



**UNIVALI**

**UNIVERSIDADE DO VALE DO ITAJAÍ**

## Paginação e Page Fault

Sistema Operacionais

Henrique Borovicz Reis, Henrique Cipriani Ertal

Universidade do Vale do Itajaí (UNIVALI)

Rua Uruguai, 458 – Centro, Itajaí – SC, 88302-901

[henriquereis@edu.univali.br](mailto:henriquereis@edu.univali.br)

[henriquecipriani@edu.univali.br](mailto:henriquecipriani@edu.univali.br)

## Sumário

Sumário .....	2
• Introdução.....	2
• Enunciado do Projeto.....	2
• Explicação e Contexto da Aplicação .....	3
• Códigos Importantes da Implementação .....	3
• Cliente.c .....	3
• Servidor.c.....	3
• Memory_cost.cpp .....	4
• Buffers.cpp.....	4
• Matriz.c .....	4
• Controle.py e Controle.cpp.....	4
• Análise do código M1.....	4
• Análise do código Matriz.c.....	10
• Análise do código Memory_cost.cpp .....	12
• Análise do código Buffers.cpp .....	13
• Análise e Discussão dos Resultados.....	15
• Análise do código controle.cpp e controle.py (extra) .....	18

### • Introdução

Este trabalho analisa o impacto dos page faults no desempenho de sistemas operacionais, utilizando simulações de alta demanda de memória em Windows e Linux. O objetivo é avaliar como diferentes tamanhos de alocação de memória e operações multi-thread influenciam a frequência de page faults e a eficiência do sistema. Para isso, foram desenvolvidos e adaptados programas que simulam o acesso intensivo à memória em variados cenários, incluindo alocações de múltiplos tamanhos e operações simultâneas de entrada e saída. Os resultados obtidos contribuem para uma compreensão mais profunda do gerenciamento de memória e das estratégias para otimização de aplicações de alto desempenho.

### • Enunciado do Projeto

Este projeto tem como objetivo avaliar o impacto dos page faults no desempenho de sistemas operacionais, como Windows e Linux, por meio de simulações que envolvem alocação intensiva de memória. Um page fault ocorre quando um programa tenta acessar uma página de memória que não está carregada na memória principal (RAM), exigindo que o sistema operacional recorra à memória secundária, o que pode impactar o desempenho.

Para realizar essa análise, foi utilizado um código de teste de alocação de memória fornecido pelo professor, que foi modificado para incluir diferentes tamanhos de buffers. A ideia é observar como o sistema gerencia o acesso à memória em cenários com alta demanda e

verificar se o número de page faults e o tempo de resposta variam com diferentes tamanhos de alocação e número de threads.

Além disso, o trabalho inclui testes em um código adicional que simula operações de leitura e escrita intensivas em memória, e uma simulação de servidor com múltiplas threads realizando operações de entrada e saída simultâneas. Esses experimentos foram conduzidos tanto em Windows quanto em Linux, utilizando ferramentas específicas para monitorar o desempenho e registrar os page faults.

## • Explicação e Contexto da Aplicação

O gerenciamento de memória é uma função essencial nos sistemas operacionais, especialmente em aplicações que exigem alto desempenho e eficiência. Quando uma aplicação tenta acessar dados que não estão na memória principal (RAM), ocorre o que chamamos de page fault – um evento em que o sistema precisa buscar esses dados na memória secundária, como o disco rígido ou SSD. Esse processo de transferência entre memória secundária e RAM gera uma interrupção e pode impactar diretamente o desempenho da aplicação, principalmente em contextos em que o acesso à memória é intensivo.

A análise de page faults é crucial para entender como o sistema opera em situações de alta demanda e como políticas de gerenciamento de memória podem ser ajustadas para otimizar o desempenho. Esse tipo de análise ajuda a identificar possíveis gargalos, vazamentos de memória e ineficiências nas políticas de paginação, especialmente em sistemas que suportam várias operações simultâneas ou utilizam grandes volumes de dados.

Neste trabalho, realizamos simulações com diferentes tamanhos de alocação de memória e com o uso de múltiplas threads para investigar como o aumento na demanda por memória impacta a ocorrência de page faults. Os testes foram aplicados em diferentes condições – variando desde pequenos buffers até grandes alocações e ambientes multi-thread – para observar o comportamento do sistema em cada cenário. Esses experimentos nos ajudam a visualizar como o sistema operacional gerencia a memória e responde a diferentes condições de uso, proporcionando uma análise detalhada de como os page faults e o gerenciamento de memória afetam o desempenho do sistema.

## • Códigos Importantes da Implementação

### • Cliente.c

O código do cliente cria dois pipes nomeados para se comunicar com o servidor. Ele possui duas threads: uma para enviar dados (solicitações) ao servidor e outra para ler as respostas recebidas. O cliente especifica seu próprio ID e tipo de solicitação (número ou string) e utiliza seu próprio pipe de resposta para que o servidor possa enviar respostas de forma direcionada.

### • Servidor.c

O código do servidor cria dois pipes nomeados (ou "Named Pipes") que permitem comunicação entre processos. Um pipe é usado para atender solicitações de números, enquanto o outro responde a solicitações de strings. O servidor fica em um loop infinito, aguardando conexões dos clientes. Quando uma solicitação é recebida, o servidor lê a mensagem e envia uma resposta personalizada de volta para o cliente por meio de um pipe exclusivo do cliente.

- **Memory\_cost.cpp**

Memory\_cost.cpp é o código original fornecido pelo professor, que realiza operações de alocação e liberação de memória com um buffer fixo de 32 MB. Esse código serve como referência para medir o impacto básico das operações de memória no sistema, observando o número de page faults e o tempo de execução.

- **Buffers.cpp**

Buffers.cpp é uma versão modificada do código original (Memory\_cost.cpp), agora com múltiplos tamanhos de buffers (64 MB, 128 MB, 256 MB, 512 MB). O objetivo é observar o comportamento do sistema em relação aos page faults e ao uso de CPU em diferentes alocações de memória.

- **Matriz.c**

Matriz.c aloca dinamicamente uma matriz bidimensional muito grande (16.384 x 16.384 inteiros), que consome uma quantidade significativa de memória (aproximadamente 1 GB). Ele é projetado para simular uma carga intensa de memória e provocar page faults.

- **Controle.py e Controle.cpp**

Este código simula o controle de acesso a recursos compartilhados por várias threads, implementando um mecanismo de prevenção de deadlock chamado *wound-wait*. As threads tentam acessar dois recursos (lock\_X e lock\_Y) e, caso ocorra um conflito de acesso, a thread mais "antiga" pode forçar a "mais nova" a liberar o recurso para evitar um deadlock. O código usa locks e timestamps para controlar quais threads possuem os recursos e para gerenciar a ordem de liberação dos mesmos, garantindo que o sistema funcione sem travamentos durante o acesso simultâneo aos recursos.

- **Análise do código M1**

Nesta análise do código implementado na M1, que envolve a comunicação entre processos e o uso de um thread pool, foram realizadas duas etapas em cada sistema operacional. A primeira etapa consistiu em executar o código com o número de threads previamente definido, enquanto a segunda etapa envolveu o aumento do número de threads no servidor para observar eventuais mudanças no uso de memória, swap e page faults.

