

**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:**

<b>1. Introdução e Contexto:</b>	<b>3</b>
<b>2. Análise do algoritmo “Memory Cost”</b>	<b>3</b>
2.1. Introdução ao código:	3
2.2. Classe Timer:	3
2.3. Função BusyWait:	4
2.4. Função FastMeasure (parte 01):	4
2.5. Função FastMeasure (parte 02):	5
2.6. Função FastMeasure (parte 03):	6
2.7. Função FastMeasure (parte 04):	6
2.9. Função Main:	7
<b>2.10. Resultados</b>	<b>8</b>
2.10.1. Tabela com os resultados (Windows):	8
2.10.2. Execução do algoritmo:	8
2.10.3. Código no Windows:	10
2.10.4. Tabela com os resultados (Linux):	11
2.10.5. Execução do algoritmo:	11
2.10.6. Código no Linux:	13
2.11. Conclusões	14
<b>3. Análise código M1</b>	<b>14</b>
3.1. Introdução ao código:	14
3.2. Função executeTask:	14
3.3. Função submitTask:	15
3.4. Função startThread:	15
3.5. Função startServer:	16
3.6. Funções handleStringConnection e handleNumberConnection:	17
3.6.1. handleStringConnection:	17
3.6.2. handleNumberConnection:	18
3.7. Função acceptConnection:	19
3.8. Função main:	20
3.9. Resultados	20
3.9.1. Código no Windows:	20
3.9.2. Código no Linux:	21
3.10. Conclusões	22
<b>4. Análise código reordenação de matrizes</b>	<b>22</b>
4.1. Introdução aos códigos:	22
4.2. Código Fonte (Python):	23
4.3. Código Fonte (C#):	24
4.4. Análise (Python):	25
4.5. Análise (C#):	25
4.6. Gráfico para comparação:	26
4.7. Conclusões:	26
<b>5. Conclusão Geral</b>	<b>26</b>
<b>6. Referências</b>	<b>27</b>

## 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.

```
1  class Timer
2  {
3  public:
4      Timer()
5      {
6          start = clock.now();
7      }
8      // Returns the duration in seconds.
9      double GetElapsed()
10     {
11         auto end = clock.now();
12         auto duration = end - start;
13         return std::chrono::duration_cast<std::chrono::nanoseconds>(duration).count() * 1.e-9;
14     }
15
16 private:
17     std::chrono::steady_clock::time_point start;
18     std::chrono::steady_clock clock;
19
20     Timer(const Timer &) = delete;
21     Timer operator=(const Timer *) = delete;
22 };
```

### 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.

```
1 void BusyWait(int ms)
2 {
3     auto end = std::chrono::steady_clock::now() + std::chrono::milliseconds(ms);
4     for (;;)
5     {
6         if (std::chrono::steady_clock::now() > end)
7             break;
8     }
9 }
```

### 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.

```
1 void FastMeasure()
2 {
3     printf("Busy waiting to raise the CPU frequency...\n");
4     // Busy wait for a second so that the CPUs ramp up to full speed.
5     BusyWait(500);
6     const int bufSize = 32 * 1024 * 1024; //MB
7     const int iterationCount = 100;
```

## 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.

```
1 {
2     Timer timer;
3     for (int i = 0; i < iterationCount; ++i)
4     {
5         int* p = new int[bufSize / sizeof(int)];
6         delete[] p;
7     }
8     printf("%1.4f s to allocate %d MB %d times.\n", timer.GetElapsed(), bufSize / (1024 *
1024), iterationCount);
9 }
10 {
11     Timer timer;
12     double deleteTime = 0.0;
13     for (int i = 0; i < iterationCount; ++i)
14     {
15         int* p = new int[bufSize / sizeof(int)];
16         Timer deleteTimer;
17         delete[] p;
18         deleteTime += deleteTimer.GetElapsed();
19     }
20     printf("%1.4f s to allocate %d MB %d times (%1.4f s to delete).\n", timer.GetElapsed
21     (), bufSize / (1024 * 1024), iterationCount, deleteTime);
22 }
```

## 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.

```
1 {
2     // Initialize and zero the memory.
3     int* p = new int[bufSize / sizeof(int)]();
4     {
5         // Repeatedly write to the already allocated memory.
6         Timer timer;
7         for (int i = 0; i < iterationCount; ++i)
8         {
9             memset(p, 1, bufSize);
10        }
11        printf("%.4f s to write %d MB %d times.\n", timer.GetElapsed(), bufSize / (1024 *
12        1024), iterationCount);
13    }
14    {
15        // Repeatedly read from the already allocated memory.
16        Timer timer;
17        int sum = 0;
18        for (int i = 0; i < iterationCount; ++i)
19        {
20            for (size_t index = 0; index < bufSize / sizeof(int); ++index)
21            {
22                sum += p[index];
23            }
24            printf("%.4f s to read %d MB %d times, sum = %d.\n", timer.GetElapsed(), bufSize /
25            (1024 * 1024), iterationCount, sum);
26        }
27        delete[] p;
28    }
```

