Atividade: Resolução de Race Condition com Semáforo – Aula 6

Integrantes:

- Alberto Galhego Neto – NUSP 17019141
- Júlio Cesar Braga Parro – NUSP 16879560

Descrição:

Este experimento foi criado com base no código produzido via prompt na atividade anterior, que tinha como objetivo demonstrar a ocorrência de race condition. O propósito deste trabalho é testar a ocorrência desse fenômeno e propor formas para corrigi-lo. Para isso, dividimos esta tarefa em 4 etapas: “Revisão do Código anterior”, “Planejamento de Testes e Execução dos Testes Iniciais”, “Correção e Reteste” e “Avaliação Interna”.

Etapa 1 – Revisão do Código Anterior:

O código original consiste em um contador incremental único (global) que é operado simultaneamente por duas threads (A e B), ambas possuindo prioridades idênticas (5).

Nessa implementação, cada thread realiza a mesma tarefa: ela lê o valor do contador global, armazena esse valor em uma variável local, incrementa a variável local em 1, aguarda um breve período (delay) e, em seguida, atualiza o contador global com o valor da variável local. Esta operação é repetida 1000 vezes por cada thread, sendo aguardado um delay igual ao do período de execução após cada ciclo. Ao concluir, o programa exibe o valor final do contador global, assim como o valor esperado.

Como a tarefa de incremento é executada 1000 vezes por cada thread, totalizando 2000 execuções, seria esperado que o valor final do contador fosse 2000. Contudo, devido a uma condição de corrida (race condition), o resultado obtido no código original diverge desse valor esperado, resultando em 1000.

Etapa 2 – Planejamento de Testes e Execução dos Testes Iniciais:

A tabela proposta pela atividade sofreu uma modificação: a inclusão de duas novas colunas, denominadas "Valor Encontrado" e “Conclusão”. Na sequência, os testes foram realizados de acordo com os dados da tabela, e os valores apurados durante a execução dos testes foram nela adicionados. O registro do resultado em alguns dos casos foi efetuado, através de capturas de tela (Figuras 1 e 2).

Após a execução dos testes iniciais, foi verificada a ocorrência de race condition em dois dos três cenários testados. Apenas no cenário 3, onde a prioridade de A é maior que a de B e o delay de A foi definido como 0, não foi possível constatar a ocorrência desse problema. Contudo, ao observar os logs de resultado (imagem 2), verificou-se que a ausência do problema se deu pelo fato de que a Thread B só foi iniciada após a conclusão de todos os ciclos da Thread A. Sendo assim, não ocorreu o acesso simultâneo ao recurso, o que poderia ter dado origem ao problema.

Teste 1 – Código original

Caso de Teste	Pré-Condição	Etapas de Teste	Pós-Condição Esperada	Valor Encontrado	Conclusão
1- Caso base (prioridades iguais)	-Thread A Prio. = 5 -Thread B Prio. = 5 -Thread A Delay = 2 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	1000	Houve race condition.
2-A possui maior prioridade. Mesmo delay	-Thread A Prio. = 3 -Thread B Prio. = 5 -Thread A Delay = 2 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	1000	Houve race condition.
3-A possui maior prioridade. Delay 0 para A.	-Thread A Prio. = 3 -Thread B Prio. = 5 -Thread A Delay = 0 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition, pois a thread B só iniciou após a conclusão de A. Não houve acesso simultâneo à variável.

Imagem 1 – Caso teste 1 da tabela 1 (código original, mesma prioridade entre threads):

