Problema dos Leitores e Escritores: Controle de Concorrência com Threads e Semáforos

André Kaled Duarte Coutinho Andrade,
Miguel Oliveira Moraes de Souza,
Pedro Henrique Belota Gadelha

[1]Instituto de Computação (IComp) – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio, 6200, Coroado I – 69080-900 – Manaus – AM – Brasil

{andre.duarte, miguel.moraes, pedro.belota}@icomp.ufam.edu.br

1. Introdução

O presente trabalho tem como objetivo implementar e analisar o problema clássico dos **Leitores e Escritores**, um caso fundamental de **concorrência** em Sistemas Operacionais. A proposta busca simular, por meio da linguagem C e da biblioteca `pthread`, o comportamento de múltiplas threads acessando um recurso compartilhado de forma simultânea, garantindo a integridade dos dados através de **semáforos** e mecanismos de sincronização.

O sistema desenvolvido representa um cenário onde diferentes entidades (leitores e escritores) acessam um mesmo recurso no caso, um portal com registros de informações. Os leitores têm permissão para realizar consultas simultâneas, enquanto os escritores possuem acesso exclusivo durante as operações de escrita. Essa abordagem evita condições de corrida e demonstra a importância da sincronização em sistemas multitarefa.

Além de compreender o funcionamento técnico, o trabalho visa reforçar os conceitos teóricos de exclusão mútua, sincronização e comunicação entre threads, aplicando-os em um contexto prático e didático. Os resultados obtidos permitem visualizar a diferença entre execução concorrente com e sem controle, destacando o papel essencial dos semáforos na coordenação do acesso a recursos compartilhados.

2. Leitores x Escritores

2.1. Versão 1: Leitores com Prioridade

2.1.1. Estrutura de Sincronização

O recurso compartilhado é o `Portal do Aluno`, que contém os registros dos estudantes (`RegistroAluno`). O controle de concorrência é feito por meio de um **mutex** e um **semáforo binário**:

- **Semáforo `recurso`** (Valor 1): Controla o acesso à região crítica. Garante exclusão mútua entre escritores e bloqueia escritores quando há leitores ativos.
- **Mutex `mtx_leitores`**: Protege a variável `leitores_ativos`, que conta quantos leitores estão atualmente lendo.

2.1.2. Lógica de Execução

Comportamento do Leitor (`portal_ler`):

1. Bloqueia o mutex e incrementa o contador de leitores ativos.
2. Se for o primeiro leitor, realiza `sem_wait(&recurso)` para impedir que escritores entrem.
3. Libera o mutex e entra na seção crítica (realiza leitura).
4. Ao sair, bloqueia o mutex novamente, decrementa o contador e, se for o último leitor, libera o recurso com `sem_post(&recurso)`.

2.1.3. Resultados

Nesta versão, leitores têm prioridade total. Isso garante que múltiplos leitores acessem o portal simultaneamente sem bloqueios, otimizando a concorrência para consultas. Entretanto, escritores podem sofrer **starvation**, isto é, podem nunca conseguir escrever caso haja uma sequência contínua de leitores.

Os logs de execução demonstram que diversas threads leitoras acessam o portal em paralelo, enquanto escritores aguardam a liberação completa dos leitores ativos. A integridade dos dados é mantida, porém o tempo de resposta dos escritores é maior.