## 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.

```
1 {
2     // Repeatedly allocate, write, and free memory.
3     Timer timer;
4     double deleteTime = 0.0;
5     for (int i = 0; i < iterationCount; ++i)
6     {
7         int* p = new int[bufSize / sizeof(int)];
8         memset(p, 1, bufSize);
9         Timer deleteTimer;
10        delete[] p;
11        deleteTime += deleteTimer.GetElapsed();
12    }
13    printf("%.4f s to allocate and write %d MB %d times (%.4f s to delete).\n",
14    timer.GetElapsed(), bufSize / (1024 * 1024), iterationCount, deleteTime);
15 }
```

## 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.

```
1  {
2      // Repeatedly allocate, read, and free memory.
3      Timer timer;
4      int sum = 0;
5      for (int i = 0; i < iterationCount; ++i)
6      {
7          int* p = new int[bufSize / sizeof(int)];
8          for (size_t index = 0; index < bufSize / sizeof(int); ++index)
9          {
10             sum += p[index];
11          }
12          delete[] p;
13      }
14      printf("%1.4f s to allocate and read %d MB %d times, sum = %d.\n",
15             timer.GetElapsed(), bufSize / (1024 * 1024), iterationCount, sum);
16 }
```

## 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.

```
1  int main(int argc, char* argv[])
2  {
3      FastMeasure();
4
5      std::cout << "Pressione Enter para continuar...";
6      std::cin.get();
7
8      return 0;
9  }
```

## 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:

```
@ribas →D:\..\MemoryCost 2m 4.454s & 'c:\Users\ribas\.vscode\extensions\ms-vscode.cpp-  
oft-MIEngine-Out-mlmnyb.cqt' '--stderr=Microsoft-MIEngine-Error-vec51ak2.cam' '--pid=  
Busy waiting to raise the CPU frequency...  
0.0025 s to allocate 1024 MB 100 times.  
0.0013 s to allocate 1024 MB 100 times (0.0007 s to delete).  
5.7625 s to write 1024 MB 100 times.  
22.1425 s to read 1024 MB 100 times, sum = 1073741824.  
26.0856 s to allocate and write 1024 MB 100 times (6.4008 s to delete).  
43.6671 s to allocate and read 1024 MB 100 times, sum = 0.  
Pressione qualquer tecla para continuar. . .
```

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

Code.exe	< 0.01	99.556 K	116.884 K	14072 Visual Studio Code	Microsoft Corporation	109.670
conhost.exe		1.480 K	6.552 K	37280 Host da Janela do Console	Microsoft Corporation	1.840
powershell.exe	< 0.01	85.588 K	98.840 K	46056 Windows PowerShell	Microsoft Corporation	41.355
conhost.exe		1.492 K	6.608 K	56828 Host da Janela do Console	Microsoft Corporation	1.849
powershell.exe		104.596 K	119.204 K	50892 Windows PowerShell	Microsoft Corporation	47.098
WindowsDebugLa...		16.904 K	18.184 K	56712	Microsoft Corporation	5.224
gdb.exe	< 0.01	18.772 K	29.660 K	14268		8.008
main.exe		872 K	4.308 K	59928		1.651.546
cmd.exe		2.060 K	4.036 K	50328 Processador de comandos d...	Microsoft Corporation	1.092
procexp64.exe	0.28	45.072 K	60.544 K	34700 Sysinternals Process Explorer	Sysinternals - www.sysinter...	486.635

Imagem: Page Faults no código Windows alocando 32MB



Code.exe		100.916 K	117.300 K	14072 Visual Studio Code	Microsoft Corporation	123.788
conhost.exe		1.480 K	6.552 K	37280 Host da Janela do Console	Microsoft Corporation	1.840
powershell.exe	< 0.01	85.588 K	98.856 K	46056 Windows PowerShell	Microsoft Corporation	41.359
conhost.exe		1.496 K	6.612 K	56828 Host da Janela do Console	Microsoft Corporation	1.850
powershell.exe		104.820 K	119.612 K	50892 Windows PowerShell	Microsoft Corporation	47.251
WindowsDebugLa...		17.028 K	18.160 K	50504	Microsoft Corporation	5.173
gdb.exe	< 0.01	19.624 K	30.452 K	59972		8.192
main.exe		964 K	4.348 K	728		3.301.352
cmd.exe		4.120 K	4.064 K	48100 Processador de comandos d...	Microsoft Corporation	1.091
procexp64.exe	0.47	47.064 K	62.572 K	34700 Sysinternals Process Explorer	Sysinternals - www.sysinter...	503.278

