EQUIPE: Lucas de Souza Cerveira Pereira, Mikaelle Costa de Santana e Pedro Gabriel Motta Vieira

USO DE IA: Tivemos breve apoio do Gemini para ajudar na escolha do tema da primeira parte e para decidir qual método usar pro controle (semáforo ou mutex), de modo que, através dos prompts, pudemos entender melhor o funcionamento de cada.

OBS: Para o entendimento do assunto como um todo e familiarização com os códigos, foram utilizados o conteúdo das aulas, sobretudo os slides da matéria.

Mecanismo de controle de concorrência

- Escolha: Mutex
- Justificativa:
    - **Semântica mais clara** – Indica explicitamente que apenas uma thread por vez pode acessar a seção crítica, evitando erros e facilitando a leitura do código.
    - **Mais eficiente** – As threads bloqueadas dormem até o recurso ser liberado, evitando consumo desnecessário de CPU.
    - **Controle seguro** – Apenas a thread que bloqueou o *mutex* pode liberá-lo, prevenindo liberações incorretas.
    - **Implementação simples** – O uso de `lock` e unlock é direto e fácil de entender, sem necessidade de manipular contadores.
    - **Bom desempenho** – Ideal para exclusão mútua simples, com baixo overhead e suporte nativo em bibliotecas padrão como *pthread*.
    - **Facilidade de manutenção** – O propósito do *mutex* é claro, o que torna o código mais legível e fácil de depurar.

Tema da parte 1:

Para a parte 1 foi escolhido o tema e-commerce, onde as threads leitoras são consultas dos consumidores sobre o estoque de um determinado produto e as threads escritoras são compras desse produto por outros consumidores.

Comentário extra da equipe

Foi necessário passar os argumentos para as threads a partir de structs, pois o pthread_create só pode passar uma (void *) struct como argumento para a função da thread

1.1 - Leitores sem controle (leitura suja)

Foi utilizado mutex apenas nas threads compradoras, o que garante que o estoque final seja o correto, porém as threads leitoras podem ler valores errados no espaço de tempo que a venda está sendo “processada”. Para simular esse processamento do pagamento foi utilizado sleep(1) logo depois da declaração da venda no terminal, nesse momento as threads de leitura podem ler o valor do estoque e mostra-lo desatualizadamente no terminal

Exemplo de execução:

Cenário: e-commerce onde as vendas estão protegidas pelo mutex mas as consultas não.

```
------------
Digite a quantidade de produtos no estoque inicial: 10
Digite a quantidade de threads de Consulta (leitoras): 3
Digite a quantidade de threads de Venda (escritoras): 2
Digite a quantidade de itens que CADA vendedor irá vender: 2

--- Iniciando simulação ---
Estoque inicial: 10

[VENDA 1]: Thread iniciada. Tentará vender 2 itens.
[VENDA 1]: Aguardando para entrar na região crítica...

[VENDA 1]: ENTROU NA REGIÃO CRÍTICA. O estoque neste momento é: 10
[VENDA 1]: Estoque suficiente. Processando a venda...
[VENDA 1]: QUANTIDADE APÓS A VENDA: 8
[VENDA 2]: Thread iniciada. Tentará vender 2 itens.
[VENDA 2]: Aguardando para entrar na região crítica...
[VENDA 1]: DORMINDO por 2 segundos ANTES de efetivar a baixa no estoque.
Leitores podem ver valores antigos agora!
[CONSULTA 1]: Iniciada. Fará 2 leituras para monitorar o estoque ao longo da
simulação.
   [CONSULTA 1]: Estoque lido: 10
[CONSULTA 2]: Iniciada. Fará 2 leituras para monitorar o estoque ao longo da
simulação.
   [CONSULTA 2]: Estoque lido: 10
[CONSULTA 3]: Iniciada. Fará 2 leituras para monitorar o estoque ao longo da
simulação.
   [CONSULTA 3]: Estoque lido: 10
```

[VENDA 1]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 8
[VENDA 1]: SAINDO da sua região crítica.

