**Cauã Ribas Devitte, Lucas Iago Adriano, Nilson Andrade Neto**

Universidade do Vale do Itajaí - Univali

Escola Politécnica

Ciência da Computação

{cauaribas, lucas\_02, nilson.neto} @edu.univali.br

**Sistemas Operacionais**

Trabalho M2 – Paginação e Page Fault

**Felipe Viel**

12/11/2024

**Sumário:**

[**1. Introdução e Contexto: 3**](#_5xuymngr5k55)

[**2. Análise do algoritmo “Memory Cost” 3**](#_hx0bm784dmig)

[2.1. Introdução ao código: 3](#_uh9ryixtknzm)

[2.2. Classe Timer: 3](#_y49w2xxppcoe)

[2.3. Função BusyWait: 4](#_yk4u73naq38)

[2.4. Função FastMeasure (parte 01): 4](#_vu7nbwkfxswy)

[2.5. Função FastMeasure (parte 02): 5](#_7ov826pafm3x)

[2.6. Função FastMeasure (parte 03): 6](#_lk1k8yrtxms7)

[2.7. Função FastMeasure (parte 04): 6](#_xtjh3n10d9tv)

[2.9. Função Main: 7](#_ee1tldxth0lu)

[**2.10. Resultados 8**](#_3lff9uu7okru)

[2.10.1. Tabela com os resultados (Windows): 8](#_l2gnimu8jaar)

[2.10.2. Execução do algoritmo: 8](#_t15qg7hk8f9i)

[2.10.3. Código no Windows: 10](#_hunoerwcy5oc)

[2.10.4. Tabela com os resultados (Linux): 11](#_5vayfd51ehce)

[2.10.5. Execução do algoritmo: 11](#_d5ae24upu1df)

[2.10.6. Código no Linux: 13](#_f9jnhw7ezdno)

[2.11. Conclusões 14](#_7yxo53v0tzv9)

[**3. Análise código M1 14**](#_7h50qp1qw5vh)

[3.1. Introdução ao código: 14](#_f32hdgjdl2ab)

[3.2. Função executeTask: 14](#_gnb41m3zxxry)

[3.3. Função submitTask: 15](#_bn14dtiv3tfb)

[3.4. Função startThread: 15](#_ozgpge8jwazr)

[3.5. Função startServer: 16](#_hdxz7sgdvq10)

[3.6. Funções handleStringConnection e handleNumberConnection: 17](#_x1bg1cm1hjml)

[3.6.1. handleStringConnection: 17](#_ibnug6hoaoqj)

[3.6.2. handleNumberConnection: 18](#_gsbre9vont2p)

[3.7. Função acceptConnection: 19](#_4o1qtxdlg12)

[3.8. Função main: 20](#_rlceb04s22ga)

[3.9. Resultados 20](#_b3q54j8ijdb5)

[3.9.1. Código no Windows: 20](#_n1onaka9ekqn)

[3.9.2. Código no Linux: 21](#_1qev45d2acrz)

[3.10. Conclusões 22](#_euqdrci0vptd)

[**4. Análise código reordenação de matrizes 22**](#_88sa7duo11y3)

[4.1. Introdução aos códigos: 22](#_nkfcgblxh1x)

[4.2. Código Fonte (Python): 23](#_b5der68tvgo7)

[4.3. Código Fonte (C#): 24](#_pujyjgu80198)

[4.4. Análise (Python): 25](#_onizj4obiqni)

[4.5. Análise (C#): 25](#_vdz5pzqzl1a)

[4.6. Gráfico para comparação: 26](#_rihs43jugm4j)

[4.7. Conclusões: 26](#_bopz75wkchs)

[**5. Conclusão Geral 26**](#_85myqg8wz0ef)

[**6. Referências 27**](#_jlz72vgk68a7)

# **1. Introdução e Contexto:**

A análise de page faults em sistemas operacionais é essencial para monitorar, identificar e compreender o impacto desses eventos no desempenho do sistema. Um page fault ocorre quando um programa tenta acessar uma página de memória que não está carregada na memória principal, o que pode indicar uma necessidade de otimização, como ajustes nas políticas de paginação ou identificação de vazamentos de memória. A investigação dos page faults permite coletar dados sobre sua frequência e localização, identificar suas causas e, por fim, implementar estratégias que otimizem o desempenho do sistema. Este trabalho visa analisar a ocorrência de page faults em sistemas operacionais, comparando seu comportamento em três algoritmos distintos executados nos sistemas Windows e Linux.

# 2. Análise do algoritmo “Memory Cost”

## **2.1. Introdução ao código:**

Este código mede o desempenho de operações de memória, incluindo alocação, escrita, leitura e desalocação, utilizando uma classe **Timer** para registrar tempos de execução em segundos. A função **FastMeasure** realiza testes de alocação e desalocação de um buffer de 32 MB, escrita e leitura repetidas nesse buffer, além de ciclos completos de alocação, escrita, leitura e desalocação. Uma espera ativa inicializa a CPU em alta frequência para garantir resultados consistentes. Esses testes fornecem uma análise precisa do tempo gasto em operações comuns de gerenciamento de memória em C++.

## 2.2. Classe Timer:

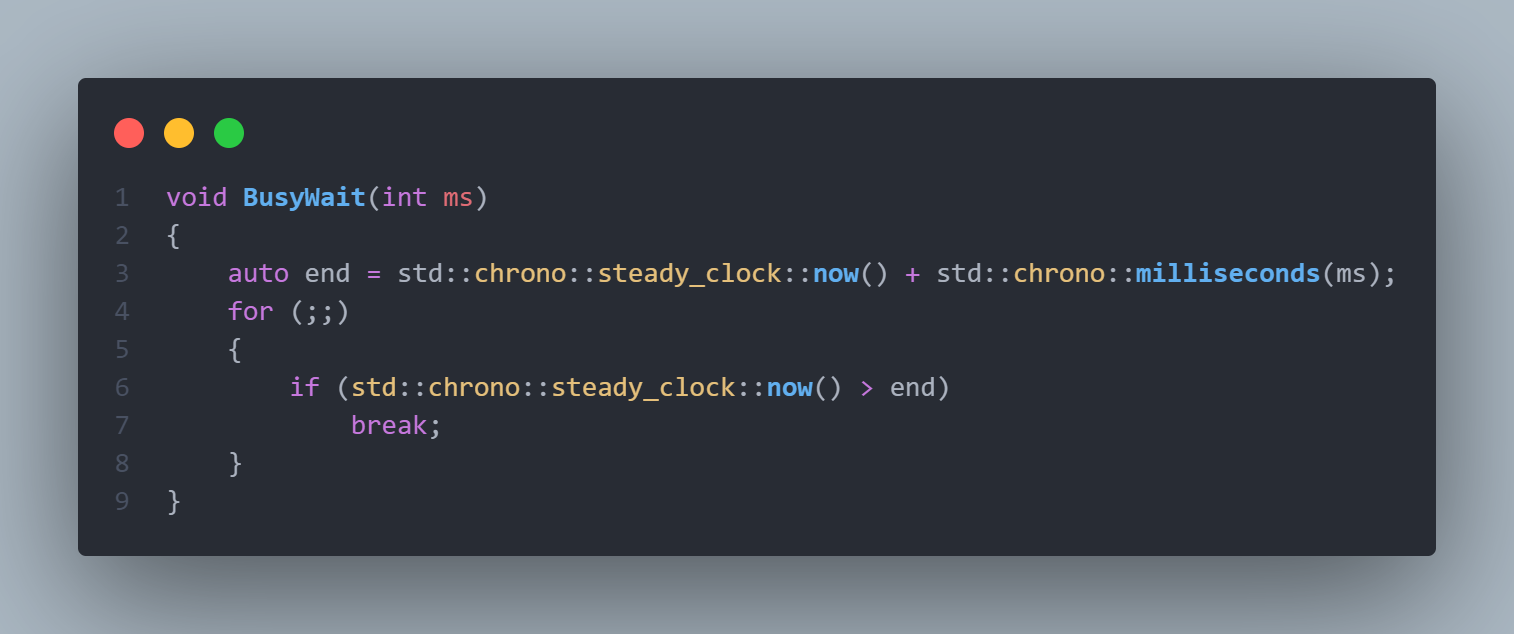
A classe **Timer** mede o tempo de execução. Quando um objeto Timer é instanciado, seu construtor armazena o tempo inicial (**start**). O método **GetElapsed** calcula a diferença entre o tempo atual e o tempo inicial, retornando a duração em segundos.