Imagem: Page Faults no código Windows alocando 64

Code.exe	< 0.01	100.956 K	117.720 K	14072 Visual Studio Code	Microsoft Corporation	140.566
conhost.exe		1.480 K	6.552 K	37280 Host da Janela do Console	Microsoft Corporation	1.840
powershell.exe	< 0.01	85.588 K	98.880 K	46056 Windows PowerShell	Microsoft Corporation	41.365
conhost.exe		1.512 K	6.628 K	56828 Host da Janela do Console	Microsoft Corporation	1.854
powershell.exe		104.596 K	119.512 K	50892 Windows PowerShell	Microsoft Corporation	47.323
WindowsDebugLa...		16.888 K	18.120 K	60140	Microsoft Corporation	5.182
gdb.exe	< 0.01	19.376 K	30.232 K	59344		8.154
main.exe		964 K	4.352 K	10844		6.600.968
cmd.exe		4.120 K	4.068 K	58624 Processador de comandos d...	Microsoft Corporation	1.093
procexp64.exe	0.38	46.212 K	61.812 K	34700 Sysinternals Process Explorer	Sysinternals - www.sysinter...	523.600

Imagem: Page Faults no código Windows alocando 128MB

Code.exe	< 0.01	101.004 K	118.160 K	14072 Visual Studio Code	Microsoft Corporation	158.735
conhost.exe		1.480 K	6.552 K	37280 Host da Janela do Console	Microsoft Corporation	1.840
powershell.exe	< 0.01	85.652 K	98.904 K	46056 Windows PowerShell	Microsoft Corporation	41.371
conhost.exe		1.520 K	6.636 K	56828 Host da Janela do Console	Microsoft Corporation	1.856
powershell.exe		94.412 K	109.348 K	50892 Windows PowerShell	Microsoft Corporation	47.396
WindowsDebugLa...		16.892 K	18.172 K	61648	Microsoft Corporation	5.197
gdb.exe	< 0.01	19.228 K	30.144 K	55448		8.133
main.exe		960 K	4.356 K	43364		13.200.201
cmd.exe		4.116 K	4.064 K	61236 Processador de comandos d...	Microsoft Corporation	1.091
procexp64.exe	0.47	46.248 K	61.832 K	34700 Sysinternals Process Explorer	Sysinternals - www.sysinter...	531.835

Imagem: Page Faults no código Windows alocando 256MB

Code.exe		101.040 K	118.452 K	14072 Visual Studio Code	Microsoft Corporation	180.904
conhost.exe		1.480 K	6.552 K	37280 Host da Janela do Console	Microsoft Corporation	1.840
powershell.exe	< 0.01	78.400 K	92.772 K	46056 Windows PowerShell	Microsoft Corporation	41.651
conhost.exe		1.520 K	6.636 K	56828 Host da Janela do Console	Microsoft Corporation	1.856
powershell.exe		78.684 K	94.024 K	50892 Windows PowerShell	Microsoft Corporation	47.575
WindowsDebugLa...		16.984 K	18.336 K	45448	Microsoft Corporation	5.271
gdb.exe	< 0.01	19.228 K	30.140 K	49292		8.119
main.exe		968 K	4.396 K	41896		26.398.677
cmd.exe		4.120 K	4.064 K	50204 Processador de comandos d...	Microsoft Corporation	1.091
procexp64.exe	0.47	46.248 K	61.844 K	34700 Sysinternals Process Explorer	Sysinternals - www.sysinter...	542.670

Imagem: Page Faults no código Windows alocando 512MB

Code.exe		105.748 K	122.500 K	14072 Visual Studio Code	Microsoft Corporation	233.728
conhost.exe		1.496 K	6.568 K	37280 Host da Janela do Console	Microsoft Corporation	1.844
powershell.exe	< 0.01	78.344 K	92.740 K	46056 Windows PowerShell	Microsoft Corporation	41.655
conhost.exe		1.544 K	6.660 K	56828 Host da Janela do Console	Microsoft Corporation	1.862
powershell.exe		77.864 K	93.220 K	50892 Windows PowerShell	Microsoft Corporation	47.658
WindowsDebugLa...		16.868 K	18.420 K	7496	Microsoft Corporation	5.285
gdb.exe	< 0.01	19.156 K	30.084 K	14920		8.115
main.exe		872 K	3.980 K	60784		52.795.594
cmd.exe		4.116 K	4.064 K	58276 Processador de comandos d...	Microsoft Corporation	1.091
procexp64.exe	0.56	48.044 K	63.728 K	34700 Sysinternals Process Explorer	Sysinternals - www.sysinter...	561.769