#### **Análise no Linux (Codespaces) - 6 Threads**

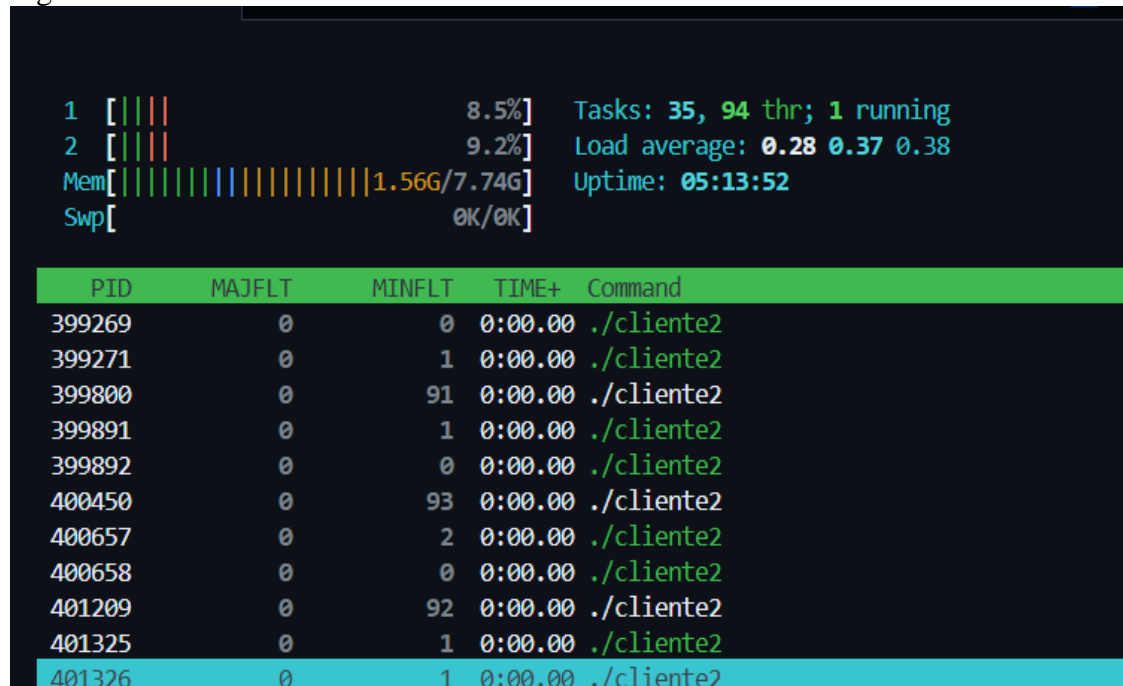
Para realizar esta análise, utilizamos o comando “htop” no terminal do Codespaces, ajustando a exibição para focar nas métricas mais relevantes e ocultar informações desnecessárias. O resultado obtido foi o seguinte:

Figura das saídas do servidor e cliente:

Figura dos resultados do servidor:

PID	MAJFLT	MINFLT	TIME+	Command
398675	0	103	0:00.05	./servidor2
398680	0	5	0:00.01	./servidor2
398676	0	1	0:00.00	./servidor2
398677	0	1	0:00.00	./servidor2
398678	0	1	0:00.00	./servidor2
398679	0	1	0:00.00	./servidor2

Figura dos resultados dos clientes:



A partir dos dados coletados com o `htop` durante a execução do servidor (4 threads) e de quatro clientes distintos (2 threads cada), observamos que o programa implementado não ultrapassou o limite de memória disponível no Codespaces. Além disso, não foi registrada a ocorrência de major page faults (MAJFLT), o que indica que não houve a necessidade de acessar o disco para buscar páginas ausentes na memória principal. Contudo, foi possível observar a ocorrência de minor page faults (MINFLT), que também indicam a falta de páginas na memória RAM, mas com a diferença de que, neste caso, a página em questão já possui um mapeamento carregado em cache ou está na área de swap, tornando desnecessário o acesso ao disco para sua recuperação.

Essas ocorrências foram mais significativas durante a execução inicial do servidor e dos clientes, quando as bibliotecas do sistema ainda não estavam totalmente carregadas na memória RAM. Nessa fase, muitas dessas páginas estavam mapeadas em cache ou localizadas na área de swap, o que resultou em um número considerável de minor page faults. No entanto, à medida que o sistema entrou em loop e os processos ficaram em execução contínua, a quantidade de minor page faults diminuiu significativamente.

Esse comportamento pode ser explicado pela dinâmica de comunicação entre o servidor e os clientes, utilizando FIFOs e threads. Durante o processo de execução, ocorre a espera por I/O entre a comunicação dos FIFOs, o preenchimento dos buffers ou até mesmo o aguardo de uma thread pelo sinal de liberação da fila. Esse tipo de operação faz com que o sistema operacional mova processos para a área de swap até que a operação de entrada/saída seja concluída. Assim, quando a página é trazida novamente para a RAM, ocorre uma minor page fault, já que a página foi previamente armazenada na área de swap ou em cache, mas não estava na memória principal no momento do acesso.

Análise no Linux (Codespaces) - 10 Threads

Para realizar esta análise, utilizamos o comando “htop” no terminal do Codespaces, ajustando a exibição para focar nas métricas mais relevantes e ocultar informações desnecessárias. O resultado obtido foi o seguinte:

Figura das saídas do servidor e cliente:

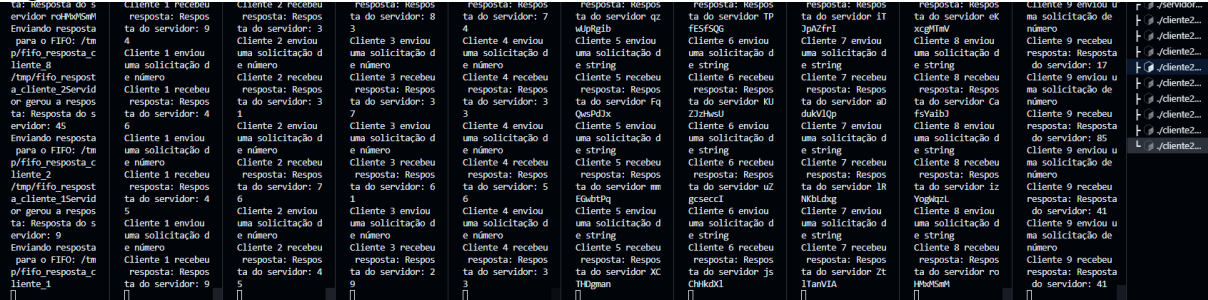


Figura dos resultados do servidor:

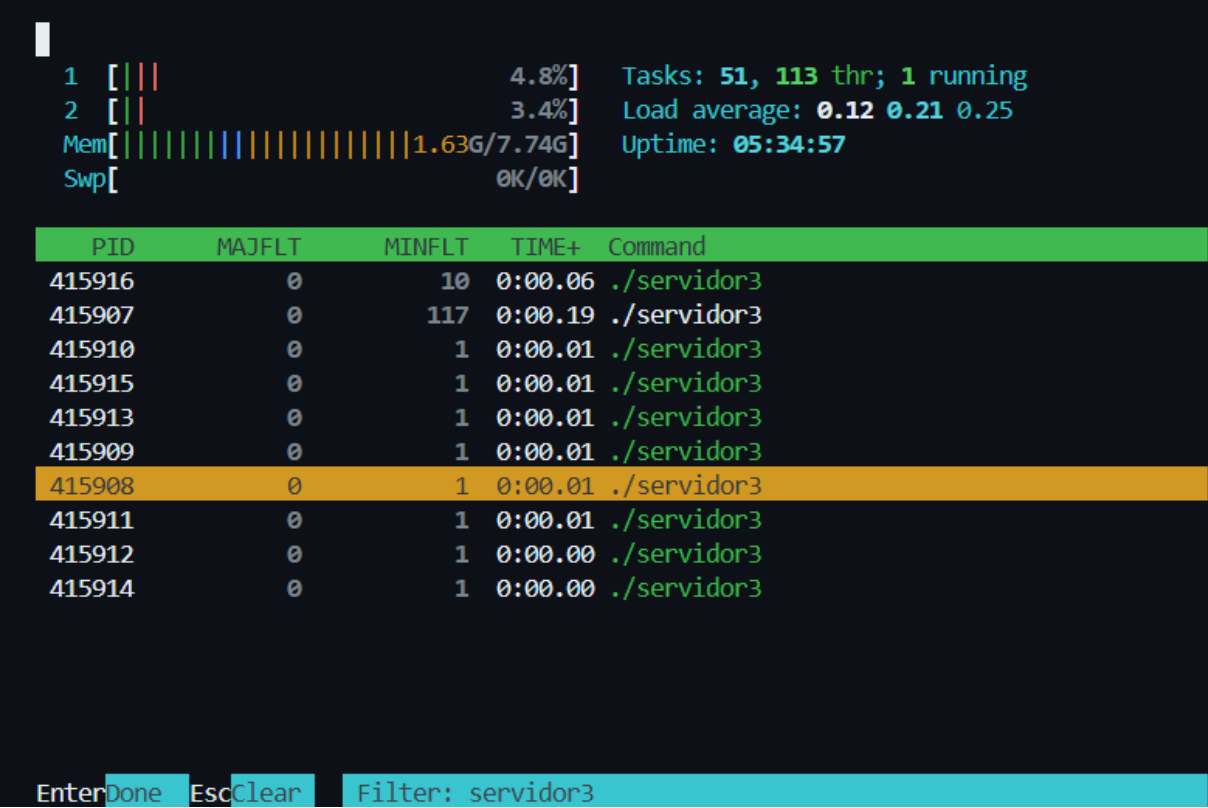
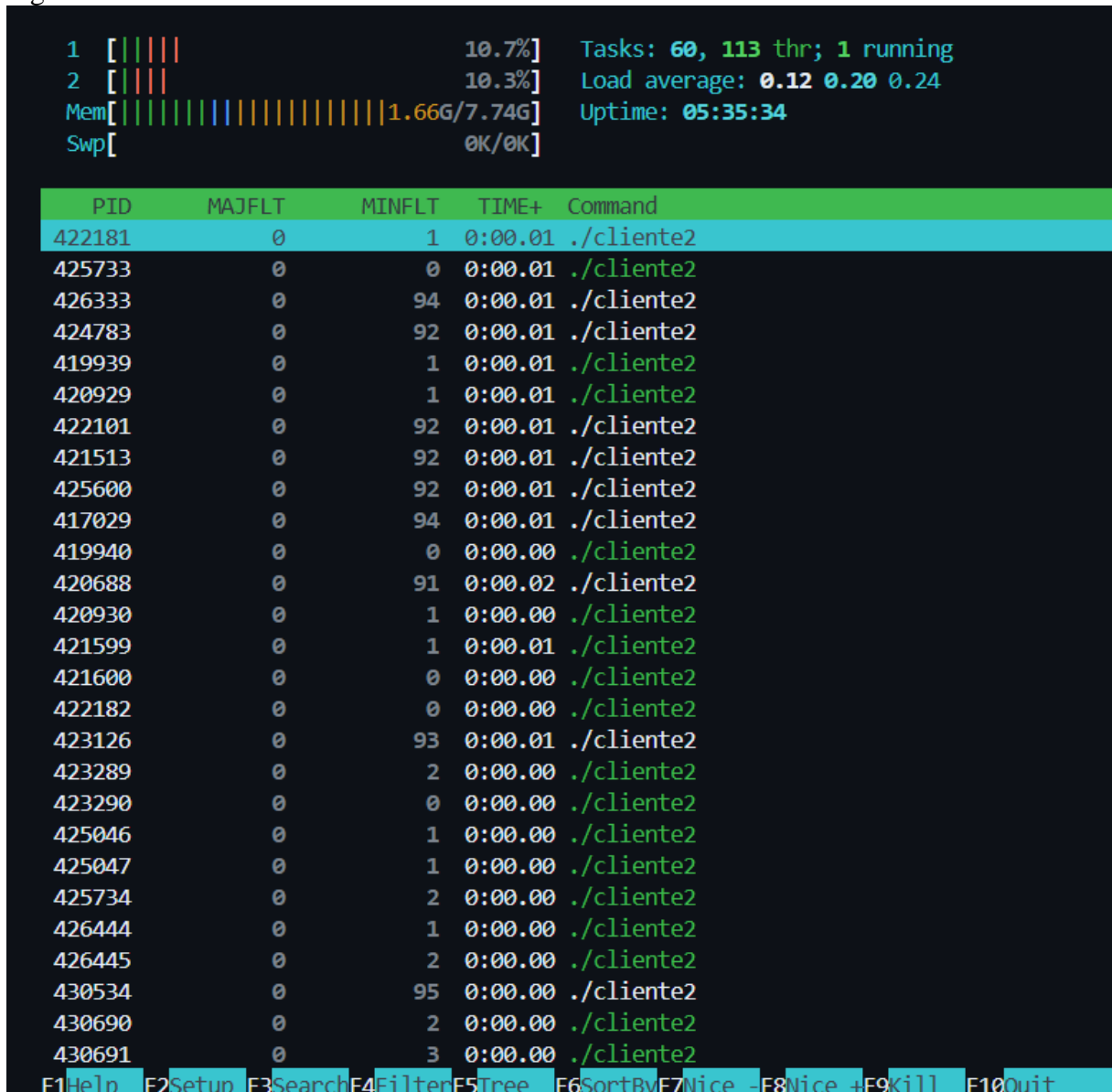


Figura dos resultados do servidor:



A partir dos dados coletados nessa etapa, com a execução de um servidor para nove clientes podemos observar que apesar do aumento significativo de threads e de clientes simultâneos a memória ainda assim não foi alocada de forma significativa, o resultou em uma quantia proporcional de page faults a etapa anterior, tendo aproximadamente 90~100 na primeira execução de cada código e após isso aproximadamente 2 minor page faults a cada iteração, resultantes também da espera por I/O das threads e dos canais de comunicação de processos.

## Análise no Windows (VSCODE) - 7 Threads

Figura das saídas de terminal:

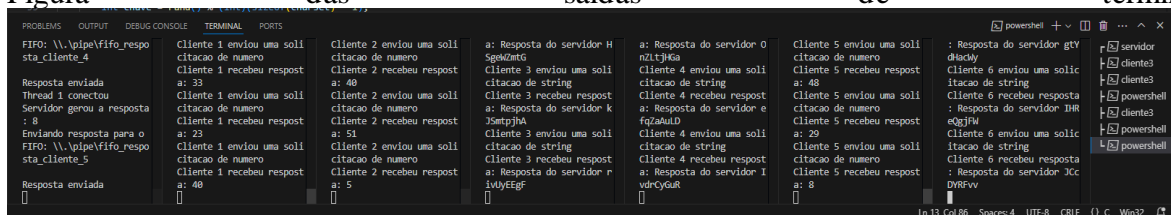
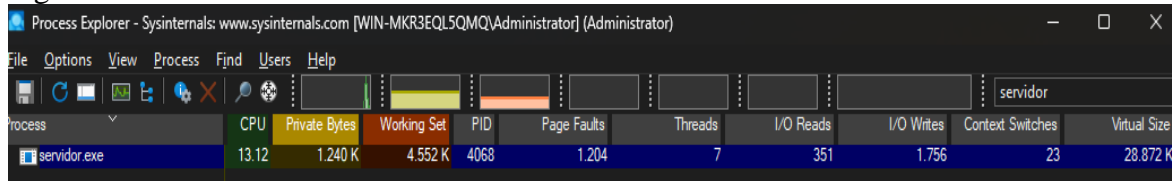


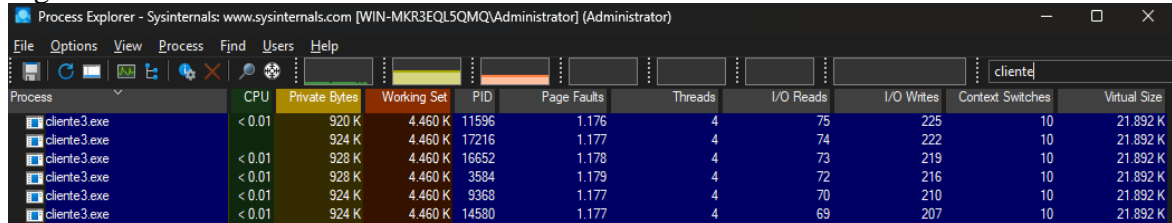


Figura do resultado do servidor:



Process	CPU	Private Bytes	Working Set	PID	Page Faults	Threads	I/O Reads	I/O Writes	Context Switches	Virtual Size
servidor.exe	13.12	1.240 K	4.552 K	4068	1.204	7	351	1.756	23	28.872 K

Figura dos resultados dos clientes:



Process	CPU	Private Bytes	Working Set	PID	Page Faults	Threads	I/O Reads	I/O Writes	Context Switches	Virtual Size
cliente3.exe	< 0.01	920 K	4.460 K	11596	1.176	4	75	225	10	21.892 K
cliente3.exe	< 0.01	924 K	4.460 K	17216	1.177	4	74	222	10	21.892 K
cliente3.exe	< 0.01	928 K	4.460 K	16652	1.178	4	73	219	10	21.892 K
cliente3.exe	< 0.01	928 K	4.460 K	3584	1.179	4	72	216	10	21.892 K
cliente3.exe	< 0.01	924 K	4.460 K	9368	1.177	4	70	210	10	21.892 K
cliente3.exe	< 0.01	924 K	4.460 K	14580	1.177	4	69	207	10	21.892 K

Para adaptar o programa originalmente desenvolvido em Linux para o Windows, foi necessário fazer diversas modificações no código-fonte, o que acabou impactando os resultados. As principais mudanças incluem o uso da API **pthread**s no lugar das alternativas do Linux, a troca dos **FIFOs** por **named pipes** e ajustes na estrutura lógica do código para garantir o funcionamento correto no Windows.

Uma alteração importante foi a necessidade de recriar os **named pipes** a cada iteração do processo cliente/servidor para permitir a troca de mensagens. Esse processo de recriação constante dos pipes pode ter gerado um aumento no número de *page faults*, pois o sistema precisa alocar e desalocar memória a todo momento.

1. **Recriação constante dos pipes:** No Windows, como os pipes precisam ser recriados a cada iteração, isso gera mais alocações de memória e, consequentemente, mais *page faults*. No Linux, os FIFOs são mais leves e não exigem tanta recriação.
2. **Maior uso de threads:** A utilização de mais threads no Windows pode aumentar o número de trocas de contexto e de acessos à memória, o que também pode contribuir para um maior número de *page faults*.
3. **Gerenciamento de memória diferente:** O Windows tem uma abordagem mais conservadora com o gerenciamento de memória, o que pode resultar em mais *page faults*, já que o sistema precisa alocar e desalocar mais páginas de memória.
4. **Ambiente Controlado:** A mudança de um ambiente controlado (Codespaces) para um ambiente não controlado que seria o caso do Windows, pode gerar uma diferença muito grande no gerenciamento de memória para o programa, visto que ele está competindo por memória com todos os outros processos do SO, e, consequentemente ter mais ocorrência de *page faults*.

Esses fatores combinados podem ter gerado o aumento nos *page faults* observados no Windows, quando comparado com o Linux, onde o programa apresentava um comportamento mais eficiente nesse aspecto.

## Análise no Windows (VSCODE) - 11 Threads

Figura das saídas de execução:

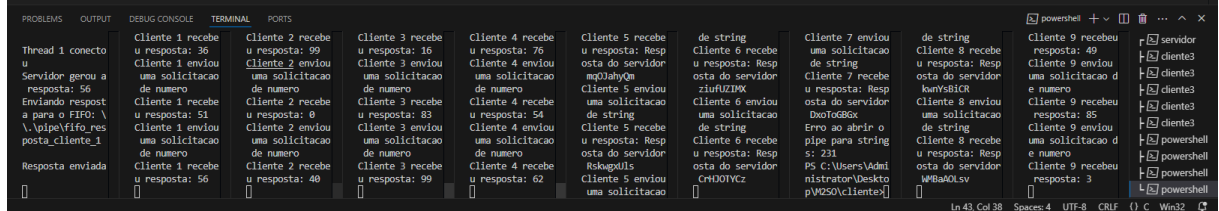


Figura dos resultados do servidor:

Process	CPU	Private Bytes	Working Set	PID	Page Faults	Threads	I/O Reads	I/O Writes
servidor.exe	34.16	1.504 K	4.652 K	4864	1.248	10	1.588	7.941

Figura dos resultados dos clientes:

Process	CPU	Private Bytes	Working Set	PID	Page Faults	Threads	I/O Reads	I/O Writes	Context Switches	Virtual Size
cliente3.exe		848 K	4.432 K	13508	1.185	3	261	783	10	19.588 K
cliente3.exe		848 K	4.432 K	12588	1.185	3	258	774	10	19.588 K
cliente3.exe		848 K	4.432 K	17184	1.185	3	259	777	10	19.588 K
cliente3.exe		852 K	4.432 K	1360	1.186	3	256	768	11	19.588 K
cliente3.exe		848 K	4.432 K	292	1.185	3	254	762	11	19.588 K
cliente3.exe		844 K	4.432 K	9868	1.187	3	253	759	10	19.588 K
cliente3.exe		848 K	4.432 K	20444	1.185	3	251	753	11	19.588 K
cliente3.exe		848 K	4.432 K	18612	1.188	3	249	747	11	19.588 K

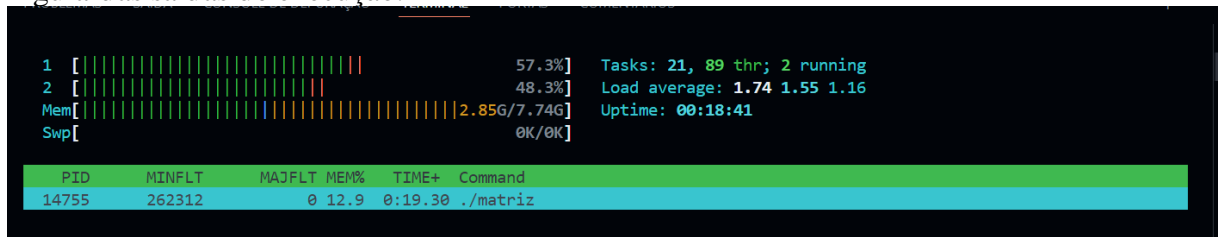
Como podemos observar, mesmo com o aumento do numero de threads e de clientes simultâneos, não obtivemos mudanças significativas na quantidade de page faults em relação a primeira execução com menos threads, embora tenha dado uma diferença grande em relação ao código executado no Codespaces.

Uma explicação para isso pode ser pelo fato de que o Codespaces é um ambiente controlado para execução destes programas, enquanto a execução dele diretamente no Windows é afetada pela concorrência de todos os processos simultâneos, fazendo com que o processo tenha competição não somente com ele mesmo, mas também com todos os outros serviços presentes no SO, o que gera uma demanda maior por memória e consequentemente uma maior ocorrência de Page Faults.

