Trabalho Prático 2
Memória Virtual

André Luiz Moreira Dutra - 2019006345

Marcos Vinicius Caldeira Pacheco - 2019006957

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

andre.dutra@dcc.ufmg.br

marcospacheco@dcc.ufmg.br

1. Introdução

A proposta do segundo trabalho prático da disciplina de Sistemas Operacionais consiste em implementar um simulador de memória virtual. Uma memória virtual é responsável por mapear o endereço lógico de um processo a uma área em disco. Desse modo, dados o tamanho da memória física, o tamanho das páginas e o algoritmo de substituição, o simulador se propõe a mapear um espaço de endereçamento de 32 bits, cada um armazenando um byte, à memória física. Para isso, foi necessário escrever um programa que fosse capaz de usar um arquivo que contenha uma sequência de endereços de memórias acessados como entrada, ler linha por linha, identificar o endereço e a operação desejada e então simular esse processamento, identificando e registrando as leituras e escritas, e utilizando o algoritmo de reposição especificado. Esse processo foi feito utilizando a linguagem C++ e um arquivo auxiliar foi criado para armazenar uma classe que irá representar a memória virtual.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. Foi incluído um Makefile para auxiliar na compilação.

2.1. Entrada e Saída

Após compilar o programa, é esperado uma série de parâmetros que devem ser inseridos juntamente com o comando de execução no terminal para que ele funcione corretamente. Esses parâmetros são, em ordem: o algoritmo de substituição a ser usado (*lru*, *fifo*, *newalg*); o arquivo contendo a sequência de endereços de memória acessados; o tamanho de cada página/quadro de memória, em kilobytes; o tamanho total da memória física disponível para o processo, também em kilobytes. Os algoritmos que podem ser inseridos são "*lru*" (least recently used) , "*fifo*" (first-in-first-out) e "*newalg*" (random). O arquivo deve conter apenas linhas com um endereço de memória acessado, seguido dos caracteres R ou W (linhas sem o caractere não garantem uma execução correta do programa) . O tamanho das páginas deve ser de 2 a 64, e o tamanho total da memória física deve ser de 128 a 16384.

A partir desses valores de entrada passados corretamente, o programa é inicializado. Primeiramente ele informa que o simulador foi inicializado, e inicia sua execução. No final, são impressos os dados que foram recebidos para realizar a simulação, a quantidade de páginas lidas da memória secundária (*page faults*), e a quantidade de páginas escritas na memória secundária. Após esse processo, o programa é finalizado.

2.2. Estruturas de Dados

Para garantir o funcionamento correto do simulador da memória, foram criados dois arquivos principais que são responsáveis pela resolução do problema: "memory.h" e "memory.cpp". Eles consistem na classe "VirtualMemory" responsável por simular a memória virtual, e contém todas as funções que coordenam seu funcionamento.

A classe consiste, basicamente, em um vetor de unsigned chars, representando a memória física, byte a byte, um vetor de inteiros, representando a tabela de páginas, e dois vetores de booleanos representando os bits válido e sujo da tabela de páginas, mais algumas variáveis auxiliares aos demais métodos. Há também uma lista de inteiros padrão da STL usada como fila de controle pelos algoritmos.

A memória física, como descrito, é definida em bytes, representados por unsigned chars. Desse modo, seu tamanho é o tamanho da memória em bytes, ou seja, se foi escolhida uma memória de 128kB, o vetor terá 128*1024 = 131.072 entradas. Como essa informação não é dada nos arquivos, nenhum dado é de fato escrito no vetor, ele só existe para representar a memória. Aproveitando-se disso, os três primeiros bytes de cada quadro são usados para guardar o endereço da sua página associada, para que os algoritmos saibam de qual página alterar os bits de válido e sujo após selecionar um quadro como vítima. Embora os algoritmos mostrados em aula dêem a entender que uma estrutura de informação de quadro para página exista (nos exemplos os quadros contêm os números das páginas), como uma tabela de páginas reversa, nada do tipo foi mencionado explicitamente nas aulas. Então, nesse caso, a própria memória física, que estava vaga, foi usada com esse propósito. Em uma futura

implementação na qual a informação da memória seja necessária, uma estrutura separada será usada para guardar essa informação.

