# Trabalho Prático - Memória Virtual

**Lucas M. Barreto Rezende[1], Luiza Sodré Salgado[1]**

[1]Departamento de Ciência de Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{lucasmbr,luiza-salgado}@ufmg.br

***Abstract.*** *This project aims to implement some concepts from the second part of the course "DCC605 - Operating Systems." In this sense, concepts of virtual memory, replacement algorithms and memory management were used.*

***Resumo.*** *Este projeto visa implementar alguns conceitos da segunda parte da disciplina 'DCC605 - Sistemas Operacionais'. Nesse sentido, foram utilizados conceitos de memória virtual, algoritmos de substituição e gerência de memória.*

## 1. Algoritmos escolhidos

Para este trabalho, foram utilizados os algoritmos RAND, LRU, LFU e MFU. Nesse viés, segue uma explicação sucinta para a escolha de cada um dos algoritmos de substituição.

### 1.1. Random (RAND)

Esse algoritmo escolhe uma página aleatória para substituir. Ele é usado como um grupo de controle para comparação com os outros algoritmos.

### 1.2. LRU

O LRU (Least Recently Used) substitui a página que foi usada menos recentemente. Esse algoritmo foi implementado usando um contador de acessos que é incrementado a cada leitura do arquivo e foi escolhido por ser um clássico simples e eficiente.

### 1.3. LFU

O LFU (Least Frequently Used) substitui a página com o menor número de acessos. Ele foi escolhido por ser semelhante ao algoritmo anterior, porém olhando um parâmetro diferente. No caso de empate, ele usa o LRU como critério para desempate.

### 1.4. MFU

O MFU (Most Frequently Used) é o oposto do algoritmo anterior, substituindo a página com maior número de acessos. Ele foi escolhido com objetivo de entender as diferenças de desempenho dessas duas técnicas na prática.

## 2. Resumo e decisões de projeto

As estruturas de dados principais do simulador de memória virtual são:

- Config, que armazena os parâmetros iniciais e os cálculos derivados da execução

- Stats, responsável por coletar métricas de desempenho como o total de acessos e a contagem de page faults e páginas escritas.
- Frame, que representa a unidade de memória física e guarda metadados sobre a página residente.
- PageTable, que gerencia o mapeamento entre o espaço de endereçamento lógico e o espaço físico, contendo as PageTableEntry (PTE) que realizam a tradução de endereço.

Foi utilizado um contador de tempo (inteiro) que é incrementado a cada leitura dos arquivos de log invés de usar o tempo de relógio do computador para simplificar a implementação. O modo de depuração imprime todas as operações de HIT e FAULT para todos os acessos, portanto, é melhor que ele seja armazenado em um arquivo texto à parte para análise. Por fim, o relatório final imprime, além das estatísticas agregadas solicitadas na especificação, a porcentagem de page faults e a porcentagem de páginas nas quais houve a necessidade de serem reescritas.

Na especificação, pede que o simulador tenha quatro argumentos. Porém, foi necessário a adição de um para diferenciar o tipo de paginação utilizada.

