DESCRIÇÃO

A programação paralela em linguagem Java, as threads (linhas de programação) e seu ciclo de vida, assim como conceitos e técnicas importantes para a sincronização entre threads e um exemplo de implementação com múltiplas linhas de execução.

PROPÓSITO

O emprego de threads em Java na programação paralela em CPU de núcleo múltiplo é fundamental para profissionais da área, uma vez que a técnica se tornou essencial para a construção de softwares que aproveitem ao máximo os recursos de hardware e resolvam os problemas de forma eficiente.

PREPARAÇÃO

Para melhor absorção do conhecimento, recomendamos o uso de um computador com o JDK (*Java Development Kit*) e um IDE (*Integrated Development Environment*) instalados.

OBJETIVOS

MÓDULO 1

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

MÓDULO 2

Identificar a sincronização entre threads em Java

MÓDULO 3

Aplicar a implementação de threads em Java

INTRODUÇÃO

Inicialmente, a execução de códigos em computadores era feita em lotes e limitada a uma única unidade de processamento. Sendo assim, quando uma tarefa era iniciada, ela ocupava a CPU até o seu término. Apenas nesse momento é que outro código podia ser carregado e executado. O primeiro avanço veio, então, com o surgimento dos sistemas multitarefa preemptivos. Isso permitiu que uma tarefa, ainda inacabada, fosse suspensa temporariamente, dando lugar a outra.

Dessa forma, várias tarefas compartilhavam a execução na CPU, simulando uma execução paralela. A execução não era em paralelo no sentido estrito da palavra: a CPU somente conseguia executar uma tarefa por vez, contudo, como as tarefas eram preemptadas, isto é, tiradas do contexto da execução antes de terminarem, várias tarefas pareciam estar sendo executadas ao mesmo tempo.

O avanço seguinte veio com a implementação pela Intel do hyperthreading em seus processadores. Essa tecnologia envolve a replicação da pipeline de execução da CPU, mantendo os registradores compartilhados entre as pipelines. Com isso, tarefas passaram a ser realmente executadas em paralelo. Entretanto, o compartilhamento dos registradores faz com que a execução de um código possa interferir no outro pipeline.

Finalmente, com o barateamento da tecnologia de fabricação de chips, surgiram as CPU com múltiplos núcleos. Cada núcleo possui a capacidade de execução completa de código, incluindo a replicação de pipelines, no caso das CPU Intel. Com essa tecnologia, a execução paralela de várias tarefas se popularizou, impulsionando o uso de threads. Neste conteúdo

vamos abordar apenas a visão Java de thread, incluindo nomenclaturas, características, funcionamento e tudo que se relacionar ao assunto.

MÓDULO 1

- **Reconhecer o conceito de threads e sua importância para o processamento paralelo**

CONCEITOS

Estamos tão acostumados com as facilidades da tecnologia hoje 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.

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 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.

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

CPU GENÉRICA DE NÚCLEO ÚNICO

CPU MULTINÚCLEO

Ambas as configurações têm um sistema operacional multitarefa preemptivo. Vamos examinar como softwares com uma única linha de execução são processados nessas plataformas em suas duas configurações, o que nos deve dar base para um melhor entendimento do conceito de threads.

EXECUÇÃO DE SOFTWARE POR UM COMPUTADOR TEÓRICO

CONFIGURAÇÃO: CPU GENÉRICA DE NÚCLEO ÚNICO

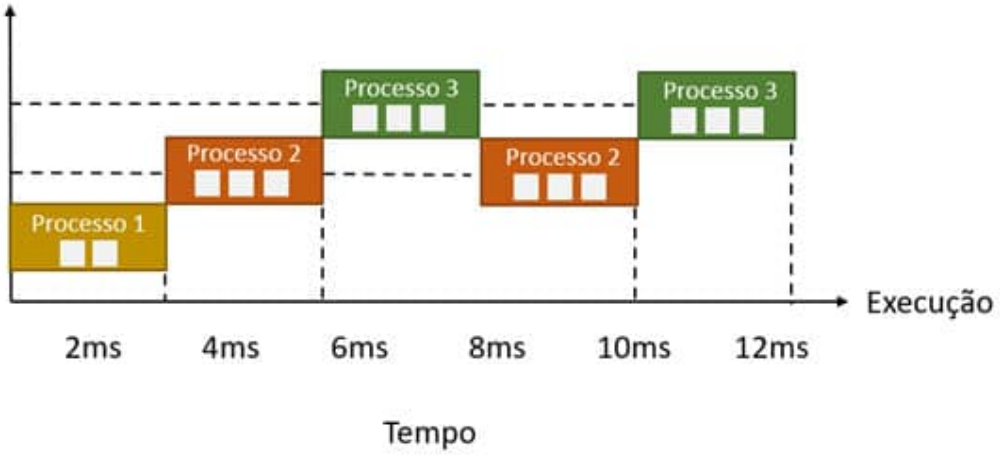
Nossa primeira configuração era muito comum há pouco mais de uma década. Imagine que você está usando o Word para fazer um trabalho e, ao mesmo tempo, está calculando a soma *hash* (soma utilizando algoritmo) de um arquivo.

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.

DICA

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. 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.

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?

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

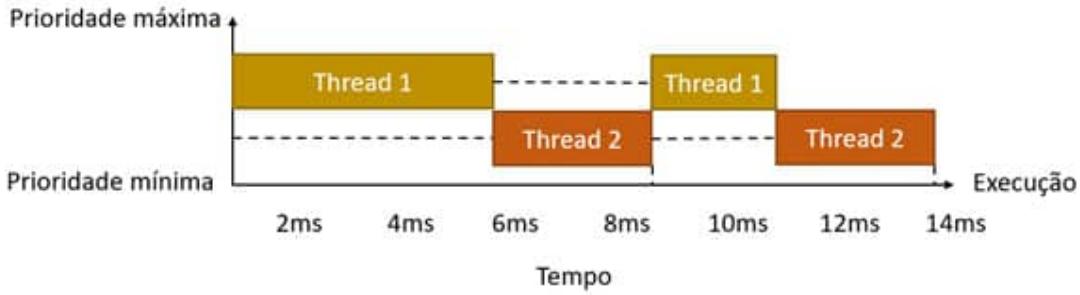
A linguagem Java é uma linguagem de programação multithread, o que significa que Java suporta o conceito de threads. Como vimos, uma thread pode ser preemptada da execução e isso é feito pelo sistema operacional que emite comandos para o hardware. Por isso, nas primeiras versões da MVJ (Máquina Virtual 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.

