia Fonseca de Sena - 2018054508
Lucas Resende Pellegrinelli Machado - 2018126673

1) Resumo

O trabalho consistiu em criar um simulador de memória virtual onde, a partir de uma entrada composta por diversos endereços de acesso, o programa deve alocar cada uma das páginas a um endereço da memória física. O tamanho das páginas e o tamanho total da memória física também são parâmetros de entrada para o algoritmo. Para fazer essa alocação, foram propostos quatro algoritmos diferentes: Least Recently Used, Second Chance, First in First out e um algoritmo aleatório.

Esses algoritmos foram implementados e comparados um com o outro em relação a tempo de execução, número de page faults e quantidade páginas sujas gerados ao longo de sua execução. O resultado desses testes serão discutidos ao longo desse relatório.

a. Algoritmo de substituição criado

Dentre os algoritmos a serem utilizados para a troca de páginas na memória física, um deles deveria ser escolhido pelo grupo. O algoritmo escolhido por nós, como citado anteriormente, é um algoritmo aleatório onde, para cada page fault encontrado, a nova página substituirá a página em um quadro aleatório na memória física.

Esse algoritmo foi escolhido devido a importância de um teste de hipótese comparando os algoritmos que se propõem a serem opções boas para o problema com uma solução meramente aleatória.

b. Análises de complexidade

Começando pelo Least Recently Used, o primeiro passo é procurar pela página nos quadros da memória. No pior caso, passa por todos os n quadros comparando as páginas sem sucesso (O(n)), e parte para a procura do quadro com o acesso menos recente. Passa-se novamente por todos os quadros (no pior caso, não encontrando um quadro vazio para colocar a página) e salva-se a posição do que foi usado menos recentemente, com uma complexidade O(2n), devido às duas comparações (if) no loop. Assim, para o algoritmo completo: O(n) + O(2n) = O(2n) = O(n).

O próximo algoritmo, First in First Out, é de mesmo custo, já que a única diferença é que no segundo loop, ao invés de procurar pelo quadro com acesso menos recente, busca o cuja página foi adicionada menos recentemente. Logo, FIFO é O(n).

Partindo para o Segunda Chance, possui o mesmo loop inicial que checa se a página já está no quadro, O(n), e caso não encontre, procura o próximo elemento da lista circular de quadros que possui bit de referência 0. No pior caso, todos os bits serão 1, e o programa terá que passar por toda a lista para voltar ao elemento em

que iniciou para encontrar o 0. Por passar por todos os elementos, possui, também, complexidade O(n), portanto o algoritmo é O(n) + O(n) = O(n).

O algoritmo aleatório criado é o mais simples, apesar de também ser O(n). Há somente um loop para checar se a página está nos quadros, e, se não estiver, é escolhido um aleatório para colocá-la. Pode-se perceber que os três algoritmos possuem então a mesma complexidade de tempo.

Considerando o programa como um todo, os algoritmos são executados para cada linha lida do arquivo passado como parâmetro. Dessa forma, sendo x o número de linhas, o custo da execução do algoritmo é O(xn). Além disso, há uma inicialização dos quadros com valores padrões, por um loop O(n). Assim, é O(n) + O(xn) = O(xn).

Quanto à complexidade de espaço, a única ocupação significativa da memória é o vetor de quadros, de tamanho n. De resto, há somente variáveis simples. Desse modo, a complexidade é O(n).

2) Decisões de projeto

A primeira decisão a ser tomada durante a implementação do projeto foi como representar as páginas e quadros no algoritmo de forma a facilitar (e manter a complexidade de tempo baixa) durante a execução do programa. A representação escolhida é a mais intuitiva possível e garante que ao atualizar os quadros, o algoritmo será assintoticamente proporcional à quantidade de quadros na memória principal e não à quantidade de páginas possíveis, garantindo eficiência na execução.

Primeiramente temos um `struct pagina` que guarda informações como o índice da página, se a página está suja e quando ela foi adicionada na memória física. Essa estrutura será criada toda vez que precisamos colocar uma nova página na memória física.