## 

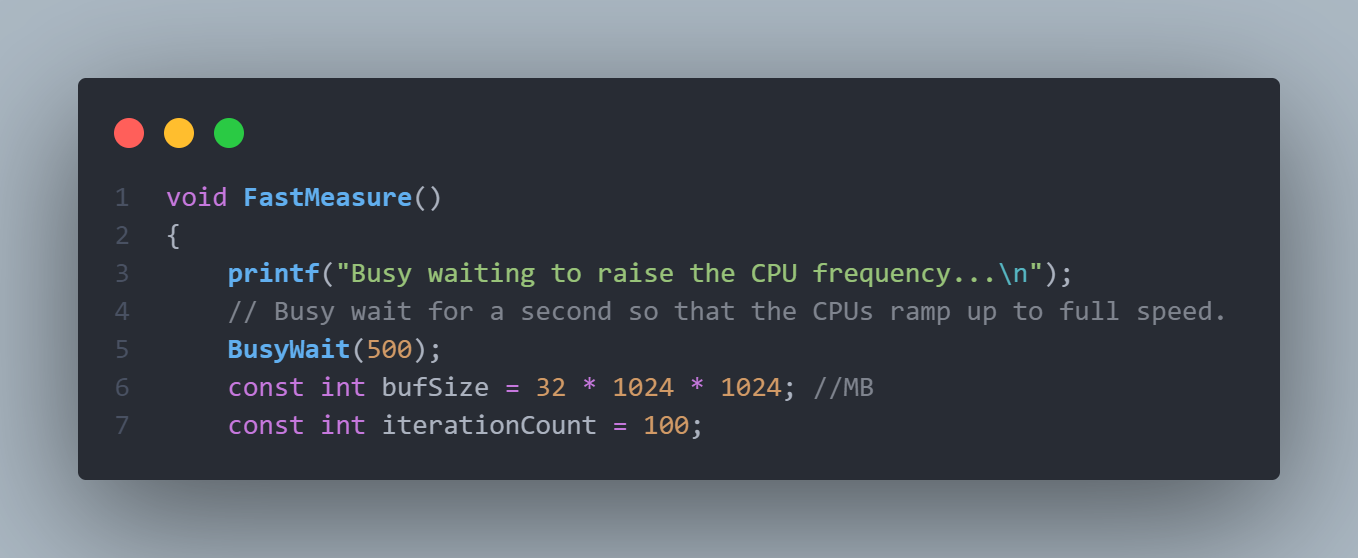
## 2.3. Função BusyWait:

A função **BusyWait** implementa uma espera ativa de milissegundos, onde o loop fica em execução até o tempo atual ultrapassar o valor **end** calculado. Este processo mantém a CPU ocupada durante o tempo especificado.



## 2.4. Função FastMeasure (parte 01):

Esta função mede o tempo de execução para várias operações de alocação, escrita e leitura de memória. Primeiro ela informa o início do teste com uma mensagem de espera ativa para aumentar a frequência da CPU. Chama **BusyWait(500)** para garantir que a CPU está no desempenho máximo, e define o tamanho da alocação de memória em MB.



## 2.5. Função FastMeasure (parte 02):

Testa a alocação e desalocação de um buffer de 32 MB (bufSize) repetidamente. Um **Timer** mede o tempo total para realizar **iterationCount** alocações e desalocações de um buffer. O resultado é o tempo médio para cada ciclo.

****

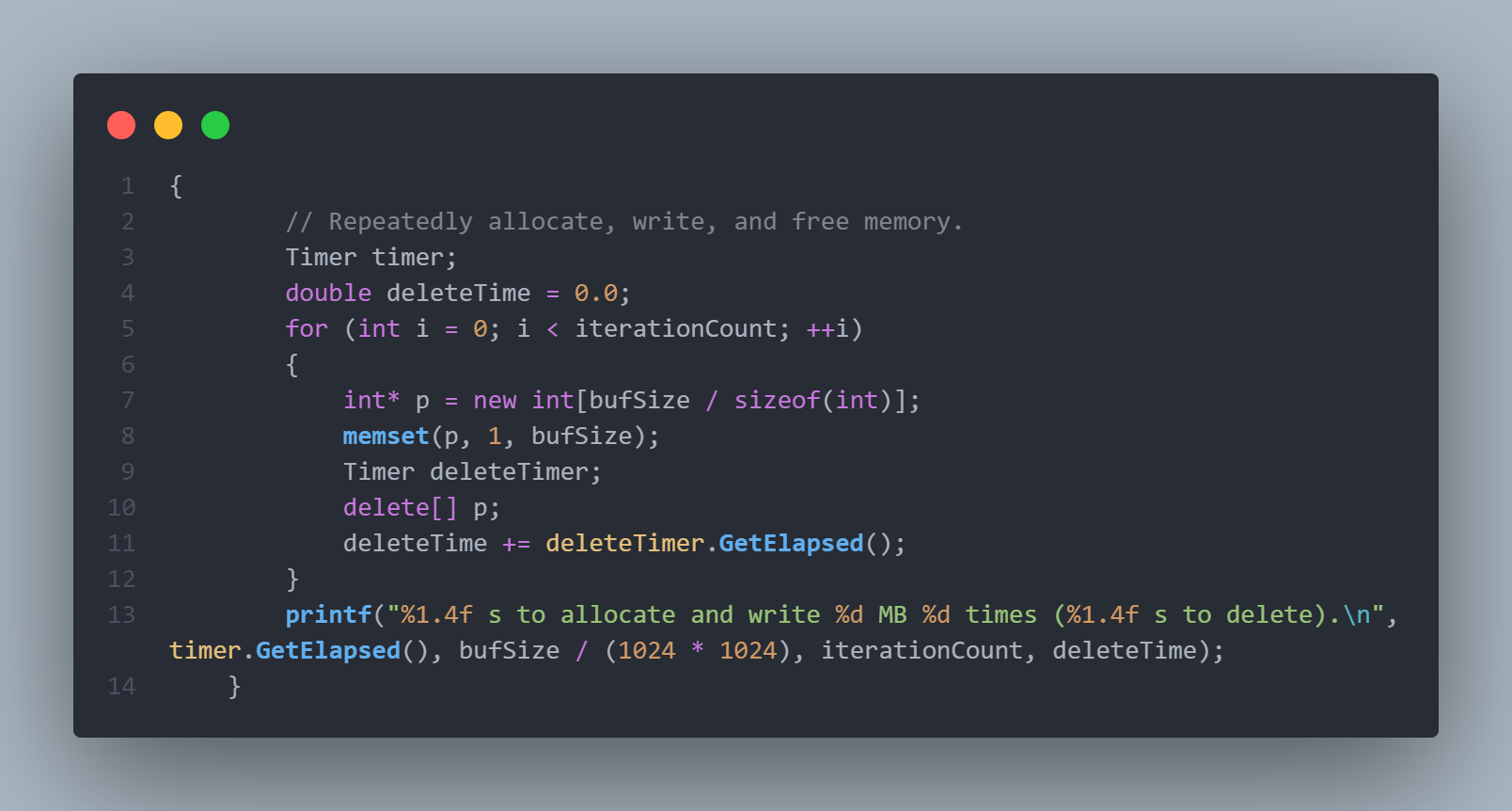
## 2.6. Função FastMeasure (parte 03):

Similar ao bloco anterior, mas mede separadamente o tempo de desalocação usando um **Timer** adicional (**deleteTimer**) em cada desalocação de **p**. O tempo acumulado para desalocação (**deleteTime**) é incluído na saída.