## 3. Implementação

Conforme solicitado na especificação do trabalho, a implementação foi feita em ambientes Linux (Ubuntu e Debian), visando garantir o funcionamento. Para testar com mais facilidade, também foi utilizado um *script .sh* para validar as tabelas e algoritmos no que tange aos .log disponibilizados. Por fim, também foi utilizado o Valgrind para verificar se não estaria havendo nenhum Memory Leak que pudesse interferir no resultado final.

```
==11134==
==11134== HEAP SUMMARY:
==11134==     in use at exit: 0 bytes in 0 blocks
==11134==   total heap usage: 7 allocs, 7 frees, 8,398,080 bytes allocated
==11134==
==11134== All heap blocks were freed -- no leaks are possible
==11134==
==11134== For lists of detected and suppressed errors, rerun with: -s
==11134== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

*Figura 1 - Validação de limpeza/gerenciamento de memória dado um .log como input*

## 4. Resultados esperados

Em geral, é esperado que o **LRU** seja o algoritmo mais robusto, i.e., é esperado que o LRU possua o menor número de Page Faults, independentemente do tipo de log de acesso, visto que explora tanto a localidade espacial quanto temporal. O **LFU** vai funcionar melhor para programas com loops longos e acessos repetitivos a poucas páginas como o matriz.log.

No que tange ao **MFU**, é esperado que ele seja o pior dos algoritmos, pois ele remove as páginas que mais necessitarão de acesso novamente em um curto período de tempo, porém, em alguns casos específicos, ele pode funcionar melhor que os outros. O

**Random (RAND)** vai depender, literalmente, de sorte, visto que é aleatório, mas espera-se que ele tenha mais page faults e, em alguns casos, tenha alto desempenho em programas sem localidade de referência.

A **tabela densa** tem o lookup mais rápido, visto que é um vetor de acesso direto, porém deve exigir um custo de memória mais alto para armazenar entradas para cada página possível, especialmente com páginas pequenas. A **tabela hierárquica (Multinível)** deve reduzir o custo de memória ao armazenar apenas páginas em uso, mas aumentar o tempo de lookup, exigindo múltiplos acessos à memória. Por fim, a **tabela invertida** deve gastar a menor quantidade de memória, sendo proporcional ao tamanho da memória física e independente do tamanho do espaço de endereçamento lógico, mas o desempenho do lookup é o mais complexo, pois requer um mecanismo de hashing.