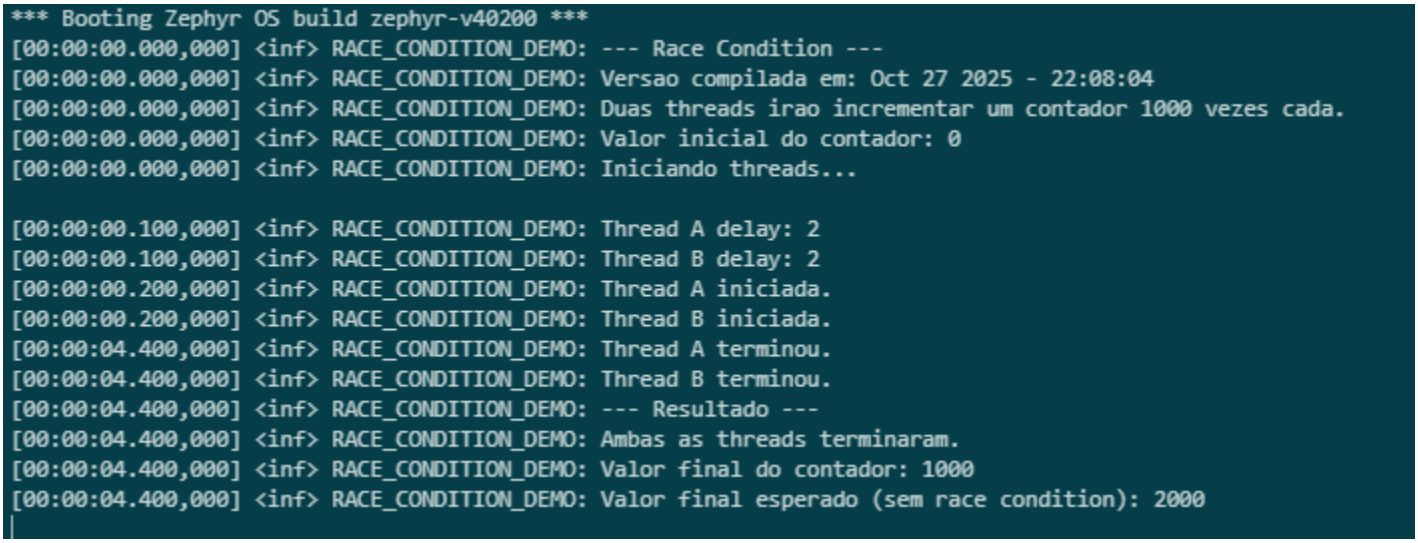
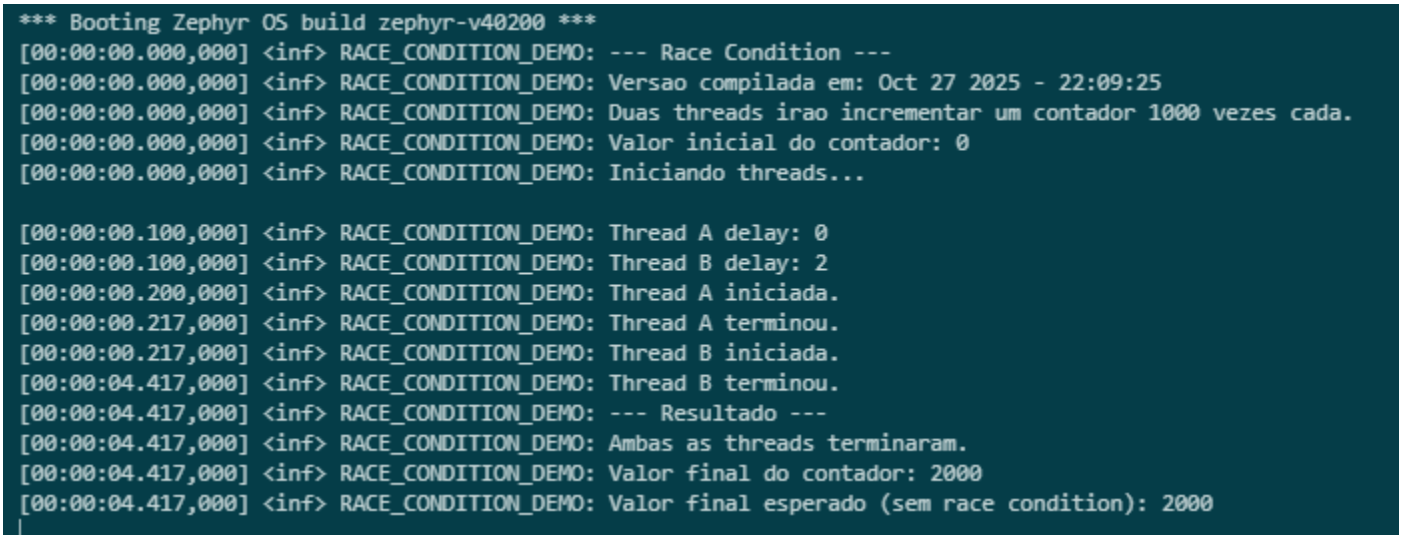


Imagem 2 – Caso teste 3 da tabela 1 (código original, A possui maior prioridade e 0 de delay):



Etapa 3 – Correção e Reteste:

Duas soluções foram propostas para os problemas encontrados: a utilização de mutex e o uso manual da função k_sched_lock(). Após a implementação das modificações, cada uma dessas situações foi submetida às mesmas condições de teste do código original, com os resultados sendo adicionados às tabelas abaixo.

Ambas as abordagens resolvem o problema da mesma forma: impedindo o acesso simultâneo de ambas as threads ao mesmo recurso. O uso de mutex torna-se particularmente interessante para projetos mais complexos, onde se exige o controle de acesso em diversas fases do código. Em contrapartida, para projetos simples como este, o bloqueio manual do scheduler acaba sendo mais simples e pode ser efetuado utilizando a função k_sched_lock().