[VENDA 2]: ENTROU NA REGIÃO CRÍTICA. O estoque neste momento é: 8
[VENDA 2]: Estoque suficiente. Processando a venda...
[VENDA 2]: QUANTIDADE APÓS A VENDA: 6
[VENDA 2]: DORMINDO por 2 segundos ANTES de efetivar a baixa no estoque.
Leitores podem ver valores antigos agora!
  [CONSULTA 1]: Estoque lido: 8
  [CONSULTA 2]: Estoque lido: 8
  [CONSULTA 3]: Estoque lido: 8

[VENDA 2]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 6
[VENDA 2]: SAINDO da sua região crítica.

[CONSULTA 1]: Finalizada.
[CONSULTA 3]: Finalizada.
[CONSULTA 2]: Finalizada.

--- Simulação finalizada ---
Estoque inicial: 10
Estoque final: 6

Observe se alguma thread de consulta leu o valor errado após uma thread de venda marcar que ele (o valor) seria mudado após uma venda entrar em sua região crítica.

==========
Note que o estoque final está correto (consequência do uso de mutex nas threads escritoras) porém, há múltiplos casos em que as leitoras leem valores desatualizados, como visto nesse trecho:

[VENDA 2]: ENTROU NA REGIÃO CRÍTICA. O estoque neste momento é: 8
[VENDA 2]: Estoque suficiente. Processando a venda...
[VENDA 2]: QUANTIDADE APÓS A VENDA: 6
[VENDA 2]: DORMINDO por 2 segundos ANTES de efetivar a baixa no estoque.
Leitores podem ver valores antigos agora!
  [CONSULTA 1]: Estoque lido: 8 **< - - - -**
  [CONSULTA 2]: Estoque lido: 8 **< - - - -**
  [CONSULTA 3]: Estoque lido: 8 **< - - - -**

1.2 - Ambos com controle de concorrência

Foi utilizado mutex nos dois tipos de thread, o que garante que todas as threads só poderão acessar o estoque quando tiverem o controle exclusivo sobre elas, assim não ocorre leitura suja nem compras não registradas. Apesar do sleep(1) para simular o processamento, nenhuma thread leitora mostra um valor errado no terminal.

Exemplo de execução:

Cenário: e-commerce onde as vendas e as consultas do estoque estão protegidas pelo mutex.

```
----------
Digite a quantidade de produtos no estoque inicial: 10
Digite a quantidade de threads de Consulta (leitoras): 3
Digite a quantidade de threads de Venda (escritoras): 2
Digite a quantidade de itens que CADA vendedor irá vender: 2

--- Iniciando simulação ---
Estoque inicial: 10

[CONSULTA 3]: Iniciada. Tentará consultar a quantidade disponível no estoque.
[CONSULTA 3]: Aguardando para entrar na região crítica...

[CONSULTA 3]: ENTROU NA REGIÃO CRÍTICA.
   [CONSULTA 3]: Lendo o estoque...
[VENDA 2]: Thread iniciada. Tentará vender 2 itens.
[VENDA 2]: Aguardando para entrar na região crítica...
[CONSULTA 2]: Iniciada. Tentará consultar a quantidade disponível no estoque.
[CONSULTA 2]: Aguardando para entrar na região crítica...
[VENDA 1]: Thread iniciada. Tentará vender 2 itens.
[VENDA 1]: Aguardando para entrar na região crítica...
[CONSULTA 1]: Iniciada. Tentará consultar a quantidade disponível no estoque.
[CONSULTA 1]: Aguardando para entrar na região crítica...
   [CONSULTA 3]: Estoque lido: 10
[CONSULTA 3]: SAINDO da sua região crítica.

[CONSULTA 3]: Finalizada.

[VENDA 2]: ENTROU NA REGIÃO CRÍTICA. O estoque neste momento é: 10
   [VENDA 2]: Estoque suficiente. Processando a venda...
   [VENDA 2]: QUANTIDADE APÓS A VENDA: 8
   [VENDA 2]: DORMINDO por 2 segundos ANTES de efetivar a baixa no estoque.
Leitores poderiam ver valores antigos agora se não tivesse o mutex!

   [VENDA 2]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 8
```

[VENDA 2]: SAINDO da sua região crítica.

[CONSULTA 2]: ENTROU NA REGIÃO CRÍTICA.
   [CONSULTA 2]: Lendo o estoque...
   [CONSULTA 2]: Estoque lido: 8
[CONSULTA 2]: SAINDO da sua região crítica.

[CONSULTA 2]: Finalizada.