Escalonador de processos.

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.



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

USER

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.).

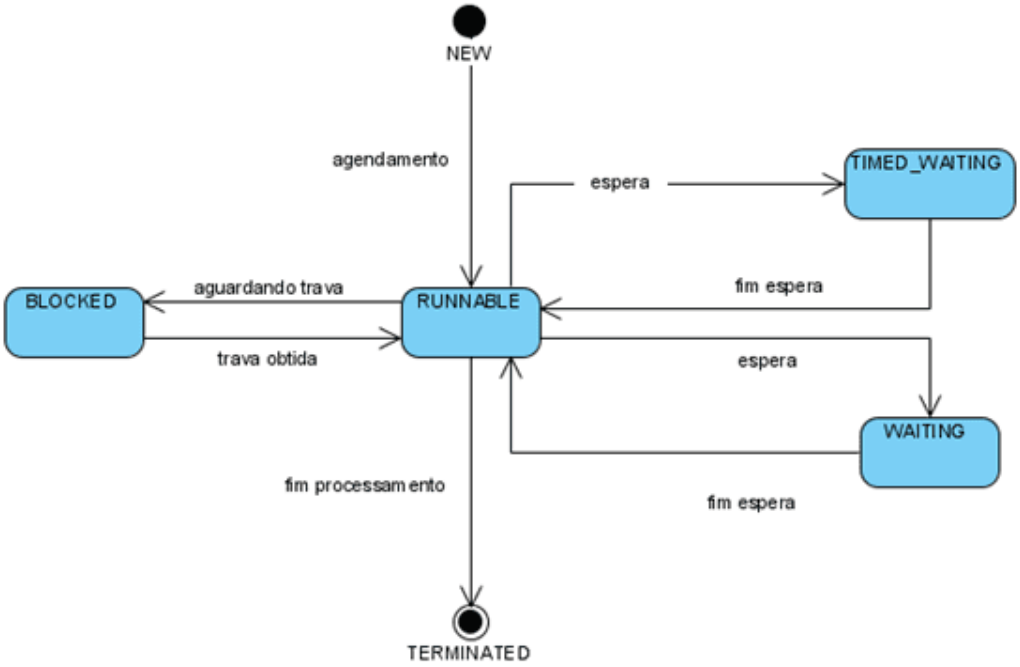
Há duas maneiras de se criar uma thread em Java:

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, conforme vemos na máquina de estados retratada na figura a seguir.



Escalonador de processos.

Como podemos observar na imagem, os 6 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 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).

💬 COMENTÁRIO

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

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

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.

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

A seguir, veremos 3 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”.

```
class ThreadSubclasse extends Thread {
      long numero;
      ThreadSubclasse  (long numero) {
         this.numero = numero;
      }

      public void run() {
// Implementa o comportamento apropriado
      }
}
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

Código 2: definindo threads por implementação de Runnable.

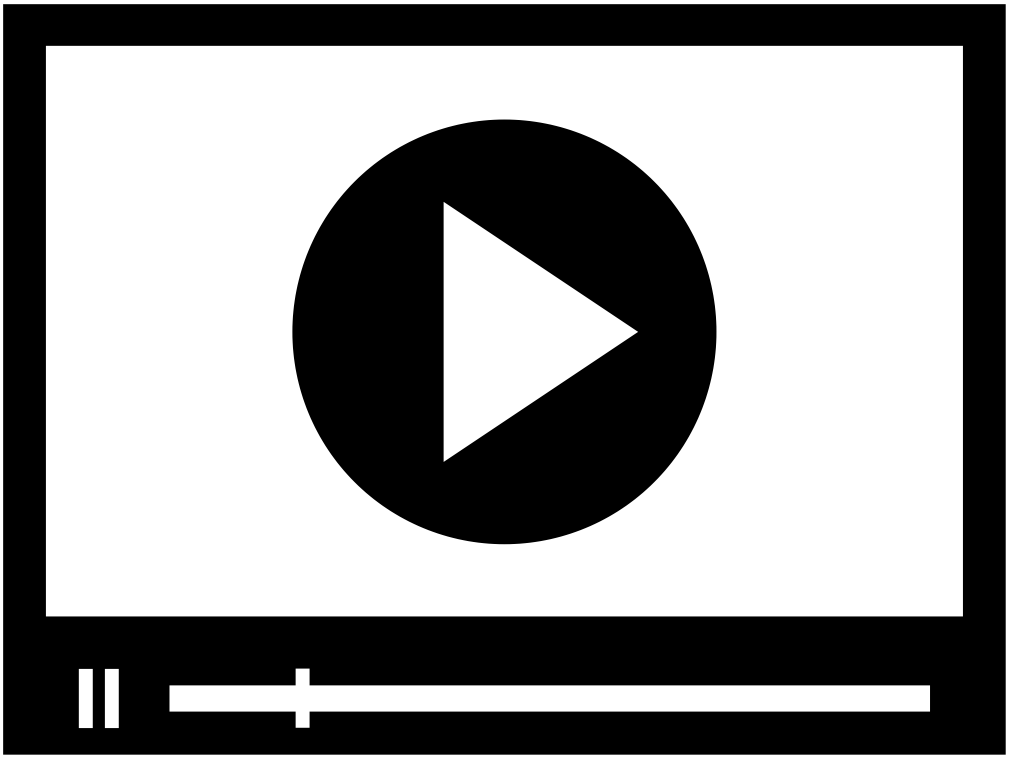
```
class ThreadInterface implements Runnable {
      long numero;
      ThreadInterface  (long numero) {
         this.numero = numero;
      }

      public void run() {
// Implementa o comportamento apropriado
      }
}
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

Código 3: criando threads.

```
...
// Extensão de Thread
ThreadSubclasse novaT = new ThreadSubclasse (200);
novaT.start ();
// Implementação de Runnable
ThreadInterface novaT = new ThreadInterface (200);
new Thread ( novaT ).start ();
...
```

Atenção! Para visualização completa do código utilize a rolagem horizontal



PROCESSAMENTO PARALELO COM THREADS EM JAVA

O vídeo a seguir aborda o conceito de threads e seu suporte pela linguagem Java, além de descrever os estados que uma thread pode ter em Java, assim como a forma de criá-la.