Teste 2 - Uso manual da função k_sched_lock():

Caso de Teste	Pré-Condição	Etapas de Teste	Pós-Condição Esperada	Valor Encontrado	Conclusão
1- Caso base (prioridades iguais)	-Thread A Prio. = 5 -Thread B Prio. = 5 -Thread A Delay = 2 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition.
2-A possui maior prioridade. Mesmo delay	-Thread A Prio. = 3 -Thread B Prio. = 5 -Thread A Delay = 2 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition.
3-A possui maior prioridade. Delay 0 para A.	-Thread A Prio. = 3 -Thread B Prio. = 5 -Thread A Delay = 0 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition, pois a thread B só iniciou após a conclusão de A. Não houve acesso simultâneo à variável.

Teste 3 - Uso da Função k_mutex_lock():

Caso de Teste	Pré-Condição	Etapas de Teste	Pós-Condição Esperada	Valor Encontrado	Conclusão
1- Caso base (prioridades iguais)	-Thread A Prio. = 5 -Thread B Prio. = 5 -Thread A Delay = 2 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition.
2-A possui maior prioridade. Mesmo delay	-Thread A Prio. = 3 -Thread B Prio. = 5 -Thread A Delay = 2 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition.
3-A possui maior prioridade. Delay 0 para A.	-Thread A Prio. = 3 -Thread B Prio. = 5 -Thread A Delay = 0 -Thread B Delay = 2	-Definir os valores. -Compilar o código -Executar o código. -Observar o terminal	Resultado Final igual a 2.000	2000	Não houve race condition, pois a thread B só iniciou após a conclusão de A. Não houve acesso simultâneo à variável.

Imagem 3 – Caso teste 1 da tabela 2 (uso de k_sched_lock):

```
*** Booting Zephyr OS build zephyr-v40200 ***
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: --- Race Condition - Corrigido com Lock Manual ---
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Versao compilada em: Oct 27 2025 - 22:03:22
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Duas threads irao incrementar um contador 1000 vezes cada.
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Valor inicial do contador: 0
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Iniciando threads...

[00:00:00.100,000] <inf> RACE_CONDITION_DEMO: Thread A delay: 2
[00:00:00.100,000] <inf> RACE_CONDITION_DEMO: Thread B delay: 2
[00:00:00.200,000] <inf> RACE_CONDITION_DEMO: Thread A iniciada.
[00:00:00.200,000] <inf> RACE_CONDITION_DEMO: Thread B iniciada.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Thread A terminou.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Thread B terminou.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: --- Resultado ---
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Ambas as threads terminaram.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Valor final do contador: 2000
[00:00:02.301,000] <inf> RACE_CONDITION_DEMO: Valor final esperado (sem race condition): 2000
```

Imagem 4 – Caso teste 1 da tabela 3 (uso de k_mutex_lock):

```
*** Booting Zephyr OS build zephyr-v40200 ***
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: --- Race Condition - Corrigido com Mutex---
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Versao compilada em: Oct 27 2025 - 22:01:31
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Duas threads irao incrementar um contador 1000 vezes cada.
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Valor inicial do contador: 0
[00:00:00.000,000] <inf> RACE_CONDITION_DEMO: Iniciando threads...

[00:00:00.100,000] <inf> RACE_CONDITION_DEMO: Thread A delay: 2
[00:00:00.100,000] <inf> RACE_CONDITION_DEMO: Thread B delay: 2
[00:00:00.200,000] <inf> RACE_CONDITION_DEMO: Thread A iniciada.
[00:00:00.200,000] <inf> RACE_CONDITION_DEMO: Thread B iniciada.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Thread A terminou.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Thread B terminou.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: --- Resultado ---
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Ambas as threads terminaram.
[00:00:02.300,000] <inf> RACE_CONDITION_DEMO: Valor final do contador: 2000
[00:00:02.301,000] <inf> RACE_CONDITION_DEMO: Valor final esperado (sem race condition): 2000
```

Etapa 4 – Avaliação Cruzada do Código do Júlio:

- **Contexto:** duas threads independentes que acessam o mesmo contador global.
- **Problema encontrado:** Devido ao acesso simultâneo ao mesmo recurso, ocorre uma contagem duplicada, onde ao invés de incrementarem de maneira conjunta o contador, ambas as threads acabam definindo o mesmo valor. O resultado é estável, mas incorreto. Esse comportamento pode ser observado através dos valores exibidos nos Logs do programa.
- **Comportamento antes da mudança:** O resultado do contador era muito diferente do valor esperado, com o resultado variando conforme as diferentes combinações de temporização e prioridades.
- **O que mudou com a correção:** Após a implementação do uso de mutex, o contador passou a se comportar da maneira esperada, incremental e constante. Não são verificadas repetições de contagem, independente da combinação de temporização e prioridade. Resultado estável e correto.