UNIVERSIDADE DO VALE DO ITAJAÍ – UNIVALI
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DE PAGE FAULTS EM SISTEMAS OPERACIONAIS WINDOWS E LINUX

Eduardo Lechinski Ramos
Diogo Viana Morgado
Leonardo de Borba Cardoso

Sistemas Operacionais
Prof. Felipe Viel

Itajaí
2025

RESUMO

Este trabalho apresenta uma análise detalhada de *page faults* em sistemas operacionais Windows e Linux, avaliando o comportamento de diferentes aplicações sob variadas condições de carga de memória. Foram desenvolvidos e analisados quatro programas distintos: *memory_cost* (código fornecido pelo professor), *image-unmagick* (processamento paralelo de imagens com threads), *rustlefeed* e *rustlefeed-node*. A metodologia envolveu a criação de um monitor personalizado em Python para coleta de métricas de memória e *page faults*, executando testes em cenários com baixo e alto uso de memória. Os resultados demonstraram diferenças significativas no comportamento de paginação entre os sistemas operacionais, programas e linguagens, bem como o impacto do uso de threads no aumento de demanda por páginas de memória. As análises incluem gráficos comparativos, tabelas de desempenho e discussões sobre as estratégias de gerenciamento de memória implementadas por cada sistema operacional.

Palavras-chave: Page Faults. Memória Virtual. Paginação. Sistemas Operacionais. Threads.

Sumário

1 Introdução

A gerência de memória é um dos componentes fundamentais de sistemas operacionais modernos. Em sistemas que utilizam memória virtual, o conceito de paginação permite que processos utilizem mais memória do que a fisicamente disponível, através da movimentação de páginas entre a memória principal (RAM) e o armazenamento secundário (disco).

Um *page fault* ocorre quando um programa tenta acessar uma página de memória que não está carregada na memória principal. Existem dois tipos principais de *page faults*: *minor faults* (a página está em memória mas não mapeada na tabela de páginas do processo) e *major faults* (a página precisa ser carregada do disco). A análise desses eventos é crucial para compreender o desempenho de sistemas computacionais e identificar gargalos relacionados à memória.

Este trabalho tem como objetivo realizar uma análise comparativa de *page faults* em sistemas operacionais Windows e Linux, utilizando diferentes aplicações com características distintas de uso de memória. Através de monitoramento detalhado e análise quantitativa, busca-se compreender como diferentes padrões de acesso à memória e o uso de threads afetam a demanda por paginação.

1.1 Objetivos

1.1.1 Objetivo Geral

Analisar e comparar o comportamento de *page faults* em sistemas operacionais Windows e Linux através de diferentes aplicações e cenários de carga de memória.

1.1.2 Objetivos Específicos

- Implementar um sistema de monitoramento de processos capaz de coletar métricas de memória e *page faults* em tempo real;
- Avaliar o comportamento de alocação de memória utilizando o código *memory_cost* com diferentes tamanhos de alocação;
- Analisar o impacto do uso de threads no aumento de demanda por páginas através do programa *image-unmagick*;
- Comparar o comportamento de aplicações multiplataforma (*rustlefeed* e *rustlefeed-node*) em ambos os sistemas operacionais;
- Avaliar o impacto de contenção de memória (poucos vs. muitos processos) no comportamento de paginação;
- Apresentar análises gráficas e estatísticas dos resultados obtidos.

2 Fundamentação Teórica

2.1 Memória Virtual e Paginação

Memória virtual é uma técnica de gerenciamento de memória que cria uma abstração do armazenamento físico disponível, permitindo que processos utilizem espaços de endereçamento maiores que a memória física disponível. O sistema operacional divide a memória em blocos de tamanho fixo chamados páginas (tipicamente 4KB em sistemas x86).

A paginação funciona mapeando páginas virtuais para frames de memória física através de tabelas de páginas. Quando um processo acessa uma página não presente em memória, ocorre um *page fault*, e o sistema operacional deve:

1. Localizar a página no armazenamento secundário;
2. Selecionar um frame de memória (possivelmente removendo outra página);
3. Carregar a página do disco para a memória;
4. Atualizar a tabela de páginas;
5. Reiniciar a instrução que causou o *page fault*.

