Atividade - Resolução de Race Condition

Integrantes:

Júlio Cesar Braga Parro - 16879560

Alberto Galhego Neto - 17019141

Revisão do código anterior

O código original utiliza um contador global sendo incrementado por duas Threads (A e B) independentes, de mesma prioridade e tempo de espera, usando, também, uma função auxiliar que simula o tempo de execução.

Tal código resulta em uma contagem duplicada no contador, sendo: A: 1, B: 1, A: 2, B: 2, A: 3, B: 3…, no caso base gerado pela IA. O erro ocorre em toda a execução do código, desde o começo.

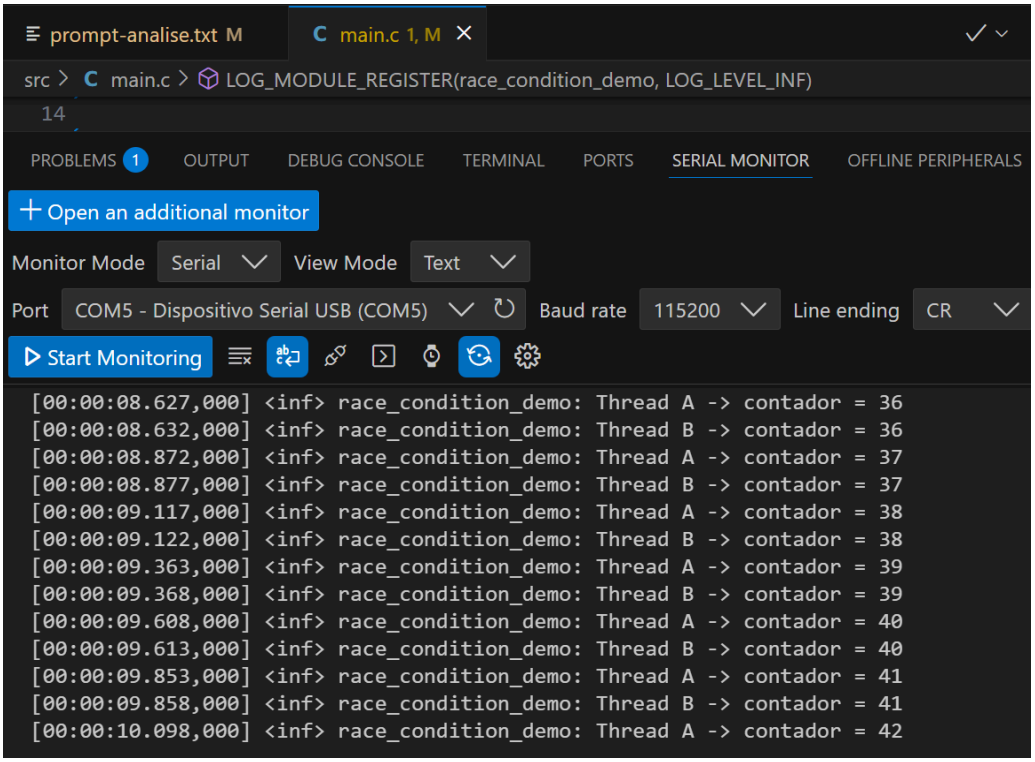
Planejamento de testes

Os casos de teste criados e executados são:

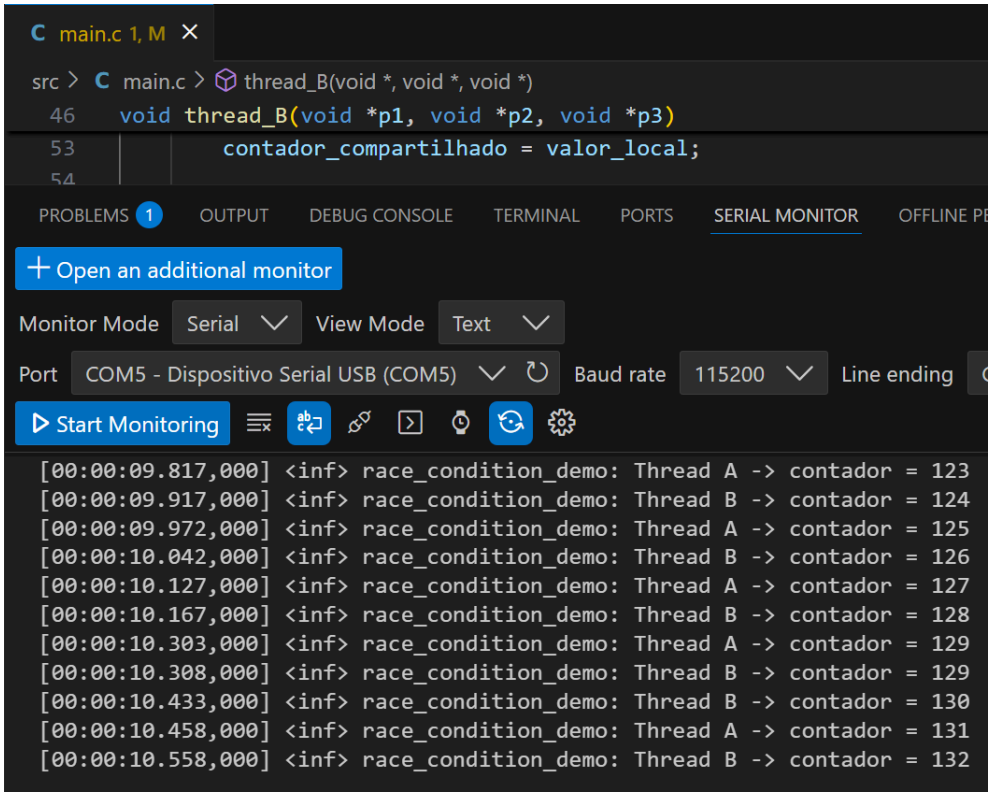
Caso de Teste	Pré-condição	Etapas de Teste	Pós-condição Observada	Conclusão
1 – Mesmas prioridades e tempos iguais (Caso base)	THREAD_A_PRIO = 5, THREAD_B_PRIO = 5 k_msleep(200) em ambas.	Compilar e executar o código original. Observar logs no terminal.	Contador cresce em sequência **duplicada** (1,1,2,2,3,3...). Nenhum salto, mas valores se repetem.	Threads acessam o contador simultaneamente. O comportamento é previsível, porém incorreto.
2 – Diferença de temporização	THREAD_A_PRIO = 5, THREAD_B_PRIO = 5 k_msleep(A)=130 ms, k_msleep(B)=100 ms.	Executar o código e observar logs.	Contador **se dessincroniza**: valores se repetem e/ou saltam (4,4,5,6,6,7...). LEDs piscam em ritmos diferentes.	O problema se manifesta claramente — cada thread sobrescreve o valor da outra.
3 – Alteração de prioridade (tentativa de mitigação)	THREAD_A_PRIO = 4, THREAD_B_PRIO = 5 (A tem prioridade mais alta).	Executar e observar o comportamento.	Contador aparece correto (1,2,3,4...), sem repetições.	A race condition foi **mascarada**: o escalonador impede a preempção, mas o problema lógico ainda existe.

Os testes realizados nas diferentes condições estão registrados abaixo em formato de printscreen dos Logs obtidos em cada caso:

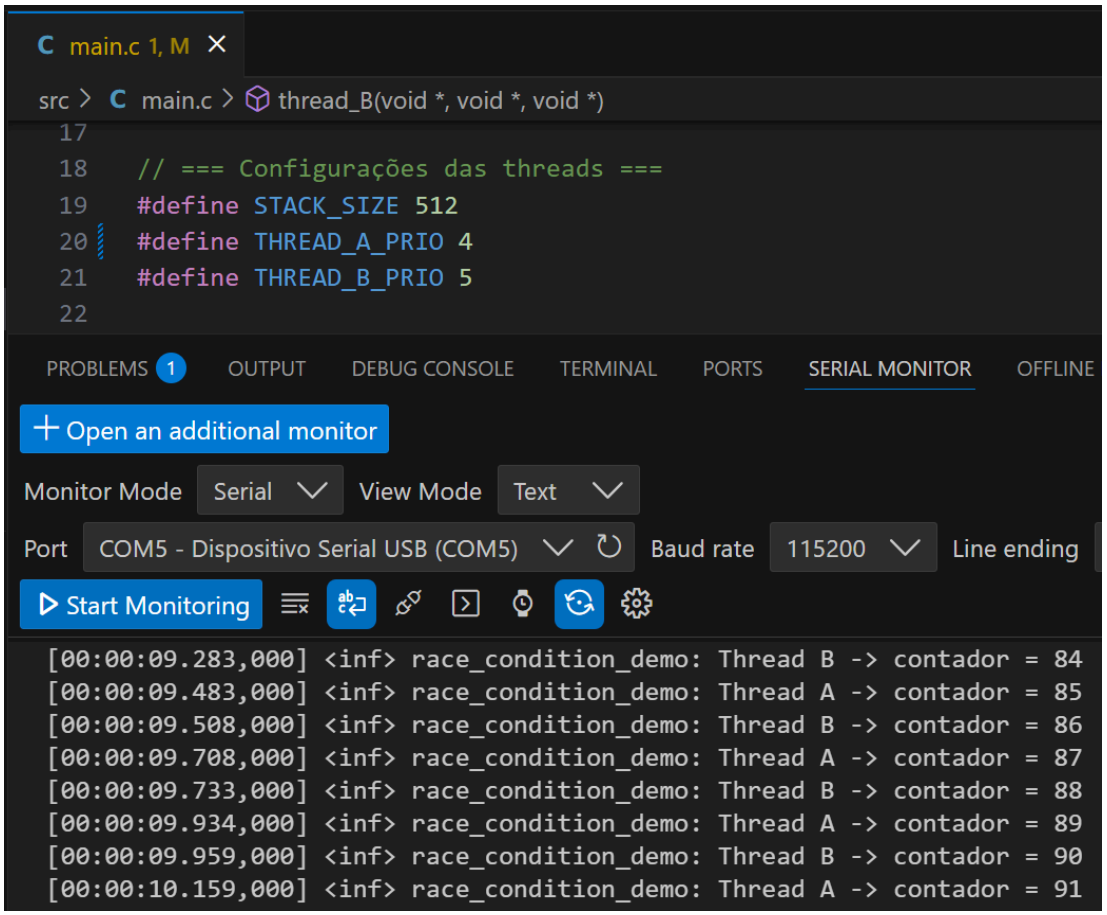
Caso 1: Mesmas prioridades e tempos iguais



