Programação paralela em Java: threads

Prof. Marlos de Mendonça Corrêa

Prof. Kleber de Aguiar

Apresentação

Programação paralela em Java usando threads permite executar múltiplas tarefas ao mesmo tempo, aumentando a eficiência do programa. Isso melhora o desempenho, especialmente em sistemas com múltiplos processadores. Threads ajudam a dividir tarefas grandes em menores e executá-las simultaneamente, o que é útil em aplicações que exigem alta performance, como jogos, processamento de imagens e servidores web.

Propósito

Objetivos

Módulo 1

Threads e processamento paralelo

Reconhecer o conceito de threads e sua importância para o processamento paralelo.

Módulo 2

Sincronização entre threads

Identificar a sincronização entre threads em Java.

Módulo 3

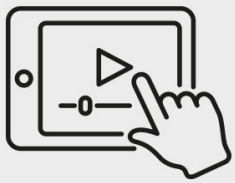
Implementação de threads

Aplicar a implementação de threads em Java.

Introdução

Neste vídeo, vamos mostrar como a programação paralela em Java com threads pode aumentar a eficiência e o desempenho dos programas. Aprenda a dividir grandes tarefas em partes menores e a executá-las ao mesmo tempo, aproveitando vários processadores. Isso é ideal para aplicações de alta performance, como jogos, processamento de imagens e servidores web.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

Download material

1 - Threads e processamento paralelo

Ao final deste módulo, você será capaz de reconhecer o conceito de threads e sua importância para o processamento paralelo.

Processamento paralelo com threads em Java

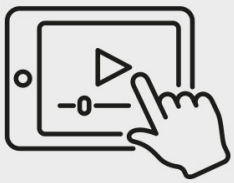
O processamento paralelo com threads em Java permite executar várias tarefas simultaneamente, aproveitando melhor os recursos do sistema. Thread é uma unidade independente de execução dentro de um programa. Usando threads, podemos dividir um programa em partes menores que rodam ao mesmo tempo, melhorando a eficiência e a velocidade.

Com threads, podemos aproveitar múltiplos núcleos de processadores modernos, tornando os programas mais rápidos. Em comparação, sem o uso de threads, o programa é executado sequencialmente, do início ao

fim, em um único núcleo do processador, demandando maior tempo de execução se comparado com o uso de threads.

Neste vídeo, abordaremos o processamento paralelo com threads em Java, que permite a execução simultânea de várias tarefas, maximizando o uso dos recursos do sistema.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Conceitos

Estamos tão acostumados com as facilidades da tecnologia que muitas vezes os mecanismos que atuam ocultos são ignorados. Isso vale também para os desenvolvedores. No início da computação, contudo, não era assim. Naquela época, bits de memória, ciclos de CPU e watts de energia gastos eram importantes. Aliás, não se podia desenvolver um programa sem perfeito conhecimento do hardware. A linguagem de máquina e a assembly (linguagem de montagem) eram os únicos recursos para programação, e operações de E/S (entrada/saída) podiam demorar minutos, devendo ser evitadas a todo custo.

Comentário

Com o passar do tempo e o avanço tecnológico (surgimento de novas linguagens de programação, barramentos mais velozes e CPUs mais rápidas), houve um aumento da complexidade e, com isso, os compiladores passaram a agilizar muito o trabalho de otimização de código, gerando códigos de máquina mais eficientes.

Na era da computação moderna, os processadores têm vários núcleos, que podem ser explorados para aumentar a eficiência de execução de programas, especialmente aqueles grandes e complexos. Ou seja, ao invés de um programa ser executado em apenas um núcleo do processador, ele pode explorar vários deles, subdividindo as tarefas. Isso é possível graças a alguns fatores: sistemas operacionais multitarefa preemptiva, linguagens que possibilitam a programação com várias threads (multithreading) e processadores com vários núcleos, físicos e virtuais (hyperthreading).

Um sistema operacional multitarefa preemptiva é um tipo de sistema que permite a execução simultânea de múltiplas tarefas (processos ou

threads) e tem a capacidade de interromper (preemptar) a execução de uma tarefa para iniciar ou continuar a execução de outra.

> **O termo thread ou processo leve consiste em uma sequência de instruções, uma linha de execução dentro de um processo.**

Para facilitar a compreensão do conceito de thread, vamos construir computadores teóricos com duas configurações:

CPU genérica de núcleo único

CPU multinúcleo

Vamos supor que os dois computadores teóricos, um com CPU genérica de núcleo único e outro com CPU multinúcleo, possuam sistemas operacionais multitarefa preempetivos, ou seja, capazes de controlar a execução das tarefas nos processadores. Como exemplo, vamos considerar um programa fictício, com apenas uma linha de código sendo executada nesses dois computadores. Esse exemplo simples fornecerá a base para a compreensão de situações mais complexas envolvendo o conceito de threads.

Execução de software por um computador teórico

Configuração: CPU genérica de núcleo único

Nossa primeira configuração era muito comum há pouco mais de uma década. Imagine que você está usando o Word para fazer um trabalho e, ao mesmo tempo, está calculando a soma hash (soma utilizando algoritmo) de um arquivo.

> Como ambos os programas podem estar em execução simultaneamente?

Na verdade, como já vimos, eles não estão. Essa é apenas uma ilusão criada pela preempção. Então, o que acontece de fato? Vamos entender!

Os sistemas operacionais multitarefa preemptiva implementam o chamado **escalonador de processos**. O escalonador utiliza algoritmos que gerenciam o tempo de CPU que cada processo pode utilizar. Assim, quando o tempo é atingido, uma interrupção faz com que o estado atual da CPU (registradores, contador de execução de programa e outros parâmetros) seja salvo em memória, ou seja, o processo é tirado do contexto da execução e outro processo é carregado no contexto.

Toda vez que o tempo determinado pelo escalonador é atingido, essa troca de contexto ocorre, e as operações são interrompidas mesmo que ainda não finalizadas. A sua execução é retomada quando o processo volta ao contexto.

Escalonador de processos

Quanto mais softwares forem executados simultaneamente, mais a ilusão será perceptível, pois poderemos perceber a lentidão na execução dos programas.

Configuração: CPU multinúcleo

Vamos, então, considerar a segunda configuração (CPU multinúcleo): imagine uma família composta por quatro pessoas que vai ao supermercado juntas no mesmo carro. Ao chegar no supermercado, cada um deles fica responsável por comprar certos itens da lista de compras. Ou seja, eles trabalham de modo independente, mas de forma colaborativa. Cada um deles passa em um caixa diferente do outro para realizar o pagamento. Fica claro que essa abordagem reduz o tempo gasto para realizar a compra da família.

Voltando para o exemplo da computação, cada membro da família corresponde a uma thread e os caixas representam os núcleos do processador, sendo um pipeline completo e independente dos demais (cada um com seus próprios registradores, sem que a execução de um código em núcleo interfira na execução desse mesmo código em outro núcleo). O conceito de pipeline em computação refere-se a uma técnica de processamento na qual uma tarefa é dividida em várias etapas sequenciais. Cada etapa executa uma parte do trabalho, e as diferentes etapas podem ser processadas em paralelo.

Claro que isso é verdade quando desconsideramos operações de I/O (entrada/saída) ou R/W (ler/escrever) em memória. E, por simplicidade, essa será nossa abordagem. Podemos considerar, também por simplicidade, que cada núcleo é idêntico ao nosso caso anterior.

Sempre que um software é executado, ele dispara um processo. Os valores de registrador, a pilha de execução, os dados e a área de memória fazem parte do processo. Quando um processo é carregado em memória para ser executado, uma área de memória é reservada e se torna exclusiva. Um processo pode criar **subprocessos**, chamados também de **processos filhos**.

Mas, afinal, o que são threads?

Vale lembrar que um programa é composto por diversas linhas de código escritas pelos desenvolvedores. Essas linhas de código são executadas pelo processador, que realiza as ações correspondentes. Mas, diante desse contexto, o que são threads e processos? As threads são linhas de execução de programa contidas nos processos. Vamos diferenciar as threads de processos:

- **Processos:** programas em execução, cada um com seu próprio espaço de memória e recursos, isolados uns dos outros.
- **Threads:** menores unidades de execução dentro de um processo, compartilham o mesmo espaço de memória e recursos.

Podemos subdividir trechos de um programa em várias threads e cada uma delas pode ser executada em um núcleo diferente do mesmo processador. Como as threads são mais simples que os processos, sua criação, finalização e trocas de contexto são mais rápidas, oferecendo a possibilidade de paralelismo com baixo custo computacional quando comparadas aos processos. O fato de compartilharem a memória também facilita a troca de dados, reduzindo a latência (atraso) envolvida nos mecanismos de comunicação interprocessos.

Atividade 1

Imagine ser convidado para assistir a um treinamento na empresa em que trabalha. A palestra aborda programação multithreading, entre outros recursos. No final, o palestrante questiona qual seria uma vantagem significativa do processamento paralelo com threads em Java, além de mostrar algumas opções descritas a seguir. Qual é a resposta correta?

A Redução do uso de memória RAM.

B Diminuição da complexidade do código.

C Aumento da eficiência ao aproveitar múltiplos núcleos de processadores.

D Melhoria na compatibilidade com sistemas operacionais antigos.

E Redução do consumo de energia.

Parabéns! A alternativa C está correta.

Utilizar threads em Java permite que diferentes partes de um programa executem simultaneamente, aproveitando os múltiplos núcleos dos processadores modernos. Isso resulta em um aumento significativo da eficiência, pois tarefas podem ser executadas paralelamente, reduzindo o tempo total de execução e melhorando a capacidade de resposta do sistema. As demais opções não refletem diretamente as vantagens específicas do processamento paralelo com threads em Java.