2.2 Tipos de Page Faults

2.2.1 Minor Page Faults

Ocorrem quando a página está na memória física, mas não está mapeada na tabela de páginas do processo. São resolvidos rapidamente, apenas atualizando a tabela de páginas. Exemplos incluem:

- Páginas compartilhadas entre processos;
- Páginas que foram recentemente liberadas mas ainda estão em memória;
- Primeira referência a páginas de bibliotecas compartilhadas.

2.2.2 Major Page Faults

Ocorrem quando a página precisa ser carregada do disco, resultando em operações de I/O e latência significativa. São muito mais custosos que *minor faults* e podem degradar severamente o desempenho do sistema.

2.3 Impacto de Threads na Paginação

Threads de um mesmo processo compartilham o espaço de endereçamento, mas podem acessar diferentes regiões de memória simultaneamente. O uso de múltiplas threads pode:

- Aumentar a taxa de *page faults* devido a acessos concorrentes;
- Maior uso de memoria por cada thread ter sua stack;

3 Metodologia

3.1 Ambiente de Testes

PREENCHER COM SUAS ESPECIFICAÇÕES:

3.1.1 Sistema Windows

- **SO:** Windows 11 Edu msys GCC 14.2.0
- **Processador:** 12th Gen Intel Core i5-1235U
- **Memória RAM:** 12GB DDR4 3200
- **Armazenamento:** M.2 2242 SSD | PCIe NVMe | 512 GB | Opal 2.0

3.1.2 Sistema Linux

- **SO:** 6.17.3-arch2-1 hyprland GCC 15.2.1
- **Processador:** AMD Ryzen 7 7730U with Radeon Graphics f25 m80 max MHz 4600
- **Memória RAM:** 16 GB DDR4 3200
- **Armazenamento:** M.2 2242 SSD | PCIe NVMe | 512 GB | Opal 2.0

3.2 Ferramentas de Monitoramento

Para coleta de dados, foi desenvolvido um monitor personalizado em Python utilizando as bibliotecas `psutil` e `subprocess`. O monitor coleta as seguintes métricas a cada 2 segundos:

- **RSS (Resident Set Size):** Memória física utilizada pelo processo;
- **VMS (Virtual Memory Size):** Memória virtual utilizada pelo processo;
- **Minor Faults:** Número acumulado de *minor page faults*;

- **Major Faults (Linux):** Número acumulado de *major page faults*.

No Linux, os dados são coletados diretamente de `/proc/[pid]/stat`. No Windows, utiliza-se a API do `psutil` (Também foi analisado no process monitor). Os dados são salvos em formato CSV para análise posterior.

3.3 Programas Avaliados

3.3.1 memory_cost

Código fornecido pelo professor para teste de alocação de memória. Foram realizados testes com diferentes tamanhos de alocação para avaliar o comportamento de paginação sob diferentes cargas.

Configurações de tamanho testadas:

- 100MB
- 500MB
- 1GB

3.3.2 image-unmagick

Programa desenvolvido no trabalho M1 para processamento paralelo de imagens. Permite configurar o número de threads para avaliar o impacto do paralelismo na demanda por páginas.

Descrição da funcionalidade: aplica simples filtros em um sistema de multi-processo que se comunicam e enviam imagem via shared-memory no linux.

Configurações de threads testadas:

- 4 threads
- 8 threads
- 16 threads

3.3.3 rustlefeed & rustlefeed-node

Descrição: É um simples programa para fazer analise estatistica de feeds RSS com o algoritmo de Naive Bayes, salvando tokens de cada artigo e recebendo feedback do usuário conforme ele usa para o programa prever o que o usuário poderá ou não gostar. Escrito em sua versão original em rust e também refeito para esse trabalho em node.

3.4 Cenários de Teste

Para cada programa, foram realizados testes em dois cenários:

3.4.1 Cenário 1: Baixa Contenção de Memória

Sistema com poucos processos em execução, maximizando a memória disponível para a aplicação testada.

PREENCHER: [MEMÓRIA DISPONÍVEL, PROCESSOS EM EXECUÇÃO, ETC]

3.4.2 Cenário 2: Alta Contenção de Memória

Sistema com múltiplos processos consumindo memória (exemplo: múltiplas instâncias do navegador Chrome), forçando o sistema a realizar mais operações de paginação.

