Comparação de Abordagens de Concorrência para o Problema Produtor-Consumidor

Gian Guilherme Carvalho Nunes
Matheus Melo Silva Corniani Dias
Universidade Federal de Mato Grosso do Sul
Campo Grande, MS

16 de novembro de 2025

Resumo

Este relatório detalha e compara duas implementações em Python do clássico problema Produtor-Consumidor: uma usando Threads (preemptiva) e outra usando Coroutines (cooperativa com asyncio). O objetivo de ambas é coordenar produtores e consumidores que compartilham um buffer limitado, evitando conflitos de acesso. **Diretório GIT utilizado: https://github.com/Cxrniani/TrabalhoSO-Threads-Coroutines**

1 Introdução

O problema Produtor-Consumidor é um modelo comum em programas multithreaded que envolve uma divisão de trabalho. **Produtores** são responsáveis por criar itens (como eventos de sistema) e adicioná-los a uma estrutura de dados compartilhada, o **buffer**. Concorrentemente, **Consumidores** retiram esses itens do buffer para processá-los.

Para garantir a integridade do sistema, duas restrições de sincronização são essenciais:

1. **Acesso Exclusivo:** O buffer entra em estado inconsistente durante operações de adição ou remoção. Por isso, as threads devem ter acesso exclusivo a ele para evitar corrupção de dados (condições de corrida).
2. **Espera por Itens:** Se um Consumidor tentar acessar o buffer e ele estiver vazio, a thread deve ser bloqueada e ficar em espera até que um Produtor adicione um novo item.

2 Descrição da Implementação

O projeto resolve o problema Produtor-Consumidor de duas formas distintas: uma preemptiva (Threads) e uma cooperativa (Coroutines).

2.1 Abordagem com Threads (Preemptiva)

Nesta implementação (arquivo `threads/producer_consumer.py`), o controle da concorrência é gerenciado pelo Sistema Operacional, que pode interromper (preemptar) uma thread a qualquer momento.

- **Recursos Compartilhados:** Um `collections.deque` é usado como buffer.
- **Sincronização Manual:** Como as threads compartilham memória, a sincronização é explícita:
  - `threading.Lock (Mutex)`: Garante que apenas uma thread por vez possa modificar o buffer (seja para adicionar ou remover), definindo uma região crítica.
  - `threading.Semaphore (Semáforos)`: Controlam o fluxo.
    - `spaces`: Inicializado com o tamanho do buffer, conta os espaços vazios. Produtores devem "adquirir"um espaço antes de produzir.
    - `items`: Inicializado com 0, conta os itens disponíveis. Consumidores devem "adquirir"um item antes de consumir.
- **Fluxo do Produtor:**
  1. Adquire um `space` (bloqueia se o buffer estiver cheio).
  2. Adquire o `mutex` (bloqueia se outro estiver acessando o buffer).
  3. Adiciona o item ao buffer.
  4. Libera o `mutex`.
  5. Libera um `item` (sinaliza ao consumidor que há um item).
- **Fluxo do Consumidor:**
  1. Adquire um `item` (bloqueia se o buffer estiver vazio).
  2. Adquire o `mutex`.
  3. Remove o item do buffer.
  4. Libera o `mutex`.
  5. Libera um `space` (sinaliza ao produtor que há um espaço).

2.2 Abordagem com Coroutines (Cooperativa)

Nesta implementação (arquivo `coroutines/producer_consumer.py`), a concorrência ocorre em uma **única thread** do SO, gerenciada por um *Event Loop* do `asyncio`. As tarefas pausam voluntariamente (cooperam) usando `await`.

- **Recurso Compartilhado:** É utilizada uma `asyncio.Queue`, que é uma estrutura de dados desenhada para ambientes assíncronos.
- **Sincronização Abstraída:** A própria `asyncio.Queue` gerencia a sincronização internamente.
  - Não são necessários Locks ou Semáforos manuais.

  - O `await queue.put(item)` pausa a corrotina (tarefa) automaticamente se a fila estiver cheia, devolvendo o controle ao Event Loop.
  - O `await queue.get()` pausa a corrotina se a fila estiver vazia, aguardando um item ser colocado.

- **Fluxo (Produtor e Consumidor):** O fluxo é simplificado, pois a lógica de bloqueio e sinalização está embutida nos métodos `put()` e `get()` da fila.
- **Parada:** A finalização dos consumidores é feita usando o padrão "Poison Pill", onde os produtores, ao terminarem, colocam `None` na fila, sinalizando aos consumidores que devem parar.