Exemplo de log obtido na execução:

```
Iniciando execucao...
ENTRADA VIA ARQUIVO ESCOLHIDO
ARQUIVO VALIDADO COM SUCESSO!
Alunos lidos: 12

--- Alunos carregados do arquivo (12) ---
Aluno [0]: ID=1 | Nota=10.00 | Faltas=0
Aluno [1]: ID=2 | Nota=9.50 | Faltas=0
Aluno [2]: ID=3 | Nota=9.00 | Faltas=0
Aluno [3]: ID=4 | Nota=8.50 | Faltas=0
Aluno [4]: ID=5 | Nota=8.00 | Faltas=0
Aluno [5]: ID=6 | Nota=7.50 | Faltas=0
Aluno [6]: ID=7 | Nota=7.00 | Faltas=1
Aluno [7]: ID=8 | Nota=6.50 | Faltas=0
Aluno [8]: ID=9 | Nota=6.00 | Faltas=0
Aluno [9]: ID=10 | Nota=5.50 | Faltas=0
Aluno [10]: ID=11 | Nota=5.00 | Faltas=0
Aluno [11]: ID=12 | Nota=4.50 | Faltas=0
----------------------------------------
[Portal] Portal iniciado com sucesso!
[L-06] leu RA=4 nota=8.5 faltas=0
[E-02] escreveu RA=4 nova_nota=8.5 delta_faltas=3
[L-05] leu RA=1 nota=10.0 faltas=0
[L-07] leu RA=7 nota=7.0 faltas=1
[E-01] escreveu RA=1 nova_nota=10.0 delta_faltas=1
```

```
[E-04] escreveu RA=10  nova_nota=6.5  delta_faltas=4
[L-08] leu RA=10  nota=6.5  faltas=4
[E-03] escreveu RA=7  nova_nota=7.0  delta_faltas=3
[L-05] leu RA=2  nota=9.5  faltas=0
[E-02] escreveu RA=5  nova_nota=7.0  delta_faltas=2
[E-01] escreveu RA=2  nova_nota=8.5  delta_faltas=4
[E-04] escreveu RA=11  nova_nota=6.0  delta_faltas=2
[L-06] leu RA=5  nota=7.0  faltas=2
[L-05] leu RA=3  nota=9.0  faltas=0
[L-07] leu RA=8  nota=6.5  faltas=0
[L-08] leu RA=11  nota=6.0  faltas=2
[E-03] escreveu RA=8  nova_nota=6.5  delta_faltas=4
[E-02] escreveu RA=6  nova_nota=6.5  delta_faltas=4
[L-06] leu RA=6  nota=6.5  faltas=4
[L-08] leu RA=12  nota=4.5  faltas=0
[E-01] escreveu RA=3  nova_nota=9.0  delta_faltas=1
[E-04] escreveu RA=12  nova_nota=4.5  delta_faltas=4
[L-07] leu RA=9  nota=6.0  faltas=0
[E-03] escreveu RA=9  nova_nota=6.0  delta_faltas=3
```

Pode-se observar que múltiplas leituras ocorrem em sequência antes de cada escrita, caracterizando a prioridade dos leitores. Nenhum conflito de dados é detectado, confirmando a correta aplicação da exclusão mútua.

2.2. Versão 2: Escritores com Prioridade

O recurso compartilhado é o **Portal do Aluno**, que contém registros de alunos (`RegistroAluno`). As threads leitoras (alunos) consultam a nota e faltas, e as threads escritoras (professores) atualizam esses dados.

A solução utiliza uma combinação de **semáforos e um mutex** para implementar a sincronização. Diferente da prioridade de leitor, que é o modelo mais comum, esta versão assegura que escritores não sofram *starvation* devido a um fluxo contínuo de leitores.

- **Semáforo `recurso`** (Valor 1): Garante a exclusão mútua na seção crítica. É responsável por bloquear todos os leitores e todos os outros escritores enquanto um escritor estiver ativo, ou enquanto houver leitores ativos (desde que o escritor ainda não tenha conseguido acessar a `catraca`).
- **Semáforo `turnstile`** (Catraca, Valor 1): Este semáforo é a chave da prioridade. Quando um escritor se aproxima, ele tenta obter o `catraca`, impedindo que novos leitores entrem no sistema de contagem e aguardem na catraca.
- **Mutex `mtx_leitores`**: Protege a variável contadora `leitores_ativos`.
- **Contador `leitores_ativos`**: Conta o número de threads leitoras atualmente na seção crítica.

2.2.1. Lógica de Execução

Comportamento do Escritor (`portal_escrever`): O escritor solicita a prioridade tomando o `turnstile` antes de tentar acessar o recurso.

1. `sem_wait(&turnstile)`: Bloqueia a catraca, impedindo que novas threads leitoras cheguem ao `recurso`.
2. `sem_wait(&recurso)`: Aguarda a liberação de todos os leitores/escritores atualmente ativos.
3. Acessa a Região Crítica (atualiza nota e faltas).
4. `sem_post(&recurso)`: Libera o recurso.
5. `sem_post(&turnstile)`: Libera a catraca para novos leitores/escritores.

A ordem de aquisição dos semáforos é fundamental para garantir que, caso um escritor esteja esperando pelo `recurso`, ele force os leitores subsequentes a esperarem por ele na `catraca`.

Comportamento do Leitor (`portal_ler`): O leitor deve respeitar a catraca antes de tentar contar sua chegada.