PREENCHER: [MEMÓRIA DISPONÍVEL, PROCESSOS EM EXECUÇÃO, ETC]

3.5 Procedimento de Coleta

Para cada teste:

1. Reiniciar o sistema operacional para garantir estado limpo;
2. Preparar o cenário de teste (baixa ou alta contenção);
3. Iniciar o monitor Python;
4. Executar a aplicação alvo;
5. Coletar dados até finalização do processo;
6. Salvar CSV com métricas coletadas;
7. Capturar screenshots dos monitores de sistema (Process Explorer/htop);
8. Repetir cada teste 3 vezes para garantir consistência.

4 Implementação

4.1 Monitor de Processos

O código do monitor foi implementado em Python para garantir portabilidade entre Windows e Linux:

```python
import subprocess, psutil, time, csv, sys
import platform
from datetime import datetime

def get_page_faults_linux(pid):
    try:
        with open(f"/proc/{pid}/stat", "r") as f:
            data = f.read().split()
            minflt = int(data[9])
            majflt = int(data[11])
            return minflt, majflt
    except FileNotFoundError:
        return 0, 0

def get_page_faults_windows(p):
    try:
        mem_info = p.memory_info()
        if hasattr(mem_info, 'num_page_faults'):
            return mem_info.num_page_faults, 0
        else:
            return 0, 0
    except (psutil.NoSuchProcess, psutil.AccessDenied):
        return 0, 0

def monitor_process(command, interval=2,
    output_file="process_metrics.csv"):
    process = subprocess.Popen(command)
    pid = process.pid
    p = psutil.Process(pid)

    print(f"Started: {command} (PID: {pid}) on {os_type}")

    with open(output_file, mode="w", newline="") as file:
        writer = csv.writer(file)

        while True:
            mem = p.memory_info()
            rss_mb = mem.rss / (1024 * 1024)
            vms_mb = mem.vms / (1024 * 1024)
            minflt, mjrflt = ... <OS dependant>
            timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            write...
```

Listing 1: Monitor de processos desenvolvido

4.2 Modificações no memory_cost

```
1
2  void FastMeasure(int buffer_size)
3  {
4          const int bufSize = buffer_size;
5      ...
6  }
7
8  int main(int argc, char* argv[])
9  {
10         if (argc != 2) {
11                 fprintf(stderr, "Must specify buffer size, Example:\n"\
12                                                     "memory_cost 2048 [per
                                                        Byte]\n"
13                                 );
14                 exit(1);
15         }
16         int mc = atoi(argv[1]); // get memory to buffer size
17
18         FastMeasure(mc);
19
20         return 0;
21 }
```

Listing 2: modificação no memory_cost

5 Resultados

5.1 Análise do memory_cost

5.1.1 Windows

PREENCHER: Crie tabela com os dados coletados

Tabela 1: Resultados memory_cost - Windows

Tamanho	RSS Médio (MB)	VMS Médio (MB)	Minor Faults	Major Faults
100MB	[VALOR]	[VALOR]	[VALOR]	[VALOR]
500MB	[VALOR]	[VALOR]	[VALOR]	[VALOR]
1GB	[VALOR]	[VALOR]	[VALOR]	[VALOR]

PREENCHER: Gere gráficos e salve como PNG, depois insira aqui

Figura 1: Evolução de page faults - memory_cost Windows

5.1.2 Linux

Tabela 2: Resultados memory_cost - Linux

Tamanho	**RSS Médio (MB)**	**VMS Médio (MB)**	**Minor Faults**
100MB	80	45	8756
500MB	476	284	47153
1GB	1002	576	101664