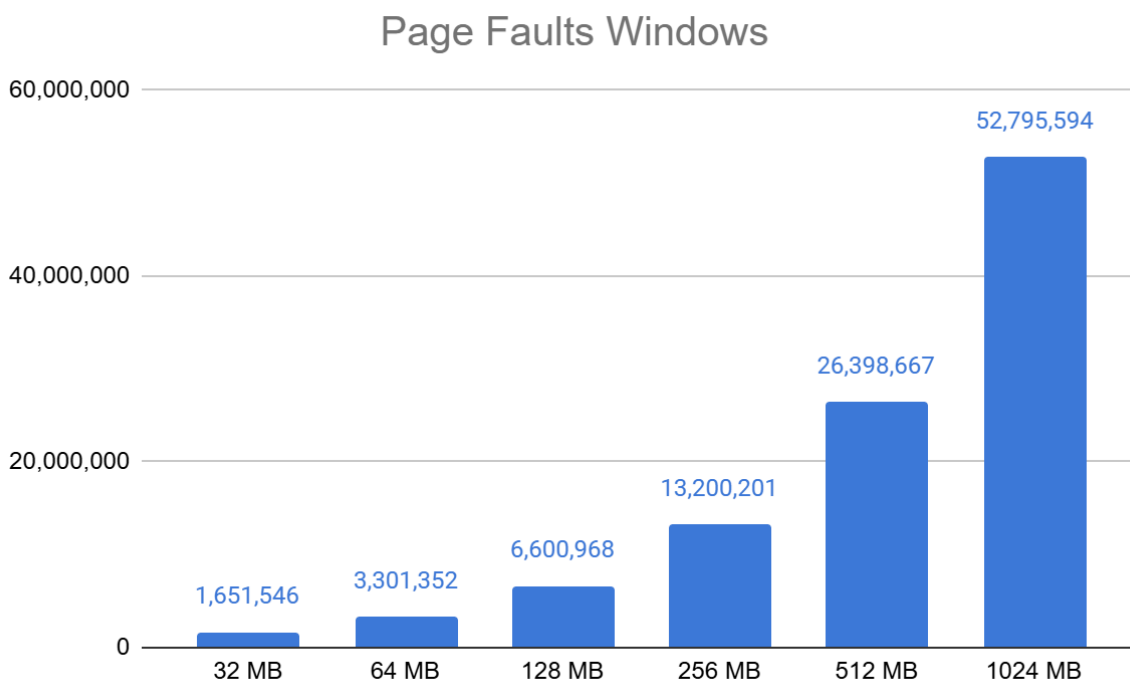
Imagem: Page Faults no código Windows alocando 1024MB

### 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.



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:

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $ ./main
PID: 4858
Busy waiting to raise the CPU frequency...
0.0013 s to allocate 32 MB 100 times.
0.0013 s to allocate 32 MB 100 times (0.0008 s to delete).
0.1371 s to write 32 MB 100 times.
0.7787 s to read 32 MB 100 times, sum = 838860800.
0.3505 s to allocate and write 32 MB 100 times (0.0242 s to delete).
0.8493 s to allocate and read 32 MB 100 times, sum = 0.
Pressione Enter para continuar...[]
```

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
ps -p 4858 -o minflt,majflt
MINFLT MAJFLT
106474      0
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
```

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

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $ ./main
PID: 10100
Busy waiting to raise the CPU frequency...
0.0013 s to allocate 64 MB 100 times.
0.0015 s to allocate 64 MB 100 times (0.0010 s to delete).
0.2892 s to write 64 MB 100 times.
1.6518 s to read 64 MB 100 times, sum = 1677721600.
0.7563 s to allocate and write 64 MB 100 times (0.0374 s to delete).
1.6850 s to allocate and read 64 MB 100 times, sum = 0.
Pressione Enter para continuar...[]
```

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
ps -p 10100 -o minflt,majflt
MINFLT MAJFLT
109691      0
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
```

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

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $ ./main
PID: 9202
Busy waiting to raise the CPU frequency...
0.0013 s to allocate 128 MB 100 times.
0.0014 s to allocate 128 MB 100 times (0.0009 s to delete).
0.5340 s to write 128 MB 100 times.
3.1715 s to read 128 MB 100 times, sum = -939524096.
1.6337 s to allocate and write 128 MB 100 times (0.0630 s to delete).
3.4307 s to allocate and read 128 MB 100 times, sum = 0.
Pressione Enter para continuar...

@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
● ps -p 9202 -o minflt,majflt
MINFLT MAJFLT
116121 0
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
```

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

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $ ./main
PID: 11022
Busy waiting to raise the CPU frequency...
0.0015 s to allocate 256 MB 100 times.
0.0016 s to allocate 256 MB 100 times (0.0010 s to delete).
1.0955 s to write 256 MB 100 times.
6.3227 s to read 256 MB 100 times, sum = -1879048192.
3.2983 s to allocate and write 256 MB 100 times (0.1200 s to delete).
6.4188 s to allocate and read 256 MB 100 times, sum = 0.
Pressione Enter para continuar...

@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
● ps -p 11022 -o minflt,majflt
MINFLT MAJFLT
130518 0
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
```

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

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $ ./main
PID: 12283
Busy waiting to raise the CPU frequency...
0.0018 s to allocate 512 MB 100 times.
0.0018 s to allocate 512 MB 100 times (0.0012 s to delete).
2.1940 s to write 512 MB 100 times.
12.6066 s to read 512 MB 100 times, sum = 536870912.
8.1927 s to allocate and write 512 MB 100 times (0.4506 s to delete).
12.8554 s to allocate and read 512 MB 100 times, sum = 0.
Pressione Enter para continuar...

