Relatório - Trabalho Prático 2
Sistemas Operacionais
Memória Virtual

Rafael Araujo Magesty
Arthur Araujo Rabelo
Universidade Federal de Minas Gerais

2025/1

1 Introdução

O presente trabalho tem como objetivo a implementação de um simulador de memória virtual, no contexto da disciplina Sistemas Operacionais (2025/1). O simulador foi desenvolvido em linguagem C, sem o uso de recursos de C++, e tem como propósito simular o comportamento de gerenciamento de memória, através da manipulação de tabelas de páginas, quadros físicos, e algoritmos de substituição de páginas.

A entrada do simulador é um arquivo contendo uma sequência de acessos à memória, compostos por endereços de 32 bits seguidos por operações de leitura (R) ou escrita (W). O simulador deve processar os acessos, detectar *page faults*, aplicar os algoritmos de substituição em diferentes tabelas de páginas e, ao final, gerar estatísticas de desempenho.

2 Escolha dos algoritmos de substituição

Os algoritmos foram escolhidos com base na familiaridade dos desenvolvedores, visto que estes foram alguns dos que tivemos contato durante o curso. Foram eles:

- **Random**: Algoritmo simples que seleciona aleatoriamente um quadro de memória para substituição. Sua principal função no projeto é servir como referência de comparação (baseline), uma vez que não leva em consideração o histórico de uso, apenas seleciona qualquer quadro da memória. O desempenho desse algoritmo é imprevisível, dependendo da sorte, visto que ignora qualquer localidade.
- **LRU (Least Recently Used)**: Algoritmo clássico que substitui a página que não é acessada há mais tempo, baseado na hipótese de localidade temporal, isto é, privilegiando as páginas que foram acessadas recentemente. A implementação foi realizada com auxílio de um contador global de acessos. Esse algoritmo deve ter um maior desempenho em programas com acessos repetitivos, como em loops. Exemplo: um processo que acessa repetidamente um conjunto pequeno de páginas que cabe completamente na memória. O pior caso acontece quando o padrão de acesso às páginas não repete páginas com frequência suficiente antes que elas sejam substituídas, incorrendo em *page faults* várias vezes.

- **MFU (Most Frequently Used)**: Algoritmo que substitui a página com maior número de acessos acumulados, com base na hipótese de que páginas muito utilizadas já podem não ser mais necessárias (privilegia páginas pouco acessadas). A implementação utiliza um contador de frequência individual em cada quadro físico. O melhor caso é quando as páginas vão se tornando rapidamente inúteis, ou seja, já não são mais importantes para o programa; e o pior caso é quando páginas mais usadas continuam sendo necessárias.

- **LFU (Least Frequently Used)**: Algoritmo que substitui a página com o menor número de acessos acumulados, com base na hipótese de que páginas pouco acessadas têm menos probabilidade de serem reutilizadas. A implementação reaproveita o mesmo contador de frequência usado no MFU. No entanto, ao contrário deste, as páginas mais acessadas são privilegiadas. O melhor cenário é quando a frequência de acesso de um programa reflete bem o padrão futuro de uso. O pior cenário é quando as páginas acessadas com frequência se tornam inúteis ao programa rapidamente.

3 Resumo do Projeto e Implementação

O projeto foi modularizado com o objetivo de facilitar a adição dos diferentes algoritmos e estruturas de dados:

- **Tabelas de Páginas (Page Tables):** Implementadas em quatro tipos diferentes, utilizando a seguinte estrutura de dados:

  A estrutura principal `page_table` contém um ponteiro genérico que pode apontar para qualquer um dos quatro tipos específicos de tabelas, identificados pelo `tableType`. Os tipos específicos são:

  - **Densa**: Implementada como um vetor linear de entradas (`page_table_block`), onde cada entrada contém:
    - `valid`: bit que indica se a página está na memória física
    - `frame`: índice do quadro físico correspondente
  - **Hierárquica de 2 níveis**: Estrutura dividida em tabela externa e tabelas internas:
    - Tabela externa contém ponteiros para tabelas internas (`two_level_page_table_block`)
    - Tabelas internas têm mesma estrutura da tabela densa
    - Divisão do endereço virtual em: índice da tabela externa + índice da tabela interna + offset
  - **Hierárquica de 3 níveis**: Extensão da de 2 níveis com mais um nível:
    - Tabela raiz aponta para tabelas de nível médio
    - Tabelas de nível médio apontam para tabelas de páginas finais
    - Divisão do endereço em 3 partes + offset
  - **Invertida**: Implementada como vetor de entradas (`inverted_page_table_block`):
    - Cada entrada representa um quadro físico
    - Contém:
      - `frame`: número do quadro físico