Caso 2: Diferença de temporização



Caso 3: Prioridades diferentes



Correção e Reteste

Em seguida, foram feitas as devidas correções no código, adicionando um mutex para controlar o acesso ao recurso compartilhado (contador global), de modo a evitar a Race condition.

Dessa forma, o contador agora é incrementado corretamente, apresentando uma sequência como: A: 1, B: 2, A: 3, B: 4, A: 5, B: 6…

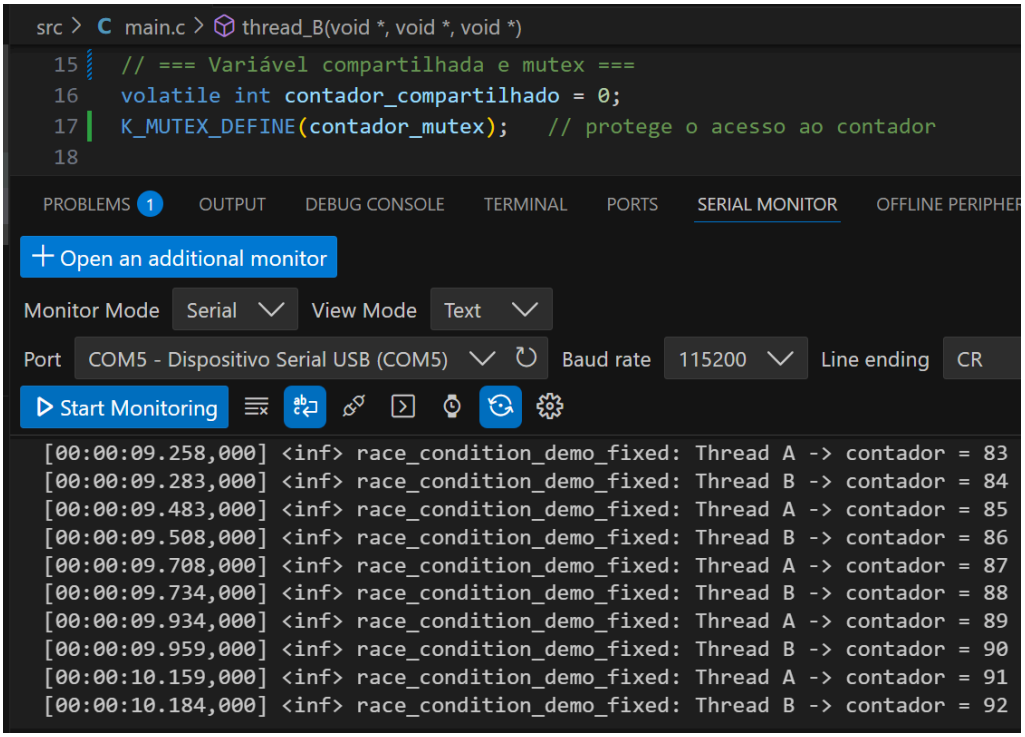
Ou seja, cada Thread incrementa o contador em 1 alternadamente, sem haver competição pelo recurso e sem causar comportamento indesejado ou imprevisível.

