Relatório

Nome: Sara ramos Scalia
RA: 22401389

Conceitos e Definições

A programação concorrente em Java tem como objetivo permitir que múltiplas tarefas sejam executadas de forma independente e potencialmente simultânea dentro de um mesmo programa. Esse modelo é especialmente útil em situações que envolvem operações paralelizáveis, esperando por eventos externos ou atividades que devem ocorrer sem bloquear o restante da aplicação. Em Java, a concorrência é implementada principalmente através de threads, que representam fluxos distintos de execução. Cada thread possui seu próprio percurso de vida, desde sua criação até a finalização, e executa instruções de maneira independente, embora compartilhe o mesmo espaço de memória quando pertence ao mesmo processo.

Para criar threads, Java oferece duas abordagens tradicionais: estender a classe Thread ou implementar a interface Runnable. A interface Runnable é normalmente preferida pois separa a lógica de execução do controle de criação da thread, promovendo um design mais modular e permitindo o uso de herança quando necessário. Em ambos os casos, o comportamento da thread é definido pelo método run(), que contém o código que será executado. É importante diferenciar o uso de run() e start(): enquanto o primeiro apenas chama o método diretamente, sem criar concorrência, o método start() é o responsável por solicitar ao escalonador da JVM que a thread seja realmente iniciada de forma concorrente. Durante sua existência, uma thread passa por diferentes estados, como os estados de nova, pronta, executando, bloqueada, dormindo, aguardando e finalizada. A transição entre esses estados depende do escalonador da JVM, que controla a distribuição do tempo de CPU entre as threads ativas. Embora seja possível atribuir diferentes prioridades às threads, essas prioridades funcionam apenas como sugestões ao escalonador, e não como garantias rígidas de execução, já que o comportamento final pode variar conforme o sistema operacional e a JVM utilizada.

A concorrência naturalmente introduz uma série de desafios clássicos. Entre eles, destaca-se a race condition, ou condição de corrida, que ocorre quando duas ou mais threads acessam e modificam um recurso compartilhado sem a devida coordenação, produzindo resultados incorretos ou inconsistentes. Outro problema comum é o deadlock, no qual duas threads ficam aguardando indefinidamente a liberação de recursos uma pela outra, impedindo o progresso do sistema. A starvation também pode ocorrer quando uma thread é sistematicamente impedida de executar, muitas vezes porque outras monopolizam os recursos disponíveis.

Para evitar esses problemas, Java oferece mecanismos de sincronização que regulam o acesso a recursos compartilhados. O mais conhecido é a palavra-chave synchronized, que garante exclusão mútua, isto é, assegura que apenas uma thread por

vez possa executar determinado bloco ou método. Essa sincronização utiliza os monitores internos de cada objeto Java, que funcionam como travas responsáveis por controlar o acesso concorrente. Além de garantir que apenas uma thread execute a seção crítica por vez, synchronized também assegura visibilidade adequada das mudanças de memória entre threads.

Além da exclusão mútua, Java fornece ferramentas para comunicação entre threads. Os métodos wait(), notify() e notifyAll() permitem que uma thread aguarde um evento e que outras sejam capazes de acordá-la quando o evento ocorrer. Esses métodos atuam em conjunto com o monitor associado ao objeto e devem ser utilizados dentro de blocos sincronizados, garantindo assim a coordenação adequada entre atividades concorrentes. Com o surgimento do pacote java.util.concurrent, a partir da versão 5 da plataforma, a manipulação de concorrência tornou-se mais robusta e moderna. Em vez de trabalhar diretamente com threads individuais, tornou-se possível utilizar abstrações como o ExecutorService, que administra pools de threads e simplifica o envio, o agendamento e o encerramento de tarefas concorrentes. Essa API também introduziu interfaces como Callable, que permite retorno de valores pelas tarefas, e Future, que possibilita o gerenciamento e a recuperação de resultados assíncronos. Para operações de incremento ou manipulação de valores numéricos compartilhados, classes como AtomicInteger e AtomicLong oferecem operações atômicas e thread-safe sem a necessidade de bloqueios explícitos, garantindo um desempenho superior em muitas situações.

De forma geral,destaca que a programação concorrente exige atenção especial ao acesso à memória compartilhada, à sincronização adequada e à prevenção de condições indeterminadas. A partir dos conceitos apresentados, torna-se possível compreender tanto os riscos da execução paralela quanto às soluções estruturadas fornecidas pela linguagem.

Soma Paralela em Java: Explicação Detalhada

A implementação da soma paralela apresentada utiliza a API de concorrência do Java, baseada em ExecutorService e Callable, permitindo dividir um vetor em partes, processar cada parte em threads separadas e, posteriormente, agregar os resultados. Esta estratégia permite explorar múltiplos núcleos de processamento, reduzindo o tempo total necessário para processar grandes quantidades de dados.
O código inicia com os imports das bibliotecas necessárias:

```
import java.util.concurrent.*;
import java.util.*;
```

Esses pacotes fornecem classes essenciais para execução de tarefas concorrentes, tais como ExecutorService, Callable, Future e estruturas auxiliares como listas e geradores aleatórios.

