Sistemas Operacionais
Relatório de Análise de Page Faults

Gabriel Sereia

Outubro de 2025

Sumário

Lista de Figuras

Lista de Tabelas

1 Enunciado do Projeto

O projeto tem como objetivo analisar *page faults* em sistemas operacionais, compreendendo o comportamento da memória durante a execução de processos. Busca-se aplicar na prática os conceitos de gerenciamento de memória e políticas de paginação, observando o impacto das falhas de página no desempenho do sistema.

Um *page fault* ocorre quando um processo acessa uma página que não está presente na memória principal, obrigando o sistema operacional a buscá-la no disco. Isso gera atraso na execução e influencia a eficiência da aplicação e do sistema.

O trabalho utiliza o código base disponibilizado pelo professor (*MemoryCost*) para realizar testes de alocação e acesso à memória, variando parâmetros de alocação e quantidade de threads.

Os resultados serão organizados em gráficos e tabelas para análise comparativa dos cenários, buscando identificar padrões e eficiência das políticas de paginação.

2 Explicação e Contexto da Aplicação

O projeto analisa *page faults*, que representam eventos em que o sistema operacional precisa buscar páginas de memória no disco por não estarem na RAM. Esses eventos impactam diretamente o desempenho de aplicações e do sistema, podendo causar atrasos significativos quando ocorrem com frequência.

A aplicação utilizada (*MemoryCost*) realiza alocação e acesso intensivo à memória, simulando cenários de uso realista de sistemas multi-threaded. A escolha do código base permite observar o comportamento da memória sob diferentes cargas de trabalho, tamanhos de alocação e quantidade de threads, possibilitando a compreensão de como políticas de paginação influenciam a eficiência do sistema.

Também utilizado o Trabalho feito para a M1, que consiste num sistema multi-thread capaz a realizar processamento de imagem em arquivos PMG. A imagem utilizada foi uma de 279x180

Além desses dois projetos, também foi desenvolvido dois códigos com o mesmo comportamento para anlisar desempenho entre duas linguagens, sendo elas Python e JavaScript (Node.Js),

3 Resultados Obtidos

3.1 Usando MemoryCost

Testes de alocação de memória foram realizados com 32MB, 128MB e 512MB.

Tabela 1: Uso de memória e Page Faults

Uso de Memória (MB)	Minor Page Faults	Major Page Faults
32	111645	0
128	222824	0
512	419824	0

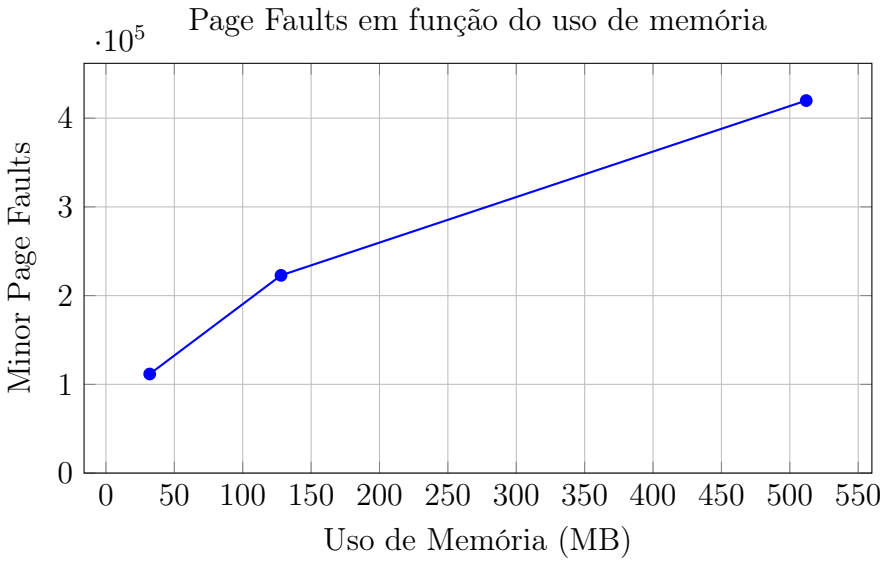

Figura 1: Relação entre uso de memória e número de Page Faults

3.1.1 Trechos de códigos importantes:

```
Timer timer;
for (int i = 0; i < iterationCount; ++i)
{
    int* p = new int[bufSize / sizeof(int)];
    delete[] p;
}
printf("%1.4f␣s␣to␣allocate␣%d␣MB␣%d␣times.\n", timer.GetElapsed(),
    bufSize / (1024 * 1024), iterationCount);
```

Listing 1: Alocação e liberação simples

```
Timer timer;
double deleteTime = 0.0;
for (int i = 0; i < iterationCount; ++i)
{
    int* p = new int[bufSize / sizeof(int)];
    Timer deleteTimer;
    delete[] p;
    deleteTime += deleteTimer.GetElapsed();
}
printf("%1.4f␣s␣to␣allocate␣%d␣MB␣%d␣times␣(%1.4f␣s␣to␣delete).\n",
    timer.GetElapsed(), bufSize / (1024 * 1024), iterationCount,
   deleteTime);
```