1. `sem_wait(&turnstile); sem_post(&turnstile)`: Se um escritor estiver esperando pelo recurso, este passo bloqueia o leitor, garantindo a prioridade do escritor.
2. O leitor entra na seção crítica se for o primeiro, ou passa livremente se já houver outros leitores ativos.
3. O último leitor a sair libera o `recurso` com `sem_post(&recurso)`.

2.2.2. Resultados

Esta implementação garante que não ocorra a **leitura suja** e resolve o problema de starvation dos escritores, que seria comum na Versão 1 (Prioridade de Leitor). Os logs de execução (`logger_log`) confirmam que threads leitoras que chegam após a espera de um escritor são bloqueadas na "catraca" (`turnstile`) até que o escritor termine sua operação exclusiva.

Os resultados mostraram que o mecanismo de Prioridade de Escritor funcionou ao impedir que novos leitores entrassem na fila, mas a concorrência ao sistema ainda permite que leitores acessem dados potencialmente desatualizados. Por exemplo, a thread Leitora L-07 acessou o registro RA=9 e depois, a thread Escritora E-03 escreveu o mesmo registro, a ordem das operações confirma que o leitor acessou o dado pouco antes de sua atualização, podendo ser devido ao Escalonador.

Log de execução:

```
Iniciando execucao...
ENTRADA VIA ARQUIVO ESCOLHIDO
ARQUIVO VALIDADO COM SUCESSO!
Alunos lidos: 12

--- Alunos carregados do arquivo (12) ---
Aluno [0]: ID=1 | Nota=10.00 | Faltas=0
Aluno [1]: ID=2 | Nota=9.50 | Faltas=0
Aluno [2]: ID=3 | Nota=9.00 | Faltas=0
Aluno [3]: ID=4 | Nota=8.50 | Faltas=0
Aluno [4]: ID=5 | Nota=8.00 | Faltas=0
```

```
Aluno [5]: ID=6 | Nota=7.50 | Faltas=0
Aluno [6]: ID=7 | Nota=7.00 | Faltas=1
Aluno [7]: ID=8 | Nota=6.50 | Faltas=0
Aluno [8]: ID=9 | Nota=6.00 | Faltas=0
Aluno [9]: ID=10 | Nota=5.50 | Faltas=0
Aluno [10]: ID=11 | Nota=5.00 | Faltas=0
Aluno [11]: ID=12 | Nota=4.50 | Faltas=0
----------------------------------------
[Portal] Portal iniciado com sucesso!
[E-04] escreveu RA=10 nova_nota=5.5 delta_faltas=4
[E-01] escreveu RA=1 nova_nota=10.0 delta_faltas=3
[E-02] escreveu RA=4 nova_nota=9.5 delta_faltas=1
[L-08] leu RA=10 nota=5.5 faltas=4
[L-05] leu RA=1 nota=10.0 faltas=3
[E-03] escreveu RA=7 nova_nota=8.0 delta_faltas=1
[L-06] leu RA=4 nota=9.5 faltas=1
[L-07] leu RA=7 nota=8.0 faltas=2
[E-04] escreveu RA=11 nova_nota=6.0 delta_faltas=3
[E-01] escreveu RA=2 nova_nota=9.5 delta_faltas=2
[L-08] leu RA=11 nota=6.0 faltas=3
[L-07] leu RA=8 nota=6.5 faltas=0
[E-02] escreveu RA=5 nova_nota=9.0 delta_faltas=4
[L-05] leu RA=2 nota=9.5 faltas=2
[E-03] escreveu RA=8 nova_nota=7.5 delta_faltas=2
[L-06] leu RA=5 nota=9.0 faltas=4
[E-04] escreveu RA=12 nova_nota=5.5 delta_faltas=0
[L-07] leu RA=9 nota=6.0 faltas=0
[L-08] leu RA=12 nota=5.5 faltas=0
[E-01] escreveu RA=3 nova_nota=10.0 delta_faltas=3
[L-05] leu RA=3 nota=10.0 faltas=3
[E-02] escreveu RA=6 nova_nota=8.5 delta_faltas=3
[L-06] leu RA=6 nota=8.5 faltas=3
[E-03] escreveu RA=9 nova_nota=5.0 delta_faltas=3
```

2.3. Versão 3: Sem Controle de Concorrência

Nesta versão, o sistema foi propositalmente implementado **sem nenhum tipo de controle de concorrência**, servindo como demonstração prática do que ocorre quando múltiplas threads acessam e modificam simultaneamente um mesmo recurso compartilhado o `Portal do Aluno`.