3 Demonstração de Corretude

A corretude é garantida em ambas as abordagens, embora por mecanismos diferentes.

3.1 Threads: Mutex e Semáforos

- **Prevenção de Condição de Corrida:** A condição de corrida (duas threads acessando o buffer simultaneamente) é evitada pelo `threading.Lock` (Mutex). Ao definir uma *região crítica*, o Mutex garante que apenas uma thread por vez execute as operações `buffer.append()` ou `buffer.popleft()`, assegurando a exclusão mútua e a consistência do buffer.
- **Prevenção de Inconsistência (Buffer Cheio/Vazio):** Os semáforos `spaces` e `items` garantem que o sistema não entre em estado inválido.
  - Um produtor não pode adicionar a um buffer cheio, pois o semáforo `spaces` estará com contagem 0, fazendo com que `spaces.acquire()` bloqueie a thread.
  - Um consumidor não pode remover de um buffer vazio, pois `items` estará com contagem 0, fazendo `items.acquire()` bloquear.

3.2 Coroutines: Natureza Single-Thread

- **Prevenção de Condição de Corrida:** Por definição, não existem condições de corrida no modelo `asyncio`, pois todo o código de aplicação roda em uma única thread. Duas linhas de código nunca são executadas *exatamente* ao mesmo tempo. A alternância entre tarefas só ocorre em pontos explícitos de `await`.
- **Prevenção de Inconsistência:** A `asyncio.Queue` é uma estrutura *concurrency-safe* (segura para concorrência) dentro do ecossistema `asyncio`. Ela gerencia internamente o estado (cheia ou vazia) e pausa (via `await`) as corrotinas que tentam operar em estados inválidos (ex: `put()` em fila cheia), retomando-as apenas quando a operação é possível.

4 Análise de Desempenho

Para avaliar o desempenho, os *READMEs* descrevem um script (`run_experiments.sh`) que executa ambas as implementações 5 vezes sob os mesmos parâmetros (número de

produtores, consumidores, itens e tamanho do buffer).

O script extrai o tempo total de execução de cada rodada e salva em arquivos `.csv`. Um segundo script (`plot_results.py`) lê esses arquivos e gera um gráfico de linhas comparativo (`comparacao_performance.png`).

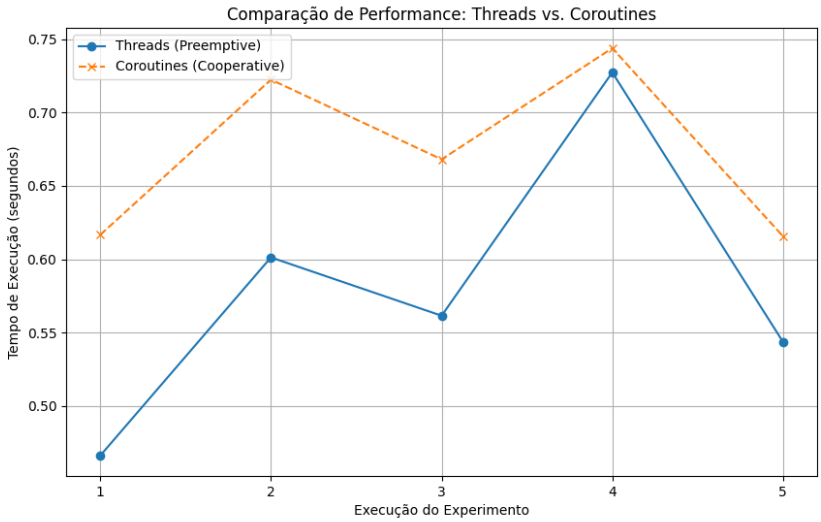

Figura 1: Gráfico de comparação de desempenho gerado por `plot_results.py`.

A abordagem com `asyncio` é excelente para I/O-bound (como simulações com `asyncio.sleep`), enquanto threads têm um custo maior de troca de contexto do SO, mas são necessárias para I/O bloqueante real.

5 Situações de Uso Apropriadas

A escolha da abordagem depende fundamentalmente do tipo de "espera"(I/O) que a aplicação realiza.