O conceito de threads em Java

Em Java, cada thread possui seu próprio fluxo de execução, compartilhando o mesmo espaço de memória do processo principal. Isso possibilita dividir um programa em partes menores que podem ser executadas em paralelo, aproveitando melhor os recursos do sistema, como os múltiplos núcleos do processador. Threads são úteis em aplicações que requerem multitarefa, como processamento de dados em tempo real, interações de usuário em interfaces gráficas, entre outros. Além disso, é possível trabalhar com prioridades e ciclo de vida de threads em Java.

Neste vídeo, abordaremos como as threads em Java lidam com a execução de programas ao permitir multitarefa simultânea. Veremos que é possível dividir programas em partes menores que executam em paralelo, beneficiando-se dos múltiplos núcleos dos processadores modernos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

A linguagem Java é uma linguagem de programação multithread, o que significa que Java suporta o conceito de threads. Como vimos, uma thread pode ser preemptada da execução e isso é feito pelo sistema operacional que emite comandos para o hardware. Por isso, nas primeiras versões da MVJ (máquina virtual Java) o uso de threads era dependente da plataforma. Logo, se o programa usasse threads, ele perdia a portabilidade oferecida pela MVJ. Com a evolução da tecnologia, a MVJ passou a abstrair essa funcionalidade, de forma que tal limitação não existe atualmente.

> Uma thread é uma maneira de implementar múltiplos caminhos de execução em uma aplicação.

Observe quantos programas estão em execução neste momento no seu computador. Talvez você esteja utilizando um navegador, um editor de texto, antivírus, entre outros. Todos eles são executados no processador. O seu computador deve ter um processador com vários núcleos e, assim, é importante tirar o máximo proveito deles. Como isso é possível? É comum que vários programas sejam executados ao mesmo tempo e o gerenciamento dessa execução fique por conta do sistema operacional (SO), que divide o tempo de execução entre eles (execução em paralelo ou de forma preemptiva). Cada programa pode ter uma ou mais partes que executam tarefas diferentes ao mesmo tempo, ou quase ao mesmo tempo.

Toda thread possui uma prioridade. A prioridade de uma thread é utilizada pelo escalonador da MVJ para decidir o agendamento de que thread vai utilizar a CPU. Threads com maior prioridade têm preferência na execução, porém é importante notar que ter preferência não é ter controle total. Suponha que uma aplicação possua apenas duas threads,

uma com prioridade máxima e a outra com prioridade mínima. Mesmo nessa situação extrema, o escalonador deverá, em algum momento, preemptar a thread de maior prioridade e permitir que a outra receba algum tempo de CPU. Na verdade, a forma como as threads e os processos são escalonados depende da política do escalonador.

Escalonador de processos.

Por que precisa ser assim?

Isso é necessário para que haja algum paralelismo entre as threads. Do contrário, a execução se tornaria serial, com a fila sendo estabelecida pela prioridade. Num caso extremo, em que novas threads de alta prioridade continuassem sendo criadas, threads de baixa prioridade seriam adiadas indefinidamente.

Atenção!

A prioridade de uma thread não garante um comportamento determinístico. Ter maior prioridade significa apenas isso. O programador não sabe quando a thread será agendada.

Em Java, há dois tipos de threads:

Daemon

São threads de baixa prioridade, sempre executadas em segundo plano. Essas threads provêm serviços para as threads de usuário (user threads), e sua existência depende delas, pois se todas as threads de usuário finalizarem, a MVJ forçará o encerramento da daemon thread, mesmo que suas tarefas não tenham sido concluídas. O Garbage Collector (GC) é um exemplo de daemon thread. Isso esclarece por que não temos controle sobre quando o GC será executado e nem se o método finalize será realizado.

User

São criadas pela aplicação e finalizadas por ela. A MVJ não força sua finalização e aguardará que as threads completem suas tarefas. Esse tipo de thread executa em primeiro plano e possui prioridades mais altas que as daemon threads. Isso não permite ao usuário ter certeza de quando sua thread entrará em execução, por isso mecanismos adicionais precisam ser usados

para garantir a sincronicidade entre as threads. Veremos esses mecanismos mais à frente.

Atividade 2

Imagine que você esteja navegando em um fórum de computação e encontra uma pergunta de um usuário sobre o que são threads em Java. Vários outros usuários postaram respostas, que são citadas a seguir. Qual das seguintes afirmações melhor descreve threads em Java?

A Threads são partes independentes de um programa que compartilham apenas o código-fonte.

B Threads são processos separados que possuem seu próprio espaço de memória e recursos.

C Threads são unidades de execução dentro de um processo que compartilham o mesmo espaço de memória.

D Threads são interfaces gráficas utilizadas para interações do usuário em Java.

E Threads são classes especiais em Java usadas para criar interfaces de usuário personalizadas.

Parabéns! A alternativa C está correta.

Em Java, threads são unidades de execução leves que operam dentro de um processo e compartilham o mesmo espaço de memória. Isso permite que múltiplas threads executem simultaneamente, dividindo tarefas e aproveitando melhor os recursos do sistema. As demais opções não descrevem corretamente o conceito de threads em Java,

que se caracteriza pelo compartilhamento de memória e execução concorrente dentro de um processo.

Ciclo de vida de thread em Java

O ciclo de vida de uma thread em Java possui seis estados: New, Runnable, Blocked, Waiting, Timed Waiting e Terminated. Há duas maneiras de se criar uma thread em Java, usando a classe Thread e o mecanismo de Herança, ou implementando a interface Runnable.

Neste vídeo, serão abordados os estados de uma thread em Java, desde a criação até a terminação. As threads transitam entre os estados New, Runnable, Blocked, Waiting, Timed Waiting e Terminated.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Quando a MVJ inicia, normalmente há apenas uma thread não daemon, que tipicamente chama o método main das classes designadas. A MVJ continua a executar threads até que o método exit da classe Runtime é chamado e o gerenciador de segurança permite a saída ou até que todas as threads que não são daemon estejam mortas (ORACLE AMERICA INC., s.d.).

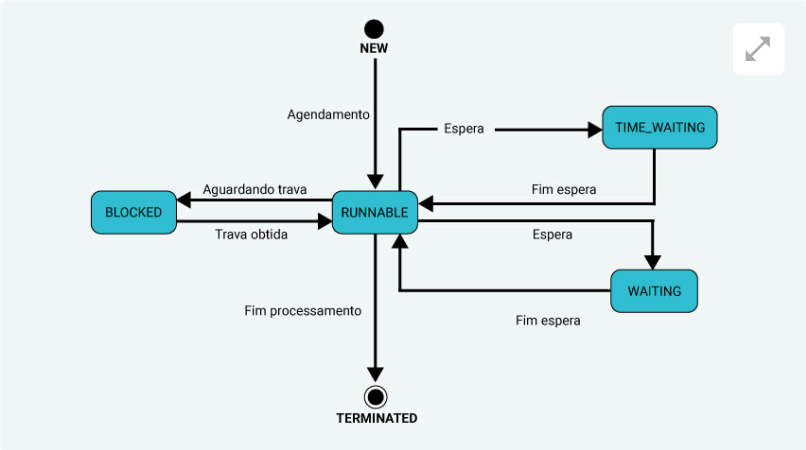
Há duas maneiras de se criar uma thread em Java:

Declarar a classe como subclasse da classe Thread.

Declarar uma classe que implementa a interface Runnable.

Toda thread possui um nome, mesmo que ele não seja especificado. Nesse caso, um nome será automaticamente gerado. Veremos os detalhes de criação e uso de threads logo mais.

Uma thread pode existir em seis estados, conforme vemos na máquina de estados retratada na imagem a seguir.



Escalonador de processos.

Como podemos observar na imagem, os seis estados de uma thread são:

NEW

A thread está nesse estado quando é criada e ainda não está agendada para execução (SCHILDT, 2014).

RUNNABLE

A thread entra nesse estado quando sua execução é agendada (escalonamento) ou quando entra no contexto de execução, isto é, passa a ser processada pela CPU (SCHILDT, 2014).

BLOCKED

A thread passa para este estado quando sua execução é suspensa enquanto aguarda uma trava (lock). A thread sai desse estado quando obtém a trava (SCHILDT, 2014).

TIMED_WAITING

A thread entra nesse estado se for suspensa por um período, por exemplo, pela chamada do método **sleep ()** (dormindo), ou quando o timeout de **wait ()** (esperando) ou **join ()** (juntando) ocorre. A thread sai desse estado quando o período de suspensão é transcorrido (SCHILDT, 2014).

WAITING

A thread entre nesse estado pela chamada aos métodos **wait ()** ou **join ()** sem timeout ou **park ()** (estacionado) (SCHILDT, 2014).

TERMINATED

A thread chega a este estado, o último, quando encerra sua execução (SCHILDT, 2014).

É possível que em algumas literaturas você encontre essa máquina de estados com nomes diferentes. Conceitualmente, a execução da thread pode envolver mais estados, e, sendo assim, você pode representar o ciclo de vida de uma thread de outras formas. Mas além de isso não invalidar a máquina mostrada em nossa figura, esses estados são os especificados pela enumeração **State** (ORACLE AMERICA INC., s.d.) da classe **Thread** e retornados pelo método **getState ()**. Isso significa que, na prática, esses são os estados com os quais você irá operar numa implementação de thread em Java.

Comentário

Convém observar que, quando uma aplicação inicia, uma thread começa a ser executada. Essa thread é usualmente conhecida como thread principal (main thread) e existirá sempre, mesmo que você não tenha empregado threads no seu programa. Nesse caso, você terá um programa single thread, ou seja, de thread única. A thread principal criará as demais threads, caso necessário, e deverá ser a última a encerrar sua execução.