| Algorithm | Brief description | Advantages | Disadvantages |
|---|---|---|---|
| RAND | A random number generator to identify an alternative object | It is the simplest algorithm and easy to implement | It is not considered a factor<br>It has an unstable performance<br>Its hit rate is low |
| LRU | The least-used items are deleted first | It is easy to implement and has a low hit rate | No factor other than the time factor is considered<br>It contains the cache contamination |
| LFU | Objects of the least frequency are removed first | It prevents cache memory contamination | Only the frequency factor is considered, and other factors are ignored<br>It is difficult to implement |
| SIZE | Large objects are removed first. | It is easy to implement, preserves small objects first, and has a high cache hit rate. | It first stores small web objects, even if they are not re-accessed<br>It has a low byte hit rate |
| GDSF | Frequency and launch factors are combined, and the age factor is produced like the time factor. | It covers the weakness of the size algorithm by deleting objects that are no longer accessible to users. | Its computational cost is low, and it has a complex parameter setting<br>It has a low byte hit rate |

*Figura 2 - Validação de limpeza/gerenciamento de memória*

## 5. Análises

Todas as análises foram feitas observando gráficos gerados por um script python que gerava as diversas combinações de tamanho de páginas, de memória física, de algoritmo de substituição utilizado e  tipo de tabela. Depois os resultados foram analisados manualmente. Abaixo segue uma visão geral dos resultados observados.

### 5.1 Page Faults

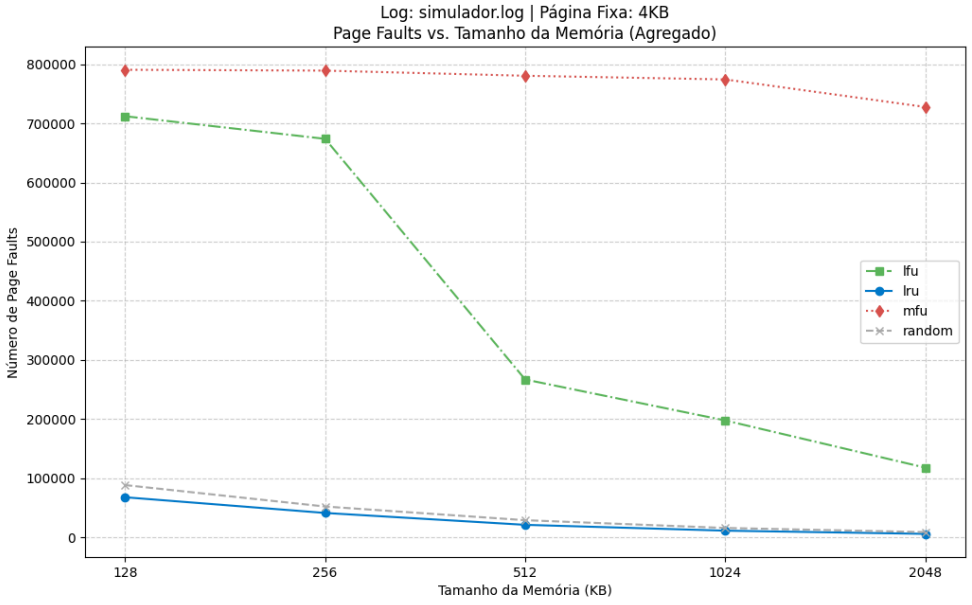