VERIFICANDO O APRENDIZADO

MÓDULO 2

- **Identificar a sincronização entre threads em Java**

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

MÉTODO 2

MÉTODO 3

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 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. 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.

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:

1

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.

2

3

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 ()

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.

```
1  public class Exemplo
2  {
3     // (...)
4     Semaphore sem = new Semaphore ( 50 , true ); //Define até 50 acessos e o usc
5     sem.acquire ( ); //Solicita 1 acesso
6     ... // Região crítica
7     sem.release ( ); //Libera o acesso obtido
8     ... //Código não crítico
9     sem.acquire ( 4 ); //Solicita 4 acessos
10    ... // Região crítica
11    sem.release ( 4 ); //Libera os 4 acessos obtidos
12    ... //Código não crítico
13 }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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

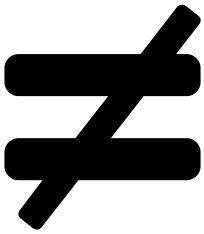
★ EXEMPLO

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

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.

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:

1

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

2

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

3

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 irá facilitar 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.

```
1  public class PingPong {
2     //Atributos
3     private Semaphore s1 , s2;
4     private Ping ping;
5     private Pong pong;
6     private Controle contador;
7     private int tamanho_partida;
8
9     //Métodos
10    public PingPong ( int tamanho_partida ) throws InterruptedException {
11       s1 = new Semaphore(0);
12       s2 = new Semaphore(0);
13       contador = new Controle ( tamanho_partida );
14       ping = new Ping ( s1 , s2 , contador );
15       pong = new Pong ( s1 , s2 , contador );
16       //juiz = new Juiz ( tamanho_partida / 2 );
17       new Thread ( ping ).start ();
18       new Thread ( pong ).start ();
19       s1.release();
20    }
21 }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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

```
public class Ping implements Runnable {
   //Atributos
   private Semaphore s1 , s2;
   private Controle contador;

   //Métodos
   public Ping ( Semaphore s1 ,  Semaphore s2 , Controle contador )
   {
      this.s1 = s1;
      this.s2 = s2;
      this.contador = contador;
   }

   @Override
   public void run() {
      try {
               System.out.println("Thread A (PING) iniciada");
               while ( contador.getControle() > 0) {
                  s1.acquire();
                  System.out.println( "PING => 0" );
                  s2.release();
                  contador.decrementa();
               }
            } catch ( InterruptedException e ) {
                e.printStackTrace();
            }
      System.out.println("Thread A (PING) terminada");
   }
}
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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

```

public class Pong implements Runnable {
   //Atributos
   private Semaphore s1 , s2;
   private Controle contador;

   //Métodos
   public Pong ( Semaphore s1 ,  Semaphore s2 , Controle contador)
   {
      this.s1 = s1;
      this.s2 = s2;
      this.contador = contador;

```

```
13      }
14
15      @Override
16      public void run() {
17          try {
18                  System.out.println("Thread B (PONG) iniciada");
19                  while ( contador.getControle() > 0) {
20                      s2.acquire();
21                      System.out.println( "0 <= PONG" );
22                      s1.release();
23                      contador.decrementa();
24                  }
25              } catch ( InterruptedException e ) {
26                  e.printStackTrace();
27              }
28          System.out.println("Thread B (PONG) terminada");
29      }
    }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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.

```
1   public class Principal {
2       //Atributos
3       private static PingPong partida;
4
5       //Métodos
6       public static void main (String args[]) throws InterruptedException {
7           partida = new PingPong ( 8 );
8       }
9   }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

RESUMINDO

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, então, 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:

```
1  Thread A (PING) iniciada
2  Thread B (PONG) iniciada
3  PING => 0
4  0 <= PONG
5  PING => 0
6  0 <= PONG
7  PING => 0
8  0 <= PONG
9  PING => 0
10 0 <= PONG
11 Thread B (PONG) terminada
12 PING => 0
13 Thread A (PING) terminada
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

```
1  Thread A (PING) iniciada
2  PING => 0
3  Thread B (PONG) iniciada
4  0 <= PONG
5  PING => 0
6  0 <= PONG
7  PING => 0
8  0 <= PONG
9  PING => 0
10 0 <= PONG
11 Thread B (PONG) terminada
12 PING => 0
13 Thread A (PING) terminada
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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.

INCREMENTO DE UMA VARIÁVEL

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

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.

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, 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. Ambos os casos são mostrados no código a seguir.