· `modified`: bit de modificação
· `page`: número da página virtual mapeada
· Informações de acesso para algoritmos de substituição
* Pesquisa linear para encontrar páginas

As funções principais incluem:

- `init_page_table()` - Inicializa qualquer tipo de tabela
- `get_page()` - Obtém uma entrada da tabela
- `set_tables_offset()` - Calcula divisão de endereços (divide o mais igualmente possível entre os diversos níveis)
- Funções específicas para cada algoritmo de substituição

- **Memória Física (Physical Frames):** Estruturada como vetor de `physical_frame`, com os seguintes campos:
  - `virtual_page`: número da página atualmente carregada;
  - `modified`: bit de modificação, indicando se houve escrita;
  - `allocated`: indica se o quadro está ocupado;
  - `last_access_moment`: instante do último acesso (usado no LRU);
  - `access_counter`: total de acessos (usado no MFU e LFU);
  - `virtual_page`: referência para a página virtual alocada.

- **Utilitários (utils):**

  O módulo de utilitários fornece funções auxiliares essenciais para o funcionamento do simulador de memória virtual, particularmente para o cálculo de offsets e manipulação de bits.

  - `calculateOffset` - Calcula o offset de página baseado no tamanho da página
    * Entrada: `page_size` (tamanho da página em bytes)
    * Saída: Número de bits necessários para representar o offset
    * Funcionamento: Calcula $\log_2$(`page_size`) contando quantas vezes o valor pode ser dividido por 2 até chegar a 1
  - `count_bits_unsigned` - Conta o número de bits necessários para representar um número
    * Entrada: `num` (número inteiro sem sinal de 32 bits)
    * Saída: Quantidade de bits necessários para representar o número
    * Caso especial: Retorna 1 para o valor 0
  - `make_mask` - Cria uma máscara de bits para extração de campos de endereço
    * Entrada: `bits` (número de bits para a máscara)
    * Saída: Máscara de N bits ativos (ex: 3 bits → 0b111)
    * Uso típico: Extração de campos de endereço virtual em tabelas hierárquicas
  - Estas funções são utilizadas principalmente por:

    - Módulo de tabelas de página para cálculo de divisão de endereços virtuais
    - Algoritmos de substituição que necessitam manipular campos de endereços
    - Funções de inicialização para configurar estruturas de dados

- **Algoritmos de reposição**:

  Para implementação dos algoritmos de substituição de páginas, tanto na memória física quanto na tabela de páginas invertida, foram utilizadas decisões que se baseiam na busca linear e em contadores:

  - LRU: utilização do contador global `access_counter`, que é incrementado a cada acesso à memória, permitindo marcar o tempo relativo de uso dos quadros pela variável `last_access_moment`. O algoritmo retorna o índice do quadro com o menor valor nessa variável.
  - LFU e MFU: a cada acesso a um quadro, a variável `access_counter` é incrementada. Os algoritmos retornam o índice do quadro com o menor (LFU) ou com o maior (MFU) valor nessa variável.
  - Random: o algoritmo apenas retorna um número aleatório entre 0 e o número total de páginas.

- **Leitura do arquivo de entrada:** Cada linha é processada por:

```
unsigned int addr;
char rw;
fscanf(file, "%x %c", &addr, &rw);
```

- **Cálculo dos `offsets` das páginas:** O deslocamento é calculado dinamicamente de acordo com o tipo de tabela e o tamanho da página pela função:

```
void set_tables_offset
```

- **Seleção de algoritmo:** O algoritmo de substituição é escolhido dinamicamente a partir dos argumentos da linha de comando:

  - `random`
  - `lru`
  - `mfu`
  - `lfu`

- **Tipo de página:** O tipo de página é escolhido dinamicamente a partir dos argumentos da linha de comando:

  - Densa: 0;
  - Hierárquica de 2 níveis: 1;
  - Hierárquica de 3 níveis: 2;
  - Invertida: 3.

- **Linha de comando (exemplos):**

- simulador lru arquivo.log 4 128 0 (tabela densa);
- simulador mfu arquivo2.log 2 1024 3 (tabela invertida);
- simulador lfu arquivo3.log 4 128 2 (tabela de 3 níveis);
- simulador random arquivo4.log 4 128 1 (tabela de 2 níveis).