@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
● ps -p 12283 -o minflt,majflt
MINFLT MAJFLT
1066850 0
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
```

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

```
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $ ./main
PID: 14544
Busy waiting to raise the CPU frequency...
0.0022 s to allocate 1024 MB 100 times.
0.0022 s to allocate 1024 MB 100 times (0.0016 s to delete).
4.3532 s to write 1024 MB 100 times.
25.1563 s to read 1024 MB 100 times, sum = 1073741824.
18.8936 s to allocate and write 1024 MB 100 times (1.2747 s to delete).
25.9906 s to allocate and read 1024 MB 100 times, sum = 0.
Pressione Enter para continuar...

@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
● ps -p 14544 -o minflt,majflt
MINFLT MAJFLT
3237422 0
@cauaribas →/workspaces/SO-Codes/M2/MemoryCostLinux (main) $
```

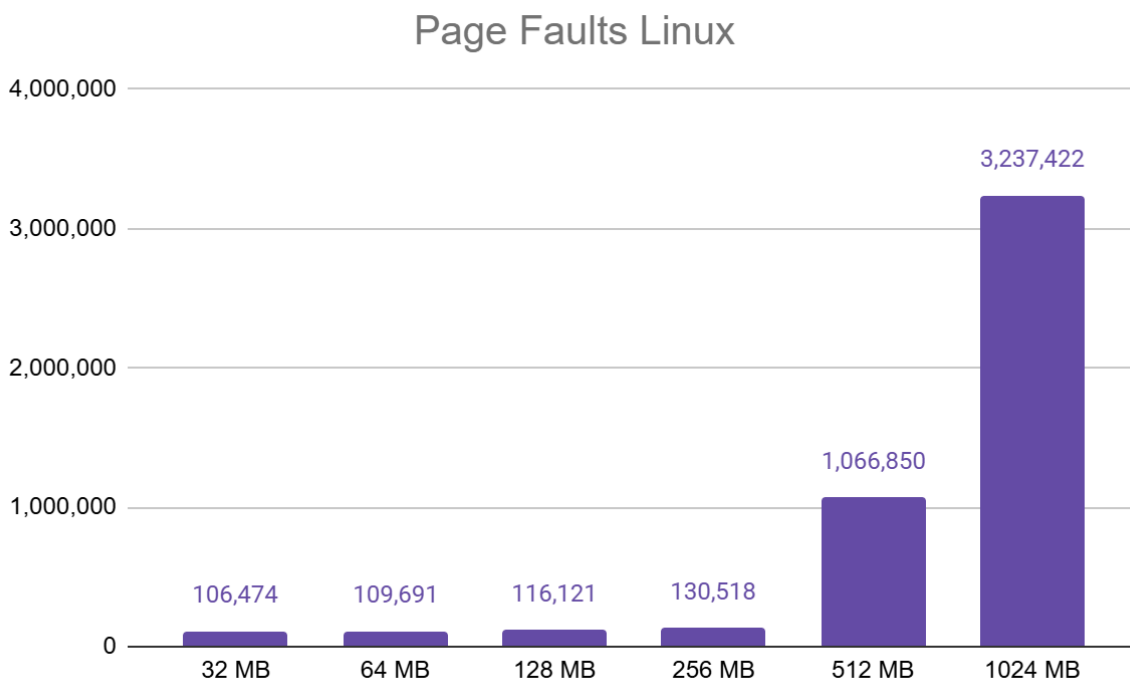
Imagem: Terminal após executar o código MemoryCost alocando 1024MB e o número de Page Faults.

### 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.



```
1 void executeTask(Task* task){
2     task->taskFunction(task->client_socket); // Executa a função de tarefa associada
3 }
```

### 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.



```
1 void submitTask(Task task){
2     pthread_mutex_lock(&mutexQueue);
3     taskQueue[taskCount] = task;
4     taskCount++;
5     pthread_mutex_unlock(&mutexQueue);
6     pthread_cond_signal(&condQueue);
7 }
```

### 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.



```
1 void* startThread(void* args){
2     while(1){
3         Task task;
4
5         pthread_mutex_lock(&mutexQueue);
6         while(taskCount == 0){ // Se não há tarefas, espera até ser sinalizado
7             pthread_cond_wait(&condQueue, &mutexQueue);
8         }
9
10        // Retira a primeira tarefa da fila
11        task = taskQueue[0];
12        for(int i = 0; i < taskCount - 1; i++){
13            taskQueue[i] = taskQueue[i + 1];
14        }
15        taskCount--;
16        pthread_mutex_unlock(&mutexQueue);
17
18        executeTask(&task); // Executa a tarefa removida da fila
19    }
20 }
```

### 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.