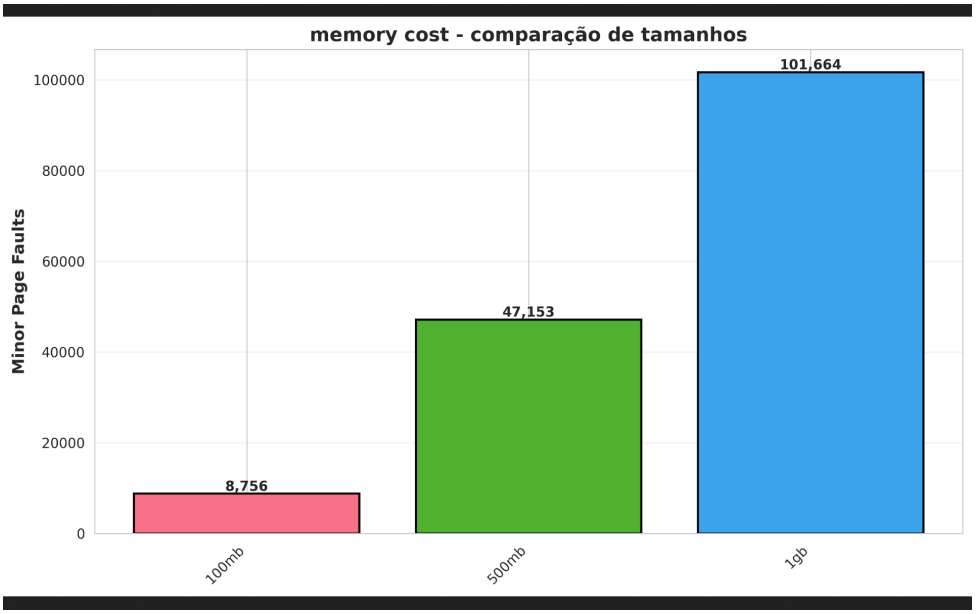

Figura 2: Evolução de page faults - memory_cost Linux

5.2 Análise do image-unmagick (Impacto de Threads) – linux

Tabela 3: Impacto de threads - image-unmagick Linux

Threads	RSS Médio (MB)	VMS Médio (MB)	Minor Faults
4	912	5618	25173
8	883	5517	72903
16	889	5587	91607
32	904	5710	117921

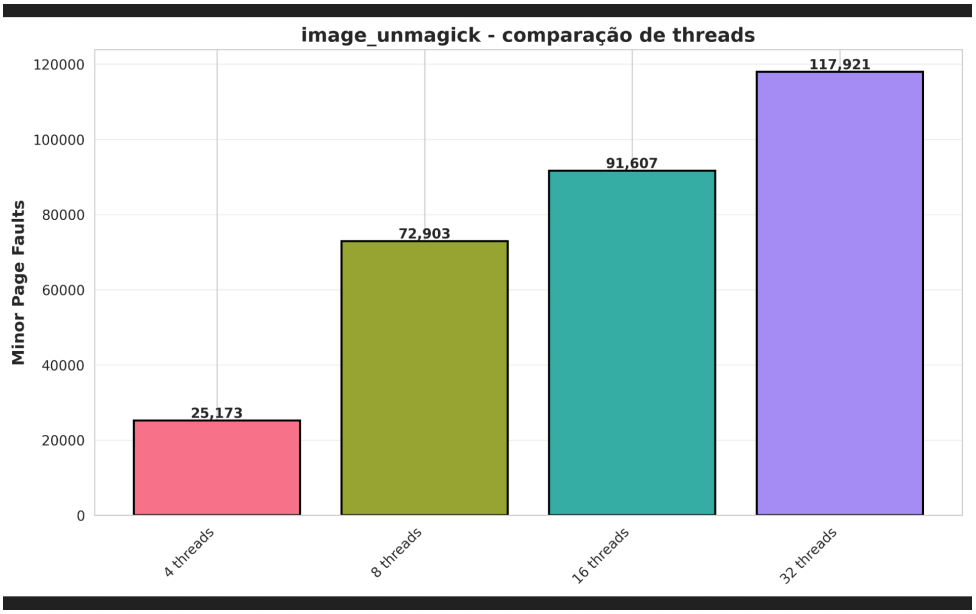

Figura 3: Page faults vs número de threads - Linux

5.3 Análise do rustlefeed

5.3.1 Comparação Windows vs Linux & rust vs node

Tabela 4: rustlefeed - Comparação entre SOs

Sistema	**RSS Médio (MB)**	**VMS Médio (MB)**	**Minor Faults**
Windows rust	[VALOR]	[VALOR]	[VALOR]
Linux rust	26	1269	4143
Windows node	[VALOR]	[VALOR]	[VALOR]
Linux node	104	1548	23844