4 Decisões de Projeto

As principais decisões de projeto tomadas foram:

- Modularização em múltiplos arquivos (`Memory.c/.h`, `PageTable.c/.h`, `utils.c/.h`, `main.c`), facilitando a extensão e manutenção do código.
- Inicialização explícita dos campos das estruturas com valores padrão (-1, false ou zero), evitando problemas de leitura de lixo de memória.
- Implementação dos contadores de tempo e frequência com variáveis simples de incremento, com custo computacional mínimo.
- Aproveitamento do mesmo campo `access_counter` tanto para o algoritmo MFU quanto para o LFU, simplificando o armazenamento de informações de acesso.
- Separação da estrutura e dos métodos da tabela invertida da estrutura de memória física, embora sejam parecidos. Esta decisão favorece o desacoplamento tanto semântico quanto a nível de execução das duas entidades.
- Separação da lógica de acesso à tabela invertida da lógica dos outros tipos de tabela, visto que a invertida "simula"a memória principal.
- Divisão mais igualitária possível dos tamanhos das tabelas hierárquicas, fazendo com que as tabelas de um nível não sejam muito maiores ou muito menores que as de outro nível.
- Alocação por demanda das tabelas internas (segundo e terceiro nível), garantindo menor utilização de memória.
- Busca linear nas tabelas de páginas e na memória, facilitando a implementação dos algoritmos de reposição, apesar de garantir tempo ótimo.
- O número de acessos a memória também considera o número de acessos às tabelas de páginas. Dessa forma, toda tabela densa tem pelo menos 1 acesso à memória; a tabela de dois níveis tem pelo menos 2; a de três níveis, 3 e a invertida 1 também. Cada `page fault` é outro acesso à memória, dessa vez à memória física.

5 Análise Experimental

5.1 Configurações de Experimento

Os experimentos serão realizados com:

- Arquivos de teste: `compilador.log`, `matriz.log`, `compressor.log`, `simulador.log`.

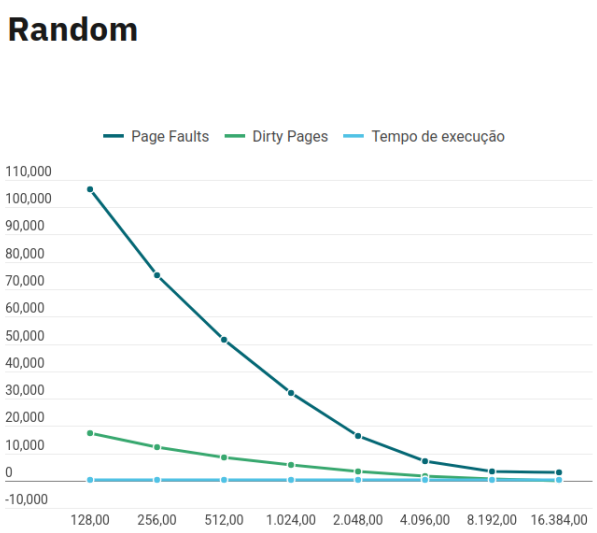
- Tamanhos de página: 2KB, 4KB, 8KB, 16KB, 32KB, 64KB.
- Tamanhos de memória: 128KB até 16MB.
- Tabelas de páginas: densa, hierárquica de dois e três níveis e invertida.
- `Valgrind`: checa se há vazamento de memória.