```
1  ...
2  private Exemplo ex = new Exemplo (); //”ex” é uma referência para objetos do tipo “
3  ...
4  //Método sincronizado
5  public synchronized void decrementa ( ) {
6     conta--;
7  }
8  ...
9  public void impressao () {
10 //Região de código sincronizada
11    synchronized (ex) {
12       ex.imprime (); //invoca o método “imprime ()” do objeto “ex” de maneira sincroniza
13    }
14 }
15 ...
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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). Ora, 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.

```
1  //Classe
2  public class Controle {
3     //Atributo
4     private int contador = 0;
5
6     //Métodos
7     public Controle ( int contador ) {
8        this.contador = contador;
9     }
10
11    public synchronized void decrementa () {
12       this.contador--;
13    }
```

```
14
15      public synchronized int getControle () {
16          return this.contador;
17      }
18  }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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.

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 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.

📢 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.

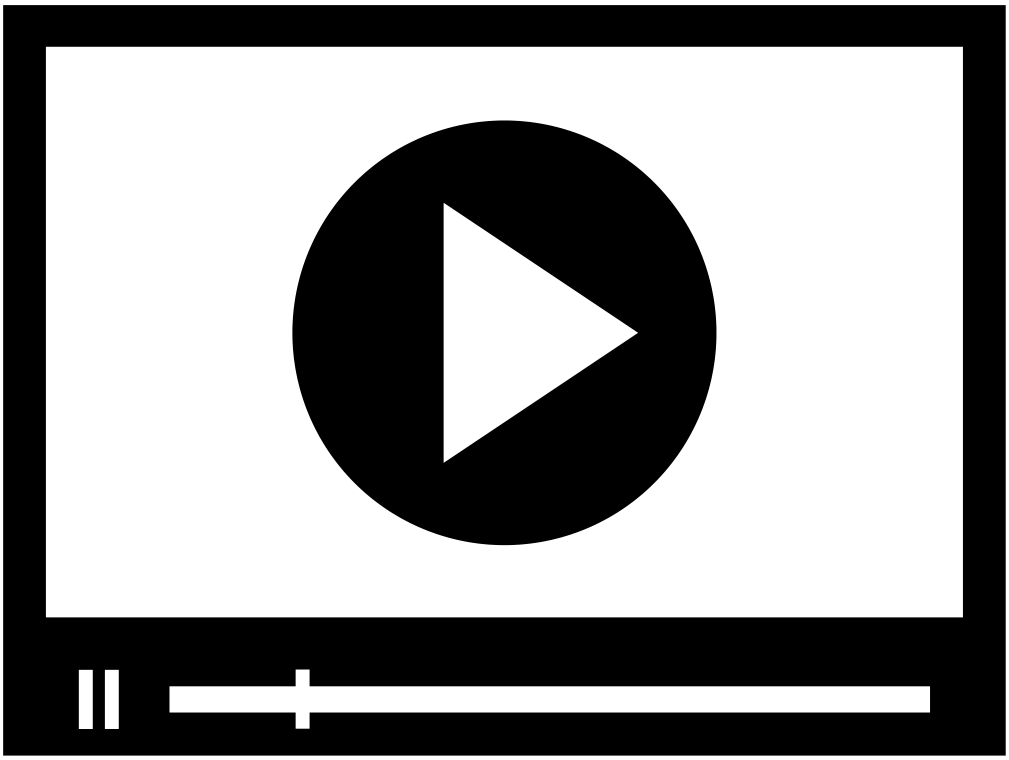
Podemos ver um exemplo de classe que define objetos imutáveis no código a seguir.

```
1  //Classe
2  public final class Aluno {
3     //Atributos
4     private final String nome;
5     private final long CPF;
6     private final int matricula;
7
8     //Métodos
9     protected Aluno ( String nome , long CPF , int matricula ) {
10       this.nome = nome;
11       this.CPF = CPF;
12       this.matricula = matricula;
13    }
14
15    protected String getNome ( ) {
16       return this.nome;
17    }
18    protected long getCPF ( ) {
19       return this.CPF;
20    }
21    protected int getMatricula ( ) {
22       return this.matricula;
23    }
24 }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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”.



MECANISMOS DE SINCRONIZAÇÃO ENTRE THREADS EM JAVA.

O vídeo a seguir aborda os mecanismos de semáforo e monitores, utilizados para sincronizar threads, além das implicações no uso de objetos imutáveis compartilhados entre threads.

VERIFICANDO O APRENDIZADO

MÓDULO 3

- **Aplicar a implementação de threads em Java**

CONCEITOS

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 auferidos 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, respectivamente, as prioridades máxima, mínima e default da thread.

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** no módulo 1 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.

ESTADOS DA THREAD

Os estados possíveis da thread são: NEW, RUNNABLE, BLOCKED, TIMED_WAITING, WAITING ou TERMINATED.

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 class “Thread” possui o método “join ()”. Esse método 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)”. A invocação de “sleep (long millis)” 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.

💬 COMENTÁRIO

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 melhorar suas habilidades em programação com threads, você irá descobrir outros recursos que a API Java oferece. Por enquanto, para consolidar o aprendizado, vamos agora apresentar um exemplo que emprega diversos conhecimentos vistos anteriormente.

IMPLEMENTAÇÃO DE THREADS EM JAVA NA PRÁTICA

Como exemplo, iremos simular uma empresa que trabalha com encomendas. Observe as regras de negócio a seguir.

As encomendas são empacotadas por equipes compostas por no mínimo duas pessoas.

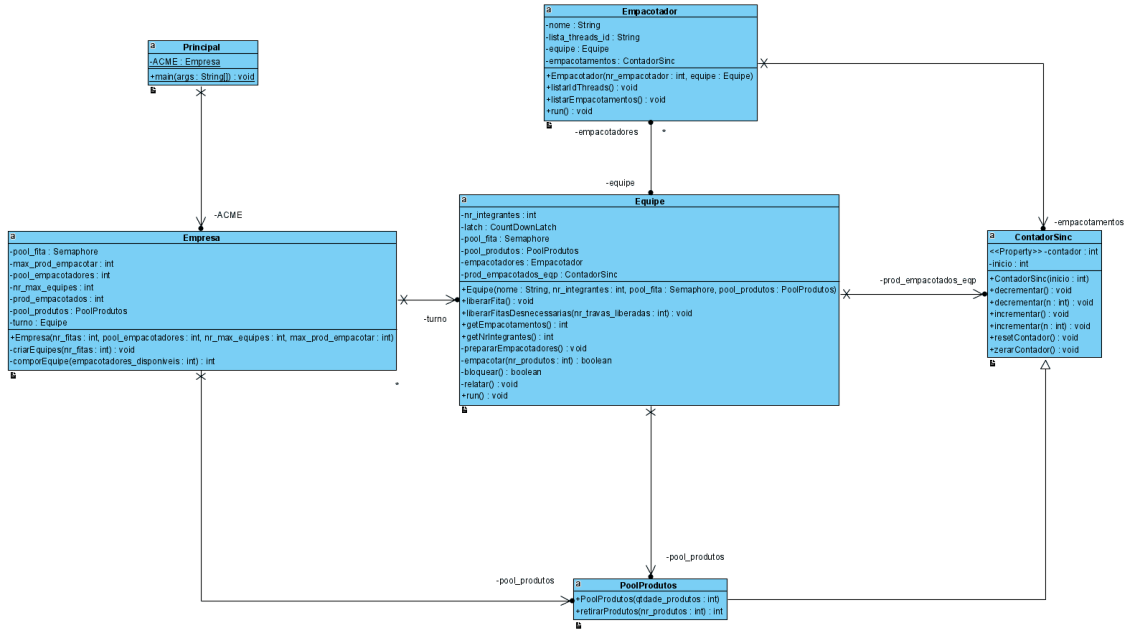
O empacotamento depende apenas do uso de fita adesiva, que será o recurso compartilhado por todas as equipes.