Nenhum `mutex` ou `semáforo` é utilizado. Assim, tanto leitores quanto escritores podem acessar a estrutura de dados ao mesmo tempo, o que inevitavelmente leva a condições de corrida e inconsistências nos registros.

2.3.1. Lógica de Execução

Como não há sincronização:

- Leitores podem acessar o registro enquanto escritores estão modificando-o, gerando a chamada **leitura suja**;
- Dois escritores podem tentar atualizar o mesmo registro simultaneamente, e o último a escrever sobrescreve o valor do outro;
- A ordem de execução torna-se totalmente dependente do escalonador de threads do sistema operacional, tornando o comportamento não-determinístico.

Essa abordagem é essencial para evidenciar a necessidade de mecanismos de controle. Ao comparar com as versões 1 e 2, é possível perceber claramente os efeitos da ausência de exclusão mútua e sincronização.

2.3.2. Resultados

Os resultados obtidos evidenciam um padrão de execução desordenado, no qual operações de leitura e escrita ocorrem de forma intercalada, sem qualquer sincronização. Um exemplo claro desse comportamento ocorre com o aluno de **RA 9**: o leitor acessa a nota *6.0* enquanto as faltas ainda permanecem em *0*, mas, imediatamente após, o escritor atualiza o mesmo registro para *faltas = 4*. Isso demonstra que a leitura ocorreu durante a modificação dos dados — um caso típico de **leitura suja**. Embora o sistema aparente funcionar corretamente à primeira vista, a ausência de controle de concorrência resulta em perda de consistência lógica nas informações.

Log de execução:

```
Iniciando execucao...
ENTRADA VIA ARQUIVO ESCOLHIDO
ARQUIVO VALIDADO COM SUCESSO!
Alunos lidos: 12

--- Alunos carregados do arquivo (12) ---
Aluno [0]: ID=1 | Nota=10.00 | Faltas=0
Aluno [1]: ID=2 | Nota=9.50 | Faltas=0
Aluno [2]: ID=3 | Nota=9.00 | Faltas=0
Aluno [3]: ID=4 | Nota=8.50 | Faltas=0
Aluno [4]: ID=5 | Nota=8.00 | Faltas=0
Aluno [5]: ID=6 | Nota=7.50 | Faltas=0
Aluno [6]: ID=7 | Nota=7.00 | Faltas=1
Aluno [7]: ID=8 | Nota=6.50 | Faltas=0
Aluno [8]: ID=9 | Nota=6.00 | Faltas=0
Aluno [9]: ID=10 | Nota=5.50 | Faltas=0
Aluno [10]: ID=11 | Nota=5.00 | Faltas=0
Aluno [11]: ID=12 | Nota=4.50 | Faltas=0
----------------------------------------
[Portal] Portal iniciado com sucesso!
[L-05] leu RA=1 nota=10.0 faltas=0
[E-02] escreveu RA=4 nova_nota=9.5 delta_faltas=4
[L-07] leu RA=7 nota=7.0 faltas=1
[E-04] escreveu RA=10 nova_nota=4.5 delta_faltas=1
```

```
[L-08] leu RA=10  nota=4.5  faltas=1
[E-03] escreveu RA=7  nova_nota=7.0  delta_faltas=4
[E-01] escreveu RA=1  nova_nota=11.0  delta_faltas=4
[L-06] leu RA=4  nota=9.5  faltas=4
[L-06] leu RA=5  nota=8.0  faltas=0
[L-05] leu RA=2  nota=9.5  faltas=0
[L-07] leu RA=8  nota=6.5  faltas=0
[E-02] escreveu RA=5  nova_nota=8.0  delta_faltas=0
[E-04] escreveu RA=11  nova_nota=4.0  delta_faltas=3
[E-03] escreveu RA=8  nova_nota=7.5  delta_faltas=4
[L-08] leu RA=11  nota=4.0  faltas=3
[E-01] escreveu RA=2  nova_nota=10.5  delta_faltas=3
[L-06] leu RA=6  nota=7.5  faltas=0
[E-04] escreveu RA=12  nova_nota=5.5  delta_faltas=2
[L-07] leu RA=9  nota=6.0  faltas=0
[E-02] escreveu RA=6  nova_nota=7.5  delta_faltas=1
[L-05] leu RA=3  nota=9.0  faltas=0
[L-08] leu RA=12  nota=5.5  faltas=2
[E-03] escreveu RA=9  nova_nota=6.0  delta_faltas=4
[E-01] escreveu RA=3  nova_nota=8.0  delta_faltas=0
```