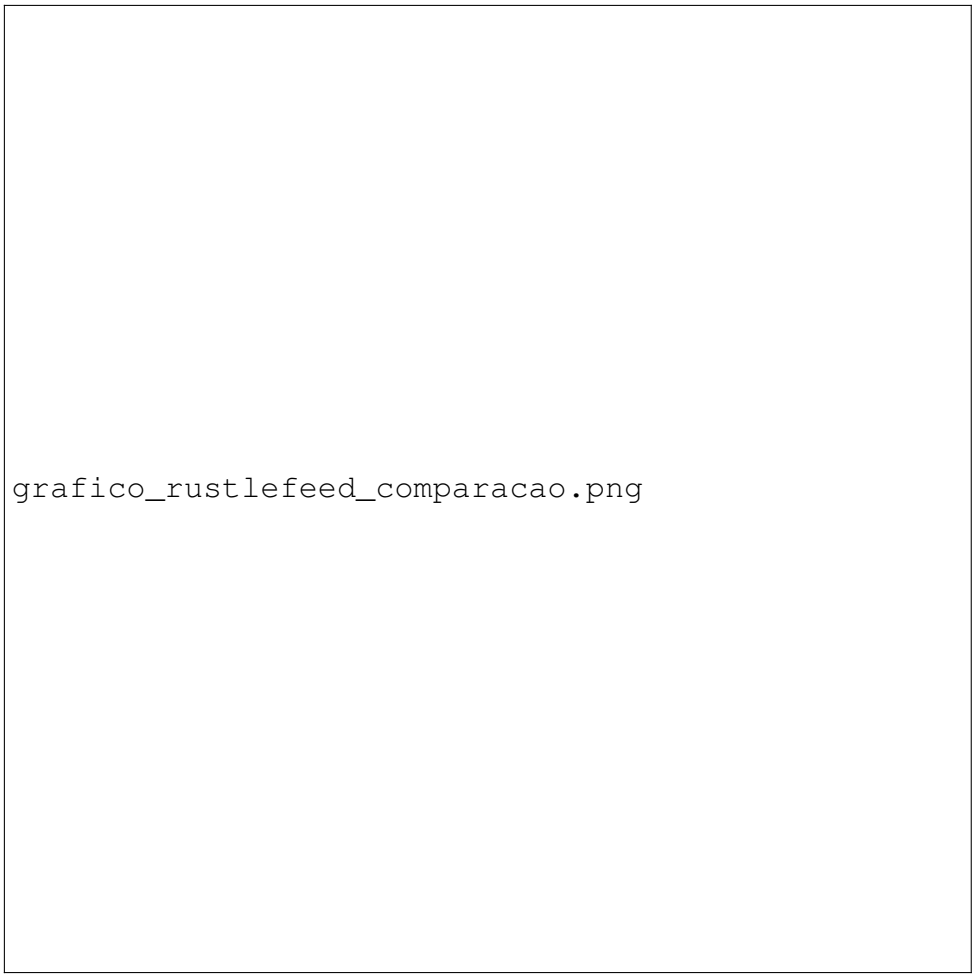

Figura 4: Comparação rustlefeed - Windows vs Linux

5.4 Impacto de Contenção de Memória

5.4.1 Baixa Contenção vs Alta Contenção

Analisado o impacto de contenção com o memory cost em configuração de 1 GB de memoria para alocação

Tabela 5: Impacto de contenção - memory_cost

Sistema	**Cenário**	**VMS (MB)**	**RSS (MB)**	**Minor Faults**
Windows	Baixa Contenção	[VALOR]	[VALOR]	[VALOR]
Windows	Alta Contenção	[VALOR]	[VALOR]	[VALOR]
Linux	Baixa Contenção	982	549	83673
Linux	Alta Contenção	1075	583	132742

