Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Curso de Ciência da Computação

Simulador de memória virtual

Adélson de Oliveira Carmo Júnior, Renan Rodrigues Sousa

1 Introdução

A crescente complexidade dos sistemas computacionais modernos tem exigido técnicas cada vez mais eficientes para o gerenciamento de memória, especialmente em ambientes onde a memória física disponível é limitada. Nesse contexto, a memória virtual surge como uma solução essencial, permitindo que os sistemas operacionais ofereçam aos processos a ilusão de dispor de uma quantidade maior de memória do que a realmente existente.

Este trabalho apresenta o desenvolvimento de um simulador de memória virtual que implementa três diferentes algoritmos de substituição de páginas: NRU (Not Recently Used), Segunda Chance e LRU (Least Recently Used). Cada um desses métodos possui características distintas para determinar quais páginas devem ser removidas da memória física quando esta atinge sua capacidade máxima. O simulador foi projetado para oferecer uma visão detalhada do comportamento desses algoritmos em cenários práticos, permitindo a comparação de suas eficácias e impactos no desempenho do sistema.

Ao longo deste documento, serão discutidos os métodos utilizados para a implementação do simulador, incluindo a estruturação das rotinas e funções que compõem o sistema. A análise do desempenho dos diferentes algoritmos de substituição de páginas também será abordada, destacando suas respectivas vantagens e limitações em ambientes simulados.

2 Implementação

Neste tópico, serão discutidos os métodos empregados para implementar o simulador de memória virtual, bem como a listagem das rotinas e funções utilizadas.

2.1 Gerenciamento de memória

Inicialmente, o programa deve ser capaz de distinguir entre endereços de leitura e escrita presentes no arquivo de entrada. Assim, cada vez que um novo endereço é lido, suas informações — endereço e tipo (leitura ou escrita) — são armazenadas para posterior inserção na tabela hash, caso seja necessário.

Com o tamanho de cada página sendo passado como parâmetro, a estrutura da tabela será definida pelo usuário. Para implementar essa parte, será utilizada uma lista hash encadeada. Cada vez que uma nova informação é lida, sua posição correspondente na tabela de páginas é calculada e, em seguida, ela é inserida. Esse processo de inserção não é exclusivo para endereços que ainda não estão na tabela; caso o endereço já esteja presente, as únicas atualizações realizadas serão no tempo do último acesso, no referenciamento da página, colocando 1 caso ela seja 0, e no indicador de presença da página no quadro. Se a inserção for realizada, dados como o endereço, o tipo (leitura ou escrita) e o tempo de execução no momento da inserção são armazenados em uma variável do tipo *NO* (um tipo criado para essa implementação), juntamente com variáveis que indicam se o endereço está presente e o tempo do último acesso.

A Figura 1 mostra um exemplo de uma tabela que foi preenchida até o limite da memória física, utilizando os endereços que têm o indicador de presença igual a 1. Observe que, nessa figura, foi adotado o valor 5 como limite da memória física. Portanto, como existem cinco endereços na tabela, ela já está completa, mesmo que haja algum desbalanceamento entre os elementos na hash.

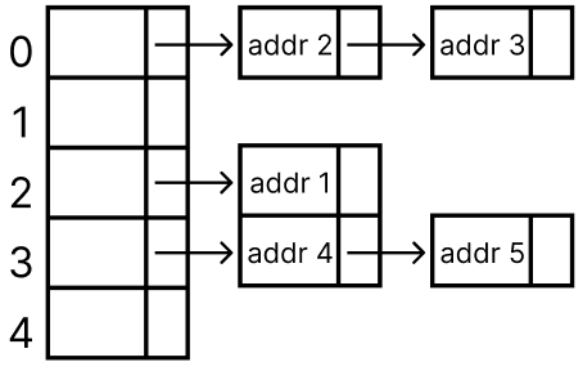

Figura 1: Moldura das páginas completa

Após atingir o limite da memória, cada nova inserção subsequente exigirá a utilização de um algoritmo de substituição, independentemente de qual seja, para que o novo dado seja integrado à tabela. As Figuras 2 e 3 ilustram esse processo, assumindo que o algoritmo tenha determinado que o endereço a ser substituído seja o "addr2"(destacado na Figura 2). O endereço "addr6"passa a integrar a tabela em sua posição correspondente na hash, como mostrado na Figura 3. Esse processo será repetido até que não haja novos endereços a serem lidos e inseridos na hash.

