Respostas

Aluno: Felipe Alves Belisário

Matrícula: 11721BCC030

Programa Fig20_04-05:

- Execução 1:

Creating threads

Threads created, starting tasks.

Tasks started, main ends.

task1 going to sleep for 695 milliseconds.

task3 going to sleep for 828 milliseconds.

task2 going to sleep for 1692 milliseconds.

task1 done sleeping

task3 done sleeping

task2 done sleeping

Process finished with exit code 0

- Execução 2:

Creating threads

Threads created, starting tasks.

Tasks started, main ends.

task1 going to sleep for 2787 milliseconds.

task3 going to sleep for 1775 milliseconds.

task2 going to sleep for 2220 milliseconds.

task3 done sleeping

task2 done sleeping

task1 done sleeping

Process finished with exit code 0

- Execução 3:

Creating threads

Threads created, starting tasks.

Tasks started, main ends.

task2 going to sleep for 2677 milliseconds.

task3 going to sleep for 175 milliseconds.

task1 going to sleep for 3336 milliseconds.

task3 done sleeping

task2 done sleeping

task1 done sleeping

Process finished with exit code 0

-- Explicação: Na execução do programa são criadas 3 threads de execução em que cada uma delas é colocada para "dormir" por um certo período de tempo gerado aleatoriamente pela função generator.nextInt (no intevalo de 0 a 5 segundos), e podemos perceber que elas não dormem na mesma ordem em que foram criadas, já que, devido ao fato de estarem executando em paralelo, essa ordem depende individualmente de cada thread.

-- Com alteração do *sleepTime*: Alterando o valor do range em que a função que gera números aleatórios pode chegar faz com que surja a possibilidade das threads dormirem por mais tempo a cada vez que aumentamos o range, e com isso demora mais para cada uma finalizar sua execução.

-- Diferença entre programas Fig26_04-05 e Fig26_06: A diferença entre eles é que o Fig26_06 possui uma melhor eficiência devido a existência de um gerenciador de threads com a utilização do ExecutorService, sobre o qual é responsável por já criar previamente um conjunto de threads preparadas para serem utilizadas e reutilizadas durante a execução do programa, sendo criadas novas apenas caso se torne necessário.

-- O que aconteceu com a Fig26_07-09: O problema deste programa está na falta de controle de utilização das threads sobre o método "add" do SimpleArray, fazendo com que uma interfira na outra durante o processo de compartilhamento de dados de cada uma, e com isso, consequentemente, os elementos não são inseridos no array na ordem correta.

-- Porque Fig26_10 resolve o problema de Fig26_07-09: Com o que foi dito anteriormente o problema foi resolvido através da atribuição do método "add" como *synchronized*, fazendo com que se tenha a característica de *mutual exclusive* entre as threads e com isso as mesmas não interfiram na outra (sendo bloqueadas) durante a inserção de dados no array iniciada previamente por alguma delas.

-- Problema de Fig26_11 até Fig26_23: Todos os programas dentre estes possuem uma maneira diferente de implementação do modelo produtor/consumidor, em que cada um possui uma forma de tratar o acesso ao buffer, onde o dado será escrito pelo produtor e depois lido pelo consumidor, a fim de resolver o problema dos conflitos das threads de fazerem esse acesso em momentos errados e acabar gerando dados inválidos.

-- Diferenças entre Fig26_16 até Fig26_23:

>> Fig26_16-17: Nessa solução ocorre o uso da classe ArrayBlockingQueue responsável pelo controle de bloqueio de acesso ao buffer feito pelo produtor e consumidor, em que se o buffer estiver cheio o método "put" espera a mesma esvaziar e se estiver vazia o método "take" espera a inserção de algum dado.

>> Fig26_18-19: Nessa solução além de ser utilizado a mesma atribuição nos métodos já citada anteriormente neste arquivo (*synchronized)*, também é utilizado uma variável booleana denominada *occupied,* inicialmente com o valor false, em que a thread do produtor só terá acesso ao buffer quando a mesma for false e, caso for, coloca o dado dentro do buffer e atribui ela como true; e o consumidor só terá acesso para leitura desse buffer se a variável for true, e logo após a leitura ela a mesma é setada para false novamente.

>> Fig26_20-21: Nessa solução o buffer é implementado pela classe CircularBuffer, em que, no caso desse programa em específico, o mesmo possui três posições no qual o produtor escreve em uma avança para a seguinte deixando aquela anteriormente escrita para que o consumidor possa lê-la. Seguindo essa ideia com a noção do caminho circular dentro do buffer a produção e consumo vai acontecendo.

>> Fig26_22-23: Nessa solução é utilizado o Lock que é responsável por interromper threads do produtor ou do consumidor que estejam em espera ou que passem de um tempo-limite previamente determinado para conseguir um bloqueio.