## 2.7. Função FastMeasure (parte 04):

Realiza operações de escrita em memória. O buffer de 32 MB é alocado uma vez e, em seguida, **memset** define o conteúdo do buffer para **1** em cada uma das **iterationCount** iterações. Um **Timer** mede o tempo total dessas operações de escrita.



**2.8. Função FastMeasure (parte 05):**

Testa operações de leitura. Um buffer é alocado e inicializado, e o código lê o conteúdo em cada iteração, acumulando a soma em **sum**. O tempo de leitura é medido com um **Timer**, e o resultado inclui o tempo total e a soma.



## 

## 2.9. Função Main:

A função principal (**main**) chama **FastMeasure** para iniciar as medições. A função **system("pause")** pausa a execução, para podermos analisar quantos **page faults** ocorreram sem que o processo seja encerrado.

## 

# 

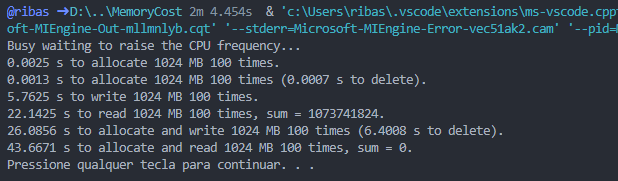
# 2.10. Resultados

## 2.10.1. Tabela com os resultados (Windows):

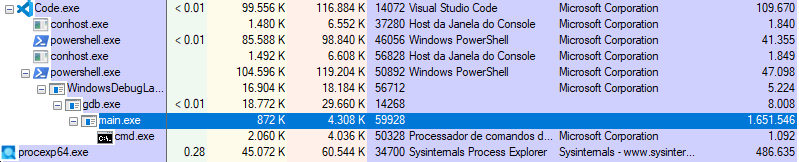
Abaixo, separamos os resultados obtidos em uma tabela para melhor visualização e análise.

| Alocação de Memória | Tempo de alocação (em segundos) | Tempo de Escrita (em segundos) | Tempo de Leitura (em segundos) | Tempo de Alocação e Escrita (em segundos) | Tempo de Alocação e Leitura (em segundos) | Page Faults |
| --- | --- | --- | --- | --- | --- | --- |
| 32MB | 0.0007 | 0.1781 | 0.6844 | 0.7601 | 1.2418 | 1.651.546 |
| 64MB | 0.0007 | 0.3592 | 1.3647 | 1.3645 | 2.5409 | 3.301.352 |
| 128MB | 0.0007 | 0.7129 | 2.7307 | 2.7163 | 5.2068 | 6.600.968 |
| 256MB | 0.0008 | 1.4510 | 5.4487 | 6.3393 | 9.0018 | 13.200.201 |
| 512MB | 0.0010 | 2.9479 | 11.2045 | 13.6353 | 21.6072 | 26.398.667 |
| 1024MB (1GB) | 0.0025 | 5.7625 | 22.1425 | 26.0856 | 43.6671 | 52.795.594 |

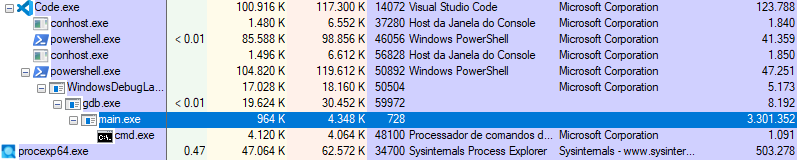
## 2.10.2. Execução do algoritmo:



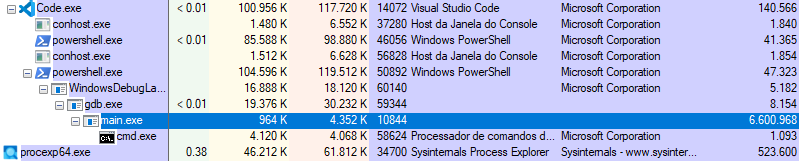
*Imagem: Terminal após executar o código MemoryCost alocando 1024MB.*

**

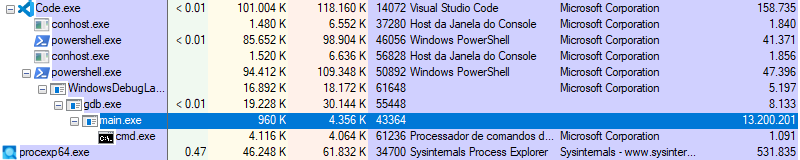
*Imagem: Page Faults no código Windows alocando 32MB*

**

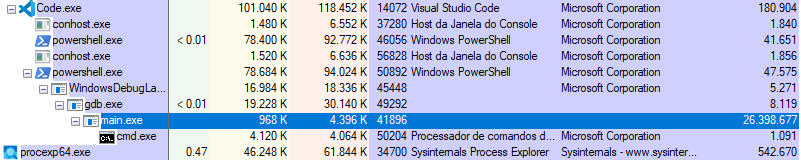
*Imagem: Page Faults no código Windows alocando 64*

**

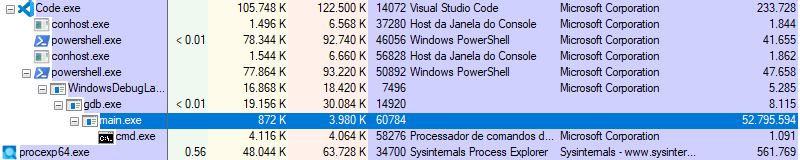
*Imagem: Page Faults no código Windows alocando 128MB*

**

*Imagem: Page Faults no código Windows alocando 256MB*

**

*Imagem: Page Faults no código Windows alocando 512MB*

**

*Imagem: Page Faults no código Windows alocando 1024MB*

## 2.10.3. Código no Windows:

Conforme vemos no gráfico, o número de page faults no Windows cresce de maneira linear com o aumento do tamanho do buffer. Esse comportamento é um reflexo direto da necessidade do sistema operacional de alocar mais páginas de memória para buffers maiores. Esse processo gera uma quantidade significativa de page faults quando a memória física se torna insuficiente, exigindo que o sistema passe a utilizar memória virtual.

Observamos que, por exemplo, ao aumentar o buffer de 32 MB para 1 GB, há um salto substancial no número de page faults, impactando diretamente o tempo de alocação e os tempos de escrita e leitura na memória. Esse aumento linear nos page faults indica que o Windows mantém uma relação proporcional entre a quantidade de memória requisitada e a quantidade de operações de swap necessárias, mas, ao mesmo tempo, sofre impacto na performance devido à dependência crescente de memória virtual. Para buffers acima de 512 MB, a pressão sobre o sistema de memória virtual se intensifica, gerando maiores tempos de espera e reduzindo a eficiência das operações de entrada e saída na RAM.