A tabela de páginas é um vetor de tamanho igual ao número de páginas da memória virtual, de 32 bits, e cada entrada, de índice igual ao índice da respectiva página, contém o índice do quadro na memória física no qual a página está armazenada. O tamanho da tabela é definido dividindo o número total de bytes da memória virtual, 2^{32}, pelo tamanho de página dado, em bytes. Associados a esse vetor estão os vetores válido e sujo. Se o endereço de um quadro guardado na tabela de páginas contém a sua página associada, o vetor válido para essa página será verdadeiro, e falso caso contrário. Já o vetor sujo será verdadeiro se o quadro associado à página na tabela for válido e tiver sido modificado por uma ação de escrita, mas essa modificação não tenha sido registrada no disco, e falso caso contrário.

Por fim, a lista de controle funciona como uma fila, na qual os índices de quadros são adicionados a cada vez que são alocados, até que o tamanho máximo seja atingido, quando ela cresce circularmente. Essa estrutura permite a execução do FIFO, e, com algumas alterações, movimentando para o fundo o mais usado, a execução do LRU. Seu tamanho máximo é o número de quadros da memória, que é obtido dividindo o tamanho da memória pelo tamanho dos quadros/páginas em bytes.

A interface da classe foi feita de modo a simular uma memória virtual. Assim, só são públicas as funções de escrita e leitura dado o endereço de 32 bits da memória virtual, além das funções de retorno dos dados solicitados. Conforme uma implementação verdadeira, detalhes de implementação não devem ser visíveis ao usuário da memória. Algumas das principais funções dessa classe estão resumidas na lista a seguir:

- VirtualMemory(int pagesize, int memsize, string algorit)

É o constructor da classe. Nele é necessário passar como parâmetro dois inteiros, representando o tamanho das páginas e da memória física em kB, e um string com o algoritmo de substituição a ser usado, sendo um dentre as opções; “fifo”, “lru” e “newalg”. As ações do construtor consistem em sua maioria em definir o tamanho das estruturas e vetores usados, guardando-os em suas respectivas variáveis, e alocar dinamicamente os vetores com seus respectivos tamanhos. Todos os vetores são inicializados com zero ou false. Algumas variáveis auxiliares, como o número de reads e writes e o string com o algoritmo usado, também são inicializadas.

- VirtualMemory(int pagesize, int memsize, string algorit, bool debug)

Tem comportamento idêntico ao construtor anterior, mas permite a iniciação do modo de depuração. Se o quinto parâmetro for true, a variável de debug é iniciada como verdadeira, ou falsa caso contrário. O construtor anterior inicializa a variável como falsa por default. Ativando o modo de depuração, cada passo do programa será anunciado com um print no console, para efeitos de teste. Aconselha-se usar o modo de depuração apenas com tamanhos pequenos de arquivo, para não poluir o console e dificultar o entendimento dos resultados. Na descrição das próximas funções, os prints do debug não serão mencionados, mas são facilmente identificáveis na implementação.

- read_mem(int address)

Representa uma leitura na memória virtual. Ela recebe o endereço de 32 bits do espaço endereçado, converte esse endereço para o endereço da página, segundo o tamanho de página dado, e seu offset, e obtém o endereço do quadro associado à página usando a função page_to_frame, que será descrita em mais detalhes posteriormente. Por fim, ela retorna o valor guardado no byte da memória física representado pelo quadro obtido, deslocado pelo offset.

- write_mem(int address, unsigned char val)

Representa uma escrita na memória virtual. Ela recebe o endereço de 32 bits do espaço endereçado, converte esse endereço para o endereço da página, segundo o tamanho de página dado, e seu offset, e obtém o endereço do quadro associado à página usando a função page_to_frame, que será descrita em mais detalhes posteriormente. A escrita na memória não é de fato feita, considerando que nenhum valor é dado nos arquivos, mas a função page_to_frame é chamada mesmo assim, para alocar um quadro na memória física e associá-lo à página acessada, caso necessário. Por fim, ela marca o vetor dirty como verdadeiro na posição relativa à página, indicando que ela foi alterada na memória física.