```
1  int startServer(const char* sockpath){
2      int server_socket;
3      struct sockaddr_un local;
4
5      // Criação do socket UNIX
6      server_socket = socket(AF_UNIX, SOCK_STREAM, 0);
7      if (server_socket < 0) {
8          perror("Falha ao criar socket");
9          return 1;
10     }
11
12     // Bind socket no endereço local
13     memset(&local, 0, sizeof(local));
14     local.sun_family = AF_UNIX;
15     strncpy(local.sun_path, sockpath, sizeof(local.sun_path) - 1); // Define o caminho do socket
16     unlink(local.sun_path); // Remove qualquer socket anterior com o mesmo caminho
17     int len = strlen(local.sun_path) + sizeof(local.sun_family);
18
19     // Vincula o socket ao caminho especificado
20     if (bind(server_socket, (struct sockaddr*)&local, len) < 0) {
21         perror("Falha ao fazer bind no socket");
22         close(server_socket);
23         return 1;
24     }
25
26     // Coloca o socket em modo de escuta por conexões
27     if (listen(server_socket, 5) < 0) {
28         perror("Erro ao escutar socket");
29         close(server_socket);
30         return 1;
31     }
32
33     return server_socket;
34 }
```



### 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 maiúsculas e envia de volta ao cliente antes de fechar o socket

```
1  void handleStringConnection(int client_socket){
2      char buffer[1024];
3
4      // Ler dados do cliente
5      if (read(client_socket, buffer, sizeof(buffer)) < 0) {
6          perror("Erro ao ler do socket");
7          close(client_socket);
8          return;
9      }
10
11     // Processa a string recebida e converte para maiúsculas
12     for (int i = 0; i < strlen(buffer); i++){
13         buffer[i] = toupper(buffer[i]);
14     }
15
16     // Enviar dados de volta ao cliente
17     write(client_socket, buffer, strlen(buffer) + 1);
18     close(client_socket);
19 }
```

### 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.



```
1 void handleNumberConnection(int client_socket){
2     char buffer[1024];
3
4     // Ler dados do cliente
5     if (read(client_socket, buffer, sizeof(buffer)) < 0) {
6         perror("Erro ao ler do socket");
7         close(client_socket);
8         return;
9     }
10
11     // Processa o numero recebido e multiplica por 2
12     int number = atoi(buffer);
13     number *= 2;
14
15     snprintf(buffer, sizeof(buffer), "%d", number);
16
17     // Enviar dados de volta ao cliente
18     write(client_socket, buffer, strlen(buffer) + 1);
19     close(client_socket);
20 }
```

### 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.