Cada membro da equipe usa somente uma fita por vez.

Ao terminar o empacotamento, ele obrigatoriamente deve devolver a fita, permitindo que outra pessoa a use.

Uma equipe só é contemplada com as fitas se puder pegar fitas para todos os integrantes ou se puder esgotar todas as encomendas que aguardam ser empacotadas. Caso contrário, a equipe aguardará até que haja fitas suficientes disponíveis.

Você deve ter em mente que estamos tratando de um exemplo didático. Outras implementações são possíveis – talvez até mais eficientes –, e tentar fazê-las é um ótimo meio de se familiarizar com as threads e consolidar os conhecimentos adquiridos. Sugerimos que você comece fazendo os diagramas de sequência e objetos, pois isso deve facilitar o seu entendimento do próprio código e a busca por outras soluções. Para ajudar, apresentamos o diagrama de classes a seguir.



📷 Escalonador de processos.

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.

```
1  public class Principal {
2     //Atributo
3     private static Empresa ACME;
4
5     //Métodos
6     public static void main ( String args [ ] ) throws InterruptedException {
7        // Empresa (número de fitas, empregados disponíveis, número máximo de equipes
8        ACME = new Empresa ( 20 , 25 , 4 , 200 );
9     }
10 }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

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 seguinte, “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”.

```
2
3  package com.mycompany.empacotadoraACME;
4
5  import java.util.ArrayList;
6  import java.util.Random;
7  import java.util.concurrent.Semaphore;
8
9  /**
10 *
11 * @author Prof Marlos M Corrêa
12 */
13 public class Empresa {
14    //Atributos
15    private final Semaphore pool_fita; //Controla o acesso ao recurso crítico (fitas).
16    private final PoolProdutos pool_produtos; //Produtos a serem empacotados.
17    private final ArrayList< Equipe > turno; //Conjunto de equipes de empacotadores.
18    private final int max_prod_empacotar; //Número máximo de produtos que serão er
      private final int pool_empacotadores; //Número de empacotadores disponíveis pa
```

```
19  private final int nr_max_equipes; //Número máximo de equipes que podem ser for
20  private int prod_empacotados; //Número de produtos empacotados.
21
22  //Métodos
23  public Empresa ( int nr_fitas , int pool_empacotadores , int nr_max_equipe
24    if ( ( nr_fitas < 1 ) || ( pool_empacotadores < 2 ) || ( nr_max_equipes
25        throw new IllegalArgumentException ( "Argumentos ilegais utilizados no co
26    else {
27      this.pool_fita = new Semaphore ( nr_fitas );
28      this.pool_empacotadores = pool_empacotadores;
29      this.nr_max_equipes = nr_max_equipes;
30      this.max_prod_empacotar = max_prod_empacotar;
31      this.pool_produtos = new PoolProdutos ( max_prod_empacotar );
32      this.turno = new ArrayList< Equipe > ();
33      this.prod_empacotados = 0;
34      criarEquipes ( nr_fitas ); //Monta as equipes alocando os empacotadores
35      turno.forEach( ( eqp ) -> eqp.start ( ) ); //Inicia todas as threads.
36      for ( Equipe eqp : turno ) //Faz o join com todas as threads de equipe.
37        try {
38            eqp.join ( ); //A thread principal deve aguardar o fim de todas as threa
39        } catch ( InterruptedException e ) {
40            e.printStackTrace();
41        }
42      for ( Equipe eqp : turno )
43        prod_empacotados = prod_empacotados + eqp.getEmpacotamentos(); //(
44      System.out.println ("TOTAL DE EMPACOTAMENTOS: " + prod_empacotac
45    }
46  }
47  private void criarEquipes ( int nr_fitas ) {
48    Equipe eqp;
49    int nr_emp_eqp;
50    int empacotadores_disponiveis = pool_empacotadores;
51    int i = 1;
52    do { //Cria as equipes com um número aleatório de integrantes e as adiciona em '
53        nr_emp_eqp = comporEquipe ( empacotadores_disponiveis );
54        if ( nr_emp_eqp > nr_fitas )
55            nr_emp_eqp = nr_fitas; //Do contrário, teríamos mais solicitações de
56        eqp = new Equipe ( "Eqp[" + String.valueOf ( i ) + "]", nr_emp_eqp ,
57        turno.add ( eqp );
58        empacotadores_disponiveis = empacotadores_disponiveis - nr_emp_
59        i++;
60    } while ( ( i < nr_max_equipes ) && ( empacotadores_disponiveis >= 2
61    /**A última equipe recebe todos os empregados restantes. Se o número de empre
62  * são alocados nr_fitas empregados (caso contrário, a equipe iria solicitar um númerc
63  * de recursos do semáforo e ficaria bloqueada).
64  */
```

```
65          if ( empacotadores_disponiveis > 0 ) {
66              if ( nr_fitas > empacotadores_disponiveis )
67                  eqp = new Equipe ( "Eqp[" + String.valueOf ( i ) + "]" , empacot
68              else
69                  eqp = new Equipe ( "Eqp[" + String.valueOf ( i ) + "]", nr_fitas
70              turno.add ( eqp );
71          }
72      }
73      private final int comporEquipe ( int empacotadores_disponiveis ) {
74          Random rnd = new Random ( );
75          if ( empacotadores_disponiveis > 2 )
76              return rnd.nextInt ( pool_empacotadores / nr_max_equipes ) + 2;
77          else
78              return empacotadores_disponiveis;
79      }
    }
```

Observe que a linha 34 percorre o “ArrayList”, que armazena as equipes iniciando as threads. A linha seguinte também percorre a estrutura, mas agora invocando o método “join”. Isso faz com que a thread inicial (a que foi criada no início da execução do programa e da qual o objeto “ACME” faz parte) seja instruída a aguardar até que as threads das equipes terminem. Logo, a thread inicial bloqueia e a linha 41 só será executada quando todas as threads correspondentes às equipes finalizarem. Você verá, mais à frente, que também impedimos que as threads das equipes finalizem antes que todo o empacotamento termine. A linha 41 é outro laço que percorre as equipes, contabilizando o número total de empacotamentos.

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