[VENDA 1]: ENTROU NA REGIÃO CRÍTICA. O estoque neste momento é: 8
   [VENDA 1]: Estoque suficiente. Processando a venda...
   [VENDA 1]: QUANTIDADE APÓS A VENDA: 6
   [VENDA 1]: DORMINDO por 2 segundos ANTES de efetivar a baixa no estoque.
Leitores poderiam ver valores antigos agora se não tivesse o mutex!

   [VENDA 1]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 6
[VENDA 1]: SAINDO da sua região crítica.

[CONSULTA 1]: ENTROU NA REGIÃO CRÍTICA.
   [CONSULTA 1]: Lendo o estoque...
   [CONSULTA 1]: Estoque lido: 6
[CONSULTA 1]: SAINDO da sua região crítica.

[CONSULTA 1]: Finalizada.

--- Simulação finalizada ---
Estoque inicial: 10
Estoque final: 6

Observe se alguma thread de consulta leu o valor errado após uma thread de venda marcar que ele (o valor) seria mudado.

==========
Note também que absolutamente nenhuma thread começa a realizar a sua função no momento em que há alguma outra thread utilizando o estoque.

1.3 - Sem controle de concorrência

Para esse código não foi implementado nenhum mutex, isso tem o objetivo de mostrar a condição de corrida. A condição de corrida é caracterizada pela “não contagem” de registros, pois uma thread pega o valor não atualizado e sobreescreve em cima do que uma outra thread acabou de escrever. Para esse

código não foram implementadas threads de consulta, pois o problema descrito ocorre somente em threads escritoras (vendedoras no nosso cenário)

Exemplo de execução:

Tema: e-commerce
Cenário: múltiplas thread de vendas (escritoras) sem nenhum controle de concorrência.
Isto irá demonstrar uma CONDIÇÃO DE CORRIDA (race condition).

Digite a quantidade de produtos no estoque inicial: 10
Digite a quantidade de threads de Venda (use cerca de 10 para produzir resultados menos monotonos): 3
Digite a quantidade de itens que CADA vendedor irá vender: 2

--- Iniciando simulação ---
Estoque inicial: 10

[VENDA 1]: Thread iniciada. Tentará vender 2 itens.
[VENDA 1]: Entrou na região crítica SEM PROTEÇÃO (SEM MUTEX).
[VENDA 1]: Leu o estoque. Valor: 10
    [VENDA 1]: Estoque suficiente. Processando a baixa...
    [VENDA 1]: QUANTIDADE APÓS A VENDA DEVERIA SER: 8
    [VENDA 1]: DORMINDO, race condition provavelmente ocorrerá agora
[VENDA 2]: Thread iniciada. Tentará vender 2 itens.
[VENDA 2]: Entrou na região crítica SEM PROTEÇÃO (SEM MUTEX).
[VENDA 2]: Leu o estoque. Valor: 10
    [VENDA 2]: Estoque suficiente. Processando a baixa...
    [VENDA 2]: QUANTIDADE APÓS A VENDA DEVERIA SER: 8
    [VENDA 2]: DORMINDO, race condition provavelmente ocorrerá agora
    [VENDA 1]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 8
[VENDA 1]: Saindo da região crítica SEM PROTEÇÃO (SEM MUTEX).
[VENDA 3]: Thread iniciada. Tentará vender 2 itens.
[VENDA 3]: Entrou na região crítica SEM PROTEÇÃO (SEM MUTEX).
[VENDA 3]: Leu o estoque. Valor: 8
    [VENDA 3]: Estoque suficiente. Processando a baixa...
    [VENDA 3]: QUANTIDADE APÓS A VENDA DEVERIA SER: 6
    [VENDA 3]: DORMINDO, race condition provavelmente ocorrerá agora
    [VENDA 2]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 8
[VENDA 2]: Saindo da região crítica SEM PROTEÇÃO (SEM MUTEX).
    [VENDA 3]: Venda de 2 itens CONCLUÍDA. Estoque atualizado para: 6
[VENDA 3]: Saindo da região crítica SEM PROTEÇÃO (SEM MUTEX).

--- Simulação finalizada ---

Estoque inicial era: 10
Estoque final: 6
Estoque final esperado: 4

Se o estoque final real for diferente do esperado, a condição de corrida ocorreu e vendas foram 'perdidas'.