Essa tendência linear sugere que o sistema de gerenciamento de memória do Windows, embora previsível, apresenta limitações na otimização de uso de memória em alocações de larga escala, especialmente quando o tamanho do buffer ultrapassa a capacidade de RAM disponível.

## Gráfico

## 

## 

## 

## 

## 

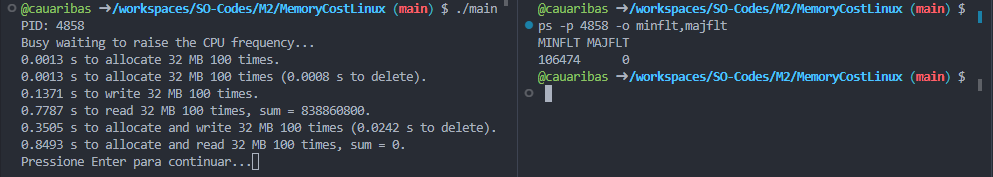
## 

## 2.10.4. Tabela com os resultados (Linux):

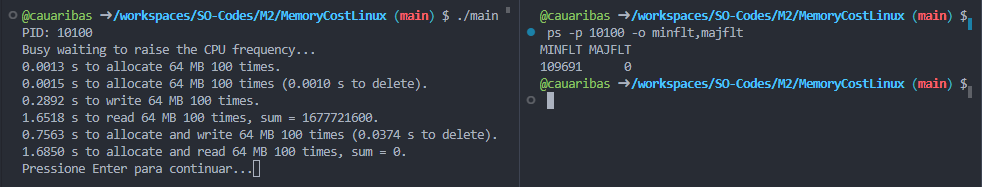
Abaixo, separamos os resultados obtidos em uma tabela para melhor visualização e análise.

| Alocação de Memória | Tempo de alocação (em segundos) | Tempo de Escrita (em segundos) | Tempo de Leitura (em segundos) | Tempo de Alocação e Escrita (em segundos) | Tempo de Alocação e Leitura (em segundos) | Page Faults |
| --- | --- | --- | --- | --- | --- | --- |
| 32MB | 0.0013 | 0.1371 | 0.7787 | 0.3505 | 0.8493 | 106.474 |
| 64MB | 0.0013 | 0.2892 | 1.6518 | 0.7563 | 1.6850 | 109.691 |
| 128MB | 0.0013 | 0.5314 | 3.1715 | 1.6337 | 3.4307 | 116.121 |
| 256MB | 0.0015 | 1.0955 | 6.3227 | 3.2983 | 6.4188 | 130.518 |
| 512MB | 0.0018 | 2.1940 | 12.6066 | 8.1927 | 12.8554 | 1.066.850 |
| 1024MB (1GB) | 0.0022 | 4.3532 | 25.1563 | 18.8936 | 25.9906 | 3.237.422 |

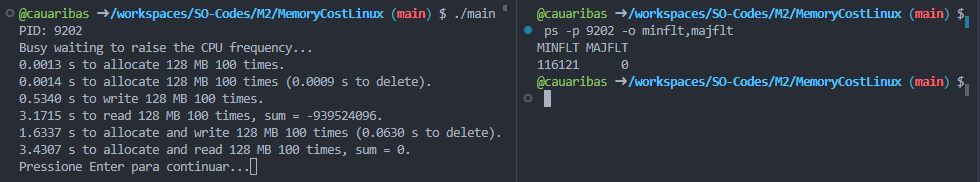
## 2.10.5. Execução do algoritmo:

****

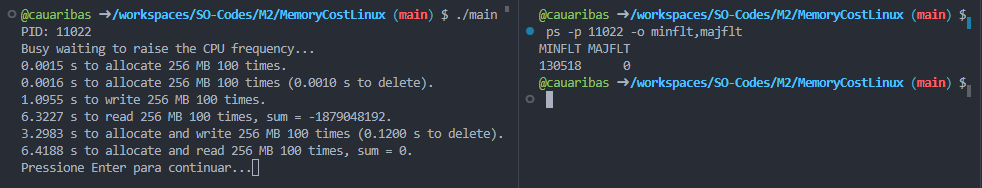
*Imagem: Terminal após executar o código MemoryCost alocando 32MB e o número de Page Faults.*

**

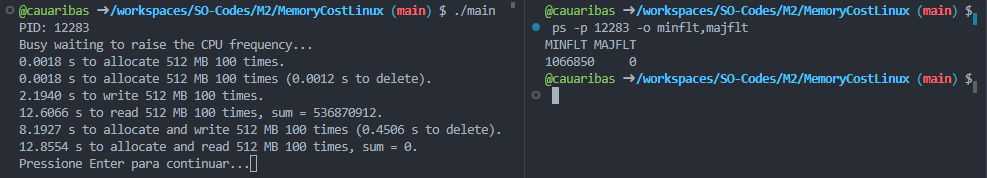
*Imagem: Terminal após executar o código MemoryCost alocando 64MB e o número de Page Faults.*

**

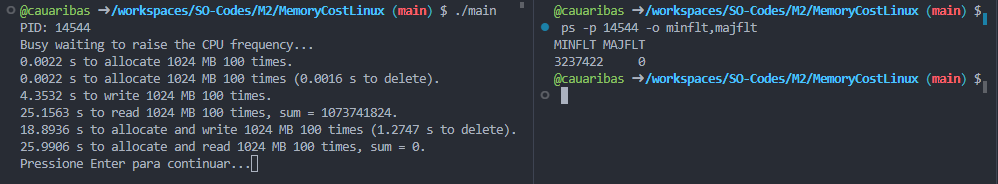
*Imagem: Terminal após executar o código MemoryCost alocando 128MB e o número de Page Faults.*

**

*Imagem: Terminal após executar o código MemoryCost alocando 256MB e o número de Page Faults.*

****

*Imagem: Terminal após executar o código MemoryCost alocando 512MB e o número de Page Faults.*

****

*Imagem: Terminal após executar o código MemoryCost alocando 1024MB e o número de Page Faults.*