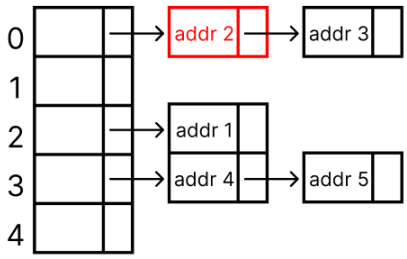

Figura 2: Destaque no elemento a ser substituido

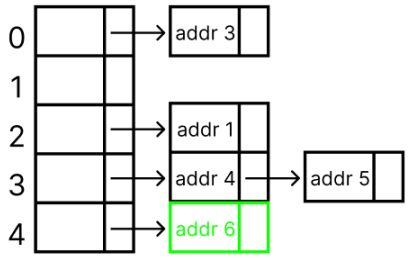

Figura 3: Destaque no novo elemento inserido

Quanto à contagem de *page faults* e *page hits*, essa parte é relativamente simples. Ao ler os endereços, se o endereço já estiver presente na tabela, o contador de acertos (*page hits*) é incrementado, e nenhum novo elemento é inserido na tabela. Por outro lado, em caso de um erro de página (*page fault*), ocorre a inserção do novo endereço na tabela, juntamente com o incremento do número de *page faults*.

É importante destacar que, embora a representação da moldura de páginas (frame) apresente um número limitado de endereços, a memória virtual contém todos os elementos, mesmo aqueles que não estão atualmente na moldura. Assim, esses elementos, embora não estejam representados na moldura, ainda existem na memória virtual e são mantidos até o final da execução.

2.2 Algoritmos de substituição

Nesta seção, discutiremos brevemente o funcionamento e a implementação dos algoritmos de substituição utilizados neste simulador de memória virtual. Vale ressaltar que a substituição não elimina definitivamente nenhum elemento da memória; em vez disso, ela altera o estado de "presente", que indica se o elemento está atualmente na moldura de páginas ou não.

2.2.1 NRU

Este método utiliza uma tabela de classes para determinar qual elemento será substituído quando um novo endereço precisar ser inserido na tabela. A tabela, exemplificada na Figura X, usa "R"para indicar se o endereço foi referenciado e "M"para indicar se ele foi modificado (0 para leitura e 1 para escrita). O elemento na tabela com o menor valor de classe será o escolhido para substituição. Se houver mais de um elemento com o menor valor, a seleção será feita de forma aleatória.

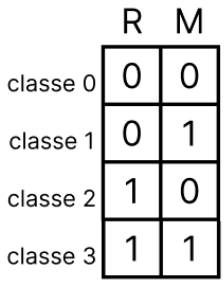

Figura 4: Tabela de classes do método NRU

2.2.2 Segunda Chance

No método da segunda chance, o parâmetro avaliado é o "R", que segue a mesma notação do método anterior. A Figura X mostra um exemplo disso, associando cada endereço ao seu valor correspondente de "R". O método da segunda chance utiliza uma fila de processos organizada pela ordem de chegada (o primeiro elemento chega primeiro, o último chega por último) para determinar se um endereço deve ser removido da tabela.