==========
Note a seguinte perda, a venda 1 será ignorada pela venda 2:

[VENDA 1]: Leu o estoque. Valor: 10
  [VENDA 1]: Estoque suficiente. Processando a baixa...
  [VENDA 1]: QUANTIDADE APÓS A VENDA DEVERIA SER: 8 **< - - - -**
  [VENDA 1]: DORMINDO, race condition provavelmente ocorrerá agora
[VENDA 2]: Thread iniciada. Tentará vender 2 itens.
[VENDA 2]: Entrou na região crítica SEM PROTEÇÃO (SEM MUTEX).
[VENDA 2]: Leu o estoque. Valor: 10
  [VENDA 2]: Estoque suficiente. Processando a baixa...
  [VENDA 2]: QUANTIDADE APÓS A VENDA DEVERIA SER: 8 **< - - - -**

Tema da parte 2:

Este tema lida com a coordenação de threads Produtoras (que adicionam dados) e Consumidoras (que retiram dados) em um buffer compartilhado (vetor de tamanho fixo). O programa ilustra o funcionamento do sistema, mostrando o "status" de cada thread ("dormindo", "produzindo", etc.)

2.1 - Vários processos produtores e 1 consumidor

Para a implementação, foram utilizados mecanismos de Variáveis de Condição combinados com o Mutex para garantir a exclusão mútua e o controle de fluxo. Dessa forma, as threads Produtoras são bloqueadas se o buffer estiver cheio e a única thread Consumidora é bloqueada se o buffer estiver vazio, focando na concorrência da produção e no controle das condições de limite do buffer.

OBS: O sleep(1) foi intencionalmente posicionado após a alteração do dado no buffer e antes da atualização do estado final (`count` e `in`/`out`). Esta escolha simula o tempo de processamento da transação, forçando o Mutex a manter o acesso exclusivo. Isso evita a race condition e garante que a thread conclua a alteração do estado compartilhado de forma atômica.

Exemplo de Execução:

```
--- Simulacao Produtores - 1 Consumidor (com Mutex + Var. de Condicao) ---
Tamanho do Buffer: 10
Digite a quantidade de threads Produtoras: 2
Digite a quantidade de itens que CADA Produtor ira gerar: 1

Total de itens a ser consumido (por 1 consumidor): 2

--- Iniciando simulacao ---
Buffer Inicial:
  [BUFFER]: [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
        -> Inserção (in): 0
        -> Remoção (out): 0
        -> Itens no buffer (count): 0
[CONSUMIDOR 1]: Status: Tentando ENTRAR na RC. Verificando se buffer esta
vazio.
[CONSUMIDOR 1]: Status: Buffer VAZIO. Indo DORMIR (cond_wait).
[PRODUTOR 1]: Status: Tentando ENTRAR na RC. Verificando se buffer esta cheio.
[PRODUTOR 1]: Status: PRODUZINDO/Dentro da RC. Inserindo item 39 na pos 0
  [BUFFER]: [ 39, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
        -> Inserção (in): 1
        -> Remoção (out): 0
        -> Itens no buffer (count): 1
[PRODUTOR 1]: Status: SAINDO da RC. Sinalizando que buffer nao esta vazio
(cond_signal).
[PRODUTOR 1]: Status: FINALIZADO.
[PRODUTOR 2]: Status: Tentando ENTRAR na RC. Verificando se buffer esta cheio.
[PRODUTOR 2]: Status: PRODUZINDO/Dentro da RC. Inserindo item 31 na pos 1
  [BUFFER]: [ 39, 31, -1, -1, -1, -1, -1, -1, -1, -1 ]
        -> Inserção (in): 2
        -> Remoção (out): 0
        -> Itens no buffer (count): 2
[PRODUTOR 2]: Status: SAINDO da RC. Sinalizando que buffer nao esta vazio
(cond_signal).
[PRODUTOR 2]: Status: FINALIZADO.
[CONSUMIDOR 1]: Status: ACORDOU. Verificando condicao novamente.
[CONSUMIDOR 1]: ENTROU na RC. Consumiu item 39 na pos 0
  [BUFFER]: [ -1, 31, -1, -1, -1, -1, -1, -1, -1, -1 ]
        -> Inserção (in): 2
        -> Remoção (out): 1
        -> Itens no buffer (count): 1
```

