Trabalho 2 de Sistemas Operacionais
Implementação de memória virtual

Gabriel Souza Baggio e Guilherme Brizzi

1. INTRODUÇÃO E OBJETIVOS

Após a implementação do trabalho 1 - cujo objetivo foi garantir o bom funcionamento do sistema operacional com múltiplos processos -, observou-se dois principais problemas:

- É necessário saber de antemão o endereço dos programas que se deseja carregar na memória, e, caso não caibam na memória principal, é impossível realizar a aplicação.
- Não é possível implementar proteção de memória, a fim de impedir o acesso de um processo em um endereço que não é o seu.

Com a implementação de memória virtual, é possível, além de resolver esses problemas, carregar programas muito maiores que a memória principal, ao deixar parte deles no disco (memória secundária). No trabalho, foram implementados o carregamento de programas na memória secundária, a transferência entre quadros da memória física e páginas da memória virtual e o armazenamento dessas informações no disco.

Durante o código, algumas das principais implementações para cumprir os itens exigidos no trabalho foram:

- Criação de um disco para memória secundária no SO.
- Uma forma de rastrear o uso da memória principal no SO.
- O carregamento dos programas (exceto ‘trata_int’) na memória secundária.
- Uma tabela de páginas única para cada processo.
- O tratamento de interrupções de erro de página.
- Algoritmos de substituição de páginas (caso a memória principal não tenha blocos disponíveis).

Além do valor didático do trabalho, o objetivo do projeto foi analisar diferentes algoritmos de substituição de páginas em diferentes configurações de tamanho de página e memória principal. Esses resultados estão dispostos na seção de análise do trabalho.

2. METODOLOGIA

2.1. Sistema operacional com memória principal e secundária

Para que o SO suportasse memória virtual, foi necessário criar uma instância adicional de memória dentro do SO, que foi chamada de disco. Nele, serão carregados todos os programas - exceto o que trata interrupções -, e, para isso, foi criada também uma variável para guardar a próxima posição ocupável do disco. Além disso, dentro do SO, foi criada uma

estrutura para rastrear as páginas físicas disponíveis dentro da memória principal. Ela guarda informações dos blocos, como por exemplo, se eles estão livres ou ocupados. Caso ocupados, também é guardado o PID do processo o qual o ocupa. Além disso, os processos agora também guardam sua própria tabela de páginas.

2.2 Criação e carregamento de programas

O programa *'trata_int.maq'* é o único carregado na memória física, ocupando, ao inicializar o SO, os dois primeiros blocos físicos. Fora ele, todos os programas são carregados diretamente no disco, na função *'so_carrega_programa_na_memoria_virtual*'. Dessa forma, é garantido que cada programa seja carregado de forma íntegra, sem perder nenhuma parte caso o seu todo não caiba dentro da memória principal.

2.3 Tratamento de interrupção *page fault*

Quando o programa procura em sua memória principal e não encontra a página que precisa, ele deve procurar na memória secundária e inserir essa página na sua memória principal, para poder trabalhar com ela. Isso é feito em blocos, ou seja, a página da memória secundária é trazida por completo para a um bloco na memória principal, sem poder acontecer pela metade. Esse problema possui duas abordagens: uma para quando há espaço disponível na memória e outra quando a memória principal está cheia.

2.3.1 Existe bloco livre na memória

Caso o programa encontre um bloco disponível na memória principal, ele pegará o endereço do disco e o endereço virtual (que é o registrador complemento do processo atual) e fará a devida transferência para o endereço físico. Esse processo basicamente passa o endereço do disco para o endereço da memória principal do SO. Após isso, o programa acerta as métricas do bloco físico que será ocupado e define a tabela de páginas do processo.