```
2
3   package com.mycompany.empacotadoraACME;
4
5   //Importações
6   import java.util.ArrayList;
7   import java.util.concurrent.CountDownLatch;
8   import java.util.concurrent.Semaphore;
9
10  /**
11   *
12   * @author Prof Marlos M Corrêa
13   */
14  public class Equipe extends Thread {
15      //Atributos
16      private final int nr_integrantes;
```

```
17    private CountDownLatch latch;
18    private final Semaphore pool_fita;
19    private final PoolProdutos pool_produtos;
20    private final ArrayList< Empacotador > empacotadores;
21    private final ContadorSinc prod_empacotados_eqp;
22
23    //Métodos
24    public Equipe ( String nome , int nr_integrantes , Semaphore pool_fita , F
25    {
26       this.setName(nome);
27       this.nr_integrantes = nr_integrantes;
28       this.pool_fita = pool_fita;
29       this.pool_produtos = pool_produtos;
30       this.empacotadores = new ArrayList< Empacotador > ();
31       prod_empacotados_eqp = new ContadorSinc ( 0 );
32       prepararEmpacotadores ( );
33    }
34    /**
35    * Realiza atomicamente as operações de decremento do latch, incremento do númerc
36    * que a equipe empacotou e libera (uma) trava sobre o semáforo "pool_fita".
37    */
38    public synchronized void liberarFita ( ) {
39       latch.countDown();
40       prod_empacotados_eqp.incrementar();
41       pool_fita.release();
42    }
43    /**
44    * Realiza atomicamente as operações de decremento do latch e libera "nr_travas" trav
45    */
46    public synchronized void liberarFitasDesnecessarias ( int nr_travas_liber
47       pool_fita.release( nr_travas_liberadas );
48       while ( nr_travas_liberadas > 0 ) {
49            latch.countDown();
50            nr_travas_liberadas--;
51       }
52    }
53    public synchronized int getEmpacotamentos ( ) {
54       return prod_empacotados_eqp.getContador();
55    }
56    public synchronized int getNrIntegrantes ( ) {
57       return nr_integrantes;
58    }
59    private void prepararEmpacotadores ( ) {
60       for ( int i = 1 ; i <= nr_integrantes ; i++ ) {
61            Empacotador emp = new Empacotador ( i , this );
62            empacotadores.add(emp);
```

```
63      }
64    }
65    /**
66    * Para cada empregador, se houver pacote disponível para empacotar, decrementa p
67    * thread (Empacotador) para realizar o trabalho de empacotamento. Do contrário, nãc
68    * @return Se for possível retirar um pacote para cada (Empacotador), retorna true, ca
69    */
70    private boolean empacotar ( int nr_produtos ) {
71      Thread thd;
72      int thd_criadas = pool_produtos.retirarProdutos ( nr_produtos ); //O n
73      boolean controle;
74      if ( thd_criadas == 0 ) { //Não há produtos disponíveis para empacotamento.
75          liberarFitasDesnecessarias ( nr_produtos ); //Devolve todas as fitas
76          return false;
77      } else {
78          for ( int i = 1 ; i <= thd_criadas ; i++ ) {
79              thd = new Thread ( empacotadores.get ( i - 1 ) );
80              thd.setPriority ( Thread.currentThread().getPriority() + 2 );
81              thd.start ( ); //Inicia a thread (Empacotador).
82          }
83          liberarFitasDesnecessarias ( nr_produtos - thd_criadas ); //Devol
84          return bloquear ( ); //Bloqueia a thread (Equipe) até que os empacotadore
85      }
86    }
87    /**
88    * Realiza o bloqueio da thread até que todas as threads (Empacotador) tenham finaliz
89    * @return boolean
90    */
91    private boolean bloquear ( ) {
92    try {
93        latch.await(); //Bloqueia a thread (Equipe) até que os empacotadores terminer
94        return true;
95    } catch ( InterruptedException e ) {
96       e.printStackTrace();
97       return false;
98      }
99    }
100   private synchronized void relatar ( ) {
101     System.out.println ( "\n/-------------------------------------\\" );
102     System.out.println ( getName () + " (thread: " + Thread.currentThread().get
103     System.out.println ( " |- Nr Integrantes: " + this.nr_integrantes );
104     System.out.println ( " |- Empacotamentos da equipe: " + this.prod_empacota
105     System.out.println ( " |- Empacotamentos por integrante:" );
106     empacotadores.forEach( (emp ) -> emp.listarEmpacotamentos ( ) );
107     System.out.println ( " |- Threads por objeto Empacotador:" );
108     empacotadores.forEach( (emp ) -> emp.listarIdThreads ( ) );
```

```
109         System.out.println ( "\\----------------------------------/\n" );
110       }
111       @Override
112       public void run() {
113        try {
114                boolean controle;
115                System.out.println ( getName () +   " PRONTA" );
116                do {
117                  pool_fita.acquire ( nr_integrantes );
118                  this.latch = new CountDownLatch ( nr_integrantes );
119                  controle = empacotar ( nr_integrantes ); //Cada integrante da equ
120              } while ( controle );
121         } catch ( InterruptedException e ) {
122             e.printStackTrace();
123         }
124       relatar ( );
          }
      }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

O primeiro ponto a se notar é que estamos estendendo a classe Thread e fazendo a implementação do método “run” na linha 110, mas vamos focar os aspectos relevantes para a programação paralela. O objeto “latch” criado pertence à classe “CountDownLatch”, da API Java. Vamos usá-lo para controlar o bloqueio da thread corrente de equipe.