## 

## 

## 

## 

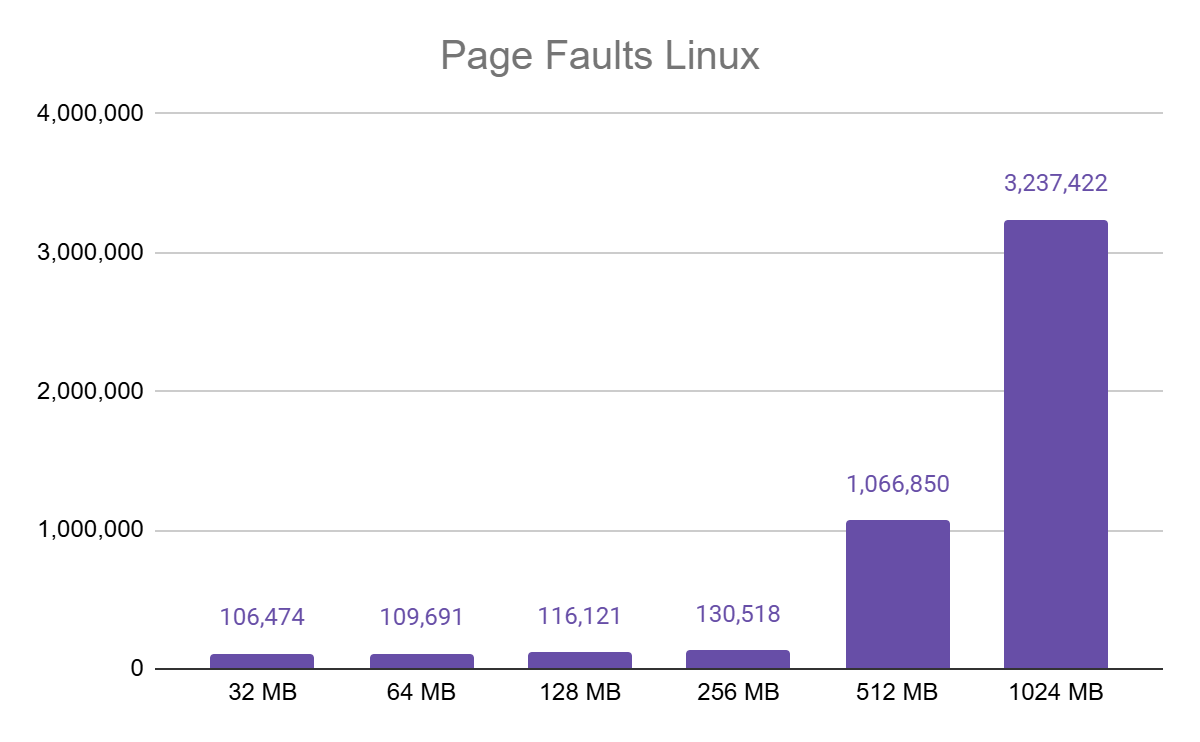
## 

## **2.10.6. Código no Linux:**

No caso do Linux, o comportamento dos page faults em relação ao tamanho do buffer é diferente e, em alguns aspectos, mais eficiente. Até o buffer de 256 MB, o número de page faults cresce apenas moderadamente, sugerindo que o Linux consegue gerenciar a memória de forma mais eficaz para alocações menores, possivelmente por meio de otimizações no cache ou outras estratégias de uso de páginas de memória.

Quando o buffer atinge 512 MB, ocorre um salto acentuado no número de page faults, refletindo uma dependência maior de memória virtual à medida que o sistema precisa lidar com volumes de dados que excedem a memória física. A partir de 1 GB, o aumento de page faults é exponencial em comparação com o Windows, o que indica que o ambiente Linux tenta manter a eficiência no uso de RAM até certo ponto, mas, quando a carga ultrapassa a capacidade física, o sistema passa a depender mais intensamente da memória virtual.

Esse comportamento diferenciado no Linux é ainda mais notável considerando as limitações do ambiente virtualizado do Codespace, que afetam a quantidade de RAM disponível. Essas limitações resultam em um aumento mais pronunciado nos page faults à medida que os buffers crescem, reforçando a ideia de que o Linux, embora mais eficiente em cargas menores, sofre com o uso intensivo de memória virtual em cenários de alto consumo de memória. Esse impacto na performance pode ser diretamente atribuído à capacidade de alocação e uso de memória em ambientes de desenvolvimento com restrições.



## 

## 

## 2.11. Conclusões

Os resultados indicam que ambos os sistemas operacionais são suscetíveis a sobrecargas de memória, mas lidam com essa situação de maneiras distintas. O Windows, com um crescimento linear de page faults, oferece previsibilidade, embora comprometa a eficiência com grandes alocações de memória. O Linux, por outro lado, consegue otimizar melhor o uso de memória para tamanhos menores, mas enfrenta dificuldades ao exceder a capacidade física de RAM, especialmente em ambientes com limitações, como o Codespace.

Essas observações destacam a importância de considerar o sistema operacional e o ambiente ao projetar e otimizar aplicações de alto consumo de memória. Em cargas leves a moderadas, o Linux pode oferecer um desempenho superior devido à sua eficiência na gestão de memória. Para cargas intensivas, ambos os sistemas enfrentam desafios, mas o comportamento previsível do Windows pode ser mais adequado, dependendo do tipo de aplicação e das exigências de desempenho.

# 3. Análise código M1

## 3.1. Introdução ao código:

Este código foi implementado como parte de um trabalho da M1, cujo objetivo era desenvolver um sistema multitarefa utilizando um pool de threads para manipulação de conexões de clientes através de sockets Unix. Ele foi projetado para processar tarefas de forma eficiente, distribuindo o trabalho entre várias threads para melhorar o desempenho e a escalabilidade do sistema.

## 3.2. Função executeTask:

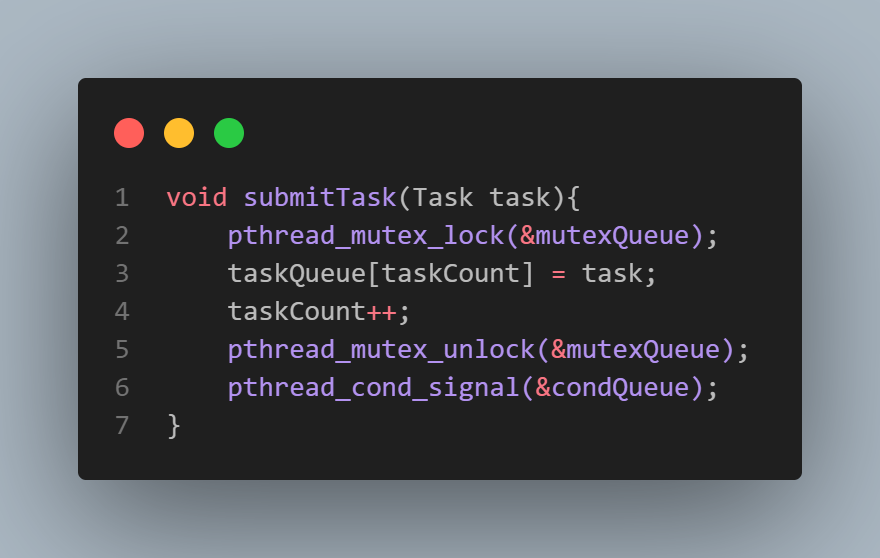
Esta função executa a função associada a uma tarefa, passando o descritor de socket do cliente como argumento. Dependendo da função de tarefa, ela pode manipular strings ou números.

****

## 

## **3.3. Função submitTask:**

Enfileira uma nova tarefa na fila de tarefas compartilhada entre as threads. A função usa um mutex para garantir que a fila seja acessada de forma segura por múltiplas threads, e depois sinaliza para as threads que há uma nova tarefa disponível.



## **3.4. Função startThread:**

Esta função define o comportamento de cada thread no pool. As threads ficam esperando novas tarefas e, quando uma tarefa é adicionada à fila, a thread a remove e a executa. A função utiliza um mutex para garantir acesso exclusivo à fila de tarefas e uma variável de condição para esperar até que novas tarefas estejam disponíveis.



## 3.5. Função startServer:

Esta função cria e inicializa um socket Unix para comunicação inter processual (IPC). Ele vincula o socket a um caminho específico no sistema de arquivos e coloca o socket em modo de escuta, pronto para aceitar conexões.