Como era esperado, o uso de uma memória física maior diminui a quantidade de page faults drasticamente. Porém, quando o tamanho da página aumenta o número de páginas que não estão presentes também aumenta. Isso não deveria ocorrer se a memória física fosse grande o suficiente para comportar páginas suficientes. Porém, como os testes foram feitos como uma memória física baixa, houve essa discrepância. Nesse viés, o

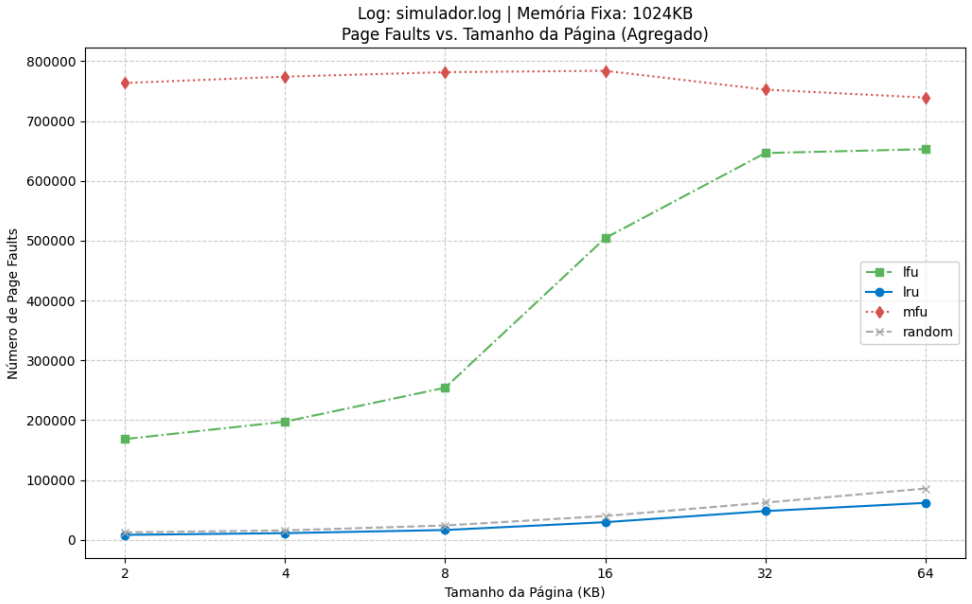
algoritmo **LRU** foi o que teve o melhor desempenho, seguido pelo **Random**. O **MFU**, em geral, foi o pior em todos os casos e o **LFU** foi relativamente melhor, apesar de inferior ao LRU e Random..

Esse fato ocorre pois, dado que o MFU deve escolher a página mais frequentemente usada como vítima, o que por definição é uma péssima política, ele remove exatamente as páginas que o programa provavelmente está usando (localidade).O **LFU** também não obteve bons resultados. A causa disso é, possivelmente, o modo como esses dois algoritmos de frequência foram implementados. Eles não apresentavam um mecanismo de envelhecimento, então depois de um certo intervalo de tempo, as frequências observadas não eram úteis.

Como consequência dos outros dois algoritmos terem aumentado muito o número de page faults, o Random pareceu ter um desempenho quase tão bom quanto o LRU. Outro possível motivo para isso é o escopo de trabalho dos programas ser reduzido.

Abaixo estão dois gráficos mostrando o desempenho de cada algoritmo com um tamanho de página fixo em (4KB) e aumentando o tamanho da memória, assim como um tamanho de memória fixa (1024KB) e páginas variadas para o log do simulador. Em geral, o formato do gráfico se mantém para todos os logs, mesmo que o número de page faults sejam levemente variados.






*Figuras 3 e 4 - Análise da relação entre Page Faults e Memória Fixa/Tamanho da página*

## 5.2 Gerência de memória

As tabelas hierárquicas gastaram mais memória que as outras, enquanto a tabela densa teve um uso equilibrado. A tabela invertida foi a que alcançou menor gasto, como era esperado. Além disso, com páginas maiores, o uso de memória foi menor, pois as tabelas de páginas possuíam menos entradas. Houve um aumento sútil quando foi modificado o tamanho da memória física, em especial na tabela invertida, pois assim era possível inserir mais páginas dentro dela.