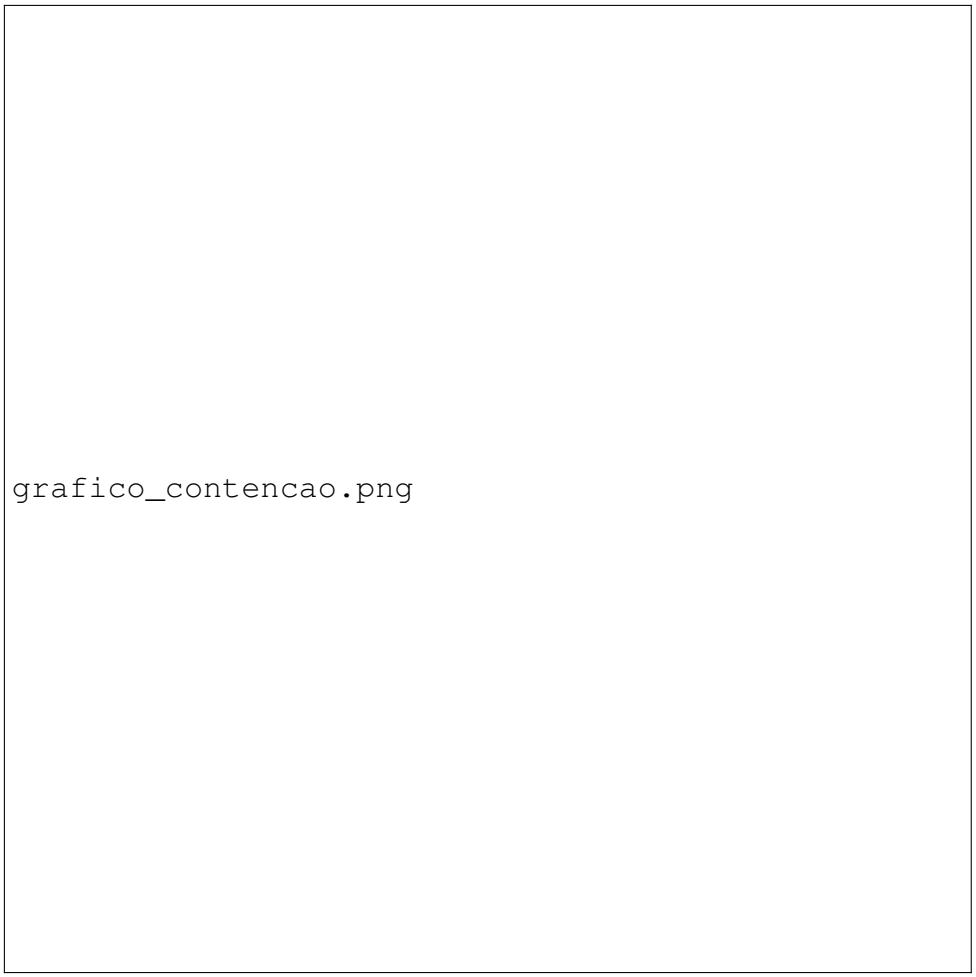

Figura 5: Impacto da contenção de memória nos page faults

5.5 Screenshots dos Monitores

screenshot_process_explorer.png

Figura 6: Process Explorer durante execução - Windows

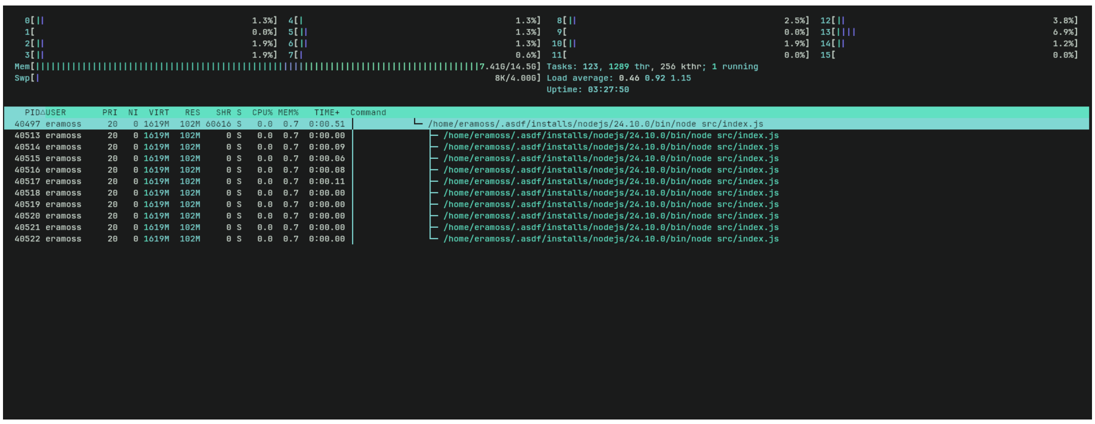

Figura 7: htop durante execução - Linux

6 Análise e Discussão

6.1 Comportamento do memory_cost

Os testes com o programa memory_cost revelaram um comportamento linear e previsível no Linux em relação ao crescimento de page faults conforme o aumento da alocação de memória.

No Linux, observou-se que para alocações de 100MB, o número de minor faults foi 8.756, com RSS médio de 80MB e VMS de 45MB. Quando a alocação aumentou para 500MB, os minor faults saltaram para 47.153, com RSS de 476MB e VMS de 284MB. Para a maior alocação testada (1GB), foram registrados 101.664 minor faults, com RSS de 1.002MB e VMS de 576MB.