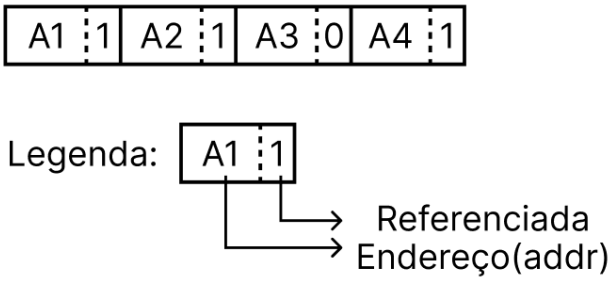

Figura 5: Exemplo de representação pro Segunda Chance

Se o endereço analisado tiver "R"igual a 1, ele recebe uma segunda chance: é movido para o final da fila e seu "R"é alterado para 0. A Figura X ilustra esse processo, onde o primeiro endereço da fila, A1, está sendo analisado. Como A1 tem "$R = 1$", ele é enviado para o fim da fila, agora com "R"igual a 0. Esse procedimento é repetido até que se encontre o primeiro endereço com "R"igual a 0, que será o endereço a ser substituído.

2.2.3 LRU

Por fim, o método LRU (Least Recently Used) utiliza o tempo do último acesso para identificar qual elemento deve ser removido da tabela. Ao percorrer a tabela, o LRU determina qual é o elemento com o menor valor de acesso, ou seja, aquele que não foi acessado há mais tempo, e o exclui da tabela.

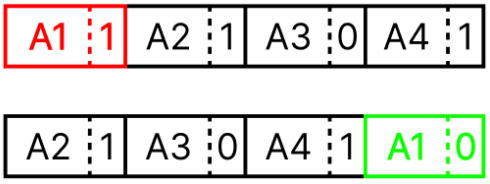

Figura 6: Segunda chance aplicada ao primeiro elemento da fila

Se o endereço lido já estiver presente na tabela, o tempo de seu último acesso é atualizado. Isso significa que, mesmo que ele estivesse prestes a ser substituído, ele não será mais considerado para remoção devido à atualização de seu tempo de acesso.

2.3 Listagem de Rotina

Nesta seção, serão descritas as funções do simulador de memória virtual, juntamente com suas respectivas funcionalidades.

2.3.1 memoria.h

Nesta subseção, estão descritas as funções contidas no arquivo memoria.h e implementadas em memoria.c. Essas funções correspondem aos algoritmos de substituição de páginas.

- **nru(TabelaHash *tabela, unsigned addr, char rw, int tempo, int *pag_sujas)**: Implementa o algoritmo NRU (Not Recently Used), que classifica as páginas em diferentes categorias com base em seus bits de referência e modificação, determinando qual página deve ser substituída.
- **segunda_chance(TabelaHash *tabela, unsigned addr, char rw, int tempo, int *pag_sujas):** Implementa o algoritmo de Segunda Chance, que percorre as páginas em uma fila circular, dando uma "segunda chance"às páginas que foram referenciadas recentemente antes de decidir substituí-las.
- **lru(TabelaHash *tabela, unsigned addr, char rw, int tempo, int *pag_sujas):** Implementa o algoritmo LRU (Least Recently Used), que substitui a página que não foi acessada há mais tempo, utilizando uma estrutura que mantém o registro dos últimos acessos.

2.3.2 tabela.h

Nesta subseção, estão descritas as funções e TADs (Tipos Abstratos de Dados) definidos no arquivo tabela.h e implementados em tabela.c.