É interessante dizer que houve variações para esse comportamento. Por exemplo, para o log da matriz, a tabela hierárquica de 3 níveis gastou bastante memória independente do tamanho da página.

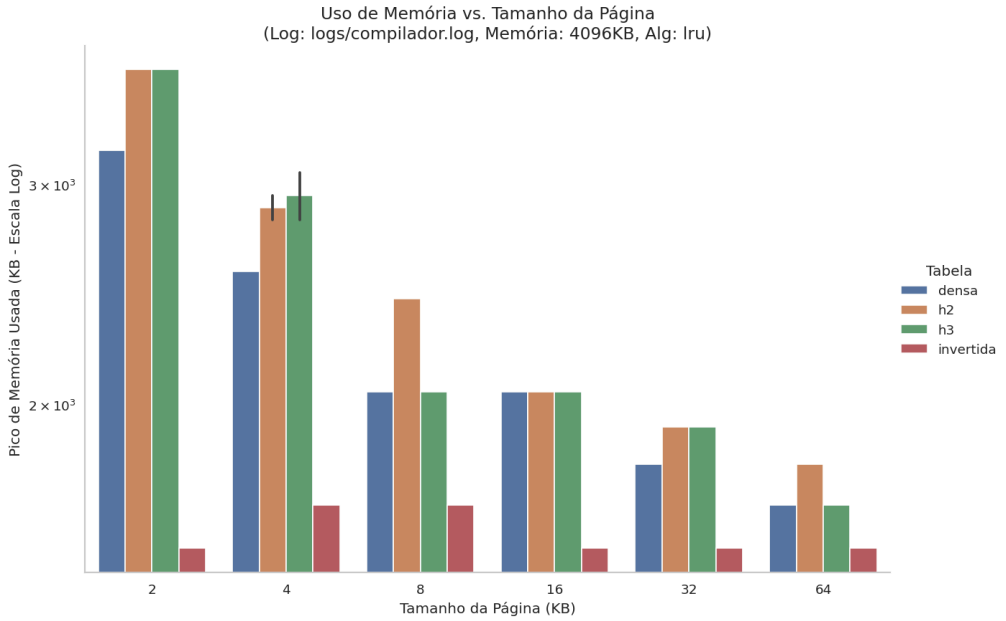



*Figura 5 - Uso de Memória Vs. Tamanho da página*

## 5.3 Gerência de tempo

Em relação ao tempo, a tabela invertida era mais custosa, pois havia uma busca pela página. A tabela densa, em geral, era a mais rápida, porém isso varia dependendo do padrão de acesso. As tabelas hierárquicas foram intermediárias, com a de dois níveis ligeiramente mais rápida que a de três. Vale ressaltar que essa diferença foi mínima nos testes realizados (menos de 0.3 segundos), porém poderia ter influenciado mais em operações com milhões ou bilhões de acessos.

Além disso, alguns casos foram diferentes, analisando também o padrão de acesso. No mesmo exemplo do ponto anterior, o log da matriz, teve um tempo mais elevado quando se utilizou a tabela hierárquica de 3 níveis.

Considerando agora a variância da memória física, mantendo uma página constante, o tempo de tradução ficou mais ou menos igual, porém para memórias muito grandes, e dependendo do tipo de acesso, algumas tabelas tiveram um desempenho melhor ou pior. Esperava-se que a tabela invertida demorasse mais pois havia mais busca, porém o tempo de execução total diminui pois houve menor taxa de page faults.

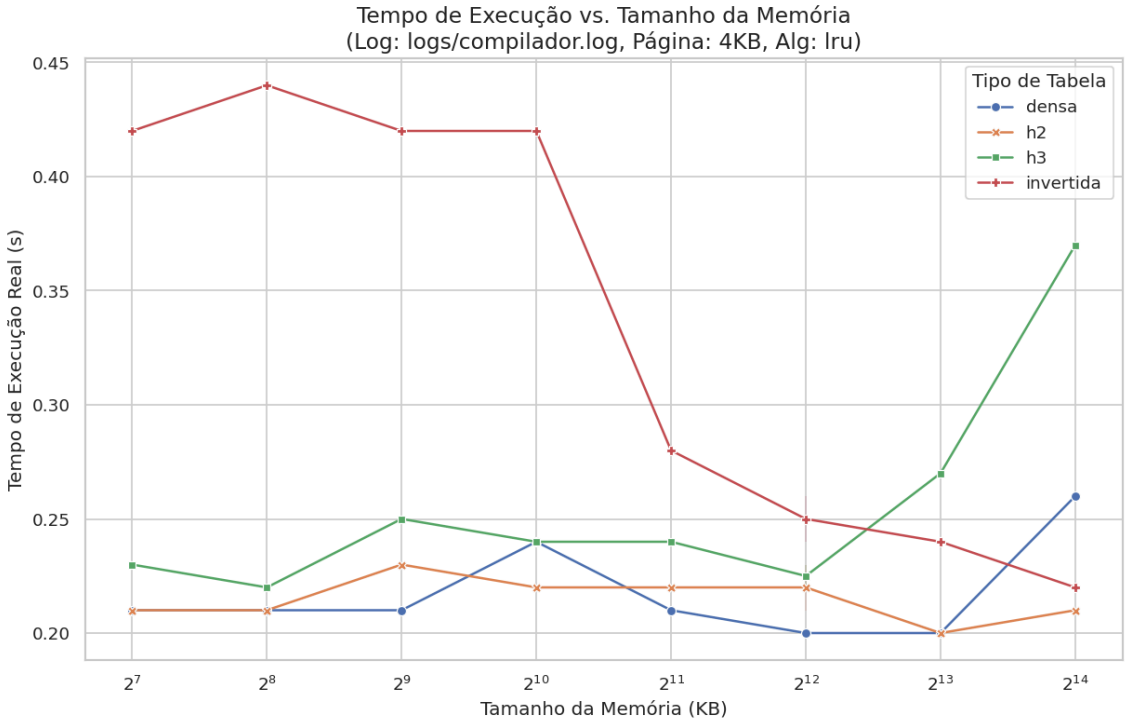



*Figura 6 - Tempo de execução Vs. Tamanho da memória*

Por fim, analisando a variância do tamanho das páginas, mantendo uma memória física constante, observou-se uma mudança mínima nos tempos. As tabelas eram menores mas a taxa de page fault aumentou ligeiramente, portanto o tempo total não mudou muito.

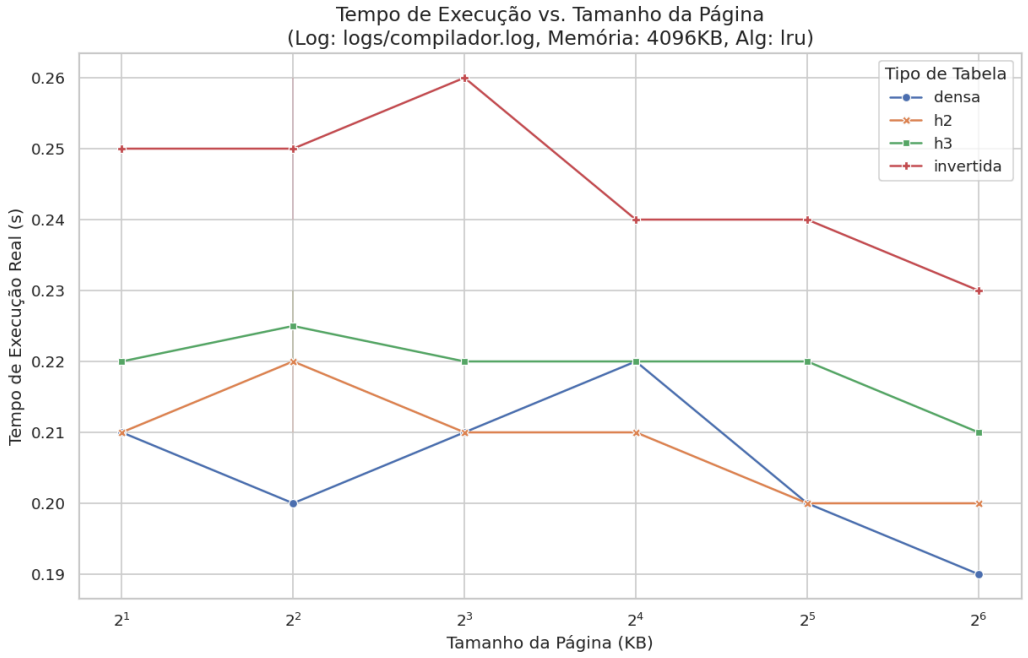



*Figura 6 - Tempo de execução Vs. Tamanho da memória*

## 6. Conclusões

Este trabalho prático permitiu uma análise interessante dos compromissos (trade-offs) envolvidos no design de sistemas de memória virtual. Foi possível observar na prática os conceitos teóricos do funcionamento da memória virtual e gerenciamento de páginas.