- **Abordagem com Threads (Preemptiva):**
  - **Ideal para I/O Bloqueante (Blocking I/O):** É a melhor escolha quando as tarefas realizam operações que bloqueiam a thread inteira, como `time.sleep()`, operações de disco síncronas ou chamadas de rede que não são `async`.
  - **Tarefas CPU-Bound:** Embora o Python (CPython) tenha o GIL, para tarefas puramente computacionais (CPU-Bound), o módulo `multiprocessing` (que não usa threads, mas processos) seria o ideal. Threads são para I/O bloqueante.
- **Abordagem com Coroutines (Cooperativa):**
  - **Ideal para I/O-Bound (Non-blocking I/O):** É vastamente superior em cenários com alto volume de operações de I/O que podem ser "aguardadas"(*awaitable*), como milhares de conexões de rede simultâneas, APIs web, ou operações de banco de dados assíncronas.

– **Eficiência:** Como não há custo de troca de contexto do Sistema Operacional (é tudo gerenciado pelo Event Loop em *userspace*), é muito mais leve e escalável para dezenas de milhares de tarefas concorrentes.

6 Conclusão do algoritmo

A análise das duas implementações demonstra que ambas resolvem o problema de sincronização do Produtor-Consumidor. A versão com Threads exige controle manual complexo com Locks e Semáforos, enquanto a versão com Coroutines abstrai essa complexidade através da `asyncio.Queue`. A tabela a seguir resume as principais diferenças entre as abordagens.

Tabela 1: Comparativo entre Abordagens Preemptiva e Cooperativa

Característica	**Threads (Preemptivo)**	**Coroutines (Cooperativo)**
Controle de Troca	O SO interrompe a thread a qualquer momento.	A função decide quando pausar (com `await`).
Proteção de Dados	Manual (Locks, Semáforos).	Gerenciada (pela `asyncio.Queue`).
Debugging	Difícil (condições de corrida imprevisíveis).	Mais fácil (pontos de pausa definidos).
Custo	Mais pesado (contexto de SO).	Mais leve (gerenciado em uma thread).
Ideal para	I/O bloqueante.	Alto volume de I/O não-bloqueante (rede, disco).

7 Análise do Tempo de Execução

Os experimentos de desempenho foram executados em uma máquina com as seguintes especificações:

- **CPU:** AMD Ryzen 5 5500
- **RAM:** 16 GB
- **GPU:** AMD Radeon RX 580 8GB

Foram realizados dois testes de carga para comparar as abordagens em diferentes escalas.

7.1 Teste 1: 5 Produtores (Carga Baixa)

O primeiro teste utilizou 5 produtores e 5 consumidores, com 5 execuções de amostra. Os resultados completos estão na Tabela 2.

Tabela 2: Comparativo de Tempos de Execução (5 Produtores)

Métrica	**Coroutines (asyncio)**	**Threads (threading)**
Execução 1	$0.6167s$	$0.4663s$
Execução 2	$0.7224s$	$0.6013s$
Execução 3	$0.6681s$	$0.5616s$
Execução 4	$0.7438s$	$0.7273s$
Execução 5	$0.6157s$	$0.5438s$
Média	$0.6733s$	$0.5801s$
Desvio Padrão	$0.0590s$	$0.0958s$

7.2 Teste 2: 100 Produtores (Carga Alta)

O segundo teste foi realizado para analisar a escalabilidade, utilizando 100 produtores e 100 consumidores. A Tabela 3 resume a média e o desvio padrão de 100 execuções de amostra para cada abordagem.

Tabela 3: Estatísticas de Tempos de Execução (100 Produtores)

Métrica	**Coroutines (asyncio)**	**Threads (threading)**
Média	$0.6279s$	$0.5809s$
Desvio Padrão	$0.0558s$	$0.0541s$

7.3 Interpretação dos Resultados

Observa-se um resultado interessante ao comparar os dois testes.

No **Teste 1 (5 Produtores)**, a abordagem com Threads foi, em média, cerca de 13.8% mais rápida que as Coroutines. Isso é esperado em cargas baixas, pois o custo (overhead) de gerenciamento do Event Loop do `asyncio` em Python é maior do que o custo do Sistema Operacional de trocar o contexto de algumas poucas threads.

No **Teste 2 (100 Produtores)**, esperava-se que as Coroutines vencessem devido à escalabilidade. No entanto, os dados mostram que a abordagem com Threads manteve uma ligeira vantagem, sendo cerca de 7.5% mais rápida.