- **TAD Quadro:** Estrutura que armazena as informações sobre o estado de cada página.
  - **modificado**: Indica se o elemento foi modificado (0 para leitura e 1 para escrita).
  - **presente**: Indica se o elemento está presente na moldura de páginas (1) ou não (0).
  - **referenciado**: Indica se o elemento foi referenciado recentemente (1 para sim, 0 para não).
  - **ultimoAcesso**: Armazena o tempo do último acesso ao elemento.
  - **endereço**: Guarda o endereço do elemento.
- **TAD NO:** Estrutura de nó usada na implementação da lista encadeada da tabela hash.
  - **pagina**: Referência ao quadro que contém as informações da página.
  - **proximo**: Ponteiro que aponta para o próximo nó na lista encadeada.
- **TAD TabelaHash:**Estrutura que representa a tabela hash que armazena as páginas de memória.
  - **tab**: Vetor que contém todas as páginas da memória.
  - **tamanho**: Indica o tamanho total da tabela hash.
  - **s**: Deslocamento necessário na tabela para encontrar uma posição.
- **criar_lista():** Inicializa uma lista encadeada vazia para ser usada na tabela hash.
- **table_hash(TabelaHash tabela, unsigned addr):** Calcula o índice na tabela hash para um determinado endereço.
- **criar_tabela_hash(int tam_Pagina, int tam_Memoria):** Cria e inicializa uma tabela hash com o tamanho especificado para a página e para a memória física.
- **inserir(TabelaHash tabela, unsigned addr, char rw, int tempo):** Insere um novo endereço na tabela hash, registrando seu tipo de acesso (leitura/escrita) e o tempo de acesso.
- **buscar(TabelaHash tabela, unsigned addr):** Busca um endereço específico na tabela hash e retorna suas informações, se encontrado.
- **imprimir_lista(Lista lista):** Imprime os elementos de uma lista encadeada, mostrando os endereços armazenados.
- **imprimir_tabela(TabelaHash tabela)**:** Exibe a estrutura completa da tabela hash, mostrando todas as páginas e seus estados.
- **criar_quadro(unsigned addr, char rw, int tempo):** Cria um novo quadro de página com as informações de endereço, tipo de acesso e tempo de último acesso.

2.3.3 main.c

Nesta subseção, está descrita a função principal do programa, contida no arquivo main.c.

- **main**: Função principal que coordena a execução do simulador de memória virtual, incluindo a inicialização das estruturas de dados, leitura do arquivo de entrada, e aplicação dos algoritmos de substituição de página.

3 Resultados

Para coletar os resultados dos três algoritmos implementados, utilizamos quatro arquivos de teste. Cada arquivo representa o registro das operações de acesso à memória observadas durante a execução de programas representativos de diferentes classes de aplicações reais:

- `compilador.log`: Registra a execução de um compilador, que geralmente utiliza um grande número de estruturas de dados internas complexas.
- `matriz.log`: Contém dados de um programa científico que realiza cálculos matriciais relativamente simples, porém sobre grandes matrizes e vetores.
- `compressor.log`: Relata a execução de um programa de compressão de arquivos, que utiliza estruturas de dados mais simples.
- `simulador.log`: Reflete a execução de um simulador de partículas, que realiza cálculos complexos sobre estruturas relativamente simples.

Cada linha desses arquivos contém um endereço de memória acessado, seguido das letras `R` ou `W`, indicando um acesso de leitura ou escrita, respectivamente. Exemplos de linhas extraídas dos arquivos são:

```
0785db58 W
000652d8 R
000652e0 R
0005df58 W
```

Nas seções seguintes, apresentamos os resultados de diversos testes realizados com variações nos parâmetros de tamanho das páginas e na capacidade total da memória. As medições observadas incluem o número de falhas de página (page faults) e a quantidade de páginas sujas. Esses resultados variam de acordo com os parâmetros ajustados e diferem para cada algoritmo avaliado.

A execução do algoritmo LRU foi a mais demorada, uma vez que ele precisa percorrer toda a tabela hash a cada vez que um novo endereço é adicionado, a fim de identificar o menos recentemente utilizado. O algoritmo NRU demonstrou ser o mais rápido. Para valores menores de memória, apresentou menos falhas de página do que o LRU. No entanto, à medida que a quantidade de memória aumentou, o LRU mostrou um desempenho superior. O algoritmo de Segunda Chance teve um tempo de execução intermediário, mas obteve bons resultados em termos de falhas de página.

3.1 Compilador

As Tabelas 1 e 2 demonstram que, ao utilizar o algoritmo LRU, uma maior diferença entre o número de páginas e o espaço de memória disponível resulta em melhorias significativas nos resultados. Em contraste, as Tabelas 3 e 4 revelam que, embora um aumento na diferença entre o número de páginas e o espaço de memória disponível ainda ofereça benefícios, a melhoria observada é mais discreta. Os valores de desempenho tornam-se bastante similares, independentemente das variações nos números de páginas e memória.