## 

## 3.6. Funções handleStringConnection e handleNumberConnection:

## 3.6.1. handleStringConnection:

Esta função lida com conexões de strings. Ela lê uma string do cliente, converte para maiusculas e envia de volta ao cliente antes de fechar o socket



## 

## 3.6.2. handleNumberConnection:

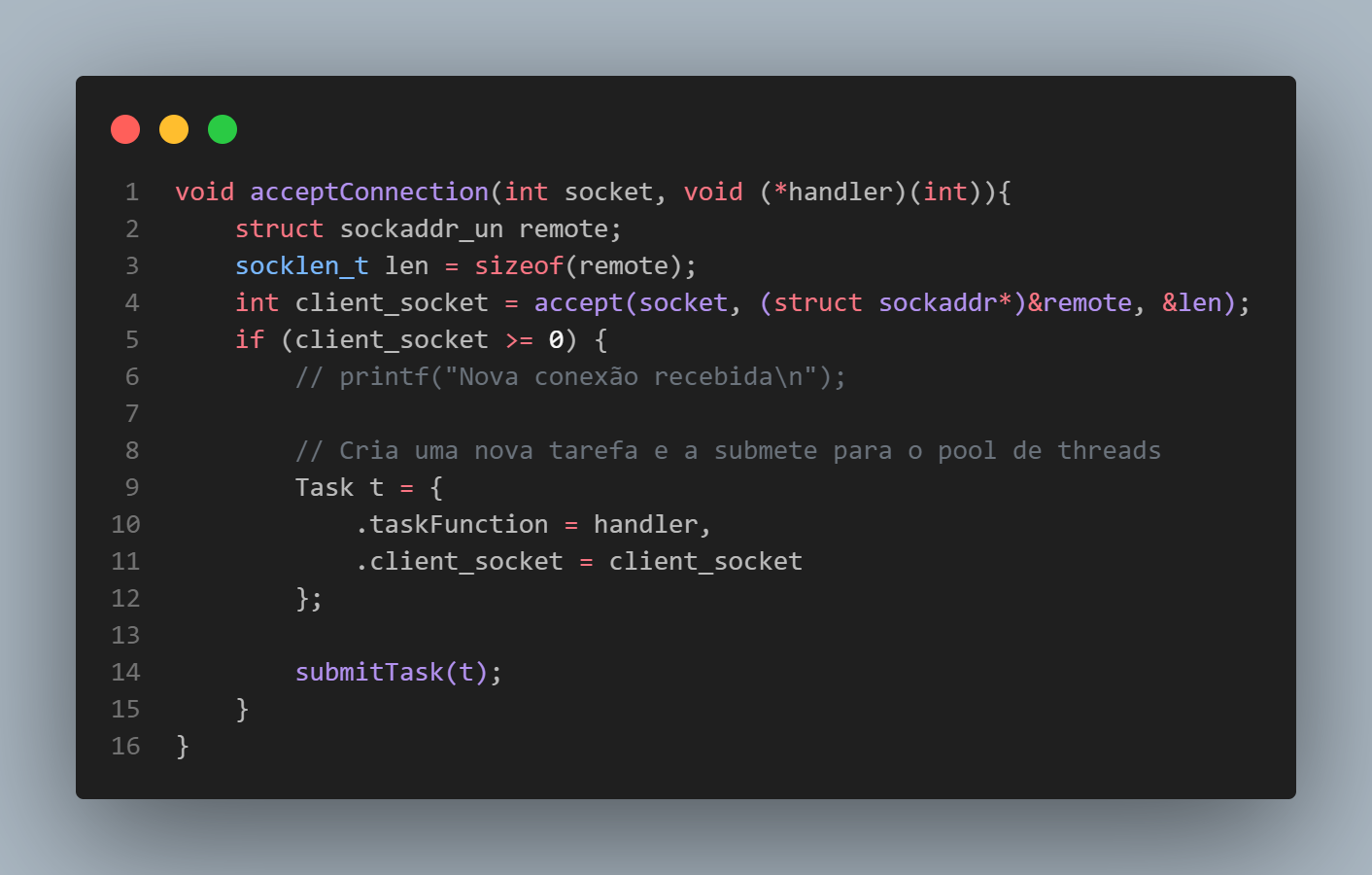
Esta função lida com conexões de números. Ela lê um número enviado pelo cliente, o multiplica por 2 e envia o resultado de volta ao cliente.

****

## 3.7. Função acceptConnection:

Esta função aceita conexões de clientes. Após aceitar a conexão, ela cria uma nova tarefa e a submete ao pool de threads. A tarefa utiliza a função apropriada (string ou número)

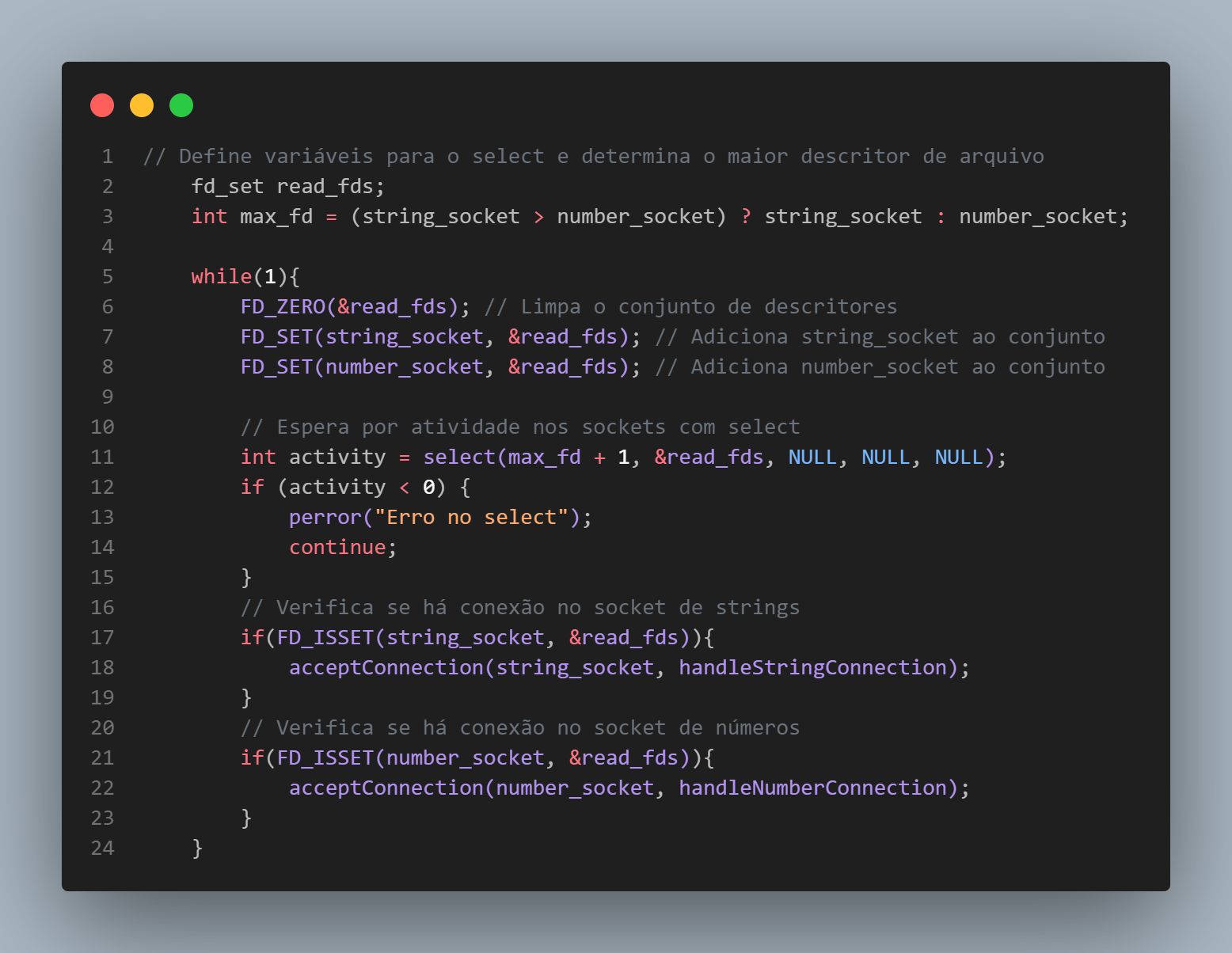
para processar a conexão.