Para armazenar as páginas que estão na memória física, temos um `struct quadro` que é responsável por guardar informações sobre uma página guardada em um dos espaços da memória. A estrutura tem como atributos uma referência a página armazenada por ele e quando aconteceu o último acesso àquele quadro.

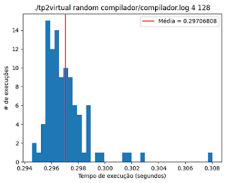
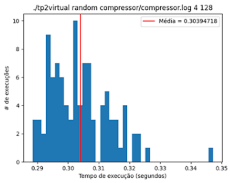
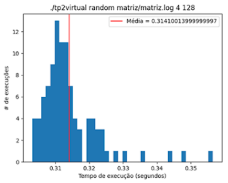
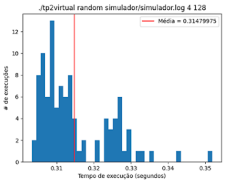
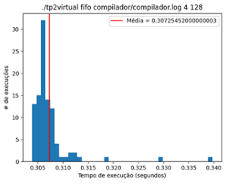
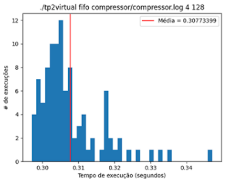
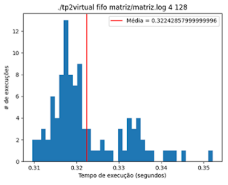
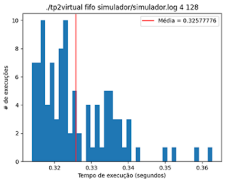
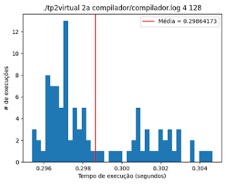
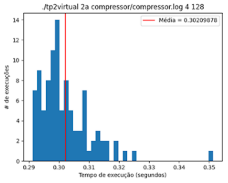
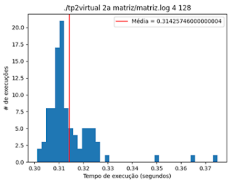
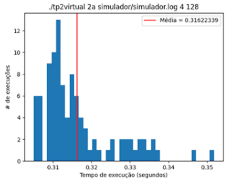
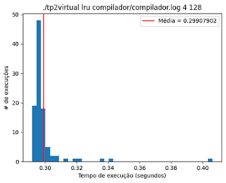
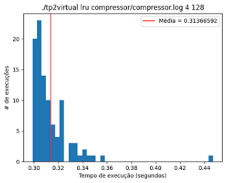
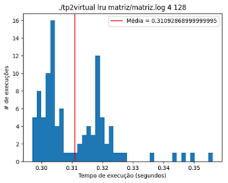
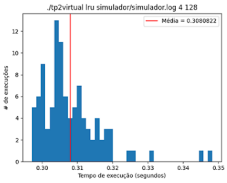
Com a estrutura de dados definida, a implementação dos algoritmos foi bastante direta. Foi decidido separar cada um deles em uma função separada que recebe como referência os quadros para serem editados e um objeto que contém cada um dos contadores necessários para gerar a estatística final de cada execução (como número de page faults, número de páginas sujas e acessos à memória).

O único algoritmo com uma peculiaridade na implementação foi o algoritmo de segunda chance, em que ao invés de guardarmos explicitamente o bit de referência na estrutura de dados da página, utilizamos os valores já armazenados do último tempo de acesso e do tempo de inclusão da página na memória para indiretamente obter o valor do bit de referência. Para identificar se o bit de referência é 0 ou 1, basta descobrirmos se o último tempo de acesso é diferente do tempo de inserção e, quando for hora de voltar uma página com o bit de referência que está em 1 para 0, simplesmente atribuímos de volta o valor do tempo do último acesso ao tempo de inserção.

Outra decisão que ficou a cargo dos alunos foi como lidar com linhas com somente o endereço, sem o ‘R’ ou ‘W’ na memória. A abordagem escolhida foi não contar esses casos como acesso à memória.

3) Análise de Desempenho

Foram executadas **100 vezes** as **16 combinações** dos **4 algoritmos** com os **4 conjuntos de endereços** e a seguir seguem os histogramas representando o **tempo de execução** para cada um deles assim como a média do tempo de execução de cada algoritmo e conjuntos de testes. Cada um dos testes usou **4KB como tamanho das páginas** e **128KB como tamanho total da memória**.