3. Processos Produtores x Consumidores

Nesta seção, abordamos o problema clássico dos **Produtores e Consumidores**. O cenário consiste em um buffer circular de tamanho fixo, compartilhado entre threads produtoras, que inserem itens, e threads consumidoras, que os removem. A sincronização é essencial para evitar que produtores insiram itens em um buffer cheio e que consumidores tentem remover itens de um buffer vazio, garantindo a integridade dos dados e a coordenação entre as threads.

A lógica de sincronização para as versões controladas é a mesma, utilizando três semáforos para gerenciar o acesso ao buffer: `vagas`, `itens` e `mutex`.

3.1. Versão 1: Múltiplos Produtores e 1 Consumidor

Esta versão testa um cenário de afunilamento (*bottleneck*), onde múltiplos produtores podem encher o buffer mais rapidamente do que o único consumidor consegue esvaziá-lo. O objetivo é observar o comportamento do sistema sob essa condição de desequilíbrio e verificar se os produtores são corretamente bloqueados quando o buffer atinge sua capacidade máxima.

3.1.1. Lógica de Execução

A dinâmica de execução é centrada na contenção pelo acesso ao buffer.

- **Threads Produtoras**: Múltiplas threads produtoras operam em paralelo. Cada uma tenta adquirir uma vaga no buffer decrementando o semáforo `vagas`. Como há muitos produtores, é esperado que o buffer se encha rapidamente. Uma vez que o buffer está cheio (o contador de `vagas` é zero), todas as threads produtoras que

tentarem inserir novos itens serão bloqueadas na chamada `sem_wait(&vagas)`, aguardando a liberação de espaço.

- **Thread Consumidora**: A única thread consumidora opera de forma contínua. Ao remover um item, ela executa `sem_post(&vagas)`, incrementando o contador de vagas e, consequentemente, liberando uma das threads produtoras que estava bloqueada. Isso cria um ciclo onde o consumidor dita o ritmo da produção.

Essa interação demonstra a eficácia dos semáforos em gerenciar um desequilíbrio entre a oferta (produção) e a demanda (consumo), garantindo que a produção pause quando o limite de capacidade é atingido.

3.1.2. Resultados

A implementação se mostrou correta. Os logs de execução demonstram que, quando o buffer fica cheio, as threads produtoras aguardam o consumidor liberar espaço. O consumidor único opera continuamente, e o sistema permanece estável e sem perda de dados, mesmo com a alta contenção por parte dos produtores.

Log de execução:

```
Produtor 1 -> produziu 42 em 0
[buffer] 42 -1 -1 -1 -1
Produtor 3 -> produziu 88 em 1
[buffer] 42 88 -1 -1 -1
Produtor 2 -> produziu 15 em 2
[buffer] 42 88 15 -1 -1
Produtor 1 -> produziu 77 em 3
[buffer] 42 88 15 77 -1
Produtor 3 -> produziu 65 em 4
[buffer] 42 88 15 77 65
Consumidor 1 <- pegou 42 de 0
[buffer] -1 88 15 77 65
Produtor 2 -> produziu 31 em 0
[buffer] 31 88 15 77 65
Consumidor 1 <- pegou 88 de 1
[buffer] 31 -1 15 77 65
Produtor 1 -> produziu 55 em 1
[buffer] 31 55 15 77 65
...
```

3.2. Versão 2: Múltiplos Produtores e Vários Consumidores

Esta versão implementa o cenário de concorrência total, onde múltiplas threads produtoras e múltiplas threads consumidoras competem pelo acesso ao buffer. Este é o caso de uso mais genérico e testa a robustez da solução de exclusão mútua, garantindo que, mesmo com vários atores, o estado do buffer permaneça consistente.

3.2.1. Lógica de Execução

Neste cenário, a concorrência ocorre em duas frentes: entre os produtores para preencher o buffer e entre os consumidores para esvaziá-lo.