Em seguida, define-se a classe ParallelSumExecutor, que contém o método responsável pela operação paralela. O método público e estático somaParalela recebe como parâmetros o vetor de inteiros A e o número de threads desejado. A palavra-chave static indica que o método pode ser invocado diretamente pela classe, sem instanciar um

objeto. Como esse método pode lançar exceções relacionadas à execução de tarefas concorrentes, declara-se throws InterruptedException e ExecutionException.

O método inicia verificando condições de segurança. Caso o vetor seja null, retorna imediatamente 0, prevenindo erros como NullPointerException. Se o número de threads informado for menor ou igual a zero, ele é ajustado para 1, evitando a criação de um pool inválido.

Define-se então n como o tamanho do vetor, o que facilita manipulações posteriores. Em seguida, cria-se um pool de threads fixo por meio do método Executors.newFixedThreadPool(numThreads), que garante a existência de um número determinado de threads reutilizáveis. Também é inicializada uma lista para armazenar objetos Future, que representarão os resultados parciais das somas.

O tamanho do pedaço (chunk) que cada thread deverá processar é calculado utilizando uma divisão arredondada para cima:

(n + numThreads – 1) / numThreads.

Esse cálculo garante que todos os elementos do vetor serão cobertos, mesmo quando n não é múltiplo do número de threads.

Dentro de um laço que percorre os índices de 0 até numThreads – 1, são calculados os valores start (início da fatia) e end (limite da fatia), ambos marcados como final para serem usados em expressões lambda. A condição if (start >= end) interrompe o laço caso não haja mais elementos disponíveis, evitando a submissão de tarefas vazias.

Para cada fatia é criado um Callable<Long>, que contém a lógica de soma parcial. A variável localSum é inicializada como zero e acumula, em um laço simples, a soma dos elementos entre start e end. Ao final, o valor é retornado. Em seguida, o callable é submetido ao pool de threads, e o Future resultante é adicionado à lista futures.

Após o envio de todas as tarefas, o método inicia a fase de agregação. Cria-se uma variável total, que acumulará a soma final. A partir disso, percorre-se a lista de futures, invocando f.get() para cada um deles. O método get é bloqueante, logo, o fluxo só prossegue quando cada soma parcial estiver concluída.

Com todas as somas parciais agregadas, o pool é encerrado com o comando shutdown, que instrui o sistema a finalizar corretamente as threads alocadas. Por fim, o método retorna o valor total calculado.

O método main apresentado serve como teste da funcionalidade. Ele cria um vetor grande com dez milhões de elementos, preenchidos com valores aleatórios. Em seguida, define quatro threads para a execução paralela, mede o tempo de processamento e compara o resultado com uma soma sequencial tradicional. Essa etapa assegura a correção do algoritmo e permite observar a diferença de desempenho entre os dois modelos.

Essa abordagem evita condições de corrida, pois cada thread acessa exclusivamente sua própria fatia, e não há escrita simultânea em variáveis compartilhadas. Além disso, o uso de ExecutorService simplifica o gerenciamento das threads, permitindo que a implementação se mantenha limpa e eficiente.

Contador Concorrente: Explicação Detalhada

O segundo problema demonstra o comportamento de um contador sendo acessado por múltiplas threads simultaneamente, destacando o fenômeno conhecido como race condition. Esse problema é clássico em programação concorrente e surge quando duas ou mais threads tentam alterar um valor compartilhado ao mesmo tempo, sem mecanismos adequados de sincronização.
A classe inicia com o import da biblioteca:

```
import java.util.concurrent.atomic.AtomicInteger;
```

Esse import permite o uso de AtomicInteger, uma estrutura de dados otimizada que garante operações atômicas sem a necessidade de blocos sincronizados explícitos.

A classe principal CounterDemo contém três implementações distintas de contador: uma insegura, uma com sincronização explícita e outra com operações atômicas.

A primeira versão, chamada UnsafeCounter, possui um atributo público inteiro e um método increment que executa count = count + 1. Embora pareça simples, essa operação envolve, na prática, três passos internos: leitura do valor, incremento e escrita. Quando múltiplas threads executam esse processo simultaneamente, há a possibilidade de leituras desatualizadas e sobrescritas incorretas, resultando em valores menores que o esperado.

A segunda versão, SafeCounterSync, corrige esse problema utilizando o modificador synchronized tanto no método increment quanto no método get. O synchronized garante que apenas uma thread por vez tenha acesso à região crítica, impedindo leituras e escritas simultâneas e preservando a consistência dos dados.

