Sincronização em Sistemas Operacionais

Ana Laura Rizzo Cardosol, Gabriel Fernando de Faria Ferreira, Fernanda Caetano Rodrigues

[1]FACOM – Universidade Federal de Mato Grosso do Sul (UFMS)

Resumo. *O objetivo deste trabalho é realizar e analisar implementações de mecanismos de sincronização preemptivos (Threads) e cooperativos (Corrotinas), utilizando o problema produtor-consumidor (4.1 Producer-Consumer) do livro The Little Book of Semaphores, a fim de avaliar vantagens e desvantagens de cada modelo.*

1. Informações iniciais

Neste trabalho, o problema produtor-consumidor será implementado de duas formas: Com sincronização preempitiva, utilizando Threads na linguagem C++ e com sincronização cooperativa, utilizando Corrotinas implementadas em Python.

1.1. Sincronização Preempitiva

Na sincronização preempitiva, o Sistema Operacional possui o controle de interromper e alternar tarefas, buscando alocar o uso do processador da maneira mais eficiente. Tal sincronização traz grandes melhoras para tarefas CPU-intensivas.

1.2. Sincronização Cooperativa

Na sincronização cooperativa, a própria tarefa deve voluntariamente cooperar com outras, executando parte de seu trabalho até que sinalize para o Sistema Operacional realizar a troca de contexto. Tal sincronização traz grandes benefícios para tarefas com grandes fluxos de entrada e saída.

2. Como foram evitadas condições de corrida e inconsistências

Um dos objetivos do trabalho é realizar ambas as implementações sem condições de corrida e sem o risco do acontecimento de inconsistências. Para realizarmos isso, os seguintes cuidados foram tomados:

2.1. Implementação com Threads

Na implementação com Threads realizada em C++, o mútex de exclusão mútua, buffer mutex, foi utilizado para proteger a região crítica. Com isso, apenas uma thread pode acessar e alterar o buffer de cada vez, evitando condições de corrida.

Além disso, o semáforo items, inicializado com 0, tem a função de contar quantos itens estão disponíveis no buffer, assim, a consumidora apenas irá consumir caso items seja maior que 0, caso seja igual a 0 ela vai esperar até que a produtora adicione um item. Um outro semáforo spaces, inicializado com o tamanho do buffer, também é utilizado para contar os espaços disponíveis no buffer. Esse semáforo tem a função de não deixar a produtora criar mais itens do que o espaço disponível, bloqueando-a quando o buffer está cheio, até que a consumidora consuma um item.

2.2. Implementação com Corrotinas

Na implementação com Corrotinas realizada em Python, a biblioteca asyncio é utilizada, eliminando a necessidade de implementar mutexes e semáforos, pois as estruturas e funções do asyncio já incorporam esses elementos em sua lógica.

O mútex também não é necessário, pois tudo roda em uma única thread, tornando impossível que duas corrotinas acessem a fila ao mesmo tempo (condição de corrida). Já os semáforos são substituídos por asyncio.Queue, que gerencia a espera através da cooperatividade.

3. Diferenças de desempenho

Para analisarmos o desempenho, realizamos um teste focado em I/O-Bound, simulando uma espera (sleep) por recursos.