[CONSUMIDOR 1]: Status: SAINDO da RC. Sinalizando que buffer nao esta cheio (cond_signal).
[CONSUMIDOR 1]: Status: Tentando ENTRAR na RC. Verificando se buffer esta vazio.
[CONSUMIDOR 1]: ENTROU na RC. Consumiu item 31 na pos 1
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        -> Inserção (in): 2
        -> Remoção (out): 2
        -> Itens no buffer (count): 0
[CONSUMIDOR 1]: Status: SAINDO da RC. Sinalizando que buffer nao esta cheio (cond_signal).
[CONSUMIDOR 1]: Status: FINALIZADO.

--- Simulacao finalizada ---
Total de itens produzidos/consumidos: 2

==========

É importante ressaltar que, assim que o Consumidor entra na Região Crítica, ele verifica a condição de limite (`count` é 0) e, logo após ver que o buffer estava vazio, é bloqueado:

[CONSUMIDOR 1]: Status: Tentando ENTRAR na RC. Verificando se buffer esta vazio.
[CONSUMIDOR 1]: Status: Buffer VAZIO. Indo DORMIR (cond_wait).

==========

Além disso, a concorrência entre os Produtores é resolvida pelo Mutex. Mesmo que o Produtor 2 inicie logo após o Produtor 1, ele só pode entrar na Região Crítica quando o Mutex for liberado, garantindo que o item 39 não seja sobrescrito.

O log mostra a inserção sequencial dos itens 39 e 31, com o buffer sendo atualizado corretamente a cada passo:

[PRODUTOR 1]: ENTROU na RC. Inserindo item 39 na pos 0
  [BUFFER]: [39, -1, -1, -1, -1, -1, -1, -1, -1, -1]
[PRODUTOR 2]: Status: Tentando ENTRAR na RC. Verificando se buffer esta cheio.
[PRODUTOR 2]: Status: PRODUZINDO/Dentro da RC. Inserindo item 31 na pos 1
  [BUFFER]: [39, 31, -1, -1, -1, -1, -1, -1, -1, -1]

Após o Produtor 2 finalizar e sinalizar, o Consumidor 1 é acordado e assume o controle do Mutex para processar os dois itens.

O Consumidor processa os dois itens em sequência, garantindo que o consumo total seja igual à produção total:

[CONSUMIDOR 1]: Status: ACORDOU. Verificando condicao novamente.
[CONSUMIDOR 1]: ENTROU na RC. Consumiu item 39 na pos 0
...
[CONSUMIDOR 1]: ENTROU na RC. Consumiu item 31 na pos 1
…

2.2 - Vários processos produtores e vários consumidores

Para esta execução, os principais detalhes da implementação anterior foram mantidos. As alterações se concentraram na função main, visto que, por se tratarem de múltiplas threads produtoras e consumidoras, algumas variáveis tiveram seus nomes e valores alterados, principalmente a variável de consumidores, visto que agora o usuário deve indicar a quantidade de consumidores. Além disso, há a implementação da divisão da carga de trabalho. Como há múltiplos consumidores, o programa primeiro calcula o total de itens a ser produzido. Em seguida, esse total é dividido pelo número de threads consumidoras para determinar o valor total a consumir por thread.

Exemplo de execução:

--- Simulacao Produtores - Consumidores ---
Tamanho do Buffer: 10
Digite a quantidade de threads Produtoras: 2
Digite a quantidade de threads Consumidoras: 2
Digite a quantidade de itens que CADA Produtor ira gerar: 1

Total de itens a ser produzido: 2
Total de itens a ser consumido POR CONSUMIDOR: 1

--- Iniciando simulacao ---
Buffer Inicial:
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        -> Inserção (in): 0
        -> Remoção (out): 0
        -> Itens no buffer (count): 0
[CONSUMIDOR 1]: Status: Tentando ENTRAR na RC. Verificando se buffer esta vazio.
[CONSUMIDOR 1]: Status: Buffer VAZIO. Indo DORMIR (cond_wait).