## • Análise do código Matriz.c

### Análise no Linux (Codespaces)

Figura das saídas de execução:



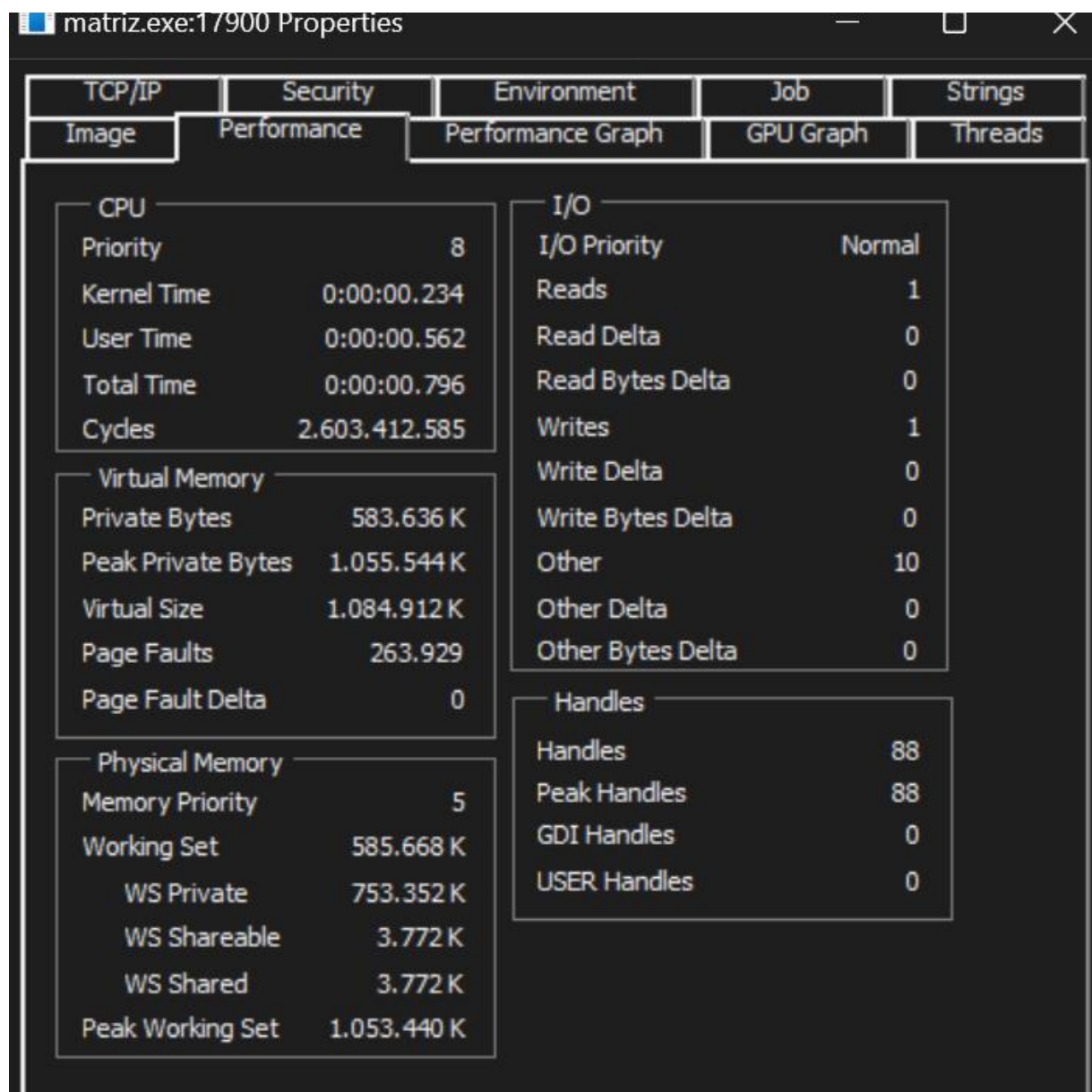
Na execução do programa matriz.c no ambiente Linux, o sistema registrou um total de 262.312 minor page faults e nenhum major page fault. Esse número elevado de minor page faults sugere que o programa utilizou intensamente a memória, provavelmente devido ao tamanho das

matrizes manipuladas. Minor page faults ocorrem quando o sistema precisa carregar páginas que já estão mapeadas, mas não na RAM, o que é comum em operações que acessam grandes quantidades de memória.

A ausência de major page faults indica que o sistema Linux conseguiu atender às solicitações de memória sem precisar buscar páginas no disco, o que é positivo para o desempenho. O programa consumiu aproximadamente 2,85 GB de memória, correspondendo a cerca de 12,9% da memória total do sistema. Esse consumo significativo demonstra que o sistema conseguiu gerenciar bem os recursos, garantindo uma execução contínua e sem interrupções.

#### Análise no Windows (Process Explorer)

Figura das propriedades do código matriz.c:



No ambiente Windows, o programa matriz.exe também gerou uma quantidade significativa de page faults, totalizando 263.929, dos quais a maioria eram minor page faults. Isso indica que o

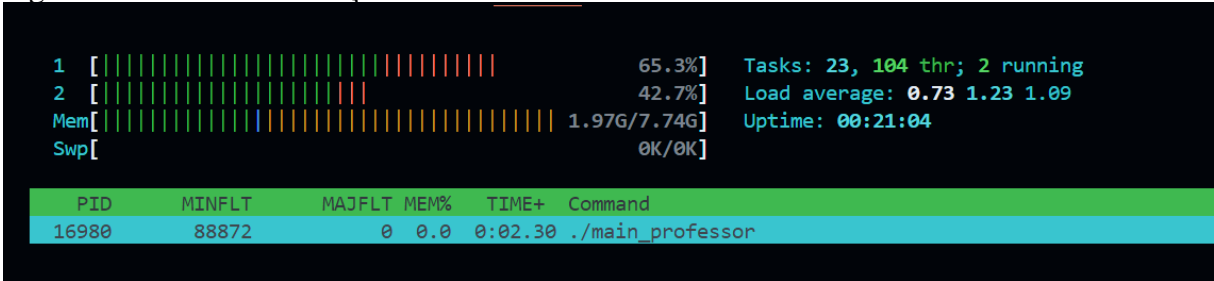
programa fez acessos frequentes a páginas de memória que estavam fora da RAM, mas mapeadas no cache ou na área de swap. O uso de memória física (Working Set) foi de 585.668 KB, com um pico de 1.053.440 KB, o que mostra uma alocação eficiente de memória pelo sistema.

A ausência de major page faults em Windows sugere que o sistema conseguiu lidar com as solicitações de memória sem acessar diretamente o disco, mantendo o desempenho estável. Esse comportamento é ideal, pois minimiza a latência causada por operações de E/S de disco, especialmente em programas que demandam grande quantidade de memória, como operações com matrizes grandes.

- **Análise do código Memory\_cost.cpp**

**Análise no Linux (Codespaces)**

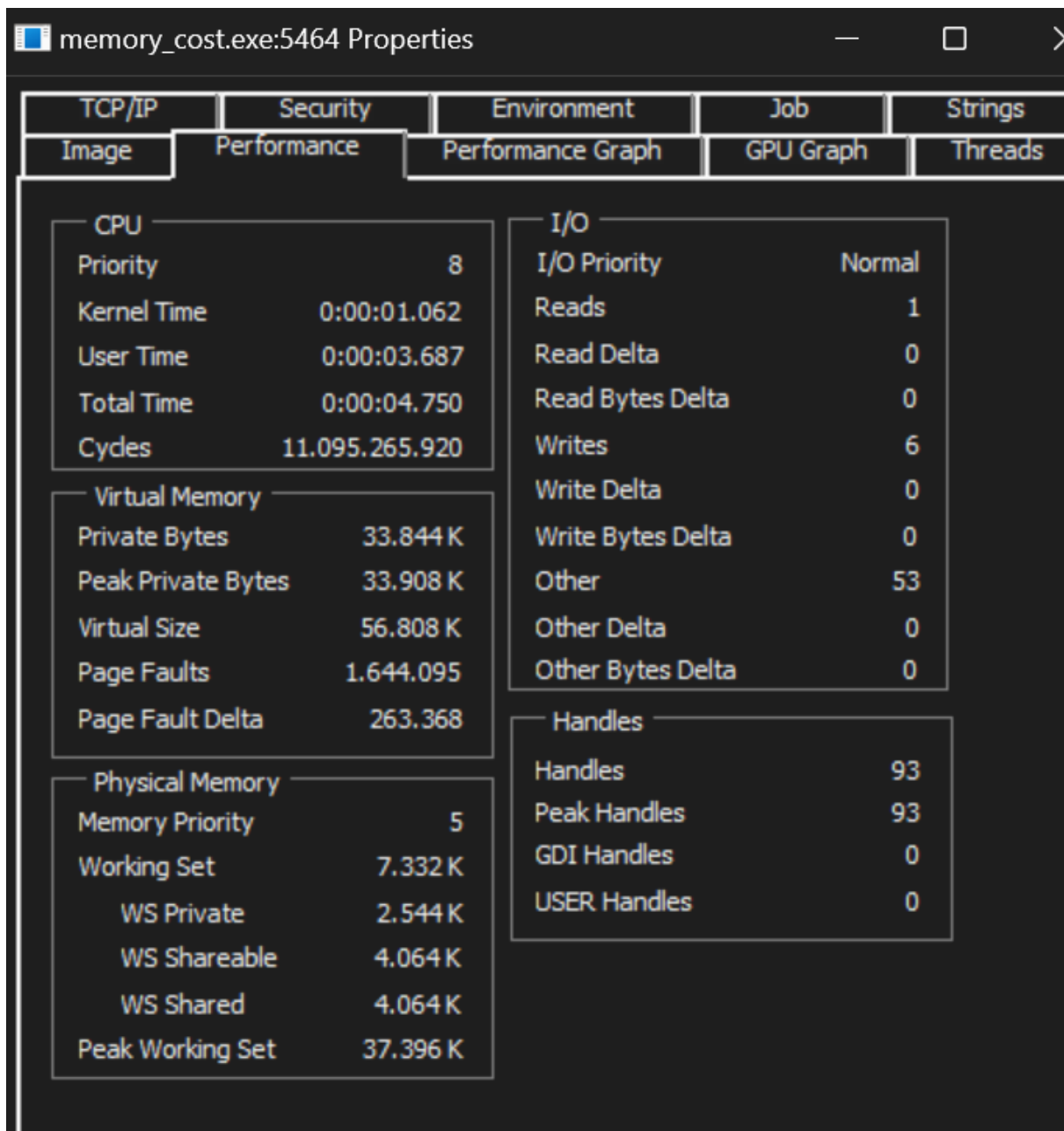
Figura das saídas de execução:



Na execução do código memory\_cost.cpp no Linux, foram registrados 88.872 minor page faults e nenhum major page fault. Esse número de minor page faults está associado ao carregamento de páginas que já estavam mapeadas no sistema, mas não necessariamente na RAM no momento do acesso. Esse comportamento é esperado em operações que utilizam memória intensivamente, como no caso desse programa. O uso de memória totalizou aproximadamente 1,97 GB, o que representa uma fração considerável da memória disponível, mas ainda dentro dos limites que evitam a necessidade de swap. A ausência de major page faults indica que o Linux conseguiu gerenciar o uso de memória sem recorrer ao disco, garantindo uma execução estável do programa.

**Análise no Windows (Process Explorer)**

Figura das propriedades do código memort\_cost.cpp:



No Windows, o programa `memory_cost.exe` registrou um total de 1.644.095 page faults, com um delta de 263.368, sendo a maioria deles minor page faults. Isso indica que houve muitos acessos a páginas que não estavam na RAM, mas que estavam disponíveis na área de cache ou no swap. O Working Set (memória física em uso) foi de 7.332 KB, com um pico de 37.396 KB, o que mostra que o programa utilizou a memória eficientemente dentro do espaço físico disponível. A ausência de major page faults também aqui confirma que o sistema conseguiu evitar acessos diretos ao disco, preservando o desempenho do programa.

- **Análise do código `Buffers.cpp`**

Análise no Linux (Codespaces)

Figura das saídas de execução:

```
1 [|||||||||||||||||90.6%] Tasks: 23, 105 thr; 2 running
2 [||||||||| 36.9%] Load average: 1.27 0.83 0.87
Mem[|||||||||2.11G/7.74G] Uptime: 00:31:39
Swp[||||| 0K/0K]

PID    MINFLT  MAJFLT MEM%  TIME+  Command
24099  204495      0  0.0  1:24.77 ./main_buffer

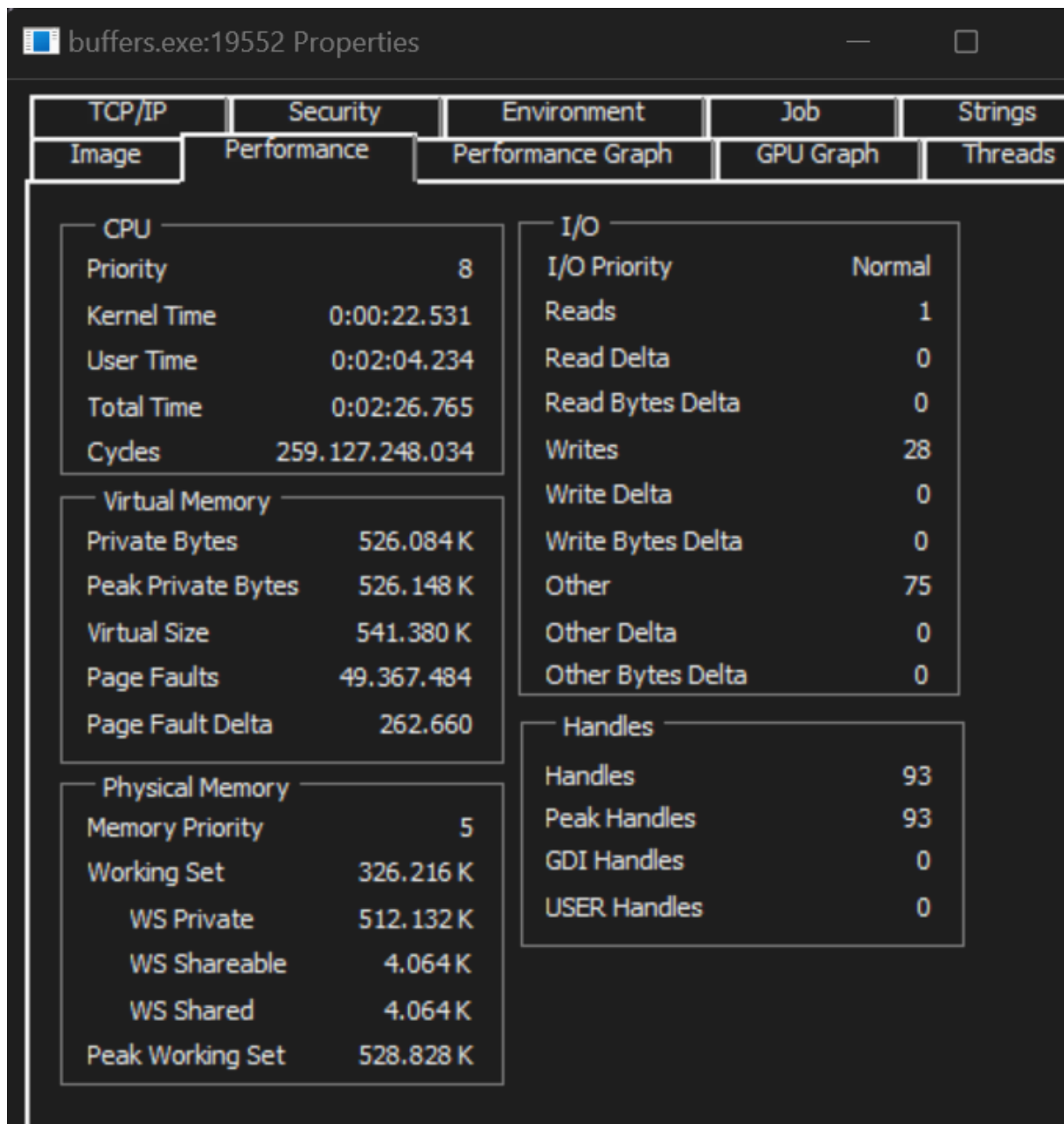
@henriquehce →/workspaces/SO---Codes/T2 (main) $ ^C
@henriquehce →/workspaces/SO---Codes/T2 (main) $ g++ -o main_buffer main_buffer.cpp
@henriquehce →/workspaces/SO---Codes/T2 (main) $ ./main_buffer
Busy waiting to raise the CPU frequency...

Testing with buffer size: 1024 MB
Before testing buffer size 1024 MB:
0.0022 s to allocate 1024 MB 100 times.
0.0022 s to allocate 1024 MB 100 times (0.0017 s to delete).
4.8171 s to write 1024 MB 100 times.
33.4785 s to read 1024 MB 100 times, sum = 1073741824.
13.1466 s to allocate and write 1024 MB 100 times (0.4203 s to delete).
33.6494 s to allocate and read 1024 MB 100 times, sum = 0.
```

A execução do Buffers.cpp no Linux exigiu uma quantidade significativa de memória, com um uso de aproximadamente 2,166 GB, sem recorrer ao swap. Durante a execução, foram registrados 204.495 minor page faults, indicando acessos rápidos a páginas que não estavam inicialmente na RAM mas estavam disponíveis em cache ou swap. Assim como nas outras execuções no Linux, não foram registrados major page faults, o que sugere que o sistema conseguiu atender a demanda sem acessar o disco. As operações de leitura foram mais lentas que as de escrita, indicando uma possível sobrecarga em leituras devido ao grande volume de dados processado.