```
1 void acceptConnection(int socket, void (*handler)(int)){
2     struct sockaddr_un remote;
3     socklen_t len = sizeof(remote);
4     int client_socket = accept(socket, (struct sockaddr*)&remote, &len);
5     if (client_socket >= 0) {
6         // printf("Nova conexão recebida\n");
7
8         // Cria uma nova tarefa e a submete para o pool de threads
9         Task t = {
10             .taskFunction = handler,
11             .client_socket = client_socket
12         };
13
14         submitTask(t);
15     }
16 }
```

### 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**).

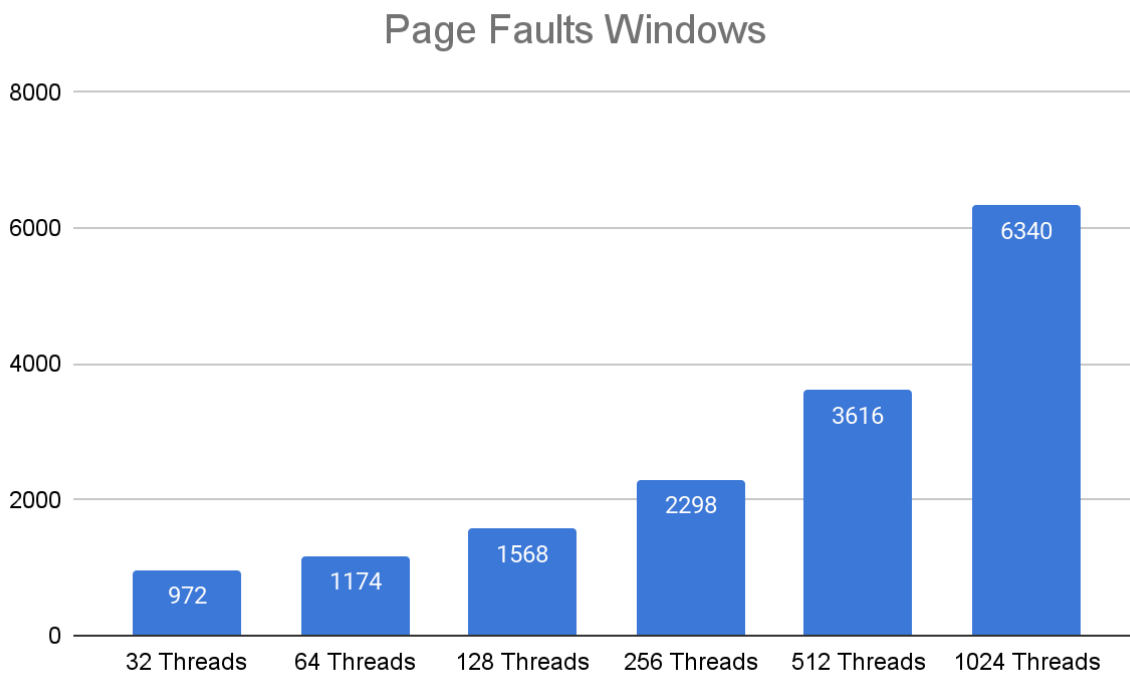
```
1 // Define variáveis para o select e determina o maior descritor de arquivo
2 fd_set read_fds;
3 int max_fd = (string_socket > number_socket) ? string_socket : number_socket;
4
5 while(1){
6     FD_ZERO(&read_fds); // Limpa o conjunto de descritores
7     FD_SET(string_socket, &read_fds); // Adiciona string_socket ao conjunto
8     FD_SET(number_socket, &read_fds); // Adiciona number_socket ao conjunto
9
10    // Espera por atividade nos sockets com select
11    int activity = select(max_fd + 1, &read_fds, NULL, NULL, NULL);
12    if (activity < 0) {
13        perror("Erro no select");
14        continue;
15    }
16    // Verifica se há conexão no socket de strings
17    if(FD_ISSET(string_socket, &read_fds)){
18        acceptConnection(string_socket, handleStringConnection);
19    }
20    // Verifica se há conexão no socket de números
21    if(FD_ISSET(number_socket, &read_fds)){
22        acceptConnection(number_socket, handleNumberConnection);
23    }
24 }
```

## 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.

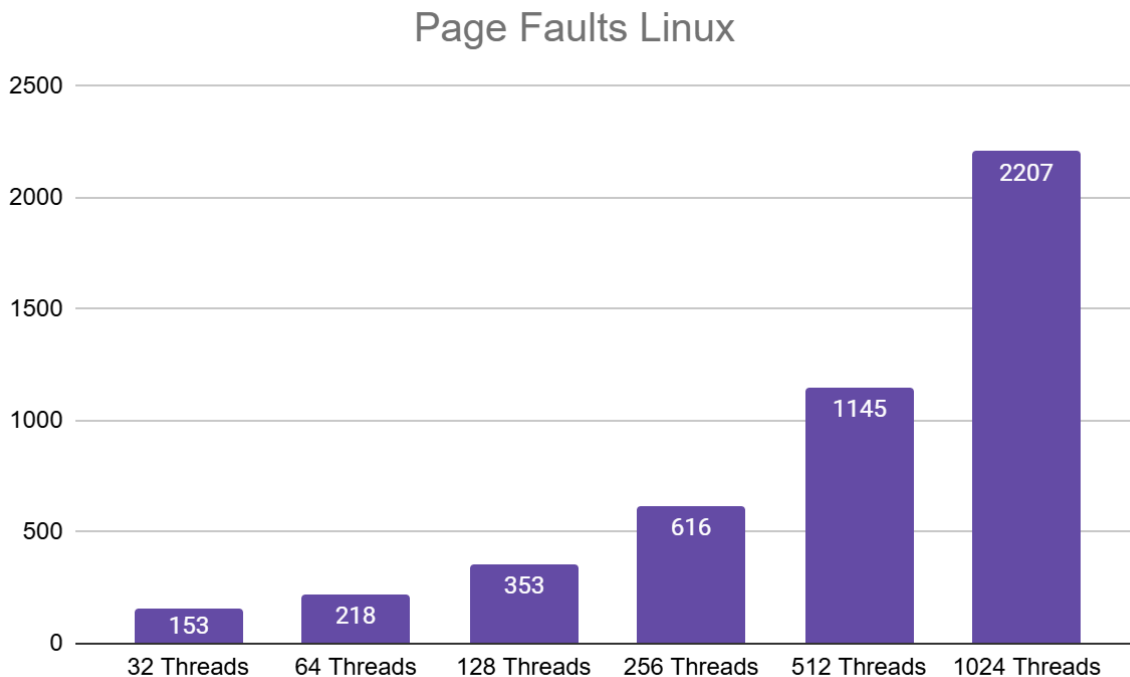
Esse aumento acentuado sugere que o Windows enfrenta limites de escalabilidade para cargas muito altas de threads, exigindo mais acessos à memória virtual e gerando latência adicional. Em resumo, o crescimento dos *page faults* ilustra o impacto negativo de cargas intensivas de threads no desempenho,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



### 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 reconstrói 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):