Observe que na linha 115 utilizamos o semáforo “pool_fita” para solicitar “nr_integrantes” (permissões de acesso). Esse semáforo foi recebido da classe “Equipe”, na qual foi instanciado na linha 26. Como cada thread de equipe recebe a referência para o mesmo semáforo, todas as threads compartilham essa estrutura. Assim, “pool_fita” controla as permissões de todas as threads de “Equipe”. Quando as permissões se esgotam, as threads bloqueiam, aguardando até que alguém libere.

O objeto “latch”, na linha 116, é criado com o contador interno igual ao número de integrantes da equipe (e, portanto, de threads criadas pelo objeto de “Equipe”). Quando cada thread correspondente a “Empacotador” finaliza, “latch” é decrementado. Quando o contador zera, a thread de “Equipe” desbloqueia. O bloqueio ocorre na linha 91, e o decréscimo do contador ocorre em dois pontos: na linha 37, quando o empacotador termina o trabalho, e na linha 46. Aliás, os métodos “liberarFita” e “liberarFitasDesnecessárias” também liberam as travas do semáforo, que ficam disponíveis para outras threads de “Equipe”.

A linha 77 cria threads de “Empacotador” em número igual à quantidade de integrantes da equipe, a linha 78 altera a prioridade dessas threads e a linha 79 as inicia. A classe “Empacotador” é mostrada no próximo código.

```
1
2  package com.mycompany.empacotadoraACME;
3
4  //Importações
5  import java.util.concurrent.CountDownLatch;
6  import java.util.concurrent.Semaphore;
7
8  /**
9   *
10  * @author Prof Marlos M Corrêa
11  */
12 public class Empacotador implements Runnable {
13    //Atributos
14    private final Equipe equipe;
15    private final ContadorSinc empacotamentos;
16    private final String nome;
17    private String lista_threads_id;
18
19    //Métodos
20    public Empacotador ( int nr_empacotador , Equipe equipe )
21    {
22       this.equipe = equipe;
23       this.lista_threads_id = new String ();
24       this.nome = "Emp[" + nr_empacotador + "]@" + equipe.getName();
25       Thread.currentThread ( ).setName ( nome );
26       empacotamentos = new ContadorSinc ( 0 );
27    }
28    public void listarIdThreads ( ) {
29       System.out.println ( " |----- Lista de threads executadas por " + nome + " : " +
30    }
31    public void listarEmpacotamentos ( ) {
32       System.out.println ( " |----- Empacotamentos feitos por " + nome + " : " + emp
33    }
34    @Override
35    public void run() {
36       try {
37             synchronized ( lista_threads_id ) {
38                lista_threads_id = lista_threads_id + "[" + Thread.currentThrea
39             }
40             System.out.println ( nome + " empacotando (" + System.currentTimeM
41             Thread.sleep ( ( int ) ( Math.random ( ) * 899 + 100 ) ); //Coloca a tl
42             System.out.println ( nome + " concluiu (" + System.currentTimeMillis()
```

```
43                empacotamentos.incrementar(); //Incrementa o contador de empacotamento
44                equipe.liberarFita();
45            } catch ( InterruptedException e ) {
46                e.printStackTrace();
47            }
48    }
}
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

A classe “Empacotador” é mais simples. Nesse caso, estamos implementando “Runnable”. O método “run” simula o empacotamento. Para isso, na linha 40 colocamos a thread para dormir. O tempo em que uma thread é colocada para dormir é aleatório. Após isso, a trava sobre o semáforo é liberada e o contador de “latch” é decrementado por meio da invocação da linha 43. Repare que na linha 24 definimos o nome da thread e que na linha 36 usamos “synchronized” aplicado à “String lista_threads_id”. Esse último ponto merece atenção. Na verdade, empregamos “synchronized” apenas para ilustrar uma forma de empregá-lo, a fim de discutir seu uso.

De fato, ele não se faz necessário nesse ponto. Você consegue explicar o porquê?

Usamos o “synchronized” para tornar uma operação atômica, evitando condições de corrida, mas no caso em questão somente a thread corrente altera essa variável. Há diversas threads de objetos “Empacotador”, mas cada thread corresponde a uma única instância de “Empacotador”. Para deixar mais claro, imagine um objeto “Empacotador”. Vamos chamá-lo de Emp1. A variável “lista_thread_id” é uma variável de instância, o que significa que cada objeto tem sua própria cópia. Quando uma thread de Emp1 altera o valor dessa variável, ela o faz somente para a variável da instância Emp1. Sabemos que essa alteração significa, de fato, a criação de um novo objeto “String” com o novo estado (“String” é imutável), logo não há condições de corrida.

Situação diversa da exposta ocorre na classe “ContadorSinc”, no código a seguir. Essa classe foi construída pensando no uso compartilhado por diversas threads, e por isso o uso de “synchronized” se faz necessário. Tomemos como exemplo o método “decrementar”, na linha 17. A operação que esse método realiza é: “contador = contador - 1”. Como já explicamos anteriormente, a ocorrência de mais de uma chamada concorrente a esse método pode levar a uma condição de corrida, e, assim, usamos “synchronized” para impedir isso, garantindo que somente uma execução do método ocorra ao mesmo tempo.

```
1    package com.mycompany.empacotadoraACME;
2
```

```
3  /**
4   *
5   * @author Prof Marlos M Corrêa
6   */
7  public class ContadorSinc {
8     //Atributo
9     private int contador;
10    private final int inicio;
11
12    //Métodos
13    public ContadorSinc ( int inicio ) {
14       this.inicio = inicio;
15       this.contador = inicio;
16    }
17    public synchronized void decrementar ( ) {
18       this.contador--;
19    }
20    public synchronized void decrementar ( int n ) {
21       this.contador = this.contador - n;
22    }
23    public synchronized void incrementar ( ) {
24       this.contador++;
25    }
26    public synchronized void incrementar ( int n ) {
27       this.contador = this.contador + n;
28    }
29    public synchronized void resetContador ( ) {
30       this.contador = this.inicio;
31    }
32    public synchronized void zerarContador ( ) {
33       this.contador = 0;
34    }
35    public synchronized int getContador ( ) {
36       return this.contador;
37    }
38 }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