## 

## **3.8. Função main:**

O **main** configura o pool de threads, cria os sockets para lidar com strings e números e entra em um loop infinito, utilizando **select** para monitorar os sockets. Dependendo de qual socket recebe uma conexão, ele chama a função apropriada **(handleStringConnection** ou **handleNumberConnection**).



## 3.9. Resultados

## 3.9.1. Código no Windows:

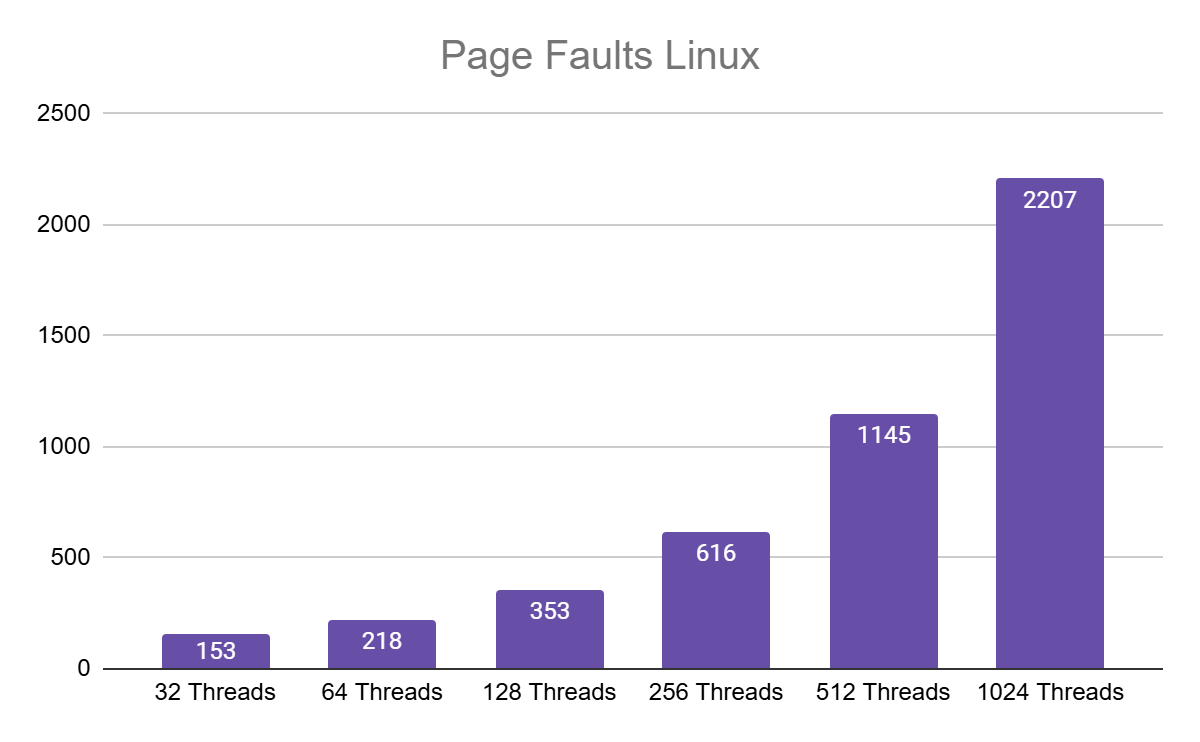
A análise dos dados de *page faults* no Windows para o código da M1 com diferentes quantidades de threads revela que o número de *page faults* aumenta progressivamente conforme o número de threads cresce. Com até 64 threads, o crescimento é moderado, mas ao atingir 256 ou mais, os *page faults* aumentam rapidamente, indicando que a sobrecarga de gerenciamento de memória se intensifica à medida que o sistema lida com mais threads simultâneas.

Esse aumento acentuado sugere que o Windows enfrenta limites de escalabilidade para cargas muito altas de threads, exigindo mais acessos à memória virtual e gerando latência adicional. Em resumo, o crescimento dos *page faults* ilustra o impacto negativo de cargas intensivas de threads no desempenho, com o sistema aproximando-se de seu limite de capacidade conforme o número de threads se eleva.

## Points scored**3.9.2. Código no Linux:**

A análise dos *page faults* para o código da M1 no Linux com um número crescente de threads revela um aumento gradual, mas controlado, no número de *page faults*. Em comparação ao Windows, o Linux apresenta uma gestão de memória mais eficiente para quantidades menores de threads, com *page faults* relativamente baixos até 128 threads. No entanto, à medida que o número de threads se aproxima de 512 e 1024, ocorre um crescimento mais acentuado, sugerindo que o sistema começa a depender mais da memória virtual.

Esse comportamento indica que, embora o Linux gerencie bem cargas moderadas de threads, ele também enfrenta aumento de pressão sobre a memória conforme a contagem de threads cresce significativamente, o que pode degradar o desempenho do sistema em cargas muito altas, exigindo mais operações de *swap* e acesso à memória virtual.



## 3.10. Conclusões

Em conclusão, a análise dos dados de *page faults* no Windows e no Linux para o código da M1 com diferentes quantidades de threads mostra comportamentos distintos na gestão de memória em ambos os sistemas operacionais. No Windows, o aumento de *page faults* é mais visível conforme o número de threads cresce, especialmente a partir de 256 threads. Isso sugere que o sistema enfrenta uma sobrecarga de gerenciamento de memória para altas cargas, exigindo maior uso de memória virtual e resultando em uma escalabilidade limitada e potencial degradação de desempenho.

Por outro lado, o Linux mostra uma gestão de *page faults* mais controlada, com aumento gradual até 128 threads. No entanto, ao se aproximar de 512 e 1024 threads, também ocorre um crescimento significativo nos *page faults*, sugerindo que o sistema começa a depender mais da memória virtual em cargas intensas. Assim, enquanto o Linux consegue manter uma eficiência maior em cargas moderadas, ambos os sistemas enfrentam desafios para manter o desempenho quando o número de threads ultrapassa suas capacidades de memória física, exigindo acesso intensivo à memória virtual.

# 4. Análise código reordenação de matrizes

## 4.1. Introdução aos códigos:

Este código realiza operações pesadas em uma matriz 2D de grande dimensão (10.000 x 10.000), exigindo significativa capacidade computacional e memória devido ao tamanho da estrutura e à natureza das operações. Ele preenche a matriz com valores incrementais, converte-a em um array unidimensional para ordenação decrescente, e então reconstroi a matriz.

Este código é altamente intensivo em recursos, demandando tanto CPU para o processamento dos dados quanto memória principal para suportar a grande estrutura da matriz e as operações de manipulação.

## 4.2. Código Fonte (Python):



## 4.3. Código Fonte (C#):



## 4.4. Análise (Python):

* **Windows**: 3.007.133 page faults
* **Linux (Codespace)**: 815.616 page faults

O número de page faults no ambiente Windows é significativamente maior do que no Linux (aproximadamente 3,7 vezes mais). Isso sugere que o gerenciamento de memória para o Python em Windows enfrenta maior pressão na memória virtual ao lidar com uma estrutura tão grande como a matriz 10.000 x 10.000.

Em sistemas Windows, Python parece gerar mais page faults quando lida com operações intensas de manipulação de dados em grandes matrizes, o que pode ser causado pela forma como o sistema operacional gerencia a memória de processos intensivos.