Quando uma thread cria outra, a mais recente é chamada de thread filha. Ao ser gerada, a thread receberá, inicialmente, a mesma prioridade

daquela que a criou. Além disso, uma thread será criada como daemon apenas se a sua thread criadora for um daemon. Todavia, a thread pode ser transformada em daemon posteriormente, pelo uso do método **setDaemon()**.

Criando uma thread

Como vimos, há duas maneiras de se criar uma thread. Em ambos os casos, o método **run ()** deverá ser sobrescrito.

> ## Então qual a diferença entre as abordagens?

Trata-se mais de oferecer alternativas em linha com os conceitos de orientação a objetos (OO). A extensão de uma classe normalmente faz sentido se a subclasse vai acrescentar comportamentos ou modificar a sua classe pai.

Então, qual abordagem seguir?

Mecanismo de herança		Implementação de “Runnable”
Utilizar o mecanismo de herança com o único objetivo de criar uma thread pode não ser a abordagem mais interessante. Mas, se houver a intenção de se acrescentar ou modificar métodos da classe Thread, então a extensão dessa classe se molda melhor, do ponto de vista conceitual.	✕	A implementação do método **run ()** da interface **Runnable** parece se adequar melhor à criação de uma thread. Além disso, como Java não aceita herança múltipla, estender a classe Thread pode complicar desnecessariamente o modelo de classes, caso não haja a necessidade de se alterar o seu comportamento. Essa razão também está estreitamente ligada aos princípios de OO.

Como podemos perceber, a escolha de qual abordagem usar é mais conceitual do que prática.

A seguir, veremos três exemplos de códigos. O Código 1 e o Código 2 mostram a definição de threads com ambas as abordagens, enquanto o Código 3 mostra o seu emprego.

Código 1: definindo thread por extensão da classe Threads.

Java

Código 2: definindo threads por implementação de Runnable.

Java

Código 3: criando threads.

Java

Atividade 3

Imagine que você esteja conversando com um colega de trabalho que está desenvolvendo um sistema em Java. Ele tem algumas dúvidas sobre ciclo de vida de threads e faz a seguinte pergunta: qual estado de uma thread em Java indica que ela está pronta para ser executada pelo escalonador, mas ainda não está em execução?

A New

B Runnable

C Blocked

D Waiting

E Terminated

Parabéns! A alternativa B está correta.

Em síntese, no estado New, a thread foi criada, mas não está em execução. No estado Runnable, a thread está pronta para ser executada pelo escalonador de threads do sistema operacional. No estado Blocked, a execução da thread é suspensa. A thread entra no estado Waiting quando ocorre a chamada aos métodos wait() ou join(). E a thread entra no estado Terminated quando encerra sua execução.

Demonstrando a preempção de processos e

threads na prática

A compreensão dos sistemas computacionais modernos, especialmente no gerenciamento de processos e threads, é muito importante. Em sistemas operacionais multitarefa como Windows, Linux ou macOS, a preempção permite suspender a execução de um processo para priorizar tarefas mais importantes ou que necessitem de recursos específicos. Esse mecanismo garante um sistema responsivo e eficiente, essencial para otimizar o desempenho, evitar bloqueios e melhorar a experiência do usuário. Além disso, o entendimento da preempção ajuda a identificar e resolver problemas de desempenho e concorrência, contribuindo para sistemas mais estáveis.

Neste vídeo, mostraremos como analisar os processos em execução no computador, como alterar a prioridade deles, além de mostrar a quantidade de threads que cada processo tem.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Roteiro de prática

Considerando o sistema operacional Windows, mais especificamente o Windows 11, utilizado na elaboração deste material, podemos observar os processos que estão em execução. Para isso, é necessário acessar o gerenciador de tarefas.

Passo a passo:

1 - Utilize as teclas CTRL + ALT + DEL e, depois, selecione a opção gerenciador de tarefas. Na sequência, selecione a opção Processos.

Exemplo de processos exibidos por meio do gerenciador de tarefas.

É possível observar vários softwares em execução no computador, mostrados em aplicativos no gerenciador de tarefas. Além deles, há vários outros processos em segundo plano. Todos eles consomem recursos de processamento, memória etc.

2 - Podemos ainda alterar a prioridade de processos. Para isso, ainda na tela do gerenciador de processos, selecione a opção Detalhes.

Mostrando os detalhes dos processos.

3 - Depois, clique com o botão direito do mouse sobre o processo que você deseja alterar a prioridade. Neste exemplo, o processo selecionado foi o Word. Escolha a opção Definir prioridade. Observamos que há várias opções e, inicialmente, o Word tem prioridade Normal. Podemos alterá-la escolhendo alguma das opções mostradas. Foi selecionada a opção Alta, e dessa forma, o Word terá maior prioridade de execução se comparado a outros processos com prioridades inferiores.

Alterando a prioridade de processos.

4 - Também é possível visualizar a quantidade de threads de cada processo em execução no seu computador. Para isso, na tela do gerenciador de tarefas, escolha a opção Detalhes e, depois, clique com o botão direito do mouse sobre as colunas, selecionando a opção Selecionar colunas, conforme esta imagem:

Habilitando a visualização de threads.

5 - Depois, selecione a opção Threads, para que seja possível visualizar a quantidade de threads. O Word tem 45 threads no momento da geração da imagem. Veja!

Visualizando a quantidade de threads.

Atividade 4

Imagine que você precise executar um software de outra empresa várias vezes, alterando algumas configurações do algoritmo de aprendizado de máquina desse software, para mitigar os problemas de classificação de fraudes em sistemas bancários. O que você pode fazer para acelerar a execução desse software no seu computador?

A Uma vez que o software esteja pronto e em execução, não há o que fazer para melhorar seu desempenho, ou seja, toda vez que o software iniciar, gastará o mesmo tempo de execução.

B O software precisa ser desenvolvido novamente, utilizando linguagem de programação funcional.

C Deve-se optar por executar o software utilizando computação em nuvem porque não é possível alterar o tempo de execução de software em um computador.

D Você deve reduzir a quantidade de threads utilizadas no escalonador do sistema operacional porque, quanto maior for a quantidade de threads, maior será o tempo de execução.

E Você pode alterar a prioridade do processo relacionado ao software no escalonador do sistema operacional, selecionando opção mais alta de prioridade.

Parabéns! A alternativa E está correta.

Quando você escolhe a prioridade Alta para um processo no gerenciador de tarefas do Windows, o sistema operacional dá a esse processo mais tempo de CPU em comparação com processos de prioridade normal ou baixa. Isso significa que o processo com prioridade alta será executado com mais frequência, potencialmente melhorando seu desempenho e tempo de resposta.

2 - Sincronização entre threads

Ao final deste módulo, você será capaz de identificar a sincronização entre threads em Java.

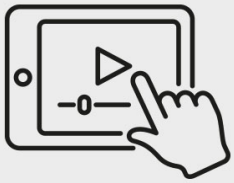
Conceitos sobre threads e paralelismo

Threads são unidades de execução independentes dentro de um programa, permitindo a realização de múltiplas tarefas simultaneamente. Em Java, as threads compartilham o mesmo espaço de memória do processo principal, facilitando a comunicação entre elas.

Paralelismo é a capacidade de executar várias operações ou tarefas ao mesmo tempo, aproveitando múltiplos núcleos de processadores modernos. Utilizando threads, um programa pode dividir suas tarefas em partes menores que rodam em paralelo, aumentando a eficiência e a velocidade de execução. Isso é essencial para aplicações que requerem multitarefa, como processamento de dados em tempo real e interações de usuário.

Neste vídeo, analisaremos mais detalhes sobre threads e paralelismo. Veremos que as threads são unidades de execução independentes dentro de um programa que permitem realizar múltiplas tarefas simultaneamente em Java. Além disso, estudaremos o conceito de paralelismo, a capacidade de executar várias operações ao mesmo tempo, aproveitando os múltiplos núcleos dos processadores modernos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Conceitos

Imagine que desejamos realizar uma busca textual em um documento com milhares de páginas, com a intenção de contar o número de vezes em que determinado padrão ocorre. Podemos fazer isso das seguintes formas:

Método 1

O método básico consiste em varrer sequencialmente as milhares de páginas, incrementando uma variável cada vez que o padrão for detectado. Esse procedimento certamente atenderia ao nosso objetivo, mas será que podemos torná-lo mais eficiente?

Método 2

Outra abordagem possível é dividir o documento em várias partes e executar várias instâncias da nossa aplicação simultaneamente. Apesar de conseguirmos reduzir o tempo de busca dessa forma, ela exige a soma manual dos resultados, o que não se mostra uma solução elegante para um bom programador.

Método 3

Podemos criar um certo número de threads e repartir as páginas do documento entre as threads, deixando a própria aplicação consolidar o resultado. Essa solução, embora tecnicamente engenhosa, é mais simples de descrever do que de fazer.

O método 3 parece ser o ideal para realizar nossa tarefa. Mas, ao paralelizar uma aplicação com o uso de threads, duas questões importantes se colocam:

Como realizar a comunicação entre as threads?

Como coordenar as execuções de cada thread?

Questões acerca do emprego de threads

Continuemos com nosso exemplo: nele, cada thread está varrendo em paralelo determinado trecho do documento. Como dissemos, cada uma faz a contagem do número de vezes que o padrão ocorre. Sabemos que threads compartilham o espaço de memória, então seria bem inteligente se fizéssemos com que cada thread incrementasse a mesma variável responsável pela contagem. Mas aí está nosso primeiro problema:

> **Cada thread pode estar sendo executada em um núcleo de CPU distinto, o que significa que elas estão, de fato, correndo em paralelo. Suponha, agora, que duas encontrem o padrão buscado ao mesmo tempo e decidam incrementar a variável de contagem também simultaneamente.**

Em um nível mais baixo, o incremento é formado por diversas operações mais simples que envolvem a soma de uma unidade, a leitura do valor acumulado e a escrita do novo valor em memória.