A tabela com os retestes realizados e seus resultados e conclusões está registrada abaixo:

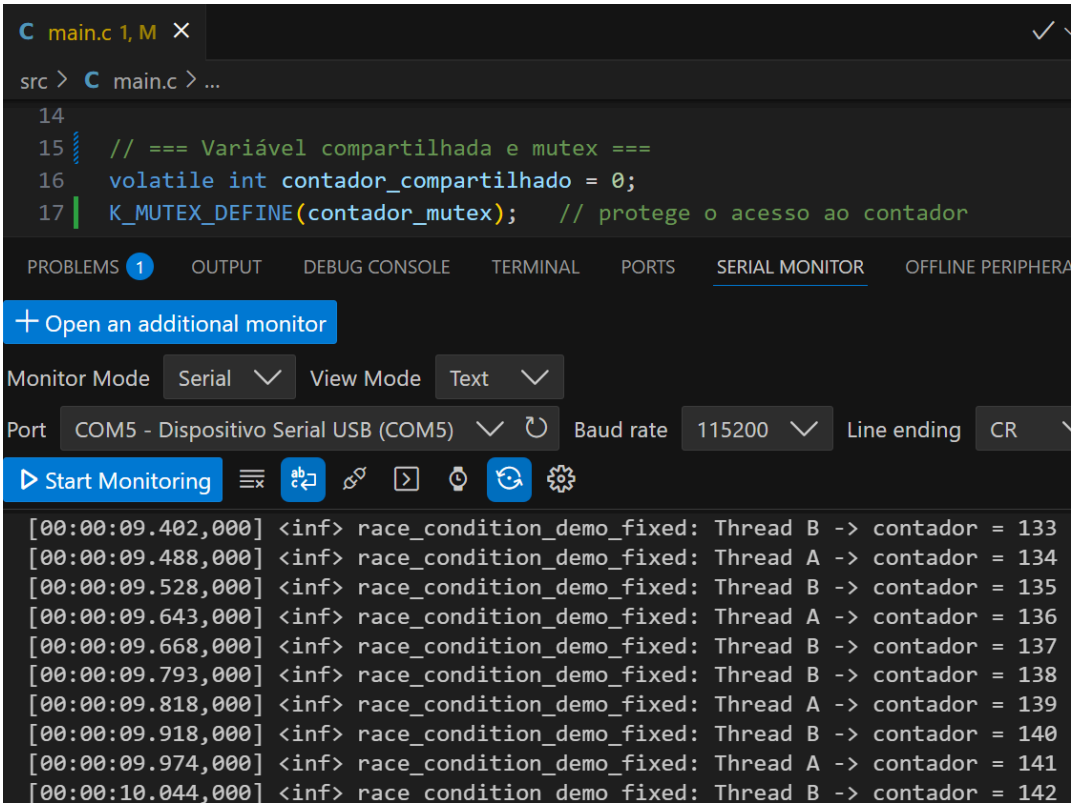
Caso de Teste	Mudança Aplicada	Etapas de Reteste	Pós-condição Esperada (resultado após correção)
1 – Mesmo cenário original (200 ms / 200 ms)	Inserção de k_mutex_lock() e k_mutex_unlock() envolvendo a manipulação de contador_compartilhado.	Executar novamente.	Contador cresce corretamente (1,2,3,4...). Nenhuma repetição, mesmo com mesma prioridade.
2 – Temporização diferente (130 ms / 100 ms)	Mesma proteção por mutex.	Executar novamente com tempos diferentes.	Contador consistente (1,2,3,4...). LEDs piscam fora de fase, mas contagem é linear.
3 – Alteração de prioridade (4 / 5)	Teste de robustez com prioridades diferentes.	Executar e observar.	Contagem correta e previsível em qualquer ordem de execução.