	Compilador	Compressor	Matriz	Simulador	
Aleatório					média 0.30748
First in first out					média 0.31580
Segunda chance					média 0.30781
Least recently used					média 0.30794
	média 0.30051	média 0.30686	média 0.31543	média 0.31622	

Como pode ser observado pela tabela a seguir, nenhum dos métodos é muito mais rápido que os outros, todos ficando na margem de erro um do outro. Isso implica que podemos escolher o algoritmo com os melhores resultados sem grandes comprometimentos em termos de velocidade de execução.

Para analisar a qualidade de resultados, a seguir temos a tabela com a quantidade de page faults e quantidade de páginas sujas durante a execução de cada algoritmo.

Número de page faults

	Compilador	**Compressor**	**Matriz**	**Simulador**
Aleatório	106644	2619	83129	88700
First in first out	98067	2497	81638	85283
Segunda chance	84178	2116	46692	67236
Least recently used	84401	2133	48254	67747

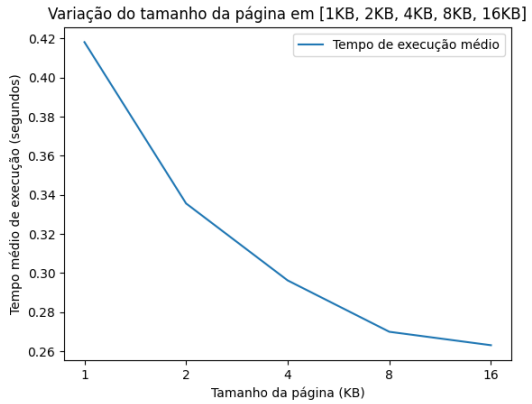
Número de páginas sujas

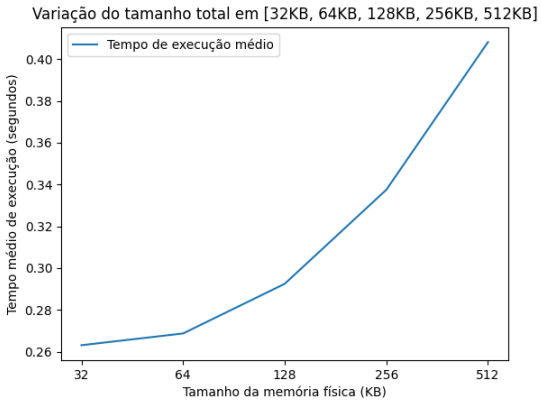
	Compilador	**Compressor**	**Matriz**	**Simulador**
Aleatório	17097	866	18028	19498
First in first out	16759	851	18172	18805
Segunda chance	11576	696	9369	13521
Least recently used	11737	702	9674	13730

Como pode ser observado pelas tabelas acima, o algoritmo que gerou o menor número de páginas sujas e o número de page faults foi o algoritmo da segunda chance, que manteve a liderança para ambas as métricas em todos os casos de teste. Isso acarreta menos acessos à memória física, os quais possuem custo maior. Percebe-se que o algoritmo aleatório e o FIFO chegaram a quase o dobro de ambos parâmetros no matriz.log, podendo haver efeito negativo de performance grande dependendo da escala do problema.

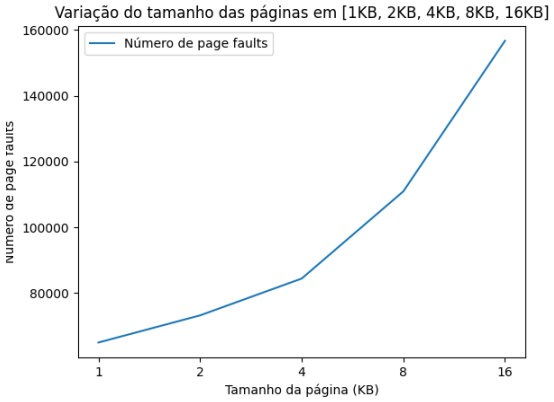
Apesar do algoritmo Segunda Chance ter ficado na frente em todos os arquivos passados, percebe-se que não houve uma diferença grande entre suas estatísticas e as do LRU, portanto ambos são alternativas boas para a reposição. Já o FIFO ficou surpreendentemente próximo do aleatório, com performances bem inferiores aos outros dois.