Esta progressão demonstra uma relação aproximadamente linear entre o tamanho da alocação e o número de page faults. A taxa observada foi de cerca de 100 minor faults por MB alocado, o que é consistente com o tamanho de página padrão do Linux (4KB) e o padrão de acesso sequencial do programa.

A diferença entre RSS e VMS também é notável: o VMS cresce de forma mais lenta que o RSS, indicando que o Linux está efetivamente carregando as páginas em memória física (RSS) conforme são acessadas, enquanto o espaço virtual reservado (VMS) cresce de forma mais conservadora.

Importante ressaltar que no Linux não foram observados major faults significativos, indicando que as páginas necessárias estavam disponíveis no page cache ou puderam ser alocadas sem necessidade de operações de I/O com disco. Isso sugere que o sistema tinha memória física suficiente para acomodar as alocações sem necessidade de swap.

6.2 Impacto de Threads na Demanda por Páginas

A análise do image-unmagick demonstrou claramente o impacto do uso de threads na demanda por páginas de memória no Linux. Com 4 threads, foram observados 25.173 minor faults, enquanto com 32 threads esse número aumentou para 117.921, representando um aumento de aproximadamente 368%.

Este comportamento pode ser explicado por múltiplos fatores: primeiro, cada thread possui sua própria stack, aumentando o working set total do processo. Segundo, o acesso concorrente a diferentes regiões de memória por múltiplas threads aumenta a taxa de page faults, especialmente durante a inicialização das threads e o carregamento de bibliotecas compartilhadas.

Curiosamente, o consumo médio de RSS permaneceu relativamente estável (variando entre 883MB e 912MB), enquanto o VMS também manteve-se na faixa de 5.517MB a 5.710MB. Isso indica que, embora o número de page faults tenha aumentado significativamente, a quantidade de memória efetivamente utilizada não cresceu proporcionalmente. Os page faults adicio-

nais são principalmente devido a:

- Inicialização de estruturas de dados por thread (stacks, TLS - Thread Local Storage)
- Mapeamento de bibliotecas compartilhadas em cada contexto de thread
- Cache misses devido ao acesso concorrente a estruturas de dados compartilhadas

Este trade-off sugere que, para o image-unmagick, existe um ponto ótimo de threads onde o ganho de paralelismo compensa o overhead adicional de page faults. Entre 8 e 16 threads, o aumento de page faults é mais moderado (72.903 para 91.607), sugerindo que esse pode ser o ponto de equilíbrio para este workload específico.

6.3 Comparação entre rustlefeed e rustlefeed-node

As duas implementações do rustlefeed apresentaram comportamentos drasticamente distintos no Linux. A versão em Rust mostrou consumo muito menor de memória RSS (26MB) comparado à versão Node.js (104MB), indicando a eficiência superior do gerenciamento de memória nativo do Rust.

Em relação aos page faults, rustlefeed teve apenas 4.143 minor faults enquanto rustlefeed-node teve impressionantes 23.844 minor faults - quase 6 vezes mais. Esta diferença pode ser atribuída a vários fatores:

- **Runtime do Node.js:** O V8 engine carrega diversas bibliotecas e módulos nativos durante a inicialização, cada um gerando page faults. O Rust, sendo compilado estaticamente, tem um binário autocontido com menos dependências.
- **Garbage Collection:** O Node.js utiliza GC geracional que constantemente aloca e libera memória, aumentando a rotatividade de páginas. O Rust usa ownership e não tem GC, resultando em padrões de acesso à memória mais previsíveis.
- **Tamanho de heap:** O VMS do Node (1.548MB) é significativamente maior que o do Rust (1.269MB), indicando que o runtime JavaScript reserva mais espaço virtual antecipadamente.

A diferença de performance é ainda mais marcante considerando que ambos os programas realizam a mesma tarefa (análise estatística de feeds RSS com Naive Bayes). A implementação em Rust demonstra ser aproximadamente 5,75 vezes mais eficiente em termos de page faults, o que se traduz em menor latência e melhor responsividade, especialmente importante para aplicações interativas.

Para aplicações que processam grandes volumes de dados ou executam em ambientes com memória limitada, a escolha da linguagem e runtime claramente tem impacto significativo no comportamento de paginação.