```
1  import time
2
3  def print_matrix(matrix, size):
4      for i in range(size):
5          for j in range(size):
6              print(matrix[i][j], end=" ")
7          print()
8
9
10 size = 10000
11 matrix = [[0] * size for _ in range(size)]
12 value = 0
13
14 for i in range(size):
15     for j in range(size):
16         matrix[i][j] = value
17         value += 1
18
19 start_time = time.time()
20
21 flat_array = []
22 for i in range(size):
23     for j in range(size):
24         flat_array.append(matrix[i][j])
25
26 flat_array.sort()
27 flat_array = flat_array[::-1]
28
29 index = 0
30 for i in range(size):
31     for j in range(size):
32         matrix[i][j] = flat_array[index]
33         index += 1
34
35 end_time = time.time()
36
37 execution_time = (end_time - start_time) * 1000
38 print(f"Execution time: {execution_time:.2f} ms")
```

### 4.3. Código Fonte (C#):

```
1  using System.Diagnostics;
2
3  void PrintMatrix(int[,] matrix, int size)
4  {
5      for (int i = 0; i < size; i++)
6      {
7          for (int j = 0; j < size; j++)
8          {
9              Console.Write(matrix[i, j] + " ");
10             }
11             Console.WriteLine();
12         }
13     }
14
15     int size = 10000;
16     int[,] matrix = new int[size, size];
17     int value = 0;
18
19     for (int i = 0; i < size; i++)
20     {
21         for (int j = 0; j < size; j++)
22         {
23             matrix[i, j] = value++;
24         }
25     }
26
27     Stopwatch stopwatch = new Stopwatch();
28     stopwatch.Start();
29
30     int[] flatArray = new int[size * size];
31     for (int i = 0; i < size; i++)
32     {
33         for (int j = 0; j < size; j++)
34         {
35             flatArray[i * size + j] = matrix[i, j];
36         }
37     }
38
39     Array.Sort(flatArray);
40     Array.Reverse(flatArray);
41
42     for (int i = 0; i < size; i++)
43     {
44         for (int j = 0; j < size; j++)
45         {
46             matrix[i, j] = flatArray[i * size + j];
47         }
48     }
49
50     stopwatch.Stop();
51
52     Console.WriteLine($"Execution time: {stopwatch.ElapsedMilliseconds} ms");
53     Console.ReadLine();
```



#### 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#):

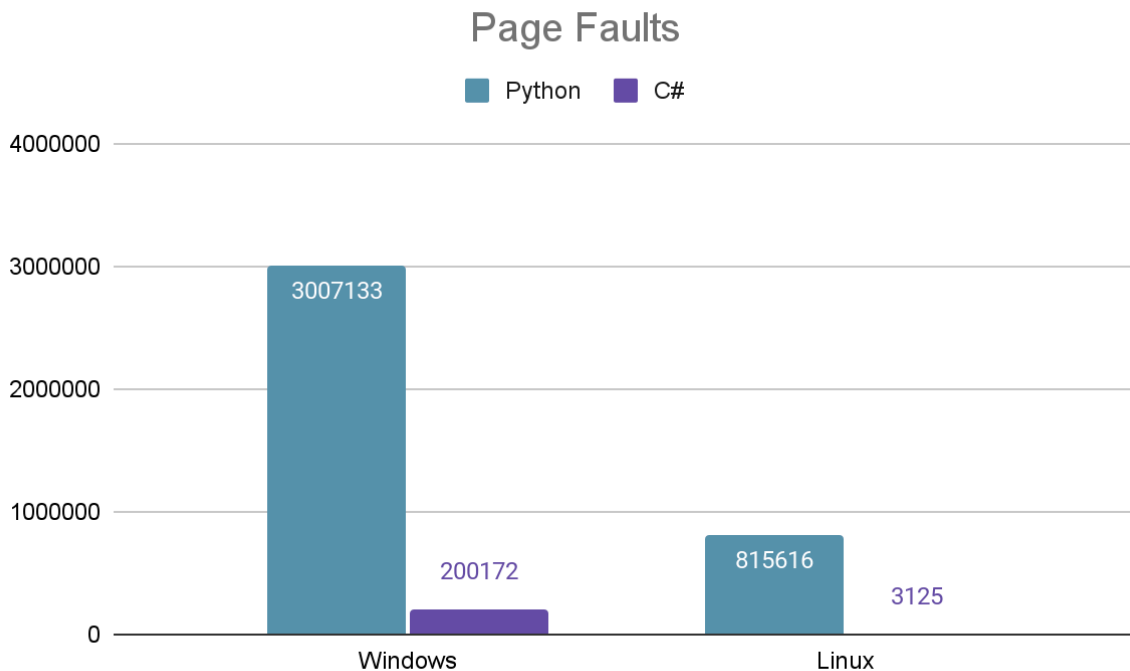
- **Windows:** 200.172 page faults
- **Linux (Codespace):** 3.125 page faults

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](https://www.cyberciti.biz/faq/linux-command-to-see-major-minor-pagefaults/#google_vignette). Acesso em: 11 nov. 2024.