### Análise no Windows (Process Explorer)

Figura das propriedades do código buffers.cpp:



No Windows, o código Buffers.cpp registrou um total de 49.367.484 page faults, todos sendo minor page faults. Esse número elevado é resultado das operações intensivas de leitura e escrita em buffers, exigindo alocações constantes e liberando memória. Com um conjunto de trabalho (Working Set) de aproximadamente 528.828 KB no pico, o sistema conseguiu lidar com a carga sem necessidade de major page faults, garantindo a continuidade do desempenho. A ausência de major page faults reforça a eficiência do sistema na gestão de memória, permitindo que o programa rodasse de forma estável apesar da alta quantidade de page faults.

## • Análise e Discussão dos Resultados

Com base na análise realizada, foi possível observar diferenças relevantes entre o comportamento dos sistemas operacionais Linux e Windows ao executar diferentes códigos, incluindo Memory\_cost.cpp, Buffers.cpp, Matriz.c, e os códigos da M1 (cliente.c e servidor.c). Cada código exigiu diferentes operações de alocação de memória, I/O, e comunicação entre



processos, o que permitiu uma avaliação abrangente da eficiência de gestão de memória de cada sistema operacional.

#### *Comparação entre Códigos e Eficiência de Page Faults:*

##### 1. **Memory\_cost.cpp:**

- Este código apresentou uma quantidade elevada de minor page faults em ambos os sistemas, mas sem major page faults. A ausência de major page faults sugere uma boa alocação de memória, onde as páginas necessárias estavam prontamente disponíveis sem necessidade de acesso ao disco. O comportamento estável indica que ambos os sistemas operacionais gerenciaram eficientemente a carga de memória.

##### 2. **Buffers.cpp:**

- Com operações intensivas de leitura e escrita em buffers grandes, este código destacou diferenças mais visíveis. No Windows, foram registrados muito mais page faults, possivelmente devido à forma como o sistema gerencia páginas em operações de I/O intensivo. No Linux, o código apresentou um número menor de page faults e uma melhor performance nas operações de leitura, indicando um gerenciamento de cache e swap mais otimizado.

##### 3. **Matriz.c:**

- Este código utilizou grandes matrizes, resultando em um uso elevado de memória em ambos os sistemas. Embora ambos os sistemas tenham mantido a estabilidade, o Linux demonstrou uma melhor otimização de memória, com menos minor page faults e um uso de memória mais eficiente. A ausência de major page faults neste código indica que ambos os sistemas puderam lidar com grandes alocações de memória sem acessar o disco.

##### 4. **Cliente.c e Servidor.c (M1):**

- Os códigos da M1, com comunicação entre processos e uso de threads, evidenciaram as capacidades de ambos os sistemas operacionais na gestão de múltiplas threads e operações de I/O. O Linux mostrou-se mais eficiente ao evitar page faults desnecessários, principalmente durante a execução inicial, quando as bibliotecas ainda estavam sendo carregadas. No Windows, o número de page faults foi maior, porém o sistema conseguiu manter a execução estável, embora um pouco mais lenta em cenários de I/O intenso.

○

#### **Comparação entre Sistemas Operacionais:**

##### 1. **Minor Page Faults:**

- O Linux apresentou consistentemente um número menor de minor page faults, principalmente em operações de I/O e comunicação entre processos. Esse comportamento reflete uma otimização no uso de cache e swap, o que reduz a necessidade de movimentação frequente de páginas na memória. No Windows, os page faults ocorreram com mais frequência, sugerindo que o sistema adota uma abordagem mais conservadora, buscando manter o desempenho sem comprometer a estabilidade.

##### 2. **Major Page Faults:**

- Nenhum dos sistemas registrou major page faults durante a execução dos códigos, o que indica uma alocação de memória eficiente e uma quantidade suficiente de RAM para suportar as operações. Essa ausência de major page faults foi um fator positivo, pois indica que ambos os sistemas conseguiram gerenciar a carga sem degradação significativa de desempenho, mesmo com operações intensivas de memória.



### 3. Gestão de Memória e Performance:

- No geral, o Linux demonstrou uma gestão de memória mais eficiente, com menor consumo de memória física e menos page faults. Essa eficiência foi mais evidente em operações intensivas de I/O e em processos de comunicação entre threads. O Windows, embora tenha registrado mais page faults, manteve a estabilidade dos programas testados, o que evidencia uma robustez na sua gestão de memória.

#### *Discussão sobre o Resultado Final*

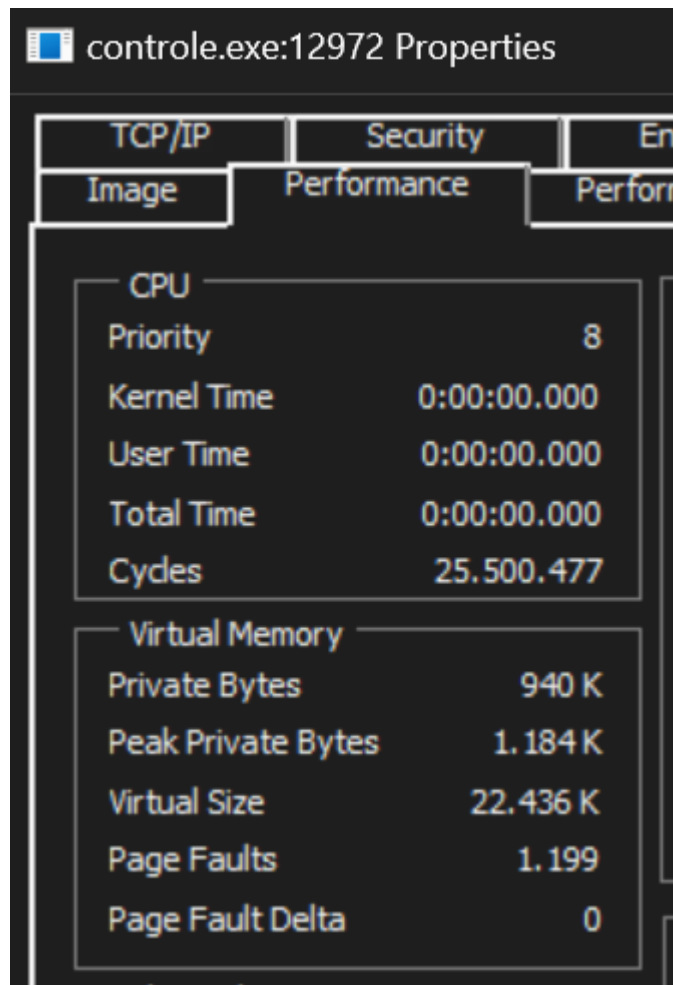
No que diz respeito a operações que exigem uso intensivo de memória, o Linux se mostrou mais eficiente, especialmente em cenários que demandam alto desempenho e uma gestão de memória mais rigorosa. Isso se deve ao fato de que o Linux possui um gerenciamento de memória mais agressivo e otimizado, o que permite um uso mais eficiente dos recursos. Por outro lado, o Windows se destaca por oferecer uma execução mais estável e robusta, sendo uma escolha sólida para situações em que a compatibilidade com diferentes softwares e a facilidade de uso são mais importantes.