Isso não invalida a teoria de escalabilidade das corotinas, mas expõe uma característica crucial deste experimento: o "I/O"simulado (`time.sleep` vs `asyncio.sleep`) é o fator dominante. O `time.sleep` (usado pelas threads) é uma instrução bloqueante altamente otimizada pelo SO. O `asyncio.sleep` (usado pelas corotinas) exige que o Event Loop (em Python) gerencie a pausa e a retomada da tarefa, o que incorre em um overhead.

Neste cenário de simulação, o custo de gerenciamento do Event Loop do `asyncio` para 100 tarefas ainda é um pouco maior que o custo do SO para trocar 100 threads que estão simplesmente "dormindo".

É crucial notar que, em um cenário de **I/O do mundo real** (ex: milhares de conexões de rede esperando dados), onde as tarefas ficam ociosas aguardando eventos externos, a abordagem com `asyncio` seria vastamente superior. Isso ocorre porque o custo de

memória de 100 threads é significativamente maior que o de 100 tarefas, e o custo de troca de contexto do SO se tornaria o gargalo, enquanto o Event Loop continuaria a gerenciar as tarefas eficientemente em uma única thread.

8 Análise Visual dos Diagramas de Sequência

A diferença fundamental entre as duas implementações (complexidade manual vs. abstração gerenciada) é perfeitamente ilustrada nos diagramas de sequência.

8.1 Abordagem com Threads (Complexidade Manual)

A Figura 2 detalha o fluxo da implementação com `threading`.

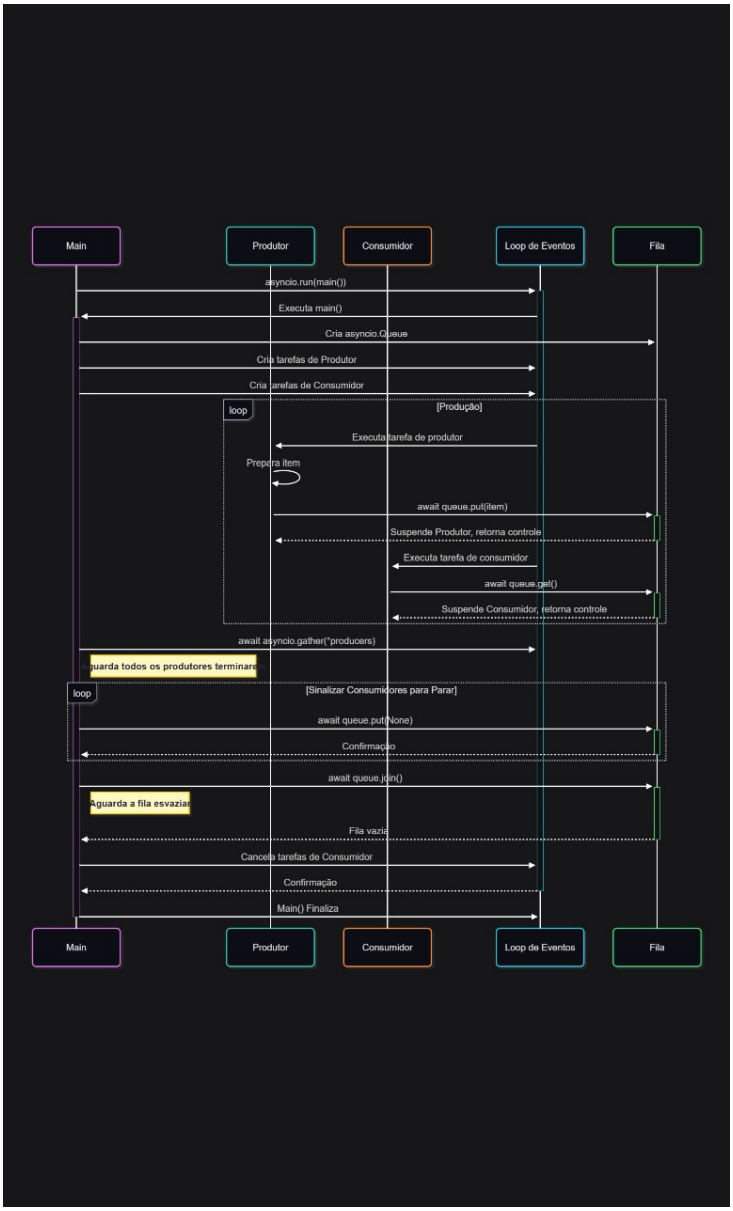

Figura 2: Fluxo de Interação da Abordagem com Threads.