Os resultados divergiram um pouco. O LFU e MFU conquistaram um desempenho pior que os demais e o Random teve um desempenho bem melhor do que o previsto, em relação a taxa de page faults. Além disso, foi interessante observar como o tipo de acesso que ocorre pode influenciar bastante no desempenho geral do algoritmo, acentuando a importância de saber exatamente quais seriam os requisitos do sistema que está sendo implementado para escolher a melhor técnica.

Outrossim, apesar de funcional, o simulador desenvolvido representa um modelo simplificado. Diversos pontos poderiam ser aprimorados para criar uma simulação de maior fidelidade e propor novas investigações. Em especial, modificações nos algoritmos de LFU e MFU deveriam ser feitas para obter um mecanismo de envelhecimento que garantisse uma execução mais robusta. Além disso, todos os algoritmos, com exceção do Random, foram implementados usando uma iteração dos quadros. Seria interessante utilizar uma lista duplamente encadeada e um hashmap para o LRU e uma fila de prioridade para o LFU e MFU. Ademais, outros algoritmos como o CLOCK, também poderiam ser implementados.

Para finalizar, o modelo atual possui uma simplificação significativa que poderia ser endereçada: O simulador atual não possui um TLB. Isso significa que todo acesso à memória (mesmo um "HIT" na RAM) paga o custo de consulta da tabela de páginas. Em um sistema real, a maior parte dos acessos são resolvidos pelo TLB. O custo da tabela de páginas só é pago em um TLB miss. Uma melhoria significativa seria implementar um pequeno cache (o TLB) que armazena os mapeamentos mais recentes. O simulador então mediria "TLB Hits" e "TLB Misses". Isso mostraria por que a localidade de referência é tão importante e como as tabelas h3 ainda são viáveis.

## 7. References


***THE GNU PROJECT.*** *Process Identification. In: The GNU C Library Reference Manual. 2001. Disponível em: https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_554.html.*

***BENGAR, D. A. Priority Cache Object Replacement by Using LRU, LFU and FIFO algorithms to Improve Cache Memory Hit Ratio. Transactions on Soft Computing****, v. 1, n. 1, p. 1–13, 2025. Disponível em: https://www.tsc.reapress.com.*