2.3.2 Memória principal cheia

Quando a memória principal está cheia, é necessário trocar um bloco pelo novo que quer ser inserido. Para isso, primeiro o programa escolhe o bloco físico que deve ser substituído. Com isso, ele verifica se houve alguma alteração nesse bloco - verifica o bit de alteração na tabela de página - e, caso haja, ele guarda o bloco alterado no disco. Após isso, ele pega as informações da página no disco e passa para esse bloco na memória principal, alterando também as métricas do bloco físico e o quadro da tabela de páginas.

2.4 Algoritmos de trocas de páginas

Dois algoritmos de substituição de páginas foram implementados. O primeiro deles é o FIFO, um algoritmo simples de fila, que substitui as últimas páginas que entraram. O segundo é um algoritmo chamado 'segunda chance'. Ele procura primeiramente os processos

que têm o bit de acesso zerado, e, caso não encontre um, ele zera um que tenha o bit de acesso em 1. O bit de acesso é levado para 1 caso seja feito um acesso ao bloco, isso faz com que os processos precisem não ser acessados duas vezes seguidas para serem removidos com preferência, garantindo assim, para os processos que foram acessados anteriormente, uma maior prioridade.

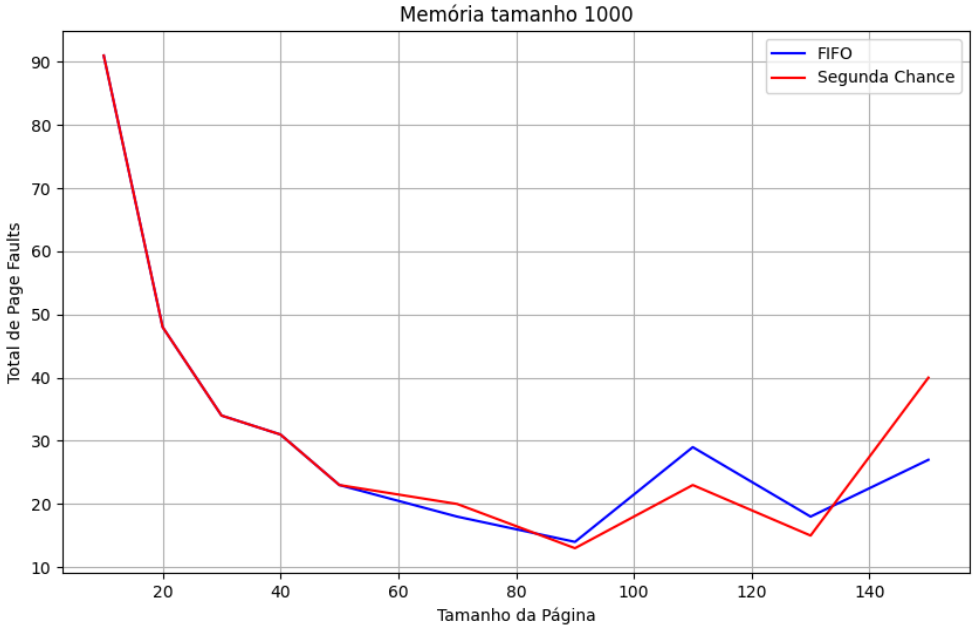
2.5 Métricas dos processos

O processo guarda novas métricas do trabalho, para mostrar no fim o número de *page faults*. Essas métricas são úteis para análise e comparação, pois permitem descobrir quais configurações e algoritmos são os melhores.