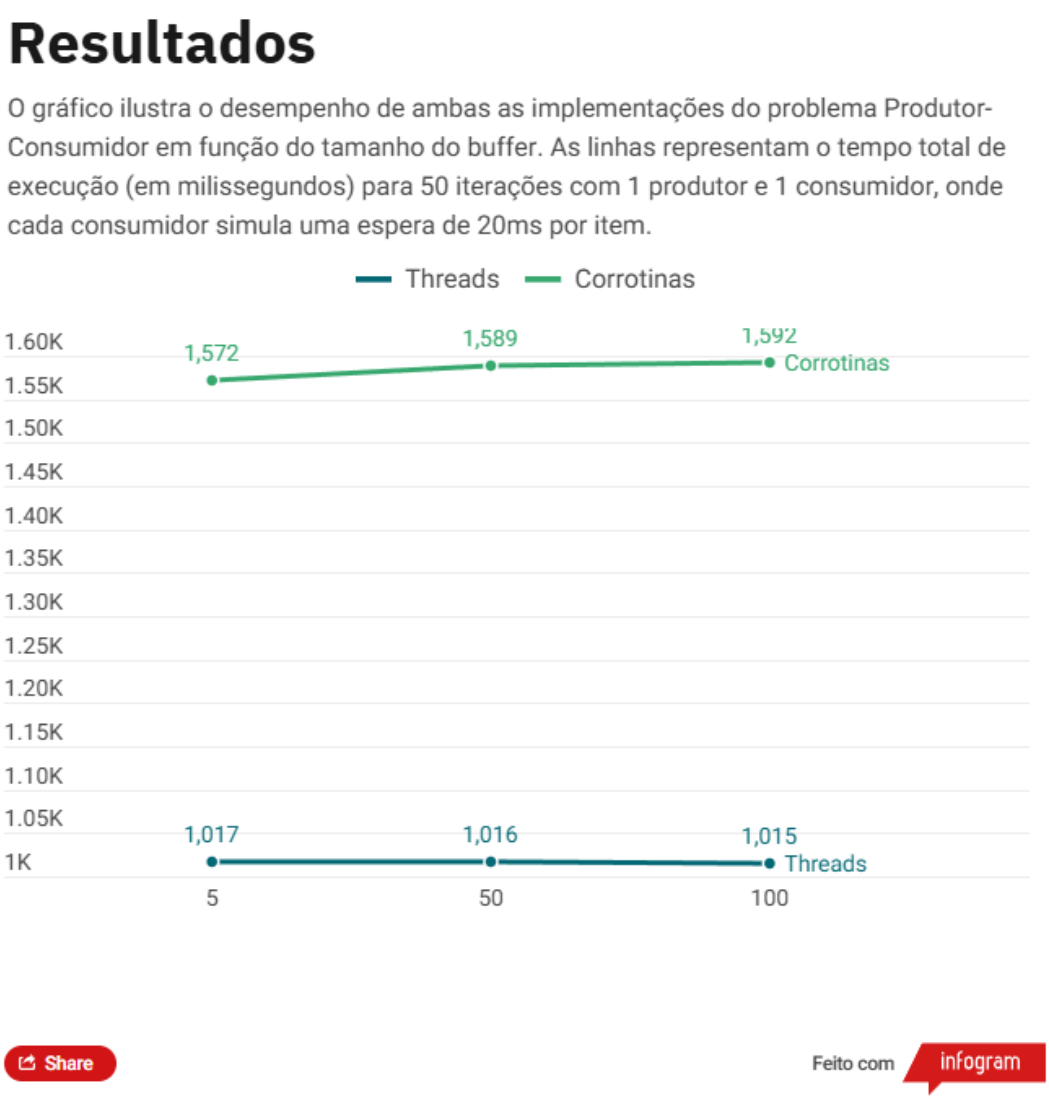

Figure 1. Resultados do teste de desempenho I/O-Bound

Com esse teste podemos observar que o tamanho dos buffers não gerou um grande

impacto nos resultados, deixando evidente as diferenças das implementações e o menor custo para o S.O gerenciar as Threads na versão em C++.

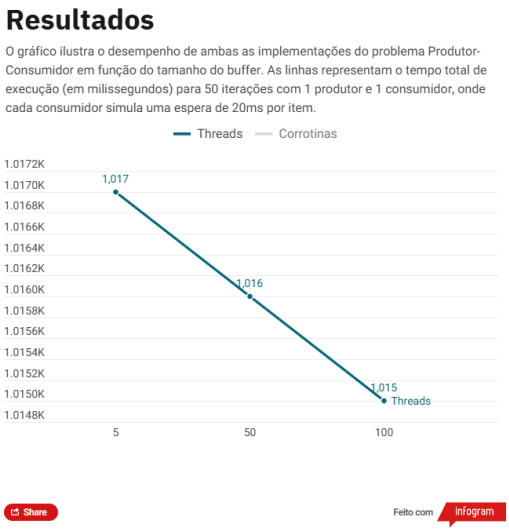

Figure 2. Detalhe do desempenho (Threads)

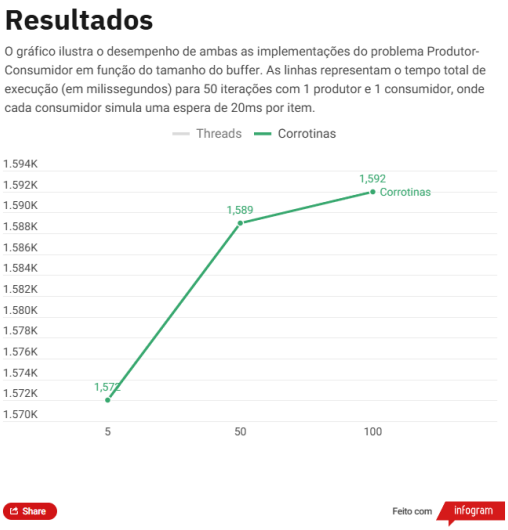

Figure 3. Detalhe do desempenho (Corrotinas)

4. Conclusão

Analisando ambas as implementações, podemos concluir que a versão em C++ (Preemptiva) exige mais atenção em sua implementação, sendo fácil cometer erros e gerar uma condição de corrida devido à necessidade de gerenciar manualmente mutexes e semáforos. Já na versão em Python a implementação é simplificada, com vários recursos já implementados automaticamente.

Nos testes que realizamos a versão em C++ foi mais eficiente devido ao baixo custo das chamadas de sistemas nativas, porém a versão em Python com Corrotinas possui uma grande capacidade para escalar, lidando muito bem com uma maior quantidade de tarefas I/O-Bound.