No Linux (Codespace), mesmo com as limitações de memória do ambiente de desenvolvimento como o Codespace, o número de page faults foi significativamente menor. Isso indica que o Linux lida de forma mais eficiente com a alocação de memória para operações intensivas em Python, possivelmente devido a uma abordagem de paginação de memória mais eficiente e um cache mais robusto para operações de manipulação de grandes arrays.

## 4.5. Análise (C#):

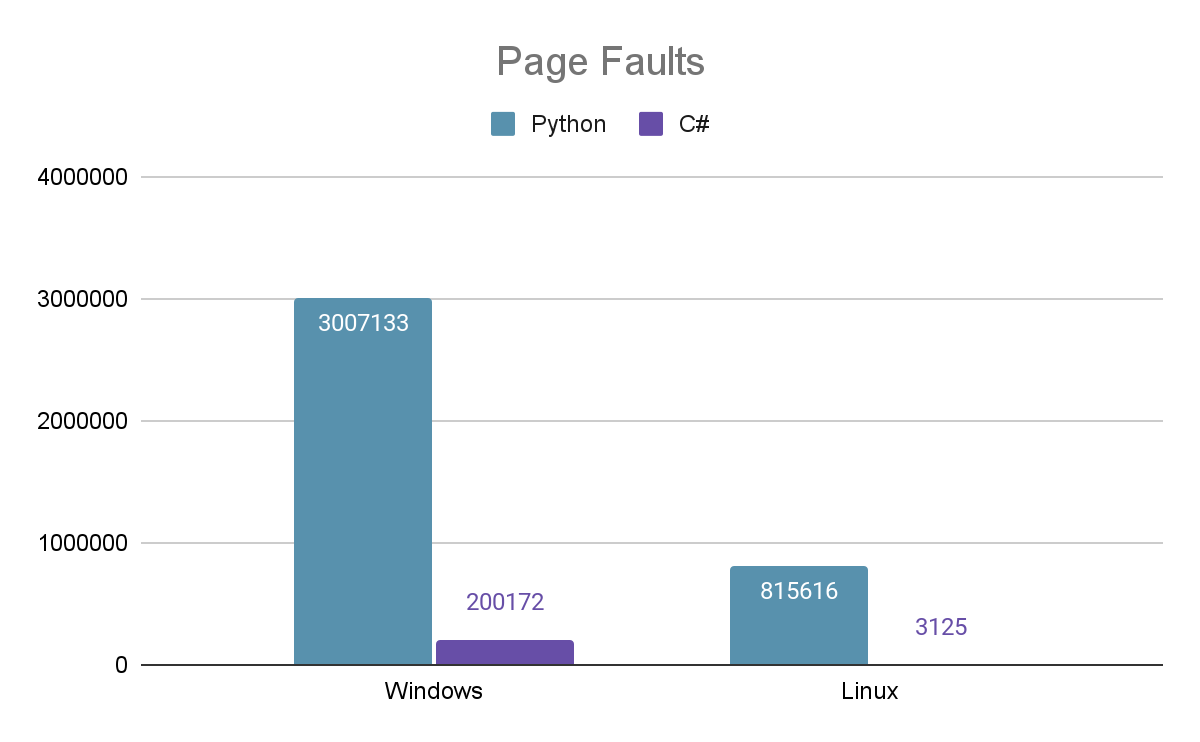
* **Windows**: 200.172 page faults
* **Linux (Codespace)**: 3.125 page faults

No caso da execução em C#, o número de page faults é muito menor tanto no Windows quanto no Linux em comparação com Python. No Windows, foram 200.172 page faults, enquanto no Linux foi ainda mais baixo, com apenas 3.125 page faults.

A diferença entre Windows e Linux é ainda mais pronunciada aqui, com Windows registrando um valor aproximadamente 64 vezes maior que o Linux. Isso sugere que o ambiente .NET no Windows, apesar de mais otimizado para C#, ainda não lida com grandes operações de reordenação de dados tão eficientemente quanto o Linux para esse caso específico.

No Linux (Codespace), o baixo número de page faults (3.125) reflete um uso eficiente da memória para a tarefa de reordenação. Isso pode ser atribuído ao fato de que o Linux gerencia memória de forma superior, minimizando page faults através de uma alocação eficiente de memória e de um sistema de cache.

## 4.6. Gráfico para comparação:



## 4.7. Conclusões:

Em conclusão, os resultados obtidos indicam que o Linux lida de maneira mais eficiente com grandes operações de memória em ambos os casos, especialmente devido ao sistema de gerenciamento de memória mais otimizado para cargas de trabalho intensivas e repetitivas.

Python tende a causar mais page faults que C# ao lidar com grandes volumes de dados, possivelmente porque a linguagem é interpretada e a forma como o Python gerencia grandes estruturas de dados, como matrizes, exige mais swaps de memória em comparação com C#.

C# em Linux foi a configuração mais eficiente em termos de page faults, sugerindo que o uso de C# em um ambiente Linux/Codespace pode ser mais apropriado para tarefas intensivas de processamento e reordenação de dados devido ao melhor gerenciamento de memória.

# 5. Conclusão Geral

Este relatório apresentou uma análise detalhada dos page faults em sistemas operacionais Windows e Linux, explorando o impacto do gerenciamento de memória em diferentes cenários e tamanhos de alocação. Os testes realizados com diferentes algoritmos evidenciam como cada sistema lida com o crescimento do uso de memória e revela aspectos importantes sobre eficiência e limitações no gerenciamento de memória em cada ambiente.

No Windows, o crescimento linear dos page faults com o aumento do buffer mostrou-se previsível, mas com um custo direto na performance em situações de alta demanda de memória, evidenciando uma dependência de memória virtual que pode impactar negativamente a eficiência em grandes alocações.

O Linux, por outro lado, demonstrou maior controle sobre page faults para buffers menores, sugerindo uma otimização mais eficiente da RAM. No entanto, essa vantagem foi reduzida em buffers grandes, onde o uso intensivo de memória virtual levou a um crescimento exponencial de page faults, especialmente devido às limitações do ambiente Codespace.

A análise comparativa também demonstrou como diferentes linguagens e configurações impactam o número de page faults, evidenciando que escolhas de linguagem, ambiente e sistema operacional podem afetar o desempenho de aplicações de alto consumo de memória. Enquanto o Windows mostrou uma escalabilidade mais previsível em grandes cargas, o Linux teve um desempenho mais vantajoso para operações moderadas.

# 6. Referências

1. Felipe Viel. SO-Codes: MemoryCost. Disponível em:<https://github.com/VielF/SO-Codes/tree/main/Memory/MemoryCost>. Acesso em: 10 nov. 2024.
2. MICROSOFT. Process Explorer. Disponível em:<https://learn.microsoft.com/pt-br/sysinternals/downloads/process-explorer>. Acesso em: 10 nov. 2024.
3. RED HAT. Linux commands: vmstat. Disponível em:<https://www.redhat.com/en/blog/linux-commands-vmstat#:~:text=Virtual%20memory%20statistics%20reporter%2C%20also,the%20array%20of%20information%20provided>. Acesso em: 11 nov. 2024.
4. PHOENIX NAP. Linux Commands to Check Memory Usage. Disponível em:<https://phoenixnap.com/kb/linux-commands-check-memory-usage>. Acesso em: 11 nov. 2024.
5. CYBERCITI. Linux Command to See Major and Minor Page Faults. Disponível em:<https://www.cyberciti.biz/faq/linux-command-to-see-major-minor-pagefaults/#google_vignette>. Acesso em: 11 nov. 2024.