Lembre-se de que, no nosso exemplo, as duas threads estão fazendo tudo simultaneamente e que, sendo assim, elas lerão o valor acumulado (digamos que seja X).

Ambas farão o incremento desse valor em uma unidade (X+1) e ambas tentarão escrever esse novo valor em memória. Duas coisas podem

ocorrer:

1. A colisão na escrita pode fazer com que uma operação de escrita seja descartada. Ou seja, quando duas ou mais threads tentam escrever em uma variável ao mesmo tempo, pode acontecer que apenas uma dessas operações seja realizada efetivamente.
2. Diferenças de microssegundos podem fazer com que as escritas ocorram com uma defasagem infinitesimal. Isso significa que o intervalo de tempo entre duas operações de escrita pode ser tão curto que o valor X+1 poderia ser escrito duas vezes.

Em ambos os casos, o resultado será incorreto (o certo é X+2).

Podemos resolver esse problema se conseguirmos coordenar as duas threads de maneira que, quando uma inicie uma operação sobre a variável, a outra aguarde até que a operação esteja finalizada. Para fazermos essa coordenação, será preciso que as threads troquem mensagens, contudo elas são entidades semi-independentes rodando em núcleos distintos da CPU. Não se trata de dois objetos instanciados na mesma aplicação. Aliás, precisaremos da MVJ para sermos capazes de enviar uma mensagem entre threads.

Felizmente esses e outros problemas consequentes do paralelismo de programação são bem conhecidos e há técnicas para lidar com eles. A linguagem Java oferece diversos mecanismos para comunicação entre threads e nas próximas seções vamos examinar dois deles:

Semáforos

Monitores

A seguir, também falaremos sobre objetos imutáveis e seu compartilhamento.

Atividade 1

Considere que você esteja participando de um processo seletivo para uma vaga de trabalho como desenvolvedor e se deparou com a seguinte

questão. Qual é o papel das threads e do paralelismo em Java? Marque a resposta correta.

A Threads permitem a execução de múltiplos programas simultaneamente. Paralelismo é a capacidade de lidar com erros de execução desses programas.

B Threads são partes de um programa que interagem com a interface do usuário. Paralelismo refere-se à interação entre diferentes threads no mesmo programa.

C Threads são unidades de execução dentro de um programa, permitindo a realização de múltiplas tarefas simultaneamente. Paralelismo é a capacidade de executar várias tarefas ao mesmo tempo, aproveitando múltiplos núcleos de processadores.

D Threads são pequenas funções que operam separadamente no código. Paralelismo envolve a compactação de múltiplos threads em um único processo.

E Threads são usadas para depurar código em Java. Paralelismo refere-se à capacidade de isolar erros de programação.

Parabéns! A alternativa C está correta.

Em Java, threads são unidades independentes de execução que permitem dividir um programa em partes menores que podem ser executadas simultaneamente. Isso possibilita o paralelismo, que é a capacidade de aproveitar os múltiplos núcleos dos processadores modernos para melhorar a eficiência e a velocidade de execução do programa. As outras opções não descrevem corretamente os conceitos de threads e paralelismo em Java.

Comunicação entre threads: semáforos e monitores

A comunicação entre threads ocorre através de mecanismos como semáforos e monitores em Java. Semáforos são contadores que controlam o acesso concorrente a recursos compartilhados, permitindo threads aguardarem ou liberarem permissões para acesso. Eles podem ser utilizados para garantir a sincronização e evitar problemas. Por outro lado, monitores são estruturas de sincronização que garantem que apenas uma thread execute um bloco de código crítico por vez, usando métodos ou blocos sincronizados em Java.

Ambos os conceitos são fundamentais para coordenar o acesso seguro a recursos compartilhados entre threads, garantindo consistência e prevenindo erros de concorrência.

Neste vídeo, estudaremos semáforos e monitores, que facilitam a comunicação entre threads em Java. Veremos como os semáforos atuam como contadores para controlar o acesso concorrente a recursos compartilhados, enquanto os monitores garantem a execução segura de blocos críticos por uma única thread por vez.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Semáforos

As técnicas para evitar os problemas já mencionados envolvem uso de travas, atomização de operações, semáforos, monitores e outras. Essencialmente, o que buscamos é evitar as causas que levam aos problemas. Por exemplo, ao usarmos uma trava sobre um recurso, evitamos o que é chamado de condição de corrida. Vimos isso superficialmente no tópico anterior.

Comentário

Problemas inerentes a acessos compartilhados de recursos e paralelismo de processamento são muito estudados em sistemas

operacionais e sistemas distribuídos. O seu estudo detalhado excederia o nosso propósito, mas vamos explorar essas questões dentro do contexto da programação Java.

Inicialmente, falaremos de maneira conceitual a respeito do **semáforo**.

Conceitualmente, o semáforo é um mecanismo que controla o acesso de processos ou threads a um recurso compartilhado. Ele pode ser usado para controlar o acesso a uma região crítica (recurso) ou para sinalização entre duas threads. Por meio do semáforo podemos definir quantos acessos simultâneos podem ser feitos a um recurso. Para isso, uma variável de controle é usada e são definidos métodos para a solicitação de acesso ao recurso e de restituição do acesso após terminado o uso do recurso obtido.

Esse processo acontece da seguinte forma:

Solicitação de acesso ao recurso

Quando uma thread deseja acesso a um recurso compartilhado, ela invoca o método de solicitação de acesso. O número máximo de acessos ao recurso é dado pela variável de controle.

Controle de acessos

Quando uma solicitação de acesso é feita, se o número de acessos que já foi concedido for menor do que o valor da variável de controle, o acesso é permitido e a variável é decrementada. Se o acesso for negado, a thread é colocada em espera numa fila.

Liberação do recurso obtido

Quando uma thread termina de usar o recurso obtido, ela invoca o método que o libera e a variável de controle é incrementada. Nesse momento, a próxima thread da fila é despertada para acessar o recurso.

Desde a versão 5, Java oferece uma implementação de semáforo por meio da classe Semaphore (ORACLE AMERICA INC., s.d.). Os métodos para acesso e liberação de recursos dessa classe são:

Acquire ()

Método que solicita acesso a um recurso ou uma região crítica, realizando o bloqueio até que uma permissão de acesso esteja disponível ou a thread seja interrompida.

Release ()

Método responsável pela liberação do recurso pela thread.

Em Java, o número de acessos simultâneos permitidos é definido pelo construtor na instanciação do objeto.

Dica

O construtor também oferece uma versão sobrecarregada em que o segundo parâmetro define a justeza (**fair**) do semáforo, ou seja, se o semáforo utilizará ou não uma fila (FIFO) para as threads em espera.

Os métodos acquire () e release () possuem uma versão sobrecarregada que permite a aquisição/liberação de mais de uma permissão de acesso.

O código a seguir mostra um exemplo de criação de semáforo em Java.

Java

Caso o semáforo seja criado com o parâmetro fair falso, ele não utilizará uma FIFO.

Exemplo

Imagine que temos um semáforo que permite apenas um acesso à região crítica e que essa permissão de acesso foi concedida a uma thread (thread 0). Em seguida, uma nova permissão é solicitada, mas como não há acessos disponíveis, a thread (thread 1) é posta em espera. Quando a thread 0 liberar o acesso, se uma terceira thread (thread 2) solicitar permissão de acesso antes de que a thread 1 seja capaz de fazê-lo, ela obterá a permissão e bloqueará a thread 1 novamente.

O exemplo anterior também mostra um caso particular no qual o semáforo é utilizado como um mecanismo de exclusão mútua, parecido com o mutex (mutual exclusion). Na prática, há diferença entre esses mecanismos:

Semáforo		Mutex
Não verifica se a liberação de acesso veio da mesma thread que a solicitou.	✕	Faz a verificação para garantir que a liberação veio da thread que a solicitou.

Como vimos, a checagem de propriedade diferencia ambos. Não obstante, um semáforo com o número máximo de acessos igual a 1 também se comporta como um mecanismo capaz de realizar a exclusão mútua.

Vamos nos valer dessa diferença quanto à checagem para utilizar o semáforo para enviar sinais entre duas threads. A ideia, nesse caso, é que a invocação de acquire () seja feita por uma thread (thread 0) e a invocação de release (), por outra (thread 1). Vamos exemplificar:

Inicialmente, um semáforo é criado com limite de acesso igual a 0.

A thread 0, então, solicita uma permissão de acesso e bloqueia.

A thread 1 invoca release (), o que incrementa a variável de controle do semáforo e desbloqueia a thread 0.

Dessa forma, conseguimos enviar um sinal da thread 1 para a 0. Se utilizarmos um segundo semáforo com a mesma configuração, mas invertendo quem faz a invocação dos métodos, teremos uma maneira de sinalizar da thread 0 para a 1.

O exemplo a seguir facilitará o entendimento acerca do uso de semáforos na sinalização entre threads. Em nosso exemplo, criaremos uma classe (PingPong) para disparar as outras threads.

Observe no código da nossa Thread Mãe (classe PingPong), a seguir, que as linhas 17 e 18 disparam as outras threads. Os semáforos são criados nas linhas 11 e 12 com número de acesso máximo igual a zero. Isso é necessário para permitir que ambas as threads (Ping e Pong) bloqueiem após o seu início.

O comando para desbloqueio é visto na linha 19.

Java

Vamos analisar os códigos das outras threads, começando pela Thread A (classe Ping).

Java

Agora, temos o código da Thread B (classe Pong).

Java

Observe a linha 19 dos códigos das threads A e B. Quando essas linhas são executadas, o comando acquire() faz com que o bloqueio ocorra e este durará até a execução da linha 19 do código da Thread Mãe, onde o comando release() irá desbloquear a Thread A. Após o desbloqueio, segue-se uma troca de sinalizações entre as threads até o número máximo definido pela linha 7 do código da classe Principal, a seguir.