[CONSUMIDOR 2]: Status: Tentando ENTRAR na RC. Verificando se buffer esta vazio.
[CONSUMIDOR 2]: Status: Buffer VAZIO. Indo DORMIR (cond_wait).
[PRODUTOR 1]: Status: Tentando ENTRAR na RC. Verificando se buffer esta cheio.
[PRODUTOR 1]: Status: PRODUZINDO/Dentro da RC. Inserindo item 62 na pos 0
  [BUFFER]: [62, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        -> Inserção (in): 1
        -> Remoção (out): 0
        -> Itens no buffer (count): 1
[PRODUTOR 1]: Status: SAINDO da RC. Sinalizando que buffer nao esta vazio (cond_signal).
[PRODUTOR 1]: Status: FINALIZADO.
[CONSUMIDOR 1]: Status: ACORDOU. Verificando condicao novamente.
[CONSUMIDOR 1]: ENTROU na RC. Consumiu item 62 na pos 0
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        -> Inserção (in): 1
        -> Remoção (out): 1
        -> Itens no buffer (count): 0
[CONSUMIDOR 1]: Status: SAINDO da RC. Sinalizando que buffer nao esta cheio (cond_signal).
[CONSUMIDOR 1]: Status: FINALIZADO.
[PRODUTOR 2]: Status: Tentando ENTRAR na RC. Verificando se buffer esta cheio.
[PRODUTOR 2]: Status: PRODUZINDO/Dentro da RC. Inserindo item 97 na pos 1
  [BUFFER]: [-1, 97, -1, -1, -1, -1, -1, -1, -1, -1]
        -> Inserção (in): 2
        -> Remoção (out): 1
        -> Itens no buffer (count): 1
[PRODUTOR 2]: Status: SAINDO da RC. Sinalizando que buffer nao esta vazio (cond_signal).
[PRODUTOR 2]: Status: FINALIZADO.
[CONSUMIDOR 2]: Status: ACORDOU. Verificando condicao novamente.
[CONSUMIDOR 2]: ENTROU na RC. Consumiu item 97 na pos 1
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        -> Inserção (in): 2
        -> Remoção (out): 2
        -> Itens no buffer (count): 0
[CONSUMIDOR 2]: Status: SAINDO da RC. Sinalizando que buffer nao esta cheio (cond_signal).
[CONSUMIDOR 2]: Status: FINALIZADO.

--- Simulacao finalizada ---
Total de itens produzidos/consumidos: 2

==========

Observe que, assim como na Versão 2.1, ambas as threads Consumidoras tentam acessar o buffer vazio e são corretamente bloqueadas em suas respectivas filas de espera:

[CONSUMIDOR 1]: Status: Buffer VAZIO. Indo DORMIR (cond_wait).
[CONSUMIDOR 2]: Status: Buffer VAZIO. Indo DORMIR (cond_wait).

==========

Além disso, o Produtor 1 insere o item 62 e envia o sinal (cond_signal). A thread que estava dormindo (consumidor 1 ou consumidor 2) é acordada.

Observa-se que o consumidor 1 é o primeiro a ser escalonado, readquirir o Mutex e consumir o item 62. O consumidor 2 só será acordado na próxima sinalização:

[PRODUTOR 1]: Status: SAINDO da RC. Sinalizando que buffer nao esta vazio (cond_signal).
[CONSUMIDOR 1]: Status: ACORDOU. Verificando condicao novamente.
[CONSUMIDOR 1]: ENTROU na RC. Consumiu item 62 na pos 0

O log mostra que o consumo é distribuído:

- O `[CONSUMIDOR 1]` consome o item 62 e FINALIZA (`total_a_consumir = 1`).
- O `[PRODUTOR 2]` insere o item 97 e sinaliza.
- O `[CONSUMIDOR 2]` (o único restante) é acordado, consome o item 97 e FINALIZA (também com `total_a_consumir = 1`).

2.3 - Buffer sem controle de concorrência

Não foi implementado nenhum controle de concorrência nesse buffer a fim de demonstrar a necessidade do mutex para combater as race conditions.

