TEMA 1: INTRODUÇÃO À POO EM JAVA

Uma classe é um modelo a partir do qual se produzem os objetos. Ela define o comportamento e os atributos que

Programação paralela em Java: Threads

Na era da computação da moderna, podemos executar múltiplas tarefas concomitantemente graças aos já mencionados hyperthreading, CPU de núcleo múltiplo e sistemas operacionais multitarefa preemptiva.

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

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.

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.

Configuração: CPU multinúcleo

Vamos então considerar a segunda configuração. Agora, temos mais de um núcleo. Cada núcleo da CPU, diferentemente do caso da hyperthreading, é um pipeline completo e independente dos demais. Sendo assim, o núcleo 0 tem seus próprios registradores, de forma que a execução de um código pelo núcleo 1 não interferirá no outro.

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.

O que são Threads

As threads são linhas de execução de programa contidas nos processos. Diferentemente deles, elas não possuem uma área de memória exclusiva, mas compartilham o mesmo espaço. Por serem 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 envolvida nos mecanismos de comunicação interprocessos.

THREADS EM JAVA

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.

A nível do sistema operacional (SO), diversos programas são executados preemptivamente e/ou em paralelo, com o SO fazendo o gerenciamento do tempo de execução. Um programa, por sua vez, pode possuir uma ou mais linhas de execução capazes de realizar tarefas distintas simultaneamente (ou quase).

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.

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.

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

As daemon threads 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

Threads de usuário 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.

CICLO DE VIDA DE THREAD EM JAVA

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 6 estados.

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 chamada aos métodos “wait ()” ou “join ()” sem timeout ou “park ()” (estacionado) leva a thread ao estado de “WAITING” (SCHILDT, 2014).

TERMINATED

O último estado ocorre quando a thread 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.

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

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.

Mecanismo de herança

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.

Implementação de “Runnable”

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.

Identificar a sincronização entre threads em Java

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 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. 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?

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.

Lembre-se 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:

A colisão na escrita pode fazer com que uma escrita seja descartada.

Diferenças de microssegundos podem fazer com que as escritas ocorram com uma defasagem infinitesimal. Nesse caso, X+1 seria 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.

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.

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:

Clique nas barras para ver as informações.

Acquire ()

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.

public class Exemplo

{

// (...)

Semaphore sem = new Semaphore ( 50 , true ); //Define até 50 acessos e o uso de FIFO

sem.acquire ( ); //Solicita 1 acesso

... // Região crítica

sem.release ( ); //Libera o acesso obtido

... //Código não crítico

sem.acquire ( 4 ); //Solicita 4 acessos

... // Região crítica

sem.release ( 4 ); //Libera os 4 acessos obtidos

... //Código não crítico

}

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.

Semáforo

Não verifica se a liberação de acesso veio da mesma thread que a solicitou.

Mutex

Faz a verificação para garantir que a liberação veio da thread que a solicitou.

MONITORES

Vamos retornar ao problema hipotético apresentado no início do módulo. Nele, precisamos proceder ao [incremento de uma variável](javascript:void(0)), garantindo que nenhuma outra thread opere sobre ela antes de terminarmos de incrementá-la.

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](javascript:void(0))

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

[Cooperação entre threads](javascript:void(0))

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.

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, o qual uma thread pode travar ou destravar. O uso de “synchronized” pode ser aplicado a um método ou a uma região menor de código.

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.

OBJETOS IMUTÁVEIS

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.

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 deve-se 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.

Aplicar a implementação de threads em Java