- get_read_times()

Retorna o número de leituras no disco, ou seja, o número de page faults. Esse valor é guardado em uma variável privada na classe.

- get_write_times()

Retorna o número de escritas no disco. Esse valor é guardado em uma variável privada na classe.

- page_to_frame(int page_address)

Essa função encapsula quase todo o comportamento da tabela de páginas. Dado o endereço de uma página na memória virtual, ela retorna o endereço na memória física do seu quadro associado na tabela de páginas. Se não houver um quadro associado, ou seja, se o bit válido da página for falso, ela seleciona um quadro vago, se houver, e o aloca para a página, definindo seu bit válido como verdadeiro. Se não houver quadros vagos, ela seleciona, segundo o algoritmo escolhido, uma página vítima, que terá seu quadro na memória física substituído pelo da página acessada. Então, sua informação é salva na memória secundária, caso seu bit sujo seja verdadeiro, e seus bits válido e sujo são definidos como falso. Por fim, o quadro é alocado para a página acessada, definindo o bit válido da página como verdadeiro. Quando um quadro é alocado, seu índice é salvo na tabela de páginas, e o índice de sua página é salvo em seus três primeiros bytes, como explicado anteriormente. Por fim, algumas rotinas do algoritmo lru são executadas, caso a página seja válida e esse algoritmo tenha sido selecionado (as rotinas serão detalhadas junto ao algoritmo) e o quadro associado à página acessada, agora definido na tabela, é retornado.

- frame_to_page(int frame_address)

Realiza o oposto da função anterior, mas não aloca quadros a páginas. Desse modo, essa função só funciona corretamente se o endereço do quadro dado estiver associado a uma página válida. A função concatena os três primeiros bits do quadro para formar o endereço da página associada, e o retorna.

- get_victim()

É responsável por executar o algoritmo de substituição escolhido (lru, fifo ou newalg) e retornar seu resultado. Seu propósito é o de servir como uma forma polimórfica dos algoritmos, podendo ser chamada nas demais funções como um algoritmo genérico para retornar a página vítima, enquanto ela seleciona o algoritmo usado segundo a inicialização da classe.

- fifo()

É a implementação do algoritmo de substituição *first-in-first-out*. Nele é utilizado uma lista de controle para ordenar qual é a fila de prioridade de remoção dos quadros, sendo que os quadros inseridos há mais tempo são os primeiros a serem removidos. Assim, quando ela é chamada, a página associada ao quadro alocado há mais tempo é retornada.

- lru()

É a implementação do algoritmo de substituição *least recently used*. A princípio ele funciona exatamente da mesma forma que a função "fifo()", mas a diferença está no uso da fila de controle chamada "control_" que armazena a prioridade de remoção dos quadros. Neste algoritmo, ao se ler um valor que já está registrado, a lista de controle é atualizada para que esse quadro tenha menor prioridade para sair na fila. Essa é a rotina mencionada na função frame_to_page. Com isso, quando a função é chamada, a página associada ao quadro utilizado há mais tempo é retornada.

- newalg()

É a implementação do algoritmo de substituição novo, o algoritmo *random*. Nele, quando ocorre um *page fault,* o novo quadro da memória para ser substituído é escolhido pseudoaleatoriamente, dentre todas as opções disponíveis. Para isso, é importada a biblioteca "ctime" e a função "rand()" é utilizada. Esse algoritmo foi selecionado como forma de analisar a influência apenas dos outros parâmetros no desempenho do programa, sem interferência do algoritmo, que nesse caso segue um comportamento aleatório.

3. Decisões de projeto

Quanto à leitura do arquivo, optamos por apenas aceitar arquivos que estejam no formato correto, com o endereço e o caractere logo após. O leitor lê o endereço, caso não haja fim do arquivo, e lê o caractere logo depois, então a falta de um caractere faria um endereço ser lido como caractere, e o próximo caractere seria lido como um endereço, gerando uma lógica errada.