Java

Em nosso exemplo, uma classe (PingPong) é criada para disparar as outras threads, conforme vemos nas linhas 17 e 18 do código da Thread Mãe. Os semáforos são criados com número de acesso máximo igual a zero (linhas 11 e 12 da Thread Mãe), para permitir que ambas as threads (Ping e Pong) bloqueiem após o seu início. O bloqueio ocorre quando a linha 19 das threads A e B são executadas. Ao executar a linha 19 do código da Thread Mãe, a Thread A é desbloqueada e, a partir daí, há uma troca de sinalizações entre as threads até o número máximo definido pela linha 7 da classe Principal.

A seguir, podemos observar duas execuções sucessivas de aplicação:

Terminal

Terminal

Você deve ter notado que as linhas 2 e 3 se invertem. Mas qual a razão disso? Como dissemos anteriormente, o agendamento da execução de uma thread não é determinístico. Isso significa que não sabemos quando ele ocorrerá. Tudo que podemos fazer é garantir a sequencialidade entre as regiões críticas. Veja que as impressões de “PING => 0” e “0 <= PONG” sempre se alternam. Isso se manterá, não importando o número de vezes que executemos a aplicação, pois garantimos a sincronia por meio dos semáforos. Já a execução da linha

7 da Thread A e da Thread B estão fora da região crítica e por isso não possuem sincronismo.

Monitores

Vamos retornar ao problema hipotético apresentado no início do módulo. Nele, precisamos proceder ao **incremento de uma variável**, garantindo que nenhuma outra thread opere sobre ela antes de terminarmos de incrementá-la.

O que precisamos fazer, para evitar problemas, é ativar um controle imediatamente antes da leitura em memória, dando início à proteção da operação. Após a última operação, o controle deve ser desativado.

incremento de uma variável

Em linhas gerais, implica ler o conteúdo em memória, acrescê-lo de uma unidade e gravá-lo novamente em memória.

> **Em outras palavras, estamos transformando a operação de incremento em uma operação atômica, ou seja, indivisível. Uma vez iniciada, nenhum acesso à variável será possível até que a operação termine.**

Para casos como esse, a linguagem Java provê um mecanismo chamado de monitor. Um monitor é uma implementação de sincronização de threads que permite:

Exclusão mútua entre threads

No monitor, a exclusão mútua é feita por meio de um mutex (lock) que garante o acesso exclusivo à região monitorada.

Cooperação entre threads

A cooperação implica que uma thread possa abrir mão temporariamente do acesso ao recurso, enquanto aguarda que alguma condição ocorra. Para isso, um sistema de sinalização entre as threads deve ser provido.

Ele recebe o nome de monitor porque se baseia no monitoramento de como as threads acessam os recursos.

Atenção!

Classes, objetos ou regiões de códigos monitorados são ditos thread-safe, indicando que seu uso por threads é seguro.

A linguagem Java implementa o conceito de monitor por meio da palavra reservada **synchronized**. Esse termo é utilizado para marcar regiões críticas de código que, portanto, deverão ser monitoradas. Em Java, cada objeto está associado a um monitor, que uma thread pode travar ou destravar. O uso de synchronized pode ser aplicado a um método ou a uma região menor de código. Ambos os casos são mostrados no código a seguir.

Java

Quando um método sincronizado (synchronized) é invocado, ele automaticamente dá início ao travamento da região crítica. A execução do método não começa até que o bloqueio tenha sido garantido. Uma vez terminado, mesmo que o método tenha sido encerrado anormalmente, o travamento é liberado. É importante perceber que quando se trata de um método de instância, o travamento é feito no monitor associado àquela instância. Em oposição, métodos static realizam o travamento do monitor associado ao objeto Class, representativo da classe na qual o método foi definido (ORACLE AMERICA INC., s.d.).

Em Java, todo objeto possui um wait-set associado que implementa o conceito de conjunto de threads. Essa estrutura é utilizada para permitir a cooperação entre as threads, fornecendo os seguintes métodos:

wait ()

Adiciona a thread ao conjunto wait-set, liberando a trava que aquela thread possui e suspendendo sua execução. A MVJ mantém uma estrutura de dados com as threads adormecidas que aguardam acesso à região crítica do objeto.

notify ()

Acorda a próxima thread que está aguardando na fila e garante o acesso exclusivo à thread despertada. Nesse momento a thread é removida da estrutura de espera.

notifyAll ()

Faz basicamente o mesmo que o método notify (), mas acordando e removendo todas as threads da estrutura de espera. Entretanto, mesmo nesse caso apenas uma única thread obterá o travamento do monitor, isto é, o acesso exclusivo à região crítica.

Você pode observar nos códigos da Thread A e Thread B de nosso exemplo anterior, que na linha 4 declaramos um objeto da classe Controle. Verificando a linha 18 fica claro que utilizamos esse objeto para contar o número de execuções das threads. A cada execução da região crítica, o contador é decrementado (linha 22). Essa situação é análoga ao problema que descrevemos no início e que enseja o uso de monitores. E, de fato, como observamos no próximo código, os métodos decrementa () e getControle () são sincronizados.

Java

Atividade 2

Imagine que você tenha acessado um fórum sobre desenvolvimento de software, mais especificamente sobre comunicação entre threads. Alguém inseriu o seguinte questionamento: Como a comunicação entre threads é facilitada em Java usando semáforos e monitores? Foram postadas várias respostas, indicadas nas opções a seguir. Qual delas está correta?

A Semáforos garantem que apenas uma thread execute um bloco de código crítico por vez.

B Semáforos são usados para executar operações aritméticas entre threads.

C Monitores controlam o número total de threads em execução simultaneamente.

D Monitores garantem que várias threads acessem um recurso compartilhado ao mesmo tempo.

E Semáforos e monitores são conceitos irrelevantes para a programação multithread em Java.

Parabéns! A alternativa A está correta.

Em Java, semáforos são utilizados como contadores para controlar o acesso concorrente a recursos compartilhados. Eles permitem que

threads aguardem ou liberem permissões para acesso, garantindo sincronização e prevenindo condições de corrida. Já monitores em Java são estruturas de sincronização que asseguram que apenas uma thread execute um bloco de código crítico (métodos ou blocos sincronizados) por vez, o que é essencial para coordenar o acesso seguro a recursos compartilhados e evitar erros de concorrência.

Objetos imutáveis

Objetos imutáveis são aqueles cujo estado não pode ser alterado após sua criação. Em programação multithread em Java, objetos imutáveis oferecem segurança ao serem compartilhados entre threads. Como não podem ser modificados, não há preocupações com condições de corrida ou sincronização. Cada thread pode acessar simultaneamente um objeto imutável sem medo de alterações inesperadas. Essa característica não apenas simplifica o desenvolvimento de software concorrente, mas também melhora a eficiência e a robustez das aplicações ao minimizar os problemas de sincronização e acesso concorrente a dados.

Neste vídeo, estudaremos as características dos objetos imutáveis, que são cruciais na programação multithread em Java. São objetos cujo estado não pode ser alterado após a criação e garantem segurança ao serem compartilhados entre threads.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Um objeto é considerado **imutável** quando seu estado não pode ser modificado após sua criação. Objetos podem ser construídos para ser imutáveis, mas a própria linguagem Java oferece classes de objetos com essa característica. O tipo String é um caso de classe que define objetos imutáveis. Caso sejam necessários objetos string mutáveis, Java disponibiliza duas classes, StringBuffer e StringBuilder, que permitem criar objetos do tipo String mutáveis (SCHILDT, 2014).

O conceito de objeto imutável pode parecer uma restrição problemática, mas na verdade há vantagens. Uma vez que já se sabe que o objeto não pode ser modificado, o código se torna mais seguro e o processo de

coleta de lixo mais simples. Já a restrição pode ser contornada ao criar um novo objeto do mesmo tipo que contenha as alterações desejadas.

No caso que estamos estudando, a vantagem é bem óbvia:

> **Se um objeto não pode ter seu estado alterado, não há risco de que ele se apresente num estado inconsistente, ou seja, que tenha seu valor lido durante um procedimento que o modifica, por exemplo. Acessos múltiplos de threads também não poderão corrompê-lo. Assim, objetos imutáveis são thread-safe.**

Em linhas gerais, se você deseja criar um objeto imutável, métodos que alteram o estado do objeto (set) não devem ser providos. Também se deve evitar que alterações no estado sejam feitas de outras maneiras. Logo, todos os campos devem ser declarados privados (private) e finais (final). A própria classe deve ser declarada final ou ter seu construtor declarado privado.

Atenção!

É preciso cuidado especial caso algum atributo faça referência a um objeto mutável. Essa situação exige que nenhuma forma de modificação desse objeto seja permitida.

Podemos ver um exemplo de classe que define objetos imutáveis no código a seguir.

Java

Após observar os dois últimos exemplos, fica claro que a classe Controle não é imutável, razão pela qual necessita do modificador synchronized. No entanto, o método usado para compartilhar o objeto contador (linhas 6 e 13 do código Thread Mãe) é o mesmo. Cada thread a acessar o objeto imutável criado deve possuir uma variável de referência do tipo da classe desse objeto. Uma vez criado o objeto, as threads precisam receber sua referência e, a partir de então, terão acesso ao objeto.

Apesar de contador não ser um objeto imutável, ele exemplifica esse mecanismo de compartilhamento de objetos entre threads. Na linha 13 da Thread Mãe ele é criado, e nas linhas 14 e 15 a referência para o objeto criado é passada para as threads Ping e Pong.

Atividade 3

Durante uma palestra sobre programação multithreading, o palestrante perguntou ao público por que os objetos imutáveis são recomendados em Java para esse tipo de programação. Vários participantes deram suas respostas, listadas nas opções a seguir. Qual delas é a correta?