3. RESULTADOS E ANÁLISE

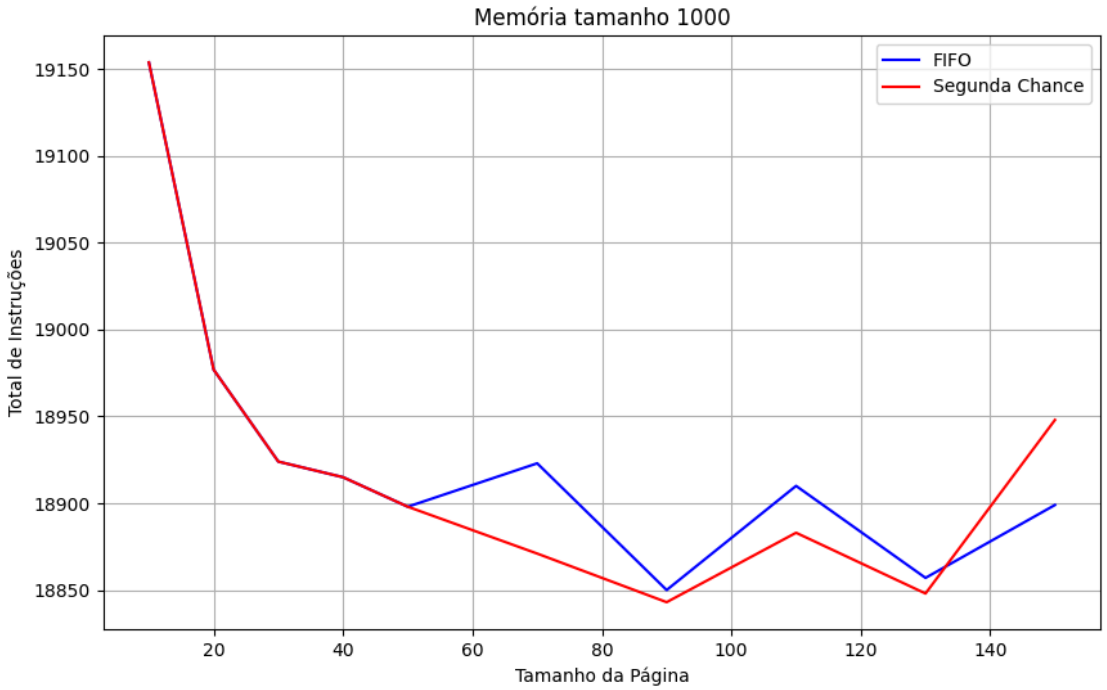
O primeiro passo para análise foi descobrir um bom tamanho para a memória principal. Ao deixar o tamanho de página em 10 e um tamanho de memória principal de 10000, foi percebido um total de 91 *page faults*. Isso quer dizer que todos os programas ocupam, juntos, por volta de 910 endereços de memória. Com essa primeira análise, é possível estipular boas métricas de tamanhos de memória para comparação.

Dessa forma, os tamanhos de memória principal decididos para análise foram: tamanho de memória 1000 (para um teste com muita memória), tamanho de memória 500 (cerca de metade do tamanho dos processos cabe na memória) e tamanho de memória 250 (uma memória bem pequena, que será necessário realizar muitas trocas de página). Nesses três tamanhos, serão feitas análises dos dois algoritmos de substituição em diferentes tamanhos de página. Vale lembrar que, em todas as análises, o *quantum* do SO foi mantido em 10 e o algoritmo de escalonamento usado foi o com prioridade.