== Simulação de race condition com buffer ==
Digite a quantidade de threads produtoras: 2
Digite a quantidade de threads consumidoras: 1
Digite a quantidade de itens que cada produtor irá gerar: 1

```
== Iniciando simulação ==
Buffer Inicial:
  [BUFFER]: [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 0, Remoção (out): 0

[PRODUTOR 1]: Status: ativo e tentando entrar na região crítica.
[PRODUTOR 1]: Planejando inserir na posição 0. Estado atual do buffer:
  [BUFFER]: [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 0, Remoção (out): 0
[PRODUTOR 1]: Status: dormindo.

[PRODUTOR 2]: Status: ativo e tentando entrar na região crítica.
[PRODUTOR 2]: Planejando inserir na posição 0. Estado atual do buffer:
  [BUFFER]: [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 0, Remoção (out): 0
[PRODUTOR 2]: Status: dormindo.

[CONSUMIDOR 1]: Status: ativo e tentando entrar na região crítica.
[CONSUMIDOR 1]: Planejando consumir da posição 0. Estado atual do buffer:
  [BUFFER]: [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 0, Remoção (out): 0
[CONSUMIDOR 1]: Status: dormindo.
[PRODUTOR 1]: Status: acordou. Retomando produção.
[PRODUTOR 1]: Status: produzindo . Inseriu item 47 na posição 0. Estado NOVO
do buffer:
  [BUFFER]: [ 47, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 0, Remoção (out): 0
[PRODUTOR 1]: Status: finalizado e saindo da região crítica.
[PRODUTOR 2]: Status: acordou. Retomando produção.
== SOBRESCRITA DETECTADA: Posição 0 já continha 47, será sobrescrito por 94
==
[PRODUTOR 2]: Status: produzindo . Inseriu item 94 na posição 0. Estado NOVO
do buffer:
  [BUFFER]: [ 94, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 1, Remoção (out): 0
[PRODUTOR 2]: Status: finalizado e saindo da região crítica.
[CONSUMIDOR 1]: Status: acordou. Retomando consumo.
[CONSUMIDOR 1]: Status: consumindo. Consumiu item 94 da posição 0.
[CONSUMIDOR 1]: Finalizou consumo na posição 0. Estado NOVO do buffer:
  [BUFFER]: [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
    -> Inserção (in): 1, Remoção (out): 0
[CONSUMIDOR 1]: Status: finalizado e saindo da região crítica.
```

[CONSUMIDOR 1]: Status: ativo e tentando entrar na região crítica.
[CONSUMIDOR 1]: Planejando consumir da posição 1. Estado atual do buffer:
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
    -> Inserção (in): 1, Remoção (out): 1
[CONSUMIDOR 1]: Status: dormindo.
[CONSUMIDOR 1]: Status: acordou. Retomando consumo.
 == CONSUMO INVÁLIDO DETECTADO: [CONSUMIDOR 1] tentou ler da posição 1 que já estava vazia ==
[CONSUMIDOR 1]: Finalizou consumo na posição 1. Estado NOVO do buffer:
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
    -> Inserção (in): 1, Remoção (out): 1
[CONSUMIDOR 1]: Status: finalizado e saindo da região crítica.

== Relatório final de corrupção ==
Total de itens que deveriam ter sido produzidos: 2
Contador global de itens produzidos (incorreto devido à concorrência): 2
Contador global de itens consumidos (incorreto): 2

Número de vezes que um dado foi explicitamente sobrescrito: 1
Itens que restaram no buffer: 0
Total de itens perdidos (Esperado - Consumidos - Restantes): 1

Estado final do buffer:
  [BUFFER]: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
    -> Inserção (in): 1, Remoção (out): 2

==========
Note como a produção do PRODUTOR 1 foi “ignorada” pelo PRODUTOR 2 aqui:

[PRODUTOR 1]: Status: produzindo . Inseriu item 47 na posição 0. Estado NOVO do buffer:
  [BUFFER]: [47, -1, -1, -1, -1, -1, -1, -1, -1, -1] **< - - - -**
    -> Inserção (in): 0, Remoção (out): 0
[PRODUTOR 1]: Status: finalizado e saindo da região crítica.
[PRODUTOR 2]: Status: acordou. Retomando produção.
== SOBRESCRITA DETECTADA: Posição 0 já continha 47, será sobrescrito por 94 ==
[PRODUTOR 2]: Status: produzindo . Inseriu item 94 na posição 0. Estado NOVO do buffer:
  [BUFFER]: [94, -1, -1, -1, -1, -1, -1, -1, -1, -1] **< - - - -**
    -> Inserção (in): 1, Remoção (out): 0