A Porque objetos imutáveis consomem menos memória durante a execução de threads.

B Porque objetos imutáveis podem ser modificados por várias threads simultaneamente sem problemas.

C Porque objetos imutáveis garantem que o estado do objeto não será alterado após sua criação, simplificando a sincronização entre threads.

D Porque objetos imutáveis são mais rápidos em comparação com objetos mutáveis.

E Porque objetos imutáveis não precisam ser inicializados antes de serem compartilhados entre threads.

Parabéns! A alternativa E está correta.

Em programação multithread em Java, objetos imutáveis oferecem segurança ao serem compartilhados entre threads. Como não podem ser modificados após sua criação, não há preocupações com condições de corrida ou sincronização. Cada thread pode acessar simultaneamente um objeto imutável sem medo de alterações inesperadas, o que simplifica a programação concorrente e minimiza os problemas de sincronização e acesso concorrente a dados.

Como evitar problemas com o uso de semáforos e monitores

O conhecimento prático em programação multithread é essencial para evitar problemas complexos e garantir a robustez das aplicações em Java. Semáforos e monitores são fundamentais para sincronizar threads e controlar o acesso a recursos compartilhados. Semáforos atuam como contadores que permitem ou bloqueiam o acesso concorrente a esses recursos, enquanto monitores garantem que apenas uma thread execute um bloco crítico de código por vez, usando métodos sincronizados ou blocos synchronized.

A aplicação incorreta ou a falta de compreensão desses mecanismos pode causar condições de corrida, levando a resultados não determinísticos e inesperados, como inconsistências nos dados, deadlocks ou falhas na lógica do programa. Também há o deadlock (bloqueio mútuo) e inconsistências nos dados compartilhados entre threads, resultando em comportamentos imprevisíveis e erros difíceis de diagnosticar. Portanto, entender e aplicar corretamente semáforos e monitores é muito importante para desenvolver aplicações robustas e eficientes em Java, garantindo a integridade dos dados e evitando problemas de concorrência que poderiam comprometer a estabilidade e o desempenho do sistema em ambientes de execução simultânea de tarefas.

Neste vídeo, estudaremos como semáforos e monitores são essenciais para garantir a estabilidade e robustez das aplicações em Java, evitando problemas complexos, como condições de corrida e deadlock.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Roteiro de prática

Imagine uma classe Contador que possua um método incrementar() responsável por incrementar um contador compartilhado entre múltiplas threads. A implementação inicial pode ser a seguinte:

Java

Aparentemente, esse código pode parecer correto. No entanto, se duas ou mais threads acessarem o método incrementar() simultaneamente, podemos enfrentar uma condição de corrida devido à natureza não atômica da operação de incremento, ou seja, envolve três operações: leitura, incremento e gravação do novo valor do incremento.

Para corrigir a condição de corrida no exemplo do contador compartilhado entre threads, podemos implementar um semáforo em Java para sincronizar o acesso ao método incrementar(). Confira um

exemplo de como isso pode ser feito usando Semaphore da biblioteca padrão java.util.concurrent:

Java

Explicação do código:

- **Semaphore:** é uma classe que mantém um contador de permissões. No exemplo, inicializamos o semáforo com 1 permissão (new Semaphore(1)), o que significa que apenas uma thread por vez pode adquirir o semáforo.
- **Método incrementar():** utiliza semaforo.acquire() para adquirir uma permissão antes de incrementar o contador. Isso garante que apenas uma thread por vez possa executar o bloco crítico dentro do método. Após a conclusão do incremento, semaforo.release() é chamado para liberar a permissão do semáforo, permitindo que outras threads possam adquiri-la.
- **getContador():** retorna o valor atual do contador de forma segura, já que a sincronização do acesso ao contador é feita através do semáforo.

Com essa implementação, garantimos que qualquer operação de incremento no contador seja realizada de forma exclusiva por uma thread de cada vez, eliminando a condição de corrida e mantendo a consistência dos dados compartilhados entre threads.

Agora, vamos implementar a classe TesteContador para podermos demonstrar a execução da classe Contador que contém um semáforo, confira:

Java

Para corrigir a condição de corrida no exemplo do contador utilizando um monitor em Java, podemos usar o bloco synchronized para garantir que apenas uma thread execute o método incrementar() por vez. Aqui está como podemos implementar isso:

Java

Explicação do código:

- **Método incrementar():** é declarado como synchronized, o que significa que apenas uma thread por vez pode executar esse método para uma instância específica de Contador. Isso garante

que não haverá condições de corrida durante a operação de incremento do contador.

- **Método getContador():** retorna o valor atual do contador de forma segura. Não há necessidade de sincronização nesse método, pois ele apenas lê o valor do contador e não altera o estado do objeto.

Com essa implementação usando um monitor (synchronized), garantimos que o acesso concorrente ao método incrementar() seja sincronizado, evitando problemas como condições de corrida e mantendo a consistência dos dados compartilhados entre as threads.

Outro problema típico é o deadlock, que ocorre quando duas ou mais threads ficam bloqueadas indefinidamente, aguardando que recursos exclusivos sejam liberados pela outra. Isso cria uma situação de impasse em que nenhuma thread pode progredir, comprometendo a execução correta do programa.

Atividade 4

Imagine que você faz parte da equipe de desenvolvimento de um sistema que está apresentando problemas. Depois de algum tempo, um colega de trabalho observou que faltava inserir a linha de código **semaforo.acquire()**, conforme mostrado no trecho de código a seguir, sendo que semaforo é um recurso do tipo Semaphore. Ele fez a correção e o sistema passou a funcionar sem problemas. Qual é a explicação para isso acontecer? Marque a resposta correta.

Java

A A linha de código semaforo.acquire() adquire a permissão do semáforo, enquanto a linha de código semaforo.release() libera a permissão do semáforo.

Sem a permissão do semáforo, há problemas na execução do software.

B Se o sistema apresentou problemas, certamente não é por causa da linha semaforo.acquire(), que simplesmente possibilita alterar a prioridade de threads em Java.

C A linha de código semaforo.acquire() possibilita acessar o atributo semaforo e, sem o método acquire(), esse atributo não tem funcionalidade.

D O problema ocorreu porque a linha de código semaforo.acquire() é utilizada para transformar o objeto semaforo em objeto imutável.

E O problema ocorreu porque se utilizou código de monitor como se fosse semáforo.

Parabéns! A alternativa A está correta.

O uso de semáforos em Java é crucial para coordenar o acesso concorrente a recursos compartilhados entre threads, garantindo sincronização e evitando condições de corrida e inconsistências nos dados. E a linha semaforo.acquire() é utilizada para adquirir a permissão do semáforo.

3 - Implementação de threads

Ao final deste módulo, você será capaz de aplicar a implementação de threads em Java.

Conceitos sobre a implementação de threads em Java

A implementação de threads em Java é essencial para desenvolver aplicações eficientes e robustas. Compreender os princípios de concorrência, sincronização, ciclo de vida das threads e gerenciamento de recursos é essencial para evitar problemas como condições de corrida, deadlock e inconsistências nos dados. Além disso, esse entendimento permite aos desenvolvedores escolher corretamente entre semáforos, monitores e variáveis de condição para garantir a sincronização adequada entre threads, melhorando a performance das aplicações em ambientes multitarefa e contribuindo para a estabilidade e segurança dos sistemas desenvolvidos.

Neste vídeo, estudaremos sobre a importância do conhecimento teórico na implementação de threads em Java para o desenvolvimento de aplicações robustas e eficientes.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

O mundo da programação paralela é vasto, e mesmo as threads vão muito além do que este conteúdo pode abarcar. Porém, desde que você tenha compreendido a essência, explorar todos os recursos que Java oferece para programação paralela será questão de tempo e prática. Para auxiliá-lo a sedimentar os conhecimentos adquiridos até o momento, vamos apresentar um exemplo que busca ilustrar os principais pontos que abordamos, inicialmente fazendo uma introdução sucinta da classe Thread e seus métodos. Em seguida, apresentaremos um caso prático e terminaremos com algumas considerações gerais pertinentes.

Classe Thread e seus métodos

A API Java oferece diversos mecanismos que suportam a programação paralela. Não é nosso objetivo explorar todos eles, contudo não podemos nos propor a examinar as threads em Java e não abordar a classe Thread e seus principais métodos. A classe é bem documentada na API e apresenta uma estrutura com duas classes aninhadas (State e UncaughtExceptionHandler), campos relativos à prioridade (MAX_PRIORITY, NORM_PRIORITY e MIN_PRIORITY) e vários métodos (ORACLE AMERICA INC., s.d.).