Por fim, ao analisar o algoritmo da Segunda Chance, conforme apresentado nas Tabelas 5 e 6, observa-se um comportamento semelhante ao do NRU, mas com uma leve melhoria geral em todos os resultados. Assim, o método mais eficaz foi o Segunda Chance, enquanto o LRU apresentou o pior desempenho, com um número maior de faltas de página. Quanto as infomações dos acessos, tem-se:

- Páginas lidas: 892816;
- Páginas escritas: 107184.

3.1.1 Compilador: LRU

Memória (KB)	Falta de Página	Páginas Sujas
128	542207	119653
8192	201621	32461
16384	109468	20533

Tabela 1: Resultados LRU com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	109468	20533
32	406134	69598
64	445631	82778

Tabela 2: Resultados LRU com tamanho de memória fixo (8192KB)

3.1.2 Compilador: NRU

Memória (KB)	Falta de Página	Páginas Sujas
128	60772	11169
8192	58705	11108
16384	56682	10138

Tabela 3: Resultados NRU com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	56682	10138
32	60474	11404
64	60651	11432

Tabela 4: Resultados NRU com tamanho de memória fixo (8192KB)

3.1.3 Compilador: Segunda Chance

Memória (KB)	Falta de Página	Páginas Sujas
128	58649	10809
8192	56635	10249
16384	54619	9894

Tabela 5: Resultados Segunda Chance com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	54619	9894
32	58426	10701
64	58554	10758

Tabela 6: Resultados Segunda Chance com tamanho de memória fixo (8192KB)

3.2 Matriz

A execução com a entrada `matriz` foi uma das mais lentas. A seguir, apresentamos os resultados obtidos para cada algoritmo avaliado. Ao todo, foram registradas as seguintes informações sobre os acessos:

- Páginas lidas: 932.830
- Páginas escritas: 67.170

3.2.1 Matriz: LRU

Memória (KB)	**Falta de Página**	**Páginas Sujas**
128	671042	83641
8192	179148	46630
16384	139413	40980

Tabela 7: Resultados LRU com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	139413	40980
32	239304	58023
64	270229	62915

Tabela 8: Resultados LRU com tamanho de memória fixo (8192KB)

3.2.2 Matriz: NRU

Memória (KB)	Falta de Página	Páginas Sujas
128	139520	40580
8192	138550	40253
16384	136429	39974

Tabela 9: Resultados NRU com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	136429	39974
32	140341	40592
64	140499	40588

Tabela 10: Resultados NRU com tamanho de memória fixo (8192KB)

3.2.3 Matriz: Segunda Chance

Memória (KB)	Falta de Página	Páginas Sujas
128	117657	32631
8192	115567	32288
16384	113209	32097

Tabela 11: Resultados Segunda Chance com tamanho de página fixo (4KB)

Tamanho da Página (KB)	Falta de Página	Páginas Sujas
2	113209	32097
32	117433	32625
64	117561	32631

Tabela 12: Resultados Segunda Chance com tamanho de memória fixo (8192KB)

3.3 Compressor

Nesta subseção, analisou-se a entrada `compressor` utilizada para os testes. A execução do compressor foi a mais rápida. A partir das tabelas dessa seção, as conclusões sobre os métodos de substituição permanecem consistentes com as análises anteriores. O LRU apresenta um mudança considerável de falta de página com o aumento da memória. A seguir, apresentamos os resultados obtidos para cada algoritmo avaliado. Ao todo, foram registradas as seguintes informações sobre os acessos:

- Páginas lidas: 877.581
- Páginas escritas: 122.419

3.3.1 Compressor: LRU

Memória (KB)	Falta de Página	Páginas Sujas
128	328115	71377
8192	19681	4936
16384	14302	3692

Tabela 13: Resultados LRU com tamanho de página fixo (4KB)

Página (KB)	**Falta de Página**	**Páginas Sujas**
2	14302	3692
32	41908	9816
64	47635	11720

Tabela 14: Resultados LRU com tamanho de memória fixo (8192KB)