A terceira versão, SafeCounterAtomic, adota um AtomicInteger. Essa classe fornece métodos como incrementAndGet(), que realiza a operação de incremento de maneira atômica e eficiente, sem a sobrecarga de bloqueios explícitos. Essa abordagem costuma produzir melhor desempenho em cenários com alto grau de concorrência.

O método auxiliar runTest executa a lógica de testar cada uma das versões. Ele cria um vetor de threads e define, para cada uma, uma operação repetida de incremento. Após iniciar todas as threads com start(), o método aguarda sua conclusão usando join(), garantindo assim que o programa só continue quando todas tiverem encerrado sua execução.

O método main configura um cenário de teste com oito threads, cada uma realizando cem mil incrementos. Para cada uma das três implementações, exibe-se o resultado esperado (número de threads multiplicado pelos incrementos por thread) e o resultado real. A versão insegura tipicamente retorna valores inferiores aos esperados, devido à perda de atualizações. As versões sincronizada e atômica, em contraste, retornam os valores corretos, demonstrando a necessidade de mecanismos de exclusão mútua ou operações atômicas para manipulação segura de dados compartilhados.

Essas três versões ilustram a importância da sincronização em cenários concorrentes. A versão insegura funciona corretamente apenas em ambientes sem paralelismo. A versão com synchronized garante correção, porém com maior custo devido ao bloqueio intrínseco. A versão com AtomicInteger oferece uma alternativa mais leve e eficiente, sendo amplamente recomendada para contadores de alta demanda.

Conclusão

A realização desta atividade permitiu compreender de forma prática e aprofundada os princípios fundamentais da programação concorrente em Java, especialmente no que diz respeito ao uso de threads, problemas de sincronização e técnicas de paralelização. A implementação da soma paralela demonstrou como a divisão de tarefas entre múltiplas threads pode acelerar operações intensivas, desde que haja uma estratégia adequada de particionamento e agregação dos resultados. A utilização de estruturas avançadas, como ExecutorService e Callable, revelou-se essencial para simplificar o gerenciamento de threads, tornando o código mais robusto, escalável e alinhado às APIs modernas da linguagem.

Além disso, o problema do contador concorrente evidenciou de maneira clara o fenômeno de race condition, mostrando que operações aparentemente simples, como incrementar uma variável inteira, podem produzir resultados incorretos em ambientes multithread. As duas soluções apresentadas ,o uso de blocos sincronizados e o emprego de tipos atômicos,demonstraram abordagens distintas para eliminar esses problemas. Enquanto synchronized garante exclusão mútua através de bloqueios explícitos, estruturas como AtomicInteger fornecem eficiência e segurança por meio de operações atômicas não bloqueantes. Essa comparação reforça a importância de compreender não apenas o funcionamento das threads, mas também os mecanismos de coordenação entre elas.

A utilização de ferramentas de Inteligência Artificial ao longo do desenvolvimento também contribuiu de forma significativa para aprimorar a aprendizagem. A IA auxiliou na geração inicial do código, oferecendo alternativas de implementação e destacando boas práticas. Entretanto, tornou-se evidente que o papel do estudante permanece essencial, seja para analisar criticamente as sugestões, realizar ajustes necessários ou adaptar o código às exigências do problema. Assim, a IA mostrou-se um instrumento valioso para o processo de autopreparação, desde que utilizada com discernimento e responsabilidade.

Em síntese, a atividade proporcionou uma experiência completa envolvendo teoria e prática, permitindo aplicar os conceitos estudados no capítulo sobre concorrência de forma concreta e contextualizada. A combinação entre estudo do material, desenvolvimento manual, experimentação com múltiplas abordagens e apoio de IA resultou em um aprendizado mais sólido, tanto do ponto de vista técnico quanto conceitual.

Prompts que eu usei

Prompt (soma paralela)

"Escreva um código Java que divida um vetor de inteiros entre N threads, cada uma calcule a soma parcial e o main agregue as somas. Use ExecutorService e Callable. Inclua um main para testar com um vetor grande e compare com soma sequencial."

Prompt (contador sincronizado)

"Mostre um exemplo em Java que demonstre race condition ao incrementar um contador por muitas threads e duas soluções (usando synchronized e AtomicInteger). Inclua saída mostrando diferença entre inseguro e seguro."

Referências

SILVEIRA, S. R.; et al. *Paradigmas de Linguagem de Programação*. Capítulo 5: Programação Concorrente. Páginas 75–91. Material utilizado como base teórica para a atividade.

DEITEL, H. M.; DEITEL, P. J. *Java: How to Program*. Conceitos de concorrência consultados para reforço teórico sobre threads, sincronização e atomicidade.

Código desenvolvido com apoio parcial de ferramenta de Inteligência Artificial (ChatGPT), com revisões, correções e adaptações manuais.