Também não foi especificado se as estruturas padrão do C++, a STL, poderiam ser usadas, ou se deveriam ser implementadas do zero. No entanto, julgamos que o uso poderia

ser feito, pois a linguagem era permitida e a STL é uma parte muito relevante de C++, além de que a implementação das estruturas básicas não está no escopo do trabalho.

Não ficou claro se os parâmetros dados seriam parâmetros do terminal ou inputs feitos por leitura no programa. Tanto no Linux quanto no Windows a chamada de execução de programa não é feita apenas com o nome (./nome no linux e nome.exe no windows), então essa dúvida foi gerada. No entanto, optamos por implementar o programa com parâmetros do terminal, pois era o que fazia mais sentido considerando que foram passados como prompt no exemplo e que o nome do programa era um dos parâmetros.

Por fim, demais decisões já foram especificadas na descrição da estrutura, como o uso dos primeiros bits do quadro para guardar o índice da página, por exemplo.

4. Análise de Desempenho

A classe criada recebe vários parâmetros, que influenciam como os dados são armazenados e acessados. Assim, como forma de observar como esses parâmetros influenciam o desempenho do programa, coletamos informações diversas combinações de parâmetros da classe e os organizamos em gráficos. Embora o desempenho geralmente seja medido usando o tempo de execução, nesse caso essa medida não seria válida, pois é sabido que em um sistema de memória real a ação com maior custo de tempo é o acesso à memória secundária, que nesse caso não foi implementado. Assim, como o tempo de acesso à memória secundária domina o tempo de todas as outras ações, o desempenho de cada combinação será analisado observando o número de acessos à memória secundária realizados, somando-se leitura e escrita.

Primeiramente, fixamos o tamanho das páginas em 4kB e coletamos o desempenho do programa para cada arquivo dado, com cada algoritmo e para tamanhos de memória variando em potências de 2, como podemos ver na figura 1:

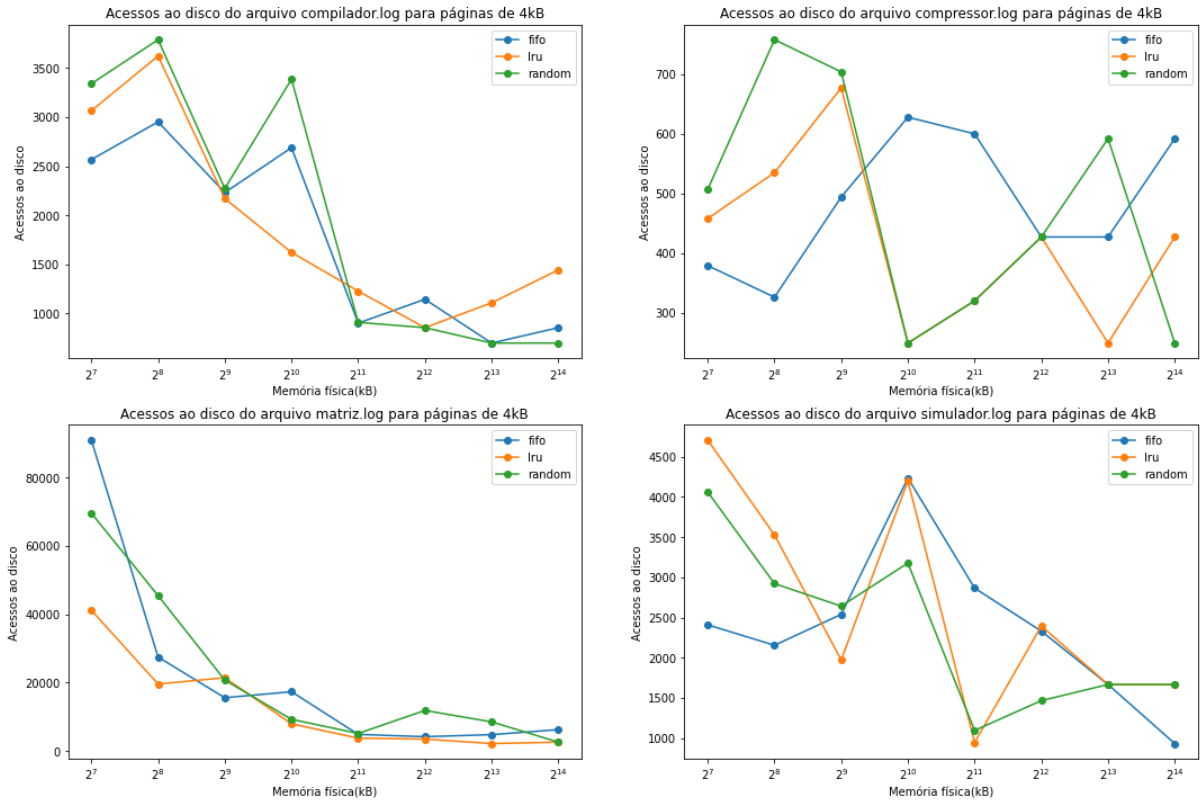

Figura 1: Acessos ao disco pelo programa fixando o tamanho das páginas em 4kB.

É notável que, num geral, tamanhos maiores de memória física levam a menos acessos à memória secundária, o que é intuitivo, considerando que há mais quadros na memória para guardar mais páginas. No entanto, o arquivo utilizado influenciou muito a estabilidade dos resultados. Os resultados dos arquivos compilador.log e matriz.log foram bem mais regulares que os demais, possivelmente porque suas estruturas de dados são maiores e se beneficiariam de maiores memórias físicas, enquanto o arquivo compressor.log foi o mais irregular. A configuração das chamadas de escrita e leitura parece influenciar bem mais o resultado que os algoritmos, pois até os resultados do algoritmo Random, que supostamente seriam imprevisíveis, parecem se comportar como os demais algoritmos nos arquivos em que houve maior estabilidade. Comparando os algoritmos entre si, o LRU tem em média um resultado melhor que o FIFO, embora o LRU tenha tido um melhor desempenho em memórias maiores, ao contrário do FIFO, que teve melhores resultados em memórias menores. Ambos apresentaram anomalias de Belady, pontos em que o aumento de quadros gerou mais page faults. Mesmo assim, as anomalias no fifo foram mais frequentes e mais acentuadas. Naturalmente, o algoritmo Random foi o mais anômalo, já que seu comportamento não segue nenhum padrão.