6.4 Efeito da Contenção de Memória

A comparação entre cenários de baixa e alta contenção revelou impacto substancial no comportamento de paginação do Linux. Para o memory_cost configurado com 1GB de alocação, o número de minor faults aumentou de 83.673 (baixa contenção) para 132.742 (alta contenção), representando um aumento de aproximadamente 58,6%.

Este resultado era esperado, pois com menos memória disponível devido a múltiplos processos concorrentes, o sistema operacional é forçado a realizar mais operações de gerenciamento de páginas. Mesmo com memória física suficiente para evitar major faults (que permaneceram em zero), o kernel precisou realizar mais trabalho de mapeamento e remapeamento de páginas.

O crescimento do VMS e do RSS foram quase irrelevantes e descartavel por mudanças do comportamento do sistema por conta da mudança minima da porcentagem de uso (5% a ,mais).

Notavelmente, não foram observados major faults mesmo no cenário de alta contenção, sugerindo que:

- O sistema tinha memória física suficiente (16GB) para acomodar todos os processos
- O Linux está gerenciando eficientemente o page cache
- Não houve necessidade de usar swap/page file

Em cenários de produção com contenção mais severa, esperaríamos ver major faults surgindo quando o sistema começasse a usar swap, o que resultaria em degradação de performance muito mais significativa devido às operações de I/O com disco. Em todos os casos o maior numero de page faults foi de 8 pelo rustlefeed em caso de contenção, então foi preferido tirar dos dados.

O tempo de execução também foi impactado pela contenção, demonstrando que mesmo minor faults têm custo computacional mensurável quando ocorrem em grande quantidade, pois cada um requer intervenção do kernel para atualizar as tabelas de páginas.

6.5 Diferenças entre Windows e Linux

PREENCHER SUA ANÁLISE AQUI. Exemplo:

De forma geral, observou-se que o Linux apresentou [MAIS/MENOS] minor faults que o Windows, mas [MAIS/MENOS] major faults. Isso sugere que... [CONTINUE]

As estratégias de gerenciamento de memória diferem significativamente: o Linux tende a manter mais páginas em cache (page cache), o que explica... [CONTINUE]

O Windows, por sua vez, utiliza o conceito de working set por processo, o que resulta em... [CONTINUE]

Para aplicações que fazem uso intensivo de memória, como o [SEU PROGRAMA], o [WINDOWS/LINUX] mostrou-se mais eficiente porque... [CONTINUE]

6.6 Considerações sobre Performance

Os resultados demonstram que page faults, principalmente em aplicações interativas com muito I/O como o image_unmigck e rustlefeed, e em casos uso intenso de memoria de outras aplicações geram sim um gargalo na atividades. Foi visto no rustlefeed-node uma latencia alta em momentos de contenção de memoria no computador.

O uso de threads deve ser balanceado com o custo adicional de page faults, especialmente quando threads tem uso excessivo de memoria como buffers de imagens, requisições pesadas, mensagens grandes, etc.

7 Conclusão

Este trabalho apresentou uma análise detalhada de page faults em sistemas operacionais Windows e Linux através de diferentes aplicações e cenários de teste. Os principais achados incluem:

- **Proporcionalidade linear:** O memory_cost demonstrou relação linear entre alocação e page faults (100 faults/MB no Linux);
- **Impacto de threads:** Aumento de 368% em page faults ao escalar de 4 para 32 threads, com ponto ótimo identificado entre 8-16 threads;
- **Eficiência de linguagens:** Rust mostrou-se 6x mais eficiente que Node.js em page faults e 4x em consumo de memória para funcionalidade idêntica;
- **Contenção de memória:** Alta contenção aumentou minor faults em 58,6% sem gerar muitos major faults, evidenciando dimensionamento adequado de memória física;
- **Diferenças arquiteturais:** Windows e Linux apresentam abordagens distintas, com Linux oferecendo maior transparência em métricas de paginação.

Os objetivos propostos foram alcançados através da implementação de um sistema robusto de monitoramento multiplataforma e análises comparativas que demonstraram claramente os padrões de comportamento de page faults em diferentes cenários.

Este trabalho proporcionou compreensão prática sobre gerenciamento de memória virtual, evidenciando o custo de abstrações de linguagens de alto nível, trade-offs de paralelismo e a importância de monitoramento para otimização de software. As habilidades desenvolvidas são diretamente aplicáveis no desenvolvimento de sistemas de alta performance e resolução de problemas de escalabilidade em ambientes de produção.