3.3.2 Compressor: NRU

Memória (KB)	**Falta de Página**	**Páginas Sujas**
128	18373	3531
8192	17049	2951
16384	15309	2371

Tabela 15: Resultados NRU com tamanho de página fixo (4KB)

Página (KB)	**Falta de Página**	**Páginas Sujas**
2	15201	2424
32	18496	3119
64	18635	3120

Tabela 16: Resultados NRU com tamanho de memória fixo (8192KB)

3.3.3 Compressor: Segunda Chance

Memória (KB)	**Falta de Página**	**Páginas Sujas**
128	17887	5344
8192	15612	4577
16384	13894	3681

Tabela 17: Resultados Segunda Chance com tamanho de página fixo (4KB)

Página (KB)	**Falta de Página**	**Páginas Sujas**
2	13894	3681
32	17626	5291
64	17835	5299

Tabela 18: Resultados Segunda Chance com tamanho de memória fixo (8192KB)

3.4 Simulador

Nesta subseção, analisou-se a entrada do simulador utilizada para os testes. Com base nas tabelas desta seção, as conclusões sobre os métodos de substituição permanecem consistentes com as análises anteriores. O LRU continua sendo o método mais sensível às diferenças entre o número de páginas e a quantidade de memória disponível, embora essa sensibilidade afete todos os métodos de substituição. Assim, o método mais eficaz para esta entrada específica é o Segunda Chance, seguido pelo NRU, com o LRU ocupando a última posição. Quanto as infomações dos acessos, tem-se:

- Páginas lidas: 839017;
- Páginas escritas: 160983.

3.4.1 Simulador: LRU

Memória (KB)	Falta de Página	Páginas Sujas
128	591439	137420
8192	138284	54571
16384	111810	49231

Tabela 19: Resultados LRU com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	111810	49231
32	290823	79981
64	351203	92532

Tabela 20: Resultados LRU com tamanho de memória fixo (8192KB)

3.4.2 Simulador: NRU

Memória (KB)	Falta de Página	Páginas Sujas
128	119946	51172
8192	121422	50809
16384	119499	49187

Tabela 21: Resultados NRU com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	119499	49187
32	123187	51375
64	123355	51408

Tabela 22: Resultados NRU com tamanho de memória fixo (8192KB)

3.4.3 Simulador: Segunda Chance

Memória (KB)	Falta de Página	Páginas Sujas
128	101814	45241
8192	99947	44264
16384	98463	43372

Tabela 23: Resultados Segunda Chance com tamanho de página fixo (4KB)

Página (KB)	Falta de Página	Páginas Sujas
2	98463	43372
32	101618	45090
64	101728	45170

Tabela 24: Resultados Segunda Chance com tamanho de memória fixo (8192KB)

4 Conclusão

Em conclusão, a principal dificuldade enfrentada durante o desenvolvimento deste trabalho foi a gestão das estruturas de dados utilizadas. Observamos que a implementação da tabela hash resultou em uma significativa ineficiência na execução dos algoritmos. Posteriormente, percebemos que a adoção de uma estrutura alternativa, como uma lista encadeada dupla para a organização das páginas por tempo, poderia ter proporcionado uma busca mais eficiente em comparação com a abordagem direta na tabela hash.

Analisando os resultados, verificou-se que o algoritmo com o melhor desempenho, para todas as entradas, foi o Segunda Chance, seguido pelo NRU e, por último, o LRU, com base no número de faltas de página. Entretanto, é importante notar que a eficiência dos métodos de substituição de página está diretamente relacionada ao arquivo de entrada, pois arquivos como matriz.log demandaram muito mais tempo para serem processados do que os demais.

De maneira geral, o trabalho contribuiu significativamente para uma melhor compreensão do simulador digital, apesar das dificuldades enfrentadas na implementação. Para projetos futuros, o conhecimento adquirido sobre a gestão de dados será valioso para evitar a repetição dos mesmos erros.

Referências

[1] TANENBAUM, A.; BOS, H. *Sistemas Operacionais Modernos*. 4ª Edição. São Paulo: Pearson Education do Brasil, 2016.