Como podemos concluir, os campos guardam as prioridades máxima, mínima e default da thread respectivamente. A seguir, vamos conhecer alguns métodos relevantes:

getPriority () e setPriority (int pri)

O método getPriority () devolve a prioridade da thread, enquanto setPriority (int pri) é utilizado para alterar a prioridade da thread. Quando uma nova thread é criada, ela herda a prioridade da thread que a criou. Isso pode ser alterado posteriormente pelo método setPriority (int pri), que recebe como parâmetro um valor inteiro correspondente à nova prioridade a ser atribuída. Observe, contudo, que esse valor

deve estar entre os limites mínimo e máximo, definidos respetivamente por MIN_PRIORITY e MAX_PRIORITY.

getState ()

Outro método relevante é o getState (). Esse método retorna o estado no qual a thread se encontra, com vimos na figura da **máquina de estados da thread** (os estados possíveis da thread são: NEW, RUNNABLE, BLOCKED, TIMED_WAITING, WAITING ou TERMINATED) no início deste estudo e está descrito na documentação da classe State (ORACLE AMERICA INC., s.d.). Embora esse método possa ser usado para monitorar a thread, ele não serve para garantir a sincronização. Isso acontece porque o estado da thread pode se alterar entre o momento em que a leitura foi realizada e o recebimento dessa informação pelo solicitante, de maneira que a informação se torna obsoleta.

getId () e getName ()

Os métodos getId () e getName () são utilizados para retornar o identificador e o nome da thread. O identificador é um número do tipo long gerado automaticamente no momento da criação da thread, e permanece inalterado até o fim de sua vida. Apesar de o identificador ser único, ele pode ser reutilizado após a thread finalizar.

setName ()

O nome da thread pode ser definido em sua criação, por meio do construtor da classe, ou posteriormente, pelo método setName (). O nome da thread é do tipo String e não precisa ser único. Na verdade, o sistema se vale do identificador e não do nome para controlar as threads. Da mesma forma, o nome da thread pode ser alterado durante seu ciclo de vida.

currentThread ()

Caso seja necessário obter uma referência para a thread corrente, ela pode ser obtida com o método currentThread (), que retorna uma referência para um objeto Thread. A referência para o próprio objeto (this) não permite ao programador acessar a thread específica que está em execução.

join ()

Para situações em que o programador precise fazer com que uma thread aguarde outra finalizar para prosseguir, a classe Thread possui o método join (), que ocorre em três versões, sendo sobrecarregado da seguinte forma: join (), join (long millis) e join (long millis, int nanos). Suponha que uma Thread A precisa aguardar a Thread B finalizar antes de prosseguir seu processamento. A invocação de B.join () em A fará com que A espere (wait) indefinidamente até que B finalize. Repare que, se B morrer, A permanecerá eternamente aguardando por B.

Uma maneira de evitar que A se torne uma espécie de “zumbi” é especificar um tempo limite de espera (timeout), após o qual ela continuará seu processamento, independentemente de B ter finalizado. A versão join (long millis) permite definir o tempo de espera em milissegundos, e a outra, em milissegundos e nanossegundos. Nas duas situações, se os parâmetros forem todos zero, o efeito será o mesmo de join ().

run ()

É o método principal da classe Thread. Esse método modela o comportamento que é realizado pela thread quando ela é executada e, portanto, é o que dá sentido ao emprego da thread. Os exemplos mostrados nos códigos das threads A e B ressaltam esse método sendo definido numa classe que implementa uma interface Runnable. Mas a situação é a mesma para o caso em que se estende a classe Thread.

setDaemon ()

O método setDaemon () é utilizado para tornar uma thread, um daemon ou uma thread de usuário. Para isso, ele recebe um

parâmetro do tipo boolean. A invocação de setDaemon (true) marca a thread como daemon. Se o parâmetro for “false”, a thread é marcada como uma thread de usuário. Essa marcação deve ser feita, contudo, antes de a thread ser iniciada (e após ter sido criada). O tipo de thread pode ser verificado pela invocação de isDaemon (), que retorna “true” se a thread for do tipo daemon.

sleep (long millis)

É possível suspender temporariamente a execução de uma thread utilizando o método sleep (long millis), o qual faz com que a thread seja suspensa pelo período de tempo em milissegundos equivalente a millis. A versão sobrecarregada sleep (long millis, int nanos) define um período em milissegundos e nanossegundos. Porém, questões de resolução de temporização podem afetar o tempo que a thread permanecerá suspensa de fato. Isso depende, por exemplo, da granularidade dos temporizadores e da política do escalonador.

start () e stop ()

Talvez o método start () seja o mais relevante depois de run (). Esse método inicia a execução da thread, que passa a executar run (). O método start () deve ser invocado após a criação da thread e é ilegal invocá-lo novamente em uma thread em execução. Há um método que para a execução da thread (stop ()), mas, conforme a documentação, esse método está depreciado desde a versão 1.2. O seu uso é inseguro devido a problemas com monitores e travas e, em consequência disso, deve ser evitado. Uma boa discussão sobre o uso de stop () pode ser encontrada nas referências deste material.

yield ()

O último método que abordaremos é o yield (). Esse método informa ao escalonador do sistema que a thread corrente deseja ceder seu tempo de processamento. Ao ceder tempo de processamento, busca-se otimizar o uso da CPU, melhorando a performance. Contudo, cabem algumas observações:

primeiramente, quem controla o agendamento de threads e processos é o escalonador do sistema, que pode perfeitamente ignorar yield (). Além disso, é preciso bom conhecimento da dinâmica dos objetos da aplicação para se extrair algum ganho pelo seu uso. Tudo isso torna o emprego de yield () questionável.

Aqui não abordamos todos os métodos da classe Thread. Procuramos apenas examinar aqueles necessários para implementações básicas usando threads e que lhe permitirão explorar a programação paralela.

A API Java oferece outras classes úteis e importantes, a Semaphore e CountDownLatch, cuja familiaridade virá do uso. Aliás, conforme você melhore suas habilidades em programação com threads, descobrirá outros recursos que a API Java oferece. Por enquanto, para consolidar o aprendizado, vamos apresentar um exemplo que emprega diversos conhecimentos vistos anteriormente.

Atividade 1

Imagine que você esteja conversando com estagiários da empresa onde trabalha como desenvolvedor de sistemas. Em um momento, você perguntou a opinião deles sobre a importância dos conceitos de threads, especificamente em Java. As respostas estão listadas a seguir. Qual delas você considera mais adequada?

A Para otimizar consultas em bancos de dados relacionais.

B Para desenvolver interfaces gráficas de usuário responsivas.

C Para evitar problemas como condições de corrida e deadlock.

D

Para implementar criptografia em transmissões de dados.

E Para acelerar operações de entrada e saída em sistemas distribuídos.

Parabéns! A alternativa C está correta.

O conhecimento teórico sobre a implementação de threads em Java é essencial para entender princípios de concorrência, sincronização, ciclo de vida das threads e gerenciamento de recursos. Isso permite aos desenvolvedores escolher corretamente entre semáforos, monitores e variáveis de condição para garantir a sincronização adequada e evitar problemas como condições de corrida (quando múltiplas threads acessam simultaneamente recursos compartilhados) e deadlock (quando duas ou mais threads ficam bloqueadas indefinidamente aguardando recursos que nunca serão liberados).

Implementação de threads em Java na prática

O conhecimento prático na implementação de threads em Java é essencial para criar aplicações eficientes e responsivas em ambientes multitarefa. Dominar aspectos como criação, sincronização, ciclo de vida e gerenciamento de threads permite que se explore ao máximo os recursos de hardware disponíveis, como múltiplos núcleos de processador. Isso resulta em aplicações capazes de executar múltiplas tarefas simultaneamente, melhorando significativamente o desempenho e a capacidade de resposta em cenários complexos. Além disso, ajuda a evitar problemas comuns em programação concorrente, como condições de corrida e deadlocks, que podem afetar a integridade dos dados e a estabilidade do sistema. Entender os mecanismos de sincronização, como semáforos, monitores e variáveis de condição, permite criar código mais robusto e confiável, garantindo que as operações concorrentes sejam executadas de maneira segura e eficiente. Em resumo, o conhecimento prático sobre implementação de threads em Java melhora a qualidade do software e prepara os desenvolvedores para enfrentar os desafios da computação paralela.

Neste vídeo, estudaremos a importância do conhecimento prático na implementação de threads em Java para o desenvolvimento de aplicações eficientes e responsivas.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Roteiro de prática

Vamos analisar um exemplo simples e puramente didático, considerando uma empresa que trabalha com encomendas. Veja o diagrama de classes!

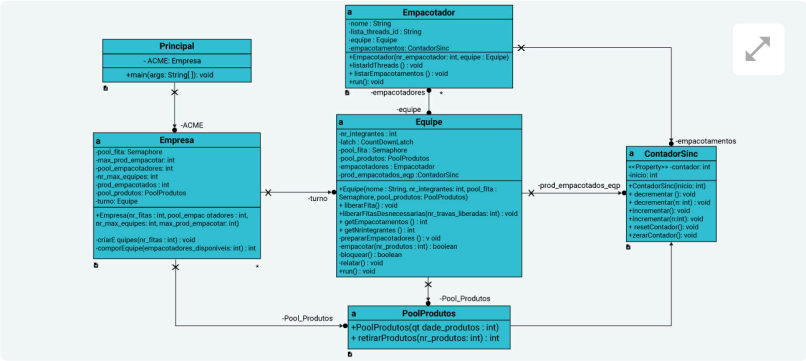


Diagrama de classes.

A classe Principal é a que possui o método main e se limita a disparar a execução da aplicação. Ela pode ser vista no código a seguir.

Java

Essa classe, que é a primeira thread a ser criada quando um programa é executado, instancia a classe Empresa. A instância ACME possui 20 fitas, 25 empregados e pode usar até 4 equipes para empacotar 200 produtos. Cada equipe formada corresponderá a uma thread, e cada empregado alocado também. Assim, a thread de uma equipe criará outras threads correspondentes aos seus membros. São os objetos Empacotador, que correspondem ao membro da equipe, que realizarão o empacotamento.

A classe Empresa realiza a montagem das equipes, distribuindo os funcionários, e inicia as threads correspondentes às equipes formadas. Os métodos comporEquipes e criarEquipes trabalham juntos para criar as equipes e definir quantos membros cada uma possuirá. Porém, o trecho que mais nos interessa nessa classe é o compreendido entre as linhas 33 e 43. Veja o código a seguir, que mostra a classe Empresa.

Java

Vejamos, então, como a classe Equipe funciona. Para isso, veja o próximo código.

Java

A classe Empacotador é mostrada no próximo código.

Java

A ocorrência de mais de uma chamada concorrente pode levar a uma condição de corrida. Para impedir isso, usamos synchronized, garantindo que somente uma execução do método ocorra ao mesmo tempo.