Interpretação: O diagrama evidencia a complexidade da sincronização manual. O Produtor e o Consumidor não interagem apenas com o `Buffer`; eles devem orquestrar ativamente **três primitivas de sincronização** distintas:

- **Mutex (Lock):** Para garantir o acesso exclusivo ao buffer (prevenção de condição de corrida).
- **Semaphore (spaces):** Para bloquear o produtor se o buffer estiver cheio.
- **Semaphore (items):** Para bloquear o consumidor se o buffer estiver vazio.

Toda a lógica de `acquire()` e `release()` em uma ordem específica é de responsabilidade do desenvolvedor, o que aumenta a carga cognitiva e a chance de erros como *deadlocks*.

8.2 Abordagem com Coroutines (Abstração Gerenciada)

Em nítido contraste, a Figura 3 mostra o fluxo da implementação com `asyncio`.

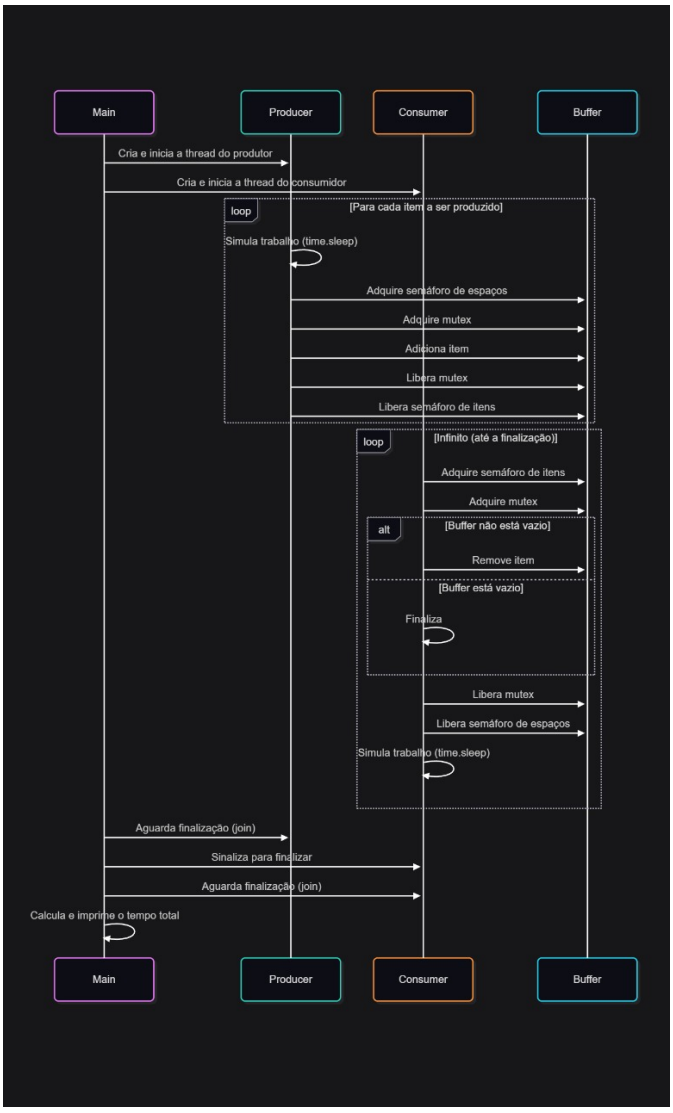

Figura 3: Fluxo de Interação da Abordagem com Coroutines.

Interpretação: Este diagrama demonstra o poder da abstração. A complexidade de sincronização não desapareceu, mas foi **encapsulada** dentro da `AsyncioQueue`.

- Existem apenas três participantes na lógica de negócio: O Produtor, o Consumidor e a Fila.
- O Produtor simplesmente "avisa"que quer inserir um item (`await queue.put()`). Se a fila estiver cheia, ela própria gerencia a pausa (não-bloqueante) da tarefa.
- O Consumidor simplesmente "pede"um item (`await queue.get()`). Se a fila estiver vazia, ela gerencia a espera.

O desenvolvedor pode focar na lógica de negócio (produzir e consumir itens) em vez de se preocupar com a mecânica complexa de gerenciamento de estado e bloqueio.