Quando comparamos a execução em ambientes como o Codespaces (um ambiente controlado e otimizado) e o Windows, percebemos uma diferença significativa no gerenciamento de memória. No Codespaces, que é baseado em uma infraestrutura gerenciada e com menos interferências de processos externos, o sistema tem um controle mais eficaz sobre os recursos, o que pode resultar em melhor desempenho e menos falhas de página (page faults). No Windows, no entanto, o processo precisa competir pela memória com todos os outros processos do sistema operacional, o que pode afetar negativamente o desempenho, especialmente em tarefas mais intensivas de memória.

Em ambos os sistemas, a adoção de boas práticas de programação, como a otimização do uso de memória e de operações de I/O, pode melhorar significativamente o desempenho, independentemente da plataforma. No entanto, o Linux ainda se destaca para aplicações que exigem máxima eficiência em termos de CPU e memória. Já o Windows é mais indicado para cenários onde a estabilidade e a compatibilidade com uma variedade de softwares são essenciais.

Em resumo, cada sistema tem suas vantagens dependendo do tipo de aplicação e do ambiente em que está sendo executado. O Linux é mais adequado para cargas de trabalho intensivas e que exigem um controle mais preciso dos recursos, enquanto o Windows é uma boa opção para ambientes mais gerais, onde a facilidade de uso e a compatibilidade com diferentes plataformas são prioritárias. Ambos demonstraram ser competentes para gerenciar as tarefas propostas, garantindo uma execução estável, mas com diferentes níveis de eficiência dependendo do cenário.

- Análise do código controle.cpp e controle.py (extra)



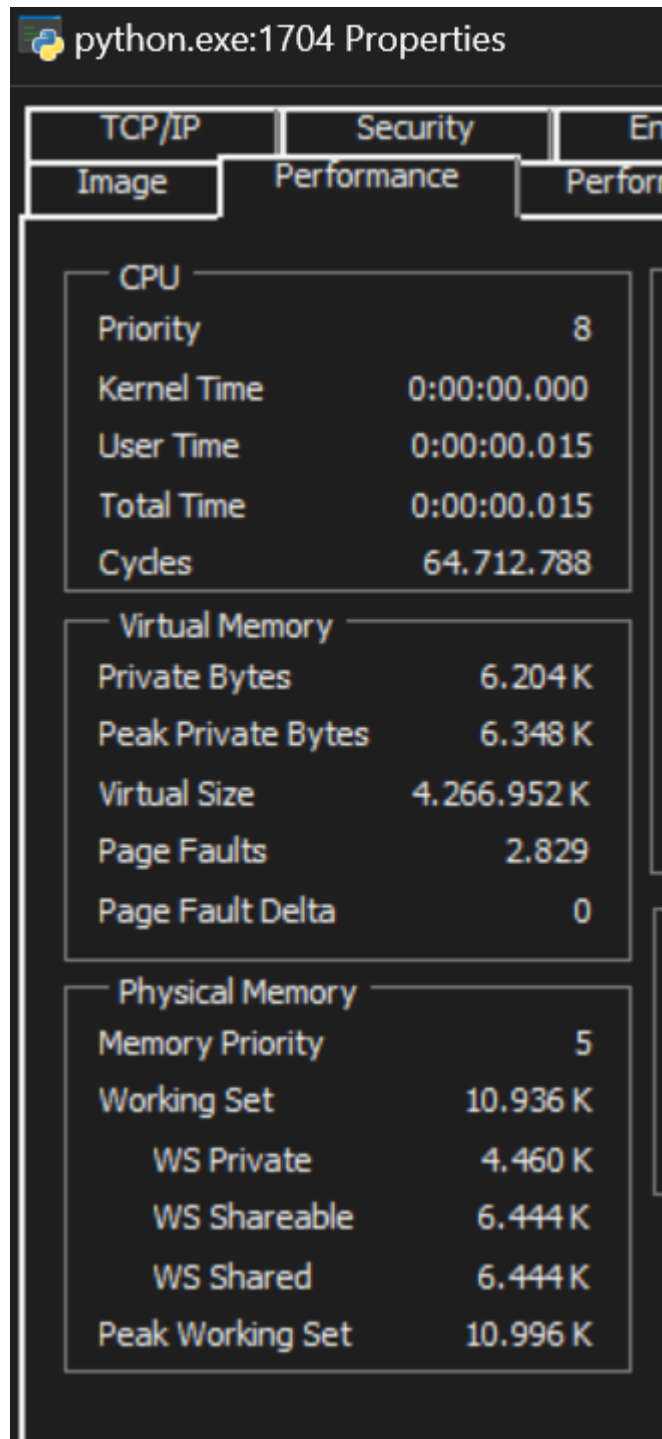
#### Análise no Windows (Process Explorer) - controle.cpp

A execução do programa controle.cpp em C++ no Windows demonstrou um uso moderado de recursos de memória e uma quantidade reduzida de page faults. Os principais resultados foram:

- **Uso de Memória Virtual:** O programa ocupou aproximadamente 22.436 KB de memória virtual e 940 KB de "Private Bytes", indicando uma alocação eficiente e leve de recursos.
- **Page Faults:** Foram registrados 1.199 page faults, todos classificados como minor page faults. A ausência de major page faults indica que o sistema não precisou acessar o disco rígido para trazer páginas para a RAM, usando diretamente o cache ou área de swap.
- **Desempenho Geral:** O programa apresentou um comportamento estável e eficiente em termos de uso de memória, e a quantidade de page faults não impactou negativamente o desempenho.

Esses resultados indicam que o código em C++ foi otimizado para utilizar uma quantidade mínima de memória, gerando poucos page faults, o que sugere uma execução eficiente em ambientes com recursos limitados.

#### Análise no Windows (Process Explorer) - controle.py



Para o programa controle.py, implementado em Python, observamos um uso de memória e page faults um pouco maior em comparação ao código em C++. Os resultados incluem:

- **Uso de Memória Virtual:** controle.py ocupou 4.266.952 KB de memória virtual, com 6.204 KB de "Private Bytes". O uso de memória foi consideravelmente maior que o do código em C++, refletindo a sobrecarga adicional associada à execução de programas em Python.
- **Page Faults:** Foram registrados 2.829 page faults, também todos minor page faults, indicando que as páginas acessadas estavam disponíveis no cache ou swap, sem a necessidade de busca no disco.

- **Desempenho Geral:** O código em Python mostrou um comportamento estável em termos de uso de CPU e memória. Embora o número de page faults tenha sido maior que no código C++, não houve major page faults, e o sistema conseguiu lidar com a execução sem degradação de desempenho.

A diferença de desempenho pode ser atribuída ao fato de que o Python, por ser interpretado e gerenciado automaticamente, utiliza mais recursos de memória do que o C++, o que resulta em mais page faults, embora o impacto tenha sido mínimo.

### **Comparação Geral**

A análise comparativa dos programas controle.cpp (em C++) e controle.py (em Python) evidencia algumas diferenças importantes:

- **Eficiência de Memória:** O código em C++ foi mais eficiente, utilizando significativamente menos memória virtual e "Private Bytes" em comparação ao código Python.
- **Quantidade de Page Faults:** Ambos os programas geraram apenas minor page faults, porém o código em Python apresentou um número maior de page faults, o que é esperado devido à sobrecarga inerente à execução de um interpretador.
- **Desempenho:** Ambos os programas foram gerenciados eficientemente pelo sistema operacional, sem registro de major page faults e mantendo um desempenho fluido.