Listing 2: Alocação e medição do tempo de liberação

```
int* p = new int[bufSize / sizeof(int)]();

// Escrita repetida
{
    Timer timer;
```

```
    for (int i = 0; i < iterationCount; ++i)
    {
         memset(p, 1, bufSize);
    }
    printf("%1.4f s to write %d MB %d times.\n", timer.GetElapsed()
       , bufSize / (1024 * 1024), iterationCount);
}

// Leitura repetida
{
    Timer timer;
    int sum = 0;
    for (int i = 0; i < iterationCount; ++i)
    {
        for (size_t index = 0; index < bufSize / sizeof(int); ++
           index)
        {
           sum += p[index];
        }
    }
    printf("%1.4f s to read %d MB %d times , sum = %d.\n", timer.
       GetElapsed(), bufSize / (1024 * 1024), iterationCount, sum);
    }

delete[] p;
```

Listing 3: Inicialização, escrita e leitura de memória já alocada

3.2 Usando trabalho M1

Testes com 4, 8 e 12 threads foram realizados usando a imagem do trabalho anterior.

Figura 2: Imagem original utilizada nos testes

Tabela 2: Quantidade de threads e Page Faults

Quantidade de Threads	Minor Page Faults	Major Page Faults
4	16005	0
8	16479	0
12	16958	0

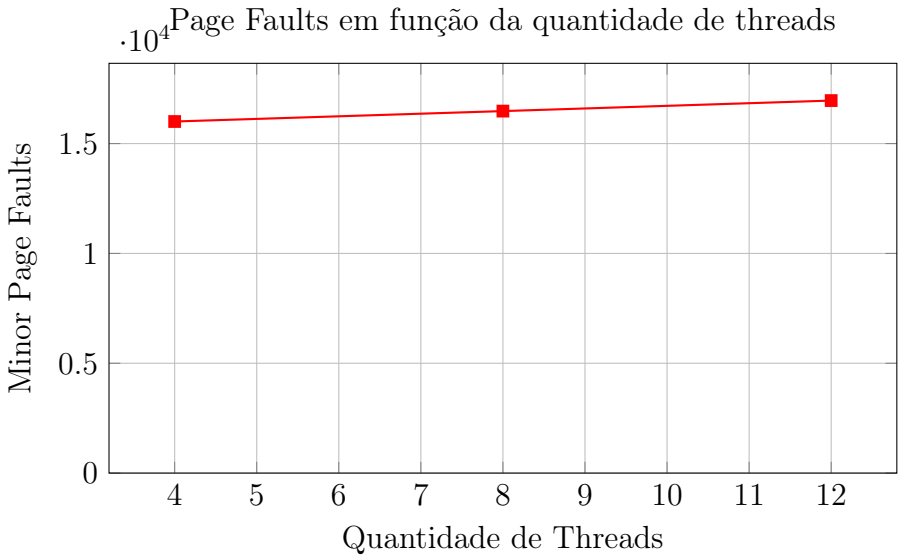

Figura 3: Relação entre número de threads e Page Faults

3.3 Comparação entre linguagens

Código desenvolvido em Python e JavaScript para comparação de alocação de memória.

Tabela 3: Linguagem e Page Faults

Linguagem	Minor Page Faults	Major Page Faults
JavaScript	61535	0
Python	2944	0

3.3.1 Código em Python

```python
import os
import time

data = ["x" * 100 for _ in range(1_000_000)]
print("Criado:", len(data), "itens", os.getpid())

while True:
    time.sleep(1)
```

Listing 4: Alocação de memória em Python

3.3.2 Código em JavaScript (Node.js)

```javascript
const data = Array.from({ length: 1000000 }, () => "x".repeat(100))
console.log("Criado:", data.length, "itens", process.pid);
process.stdin.resume();
```

Listing 5: Alocação de memória em JavaScript

4 Análise e Discussão dos Resultados Finais

Os resultados mostram que o número de *minor page faults* aumenta proporcionalmente ao tamanho da memória alocada e à quantidade de threads, enquanto os *major page faults* permanecem praticamente nulos. Isso indica que a memória RAM disponível foi suficiente para os testes realizados, e que o sistema operacional conseguiu gerenciar eficientemente o mapeamento de páginas.

A comparação entre linguagens evidencia diferenças de gerenciamento de memória: o JavaScript apresentou muito mais *minor page faults* do que Python, possivelmente devido a diferenças na alocação interna e coleta de lixo das linguagens.

Também se observa que o aumento no número de threads provoca apenas um crescimento moderado nos *page faults*, sugerindo que o paralelismo introduzido não causou saturação de memória. Isso confirma que a política de paginação e gerenciamento de memória do sistema operacional está funcionando de forma eficiente para os cenários testados.

Em resumo, a análise permite compreender o impacto da alocação de memória, paralelismo e escolha de linguagem no desempenho do sistema, fornecendo uma base para otimizações futuras em aplicações que exigem uso intensivo de memória.