Os retestes realizados com o código atualizado estão registrados abaixo, por meio de printscreen dos Logs apresentados no Serial Monitor:

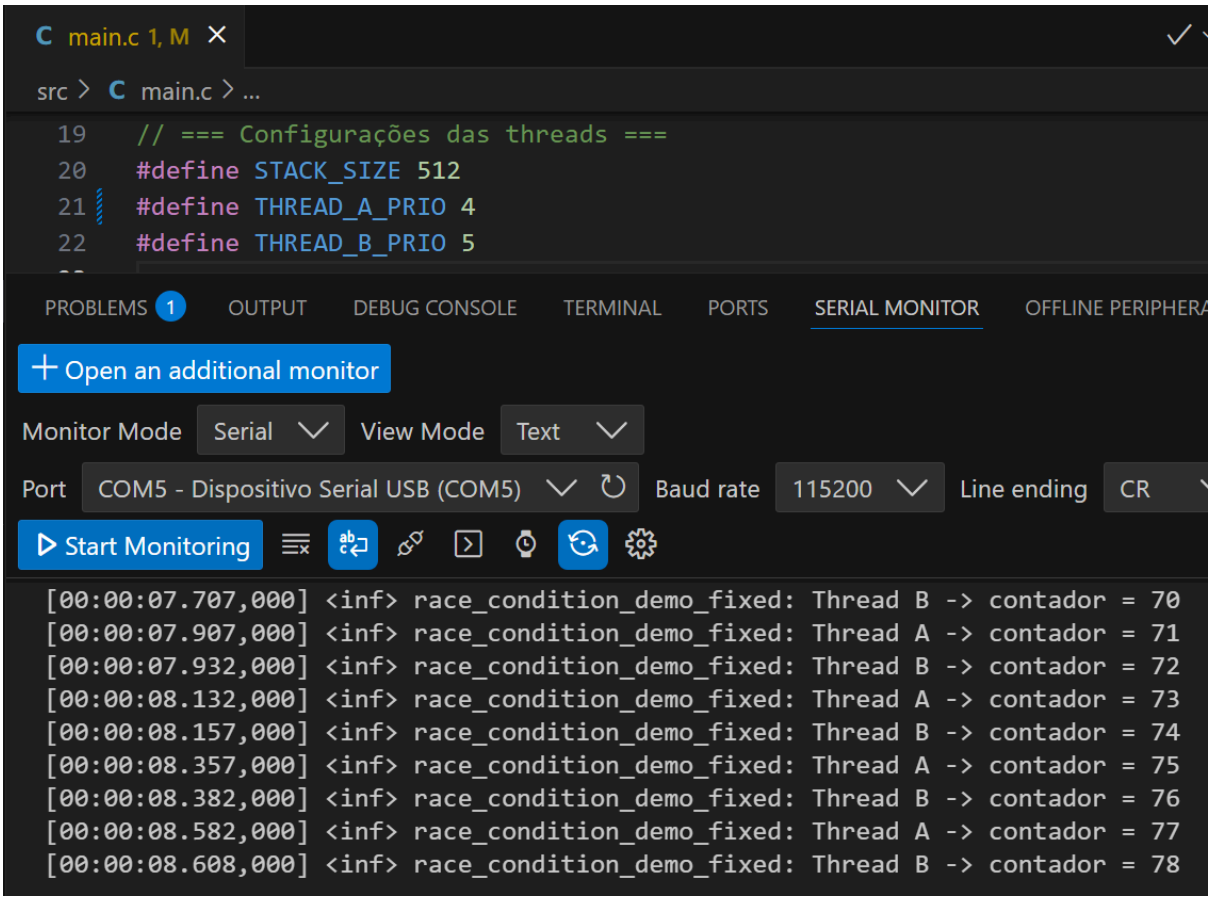
Caso 1: Mesma prioridade e tempos iguais



Caso 2: Diferença de temporização



Caso 3: Diferentes prioridades



Avaliação Cruzada do Código do Alberto

Contexto:

- Duas threads independentes que acessam o mesmo contador global.
- O código já faz um auto-diagnóstico de funcionamento, mostrando automaticamente no monitor serial o valor final do contador e o valor final esperado, de modo a evidenciar o problema.

O que estava errado antes:

- Nos cenários propostos 1 e 2, o resultado final do contador (1000) é diferente do valor final esperado após a execução do código (2000), devido à ocorrência de race condition.
- Já no cenário 3, que apresenta o resultado final igual ao esperado, a ocorrência desse problema é mascarada pela atribuição de uma prioridade maior a Thread A e pela definição de seu delay como 0. Nessas condições, não ocorreu o uso simultâneo do recurso compartilhado, pois B inicia sua tarefa somente após A ter concluído todos os seus ciclos, justificando a ausência de race condition e o resultado correto encontrado.

O que mudou com a correção:

- Tanto na versão corrigida utilizando Lock manual, quanto na versão com Mutex, o resultado final do contador global é igual ao valor final esperado (2000), em todos os cenários. Este valor é estável, sendo cada um dos cenários verificados 3 vezes em ambas as implementações.