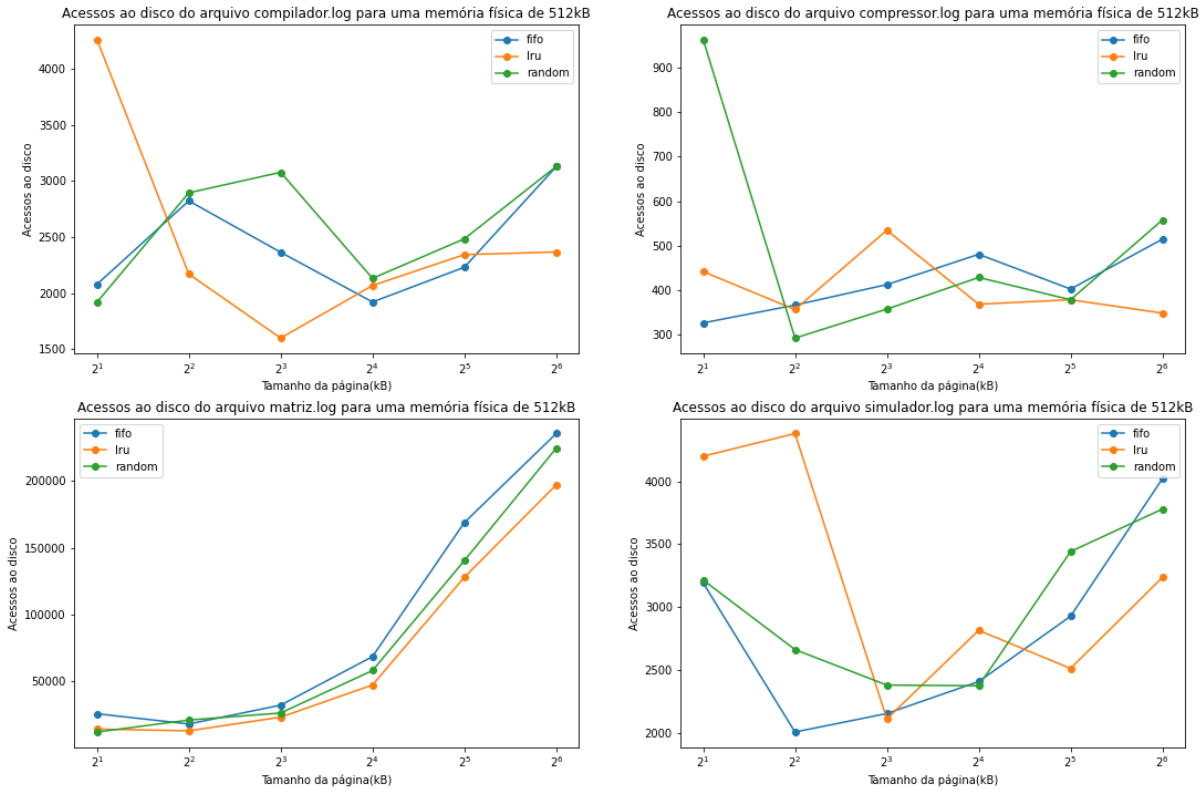

Figura 2: Acessos ao disco pelo programa fixando a memória física em 512kB.

Comparado ao caso anterior, podemos observar que os gráficos alterando o número de páginas são bem mais irregulares. Isso se dá pelo fato de que o tamanho das páginas influencia diversos fatores que influenciam o desempenho do programa de formas diferentes. Por exemplo, páginas menores fazem com que menos dados sejam carregados do disco a cada acesso e mais acessos sejam necessários. No entanto, páginas muito grandes fazem com que mais espaço seja alocado para cada página do que o que de fato é acessado, diminuindo a porcentagem de memória usada e, com isso, aumentando o número de acessos ao disco. Tirando o arquivo matriz.log, podemos observar que os gráficos de cada arquivo seguem, de fato, um formato de "U", indicando a dualidade. No caso do arquivo matriz.log, o que possivelmente ocorre é que os acessos se concentram localmente em pedaços do disco que cabem mesmo em páginas pequenas. Assim, o tamanho pequeno das páginas não influenciaria tanto no aumento de acessos quanto a memória vaga causado pelas páginas grandes, levando a curvas crescentes em todos os algoritmos, inclusive o random. Comparando os algoritmos entre si, o FIFO e o LRU tiveram desempenhos em média semelhantes. No entanto, o LRU foi melhor que o FIFO em páginas maiores, enquanto o FIFO superou o LRU em páginas menores. Curiosamente, o algoritmo Random teve um desempenho muito semelhante ao FIFO num geral, embora tenha apresentado um pico notável no arquivo compressor.log.

4. Instruções de Compilação e Execução

O programa foi testado usando Linux Ubuntu como sistema operacional. Foi disponibilizado também um arquivo Makefile para facilitar a compilação.

Para compilar, é necessário:

- Instalar o GCC

- Executar o comando “make”

Com isso, será gerado um arquivo “tp2virtual” que deve ser usado para testar o programa. Os parâmetros devem ser inseridos juntamente com o comando do executável. Para executar o programa utilizando o arquivo “compilador.log” com o algoritmo “lru” é necessário inserir manualmente no terminal o seguinte comando, por exemplo:

- Executar o comando “./tp2virtual lru compilador.log 4 128”

5. Conclusão

Após concluir esse trabalho prático, foi possível entender melhor como ocorre o processamento de uma memória virtual. Ficou evidente que o algoritmo utilizado para selecionar os quadros a serem sobrescritos nos casos de *page faults* tem um impacto direto e significativo na quantidade de itens lidos e escritos, bem como no tempo de execução total do programa. A utilização de uma classe para representar o funcionamento da memória virtual também mostrou-se de grande ajuda para melhorar tanto a legibilidade do código, quanto a organização e a compartimentalização.

Com isso, foi possível desenvolver um programa que encontra a solução do problema apresentado de maneira eficiente e para diferentes casos dentro do escopo solicitado.

6. Referências

TEODORO, George L. M. Slides da disciplina Sistemas Operacionais. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2021.