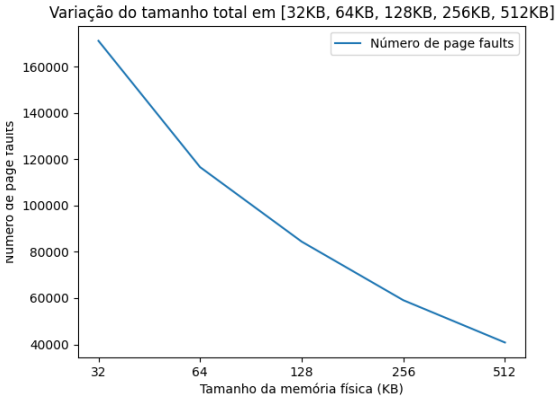
A seguir temos gráficos indicando a variação das estatísticas obtidas quando alteramos os valores do tamanho das páginas e do tamanho total da memória. Esses testes utilizaram como base o algoritmo **least recently used** com o arquivo de entrada **compilador.log**.



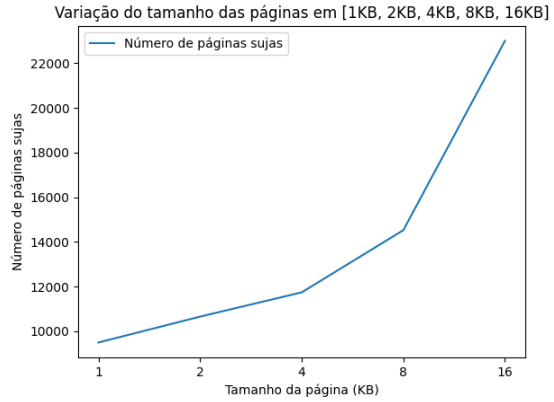


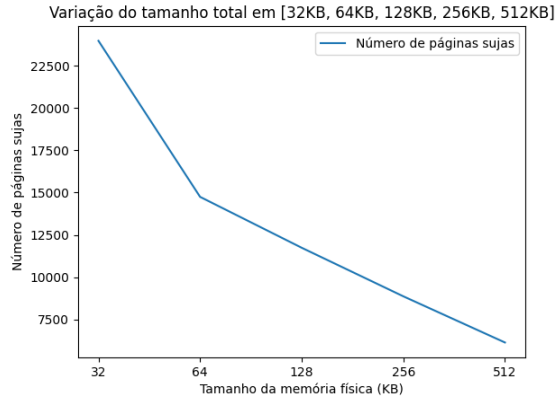
Acima podemos observar o comportamento esperado para a variação do tempo de execução do programa. Quanto menor o tamanho das páginas, mais quadros temos na memória física e, como dito antes, a implementação tem complexidade de tempo assintótica proporcional ao número de quadros. Esse mesmo efeito acontece quando aumentamos a capacidade da memória física, que também gera um aumento no número de quadros.





Acima temos a análise do número de page faults ao variar o tamanho das páginas e o tamanho da memória física e mais uma vez temos o resultado esperado. Com o aumento do tamanho das páginas, temos menos quadros na memória física, causando uma maior chance de que, caso uma nova página seja visitada, ela não esteja por lá, gerando um page fault. O mesmo efeito é observado ao diminuir a capacidade total da memória física pelos mesmos motivos.





Por fim fizemos a análise do número de páginas sujas durante a execução do algoritmo e, analogamente as últimas análises, a intuição por trás dos resultados vem da análise de quantos quadros temos na memória física. Com o aumento do tamanho das páginas, temos um decréscimo no número de quadros e assim mais endereços apontam para a mesma página, fazendo com o que mais endereços sejam passíveis de sujar as páginas carregadas na memória física. Analogamente, ao decrescer o tamanho da memória física o mesmo efeito se repete.