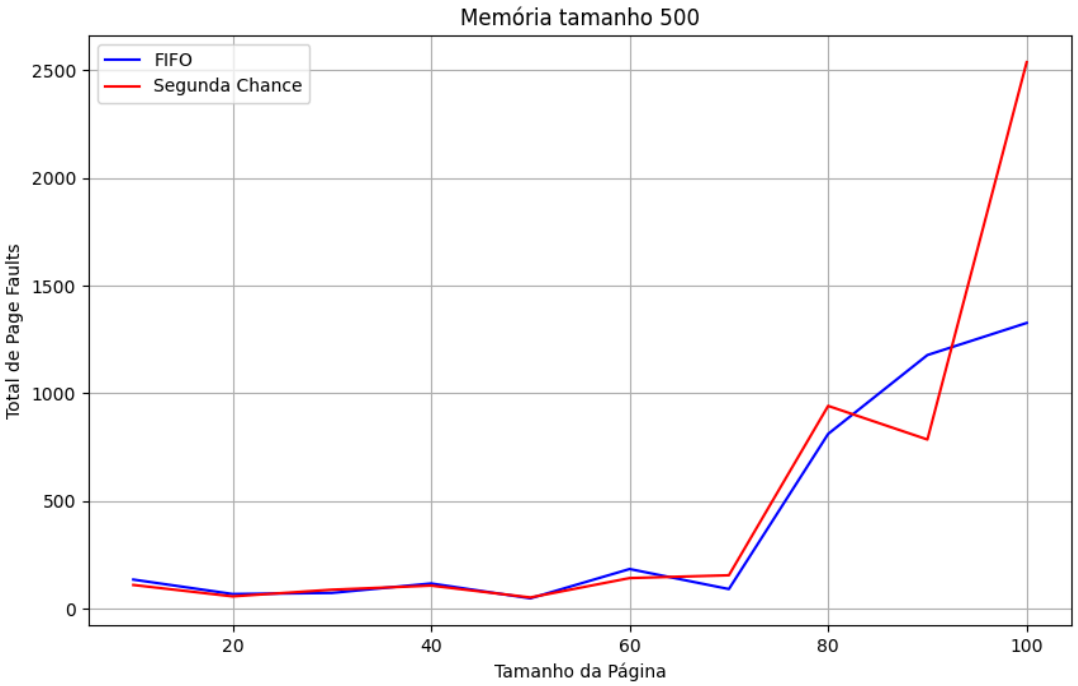
Analisando o gráfico dos testes com tamanho de memória 1000, percebe-se a grande ineficiência de se ter tamanhos de página menores que 4% do tamanho total da memória. Ao baixar de um tamanho de página de 40, o número de *page faults* aumentou drasticamente. Além disso, percebe-se que os dois algoritmos tiveram uma performance parecido quanto ao nível de *page faults*.

Por outro lado, ao analisar uma comparação do total de instruções - disposta abaixo -, o algoritmo FIFO teve um desempenho um pouco pior se comparado ao 'segunda chance', exceto no último caso.