- **Concorrência de Produtores**: Múltiplas threads produtoras competem por vagas disponíveis, bloqueando em `sem_wait(&vagas)` se o buffer estiver cheio. Além disso, o semáforo `mutex` garante que apenas uma produtora por vez possa acessar e modificar o buffer e o ponteiro de inserção, evitando que duas threads escrevam na mesma posição simultaneamente.
- **Concorrência de Consumidores**: De forma análoga, múltiplas threads consumidoras competem por itens disponíveis, bloqueando em `sem_wait(&itens)` se o buffer estiver vazio. O mesmo `mutex` impede que mais de uma consumidora acesse o buffer ao mesmo tempo, prevenindo que duas threads leiam e tentem remover o mesmo item.

O `mutex` é o mecanismo central que serializa o acesso ao buffer, garantindo que, a qualquer momento, apenas uma única thread — seja ela produtora ou consumidora — esteja dentro da região crítica. Isso é fundamental para manter a consistência dos dados e dos ponteiros do buffer em um ambiente com alta disputa por recursos.

3.2.2. Resultados

O sistema operou conforme o esperado. A combinação dos semáforos `vagas`, `itens` e `mutex` foi eficaz para coordenar todas as threads, prevenindo condições de corrida e deadlocks. As operações de inserção e remoção ocorreram de forma harmônica, com threads sendo bloqueadas e liberadas corretamente conforme o estado do buffer (cheio ou vazio).

Log de execução:

```
Produtor 1 -> colocou 42 em 0
[buffer] 42 -1 -1 -1 -1
Consumidor 1 <- pegou 42 de 0
[buffer] -1 -1 -1 -1 -1
Produtor 2 -> colocou 88 em 1
[buffer] -1 88 -1 -1 -1
Produtor 1 -> colocou 15 em 2
[buffer] -1 88 15 -1 -1
Consumidor 2 <- pegou 88 de 1
[buffer] -1 -1 15 -1 -1
Consumidor 1 <- pegou 15 de 2
[buffer] -1 -1 -1 -1 -1
...
```

3.3. Versão 3: Sem Controle de Concorrência

Nesta versão, todos os mecanismos de sincronização (semáforos) foram removidos para demonstrar os problemas inerentes ao acesso concorrente descontrolado a um recurso compartilhado, conforme solicitado.

3.3.1. Lógica de Execução

As threads produtoras e consumidoras acessam diretamente o buffer e as variáveis de índice `pos_inserir` e `pos_remover` sem qualquer tipo de bloqueio ou verificação. Múltiplas threads podem ler e escrever nessas variáveis simultaneamente, levando a um comportamento caótico e não determinístico.

3.3.2. Resultados

A ausência de controle de concorrência resulta em uma série de problemas críticos, conhecidos como **condições de corrida** (*race conditions*). Os logs de execução evidenciam claramente esses problemas:

- **Sobrescrita de Itens**: Um produtor pode inserir um item em uma posição, mas antes que o índice `pos_inserir` seja atualizado, outro produtor insere um item na mesma posição, fazendo com que o primeiro item seja perdido.
- **Consumo Repetido**: Um consumidor pode ler um item, mas antes de atualizar o índice `pos_remover`, outro consumidor lê o mesmo item da mesma posição.
- **Inconsistência dos Índices**: Os índices de inserção e remoção perdem a sincronia, fazendo com que o buffer se comporte de maneira imprevisível, pulando posições ou sobrescrevendo áreas que ainda contêm dados válidos.

O exemplo de log abaixo ilustra uma condição de corrida, onde o Produtor 2 insere o item 35 na posição 1, mas é imediatamente seguido pelo Produtor 1, que insere o item 99 na mesma posição 1 antes que o consumidor tenha a chance de ler o item 35. O dado original do Produtor 2 foi perdido.

Log de execução:

```
[SEM CTRL] Produtor 1 -> 24 em 0
[buffer] 24 -1 -1 -1 -1
[SEM CTRL] Consumidor 1 <- 24 de 0
[buffer] -1 -1 -1 -1 -1
[SEM CTRL] Produtor 2 -> 35 em 1
[buffer] -1 35 -1 -1 -1
[SEM CTRL] Produtor 1 -> 99 em 1
[buffer] -1 99 -1 -1 -1
[SEM CTRL] Consumidor 2 <- 99 de 1
[buffer] -1 -1 -1 -1 -1
...
```

Esta versão comprova a necessidade absoluta de mecanismos de sincronização em ambientes concorrentes para garantir a corretude e a previsibilidade do programa.