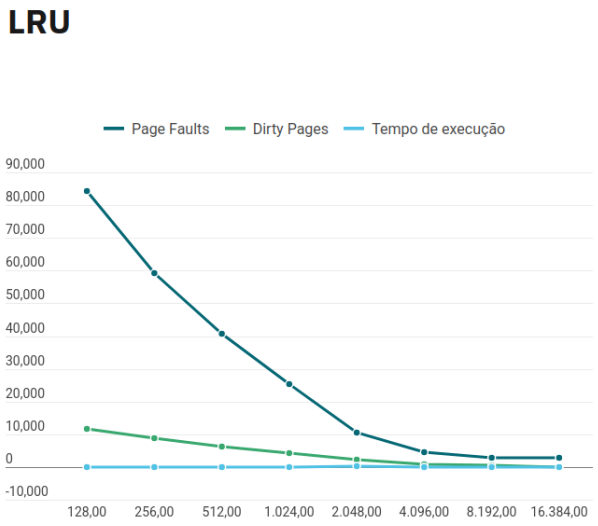
5.2 Resultados

5.2.1 Comparação entre algoritmos

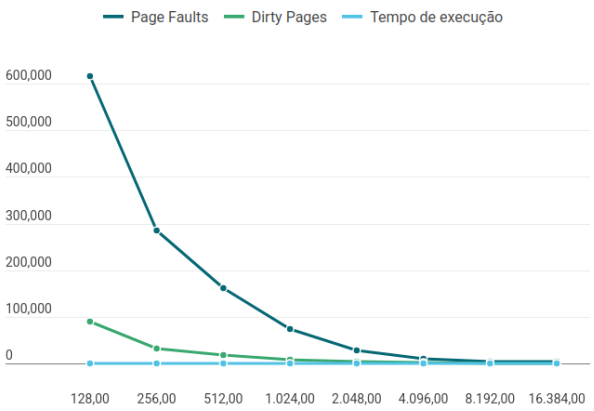
A comparação foi feita para todos os arquivos e pode ser conferida na planilha: Análise de desempenho (algoritmos) - TP2 - SO. A título de exemplo e para fins de simplificação, colocamos apenas os gráficos para o arquivo `compilador.log`:

- Memória variando entre 128KB e 16384KB, com página de 4KB:

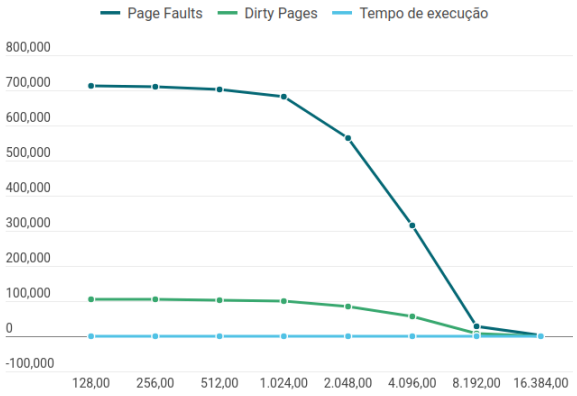




LFU

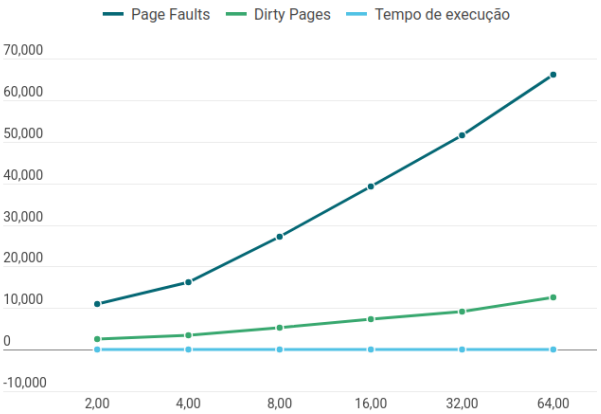


MFU

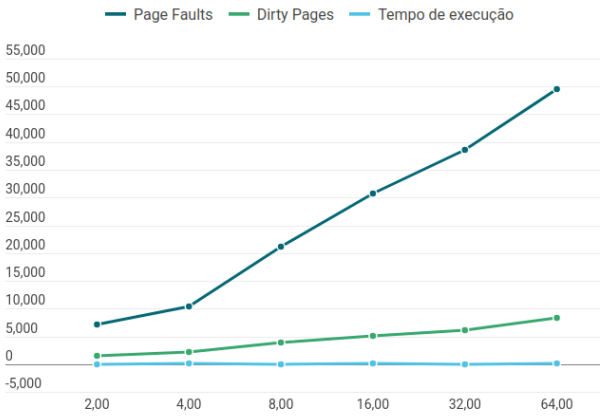


- Página variando entre 2KB e 64KB, com memória de 2048KB:

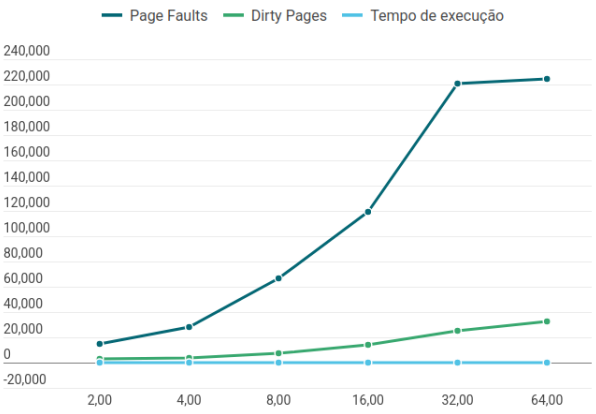
Random



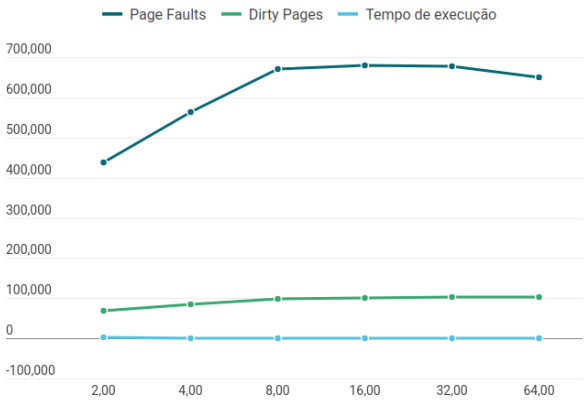
LRU


Page Faults
Dirty Pages
Tempo de execução
55,000
50,000
45,000
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
-5,000
2,00
4,00
8,00
16,00
32,00
64,00

LFU


Page Faults
Dirty Pages
Tempo de execução
240,000
220,000
200,000
180,000
160,000
140,000
120,000
100,000
80,000
60,000
40,000
20,000
0
-20,000
2,00
4,00
8,00
16,00
32,00
64,00

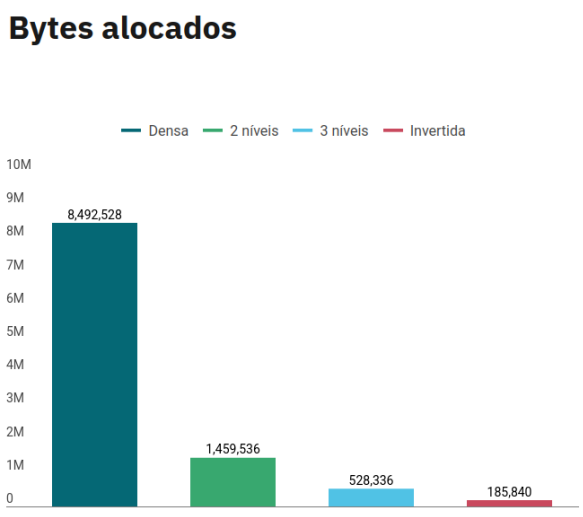
MFU


Page Faults
Dirty Pages
Tempo de execução
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0
-100,000
2,00
4,00
8,00
16,00
32,00
64,00

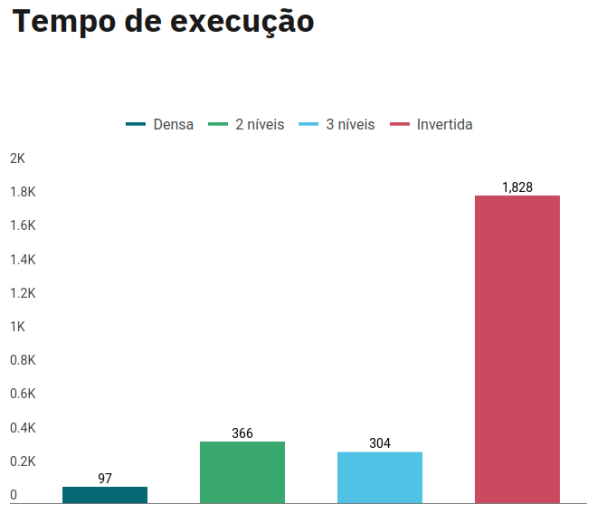
5.2.2 Comparação entre tabelas de páginas

A comparação foi feita apenas para o arquivo `compilador.log`: Análise de desempenho (tabelas) - TP2 - SO. Os gráficos abaixo foram gerados utilizando o algoritmo LRU, páginas de 4KB e memória de 16384KB.