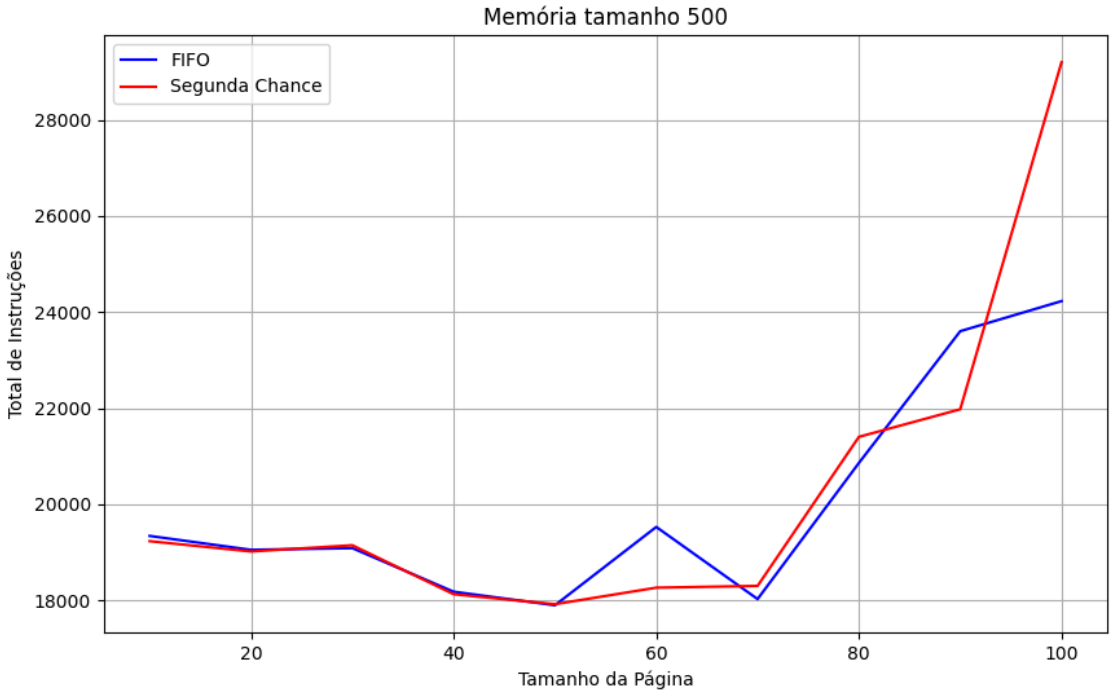


Analisando os resultados com tamanho de memória 500, nota-se um grande aumento no número de *page faults*. Isso acontece porque, ao tratar o programa com um tamanho de memória 1000, quase sempre todos os programas cabiam na memória por completo. Nessa segunda análise, é necessário fazer substituições de página para que os programas sejam executados, fazendo que os algoritmos de substituição tenham uma maior diferença.

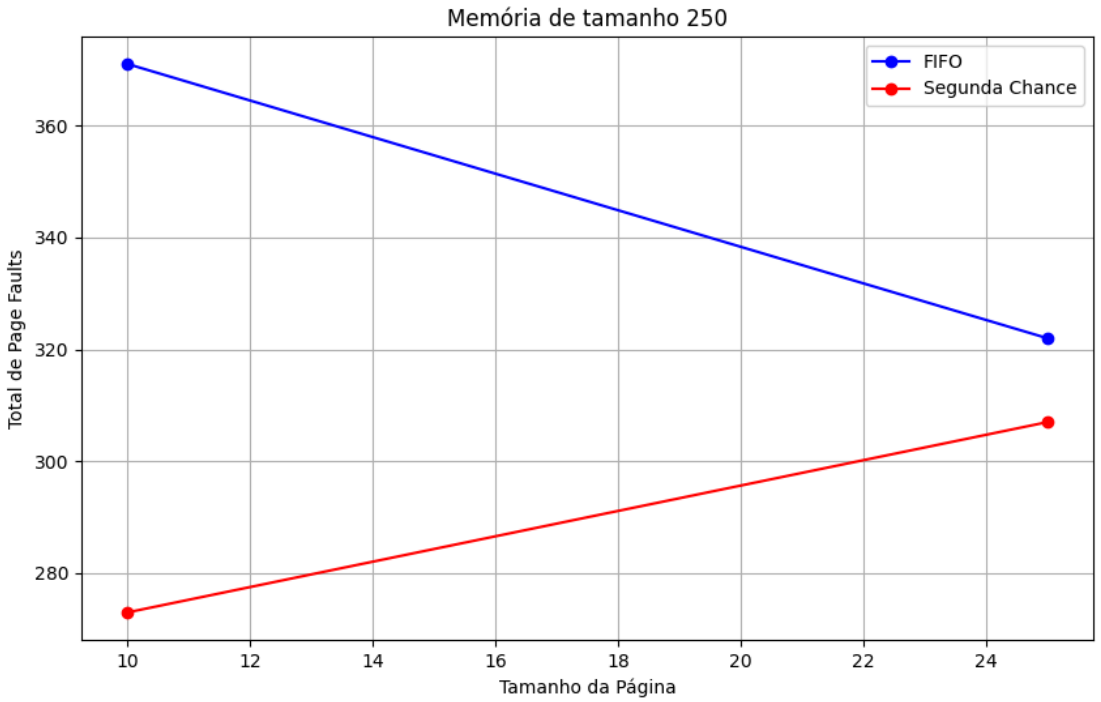
No gráfico, percebe-se que os algoritmos trabalham de forma similar, exceto com o tamanho da página 100, que o 'segunda chance' opera de forma mais ineficaz.