Java

Nossa última classe é a PoolProdutos (próximo código). Ela também é um contador que deve ser compartilhado por mais de uma thread, mas precisamos modelar um comportamento adicional, representado pelo método retirarProdutos na linha 15. Para isso, estendemos a classe ContadorSicn, adaptando-a para essa nova funcionalidade. Note que continuamos usando o synchronized, pelas mesmas razões de antes.

Java

Atividade 2

Suponha que você esteja desenvolvendo um sistema que usa programação multithreading em Java, e uma classe específica está apresentando problemas. O trecho de código a seguir se refere a essa classe.

Java

Qual será a solução do problema? Marque a resposta correta.

A Não há erros no trecho de código mostrado; o problema está em outro trecho de código.

B Na linha 1, não se trata da definição de uma classe, mas de uma interface. Portanto, a linha 1 deve ser substituída por: public interface ContadorSinc {.

C A linha 9 está errada porque deve-se utilizar Semaphore e não monitor como indicado no trecho de código. Portanto, a linha 9 deve ser substituída por: public Semaphore void decrementar ( ) {.

D A linha 9 está errada porque deve-se utilizar synchronized e não monitor como indicado no trecho de código. Portanto, a linha 9 deve ser substituída por: public synchronized void decrementar ( ) {.

E Não se pode implementar threads que realizem operação de decremento, conforme indicado no código.

Parabéns! A alternativa D está correta.

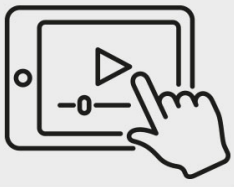
Synchronized é usado para garantir a exclusão mútua e a sincronização adequada entre threads, permitindo que a classe seja utilizada de forma segura em ambientes concorrentes. As demais opções estão incorretas.

Considerações gerais: programação paralela e o uso de threads em Java

O conhecimento sobre programação paralela e o uso de threads em Java é crucial para desenvolver aplicações eficientes e robustas em ambientes multitarefa. Entender princípios, como sincronização, concorrência e gerenciamento de recursos, permite aos desenvolvedores explorarem ao máximo o potencial de processamento dos sistemas modernos. Isso não apenas melhora o desempenho e a capacidade de resposta das aplicações, mas também garante a integridade dos dados e a estabilidade do sistema em cenários complexos. Dominar esses conceitos é fundamental para enfrentar os desafios da computação paralela e oferecer soluções escaláveis e confiáveis.

Este vídeo explora a importância do conhecimento teórico em programação paralela e uso de threads em Java para o desenvolvimento e aplicações multitarefa robustas e eficientes.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A programação paralela é desafiadora. É fácil pensar de maneira sequencial, com todas as instruções ocorrendo de forma encadeada ao longo de uma única linha de execução, mas quando o programa envolve múltiplas linhas que se entrecruzam, a situação suscita problemas inexistentes no caso de uma única linha.

A chamada **condição de corrida** frequentemente se faz presente, exigindo do programador uma atenção especial. Vimos os mecanismos que Java oferece para permitir a sincronização de threads, mas esses

mecanismos precisam ser apropriadamente empregados. Dependendo do tamanho do programa e do número de threads, controlar essa dinâmica mentalmente é desejar o erro.

> Erros em programação paralela são mais difíceis de localizar, pela própria forma como o sistema funciona.

Há algumas práticas simples que podem auxiliar o programador a evitar os erros, como:

Escolha da IDE

Atualmente, as IDE evoluíram bastante. O Apache Netbeans, por exemplo, permite, durante a depuração, mudar a linha de execução que se está examinando. Porém, como os problemas geralmente advêm da interação entre as linhas, a depuração pode ser difícil e demorada mesmo com essa facilidade da IDE.

Uso da UML

Um bom profissional de programação é ligado a metodologias. E uma boa prática, nesse caso, é a elaboração de diagramas dinâmicos do sistema, como o diagrama de sequência e o diagrama de objetos da UML (em inglês, *Unified Modeling Language*; em português, Linguagem Unificada de Modelagem), por exemplo. Esses são mecanismos formais que permitem compreender a interação entre os componentes do sistema.

Atenção aos detalhes

Há sutilezas na programação que muitas vezes passam despercebidas e podem levar o software a se comportar de forma diferente da esperada, já que a linguagem Java oculta os mecanismos de apontamento de memória. Se por um lado isso facilita a programação, por outro exige atenção do programador quando estiver trabalhando com tipos não primitivos. Por exemplo, uma variável do tipo int é passada por cópia, mas uma variável do tipo de uma classe definida pelo usuário é passada

por referência. Isso tem implicações importantes quando estamos construindo um tipo de dado imutável.

Veja a classe mostrada no código a seguir.

Java

Queremos construir uma classe que nos fornecerá um objeto imutável. Por sua simplicidade, e já que a tornamos final, assim como seu único atributo, esse deveria ser o caso. Mas examinemos melhor a linha 3. Essa linha diz que conta é uma referência imutável. Isso quer dizer que, uma vez instanciada (linha 7), ela não poderá se referenciar a outro objeto, mas nada impede que o objeto por ela apontado se modifique, o que pode ocorrer se a referência vazar ou se o próprio objeto realizar interações que o levem a tal.

Atenção!

Lembre-se: quando se trata de tipos não primitivos, a variável é uma referência de um tipo, e não o tipo em si.

Como se não bastassem todas essas questões, temos o escalonador do sistema, que pode fazer o software se comportar diferentemente do esperado, se tivermos em mente uma política distinta da do escalonador. Questões relativas à carga do sistema também podem interferir, e por isso a corretude do software tem de ser garantida. É comum, quando há falhas na garantia da sincronização, que o programa funcione em algumas execuções e falhe em outras, sem que nada tenha sido modificado. Essa sensibilidade às condições de execução é praticamente um atestado de problemas e condições de corrida que não foram adequadamente tratadas.

Por fim, um bom conhecimento do como as threads se comportam é essencial. Isso é importante para evitar que threads morram inadvertidamente, transformando outras em “zumbis”. Também é um ponto crítico quando operações de E/S ocorrem, pois são operações que muitas vezes podem bloquear a thread indefinidamente.

Atividade 3

Você está finalizando uma apresentação para um seminário sobre desenvolvimento de sistemas, com foco em programação multithreading em Java. Para reforçar a importância desse tema, você pediu sugestões aos seus colegas sobre o motivo de o conhecimento em programação paralela e o uso de threads em Java serem tão essenciais. As respostas estão listadas nas opções a seguir. Qual delas você escolheria para destacar a relevância da programação paralela e threads em Java?

A Para minimizar o consumo de energia dos dispositivos.

B Para garantir compatibilidade com diferentes sistemas operacionais.

C Para maximizar o uso de recursos de hardware e melhorar o desempenho das aplicações.

D Para simplificar a integração de novas linguagens de programação.

E Para reduzir o tempo de compilação de código-fonte.

Parabéns! A alternativa C está correta.

O conhecimento sobre programação paralela e uso de threads em Java permite aos desenvolvedores explorar ao máximo o potencial dos

sistemas modernos, como múltiplos núcleos de processador. Isso resulta na maximização do uso de recursos de hardware e na melhoria significativa do desempenho das aplicações em ambientes multitarefa. Compreender os princípios de sincronização, concorrência e gerenciamento de recursos garante a estabilidade do sistema e a integridade dos dados em cenários complexos, em que múltiplas tarefas são executadas simultaneamente.

O que você aprendeu neste conteúdo?

- Processamento paralelo e threads em Java.
- Ciclo de vida de thread em Java.
- Demonstrando a preempção de processos e threads na prática.
- Sincronização entre threads.
- Comunicação entre threads: semáforos e monitores.
- Objetos imutáveis.
- Implementação de threads.

Explore +

Como se trata de um assunto rico, há muitos aspectos que convêm ser explorados sobre o uso de threads. Sugerimos conhecer as nuances da MVJ para melhorar o entendimento sobre como threads funcionam em Java.

Busque também conhecer mais sobre escalonadores de processo e suas políticas. Veja não apenas como a MVJ implementa essas funcionalidades, mas como os sistemas operacionais o fazem. Ao estudar o agendamento de processos de sistemas operacionais e da MVJ, identifique as limitações e os problemas que podem ocorrer.

Outro ponto importante é conhecer o que a API Java oferece de recursos para programação com threads. Para isso, uma consulta à

documentação da API disponibilizada pela própria Oracle é um excelente ponto de partida.

Você pode se interessar, inclusive, em conhecer os principais problemas envolvidos em programação paralela. Aqui mencionamos superficialmente a ocorrência de condições de corrida, mas sugerimos se informar melhor sobre essa questão e outras, como deadlocks e starvation. Indicamos também que você pesquise problemas clássicos como o jantar dos filósofos — às vezes apresentado com nomes diferentes, como “filósofos pensantes”.

Por fim, tome essas sugestões como apenas um começo. Conforme você explorar esses assuntos, outros surgirão. Estude-os também. Estude sempre. Estude muito!

Referências

ORACLE AMERICA INC. **Chapter 17. Threads and Locks**. Consultado na internet em: 5 maio 2021.

ORACLE AMERICA INC. **Class Thread**. Consultado na internet em: 5 maio 2021.

ORACLE AMERICA INC. **Enum Thread.State**. Consultado na internet em: 5 maio 2021.

ORACLE AMERICA INC. **Java Thread Primitive Deprecation**. Consultado na internet em: 5 maio 2021.

ORACLE AMERICA INC. **Semaphore (Java Platform SE 7)**. Consultado na internet em: 5 maio 2021.

ORACLE AMERICA INC. **Thread (Java Platform SE 7)**. Consultado na internet em: 5 maio 2021.

SCHILDT, H. **Java - The Complete Reference**. Nova York: McGraw Hill Education, 2014.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

Download material

O que você achou do conteúdo?

☆ ☆ ☆ ☆ ☆

Relatar problema