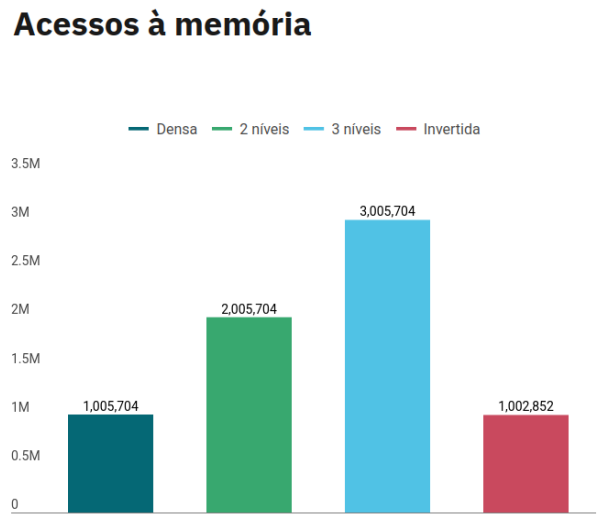
- Bytes alocados:



- Tempo de execução:



- Acessos à memória:

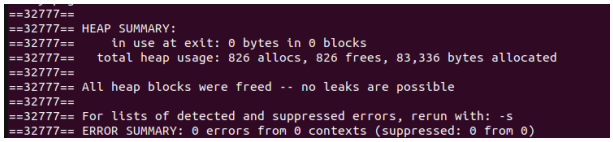


5.2.3 Verificação de vazamentos de memória

Para verificar `memory leaks`, foi utilizada a ferramente do `valgrind`. Para executá-la na linha de comando:

`valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./simulador args`.

Isso acabará retornando uma tela assim, indicando que não existem vazamentos:



5.3 Discussão dos Resultados

- Algoritmos:
  - Ao aumentar o tamanho da memória, foi possível perceber, em todos os algoritmos, que a quantidade de de `page faults` diminuiu, o que já era esperado, pois quanto maior a memória, menor a necessidade de que páginas sejam substituídas; mais páginas cabem na memória ao mesmo tempo. Também foi possível perceber que a quantidade de páginas sujas que tiveram que ser escritas no "disco"foi menor. Isso também se deu ao fato explicitado anteriormente: quanto menos substituição, menos páginas vão ter que ser escritas novamento no "disco"caso tenham sido modificadas.

    Ao aumentar o tamanho da página, a quantidade de `page faults` e de `dirty pages` também aumentou. Isso ocorreu porque à medida que o tamanho das páginas aumenta, menos páginas cabem na memória e, portanto, haverá mais falhas na busca e mais substituição.

    O tempo de execução se manteve quase constante, com alterações desprezíveis, visto que esta análise foi feita apenas com o tipo de tabela densa e utilizando a mesma estratégia de busca linear nos algoritmos de substituição.
  - Taxa de `page faults`:

Para calcular a taxa de `page faults`, fizemos a divisão entre o número de `page faults` e o número de acessos. Para esta análise, vamos considerar a média dessa taxa entre as variações de memória e de páginas.

Pela planilha, o algoritmo com a menor taxa média de `page faults` foi o LRU. Isso pode significar que o `compilador.log` tem localidade temporal, isto é, privilegia as páginas que foram acessadas recentemente. O algoritmo com a maior taxa média foi o MFU, o que pode indicar outro caráter do arquivo: páginas que são acessadas mais frequentemente não perdem sua "importância"durante o programa.

- Tabelas:

  Pelos resultados obtidos, foi possível constatar vários resultados esperados:

  – O número de memória gasta (`bytes` alocados) é maior para tabelas densas e menor para tabelas invertidas. Isso prova que, para um número de páginas muito grande, faz sentido adotar tabelas hierárquicas ou invertidas para minimizar o custo de memória.

  – O tempo de execução é menor para tabelas densas, visto que é preciso percorrer apenas um nível para verificar se a página é válida.

  – O número de acessos à memória é maior à medida que o número de níveis da tabela de páginas cresce, visto que são mais "camadas"a serem percorridas até chegar à página de fato. Na tabela invertida e densa, como são de apenas 1 nível, o número de acessos é bem parecido.

6 Conclusão

Este trabalho permitiu aplicar, de forma prática, os conceitos de gerenciamento de memória estudados em sala, como tabelas de páginas, quadros físicos e algoritmos de substituição. A implementação incremental dos algoritmos Random, LRU, MFU e LFU possibilitou compreender melhor o impacto de cada política sobre o desempenho, evidenciando como o padrão de acesso à memória influencia diretamente o número de *page faults* e páginas sujas.

Além disso, a modularização do código facilitou o entendimento de como os sistemas operacionais estruturam suas rotinas de gerenciamento de memória, aproximando a teoria da prática real. A manipulação dos logs de acesso também contribuiu para o desenvolvimento de habilidades de análise e simulação de cenários reais de execução, fundamentais para a compreensão do funcionamento interno dos sistemas.