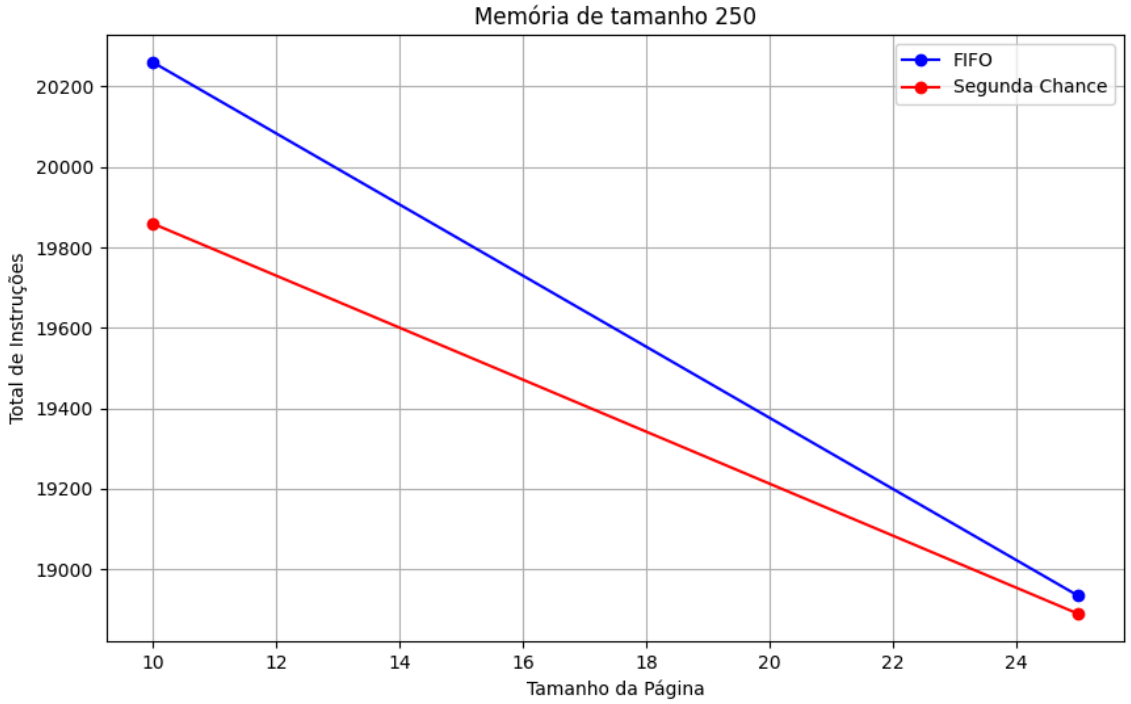
Fora isso, também se percebe o enorme aumento no número de page faults e instruções quando se aumenta o tamanho de página para um quinto do tamanho total de memória.



Observando os dados de uma memória com tamanho 250, pode-se perceber que algoritmo ‘segunda chance’ é muito mais eficaz que o FIFO com tamanhos de página em 10, e um pouco mais eficaz se elevados para 25.



Em geral, o algoritmo de segunda chance opera de forma melhor que o FIFO quando a memória tem tamanho 250 para esses programas.



Após feitos esses testes, também foram analisados os dados com uma memória de tamanho 100. Entretanto, com o tamanho da memória sendo tão pequeno, a maioria das configurações de tamanho de página não funcionaram corretamente. Diminuindo mais ainda, para um tamanho de memória de 50, a execução não funcionou corretamente, pois o tamanho de memória era muito pequeno para rodar os programas.

4. CONCLUSÃO

Após a implementação de memória física e virtual no sistema operacional, foi possível melhorar a aplicação para suportar programas maiores que a memória principal. Isso, em aplicações realísticas de sistemas operacionais, é imprescindível para o bom funcionamento do sistema, pois é impraticável armazenar todas as informações na memória principal.

Além disso, com base nas análises obtidas, percebeu-se que tomar páginas menores que 4% do tamanho total da memória principal na aplicação demonstrada aumentou drasticamente a quantidade de *page faults*. Esse mesmo aumento foi observado ao tomar páginas com tamanho superior a 20% do total da memória física.