Nossa última classe é a “PoolProdutos”, mostrada no 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. Então estendemos “ContadorSicn”, especializando-a. Veja que mantemos o uso de “synchronized”, pelas mesmas razões de antes.

```java
package com.mycompany.empacotadoraACME;

/**
 *
 * @author Prof Marlos M Corrêa
 */
public final class PoolProdutos extends ContadorSinc {

   //Métodos
   public PoolProdutos ( int qtdade_produtos ) {
      super ( qtdade_produtos );
      if ( qtdade_produtos < 1 )
           throw new IllegalArgumentException ( "Argumentos ilegais utilizados no co
   }
   public synchronized int retirarProdutos ( int nr_produtos ) {
      int aux = getContador ();
      if ( ( aux - nr_produtos ) >= 0 ) { //Há produtos disponíveis suficientes para
           decrementar ( nr_produtos );
           return nr_produtos;
      } else { //Os produtos são insuficientes ou inexistentes.
           zerarContador ();
           return aux;
      }
   }
}
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

O exemplo que apresentamos fornece uma boa ideia de como usar threads. Estude-o e faça suas próprias alterações, analisando os impactos decorrentes.

📢 ATENÇÃO

Um ponto a se destacar é que quando se trabalha com programação paralela, erros podem fazer com que a mesma execução tenha resultados diferentes. Pode ser que em uma execução o programa funcione perfeitamente e, na execução seguinte, sem nada ser alterado, o programa falhe. Isso ocorre porque a execução é afetada por vários motivos, como a carga do sistema. Programas que fazem uso de paralelismo devem ser, sempre, imunes a isso. Garantir a correção nesse caso demanda um bom projeto e testes muito bem elaborados.

CONSIDERAÇÕES GERAIS

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

USO DA UML

ATENÇÃO AOS DETALHES

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

Além dessas questões, 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.

```
1  public final class Imutavel {
2      //Atributo
3      private final Contador conta;
4
5      //Métodos
6      protected Imutavel ( ) {
7          this.conta = new Contador (0);
8      }
9  }
```

Atenção! Para visualização completa do código utilize a rolagem horizontal

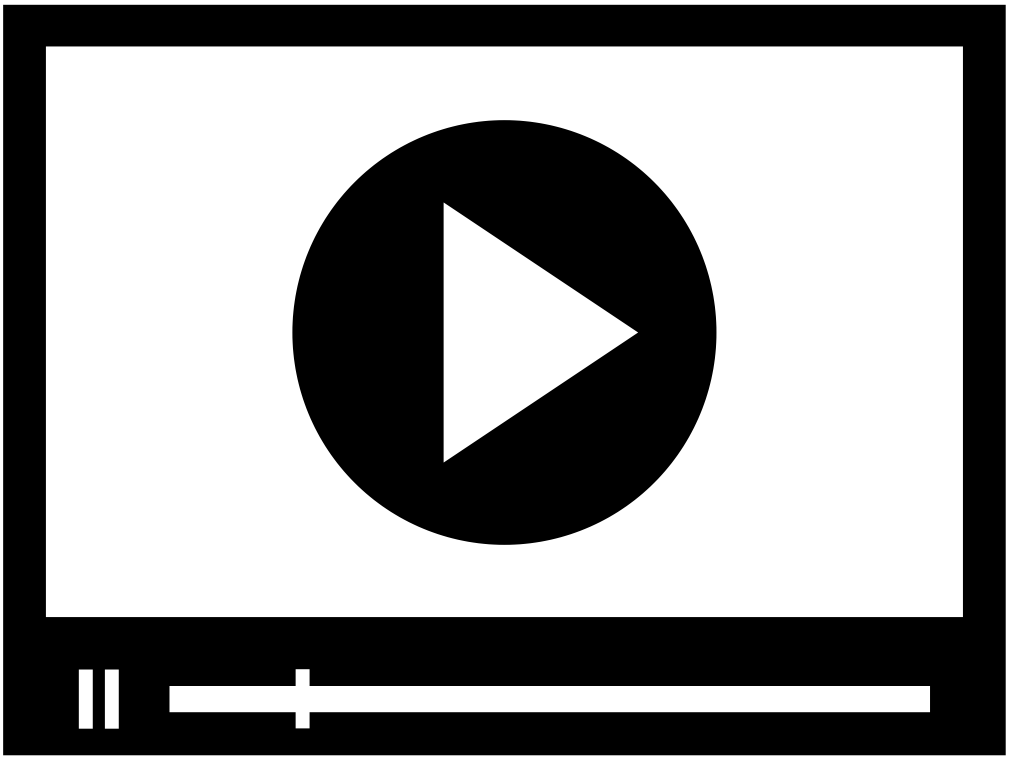
Queremos construir uma classe que nos fornecerá um objeto imutável. Por sua simplicidade, e já que a tornamos final, assim como seu único atributo, esse deveria ser o caso. Mas examinemos melhor a linha 3. Essa linha diz que “conta” é uma referência imutável. Isso quer dizer que, uma vez instanciada (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.



EXEMPLO DE USO DE THREADS EM JAVA.

O vídeo a seguir aborda o uso de threads em Java.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Como pudemos aprender neste conteúdo, as threads são um importante recurso de programação, especialmente nos dias de hoje. Compreender o seu funcionamento permite o desenvolvimento de softwares capazes de extrair o melhor que a plataforma de execução tem a oferecer. Ao mesmo tempo, o uso de múltiplas linhas de execução permite a resolução de problemas de maneira mais rápida e eficiente.

Nosso estudo iniciou-se com a apresentação do conceito de thread e uma discussão sobre sua importância para a programação paralela. Isso nos permitiu compreender o seu papel e a forma como Java lida com esse conceito. Vimos que apesar de ser um valioso recurso, o uso de threads demanda cuidados com questões que não estão presentes na programação linear. Isso nos levou ao estudo dos mecanismos de sincronização, essenciais para programação paralela.

Encerramos explorando um exemplo de uso de threads. Nele, pudemos verificar o emprego dos conceitos estudados e constatar como o ciclo de vida de uma thread se processa. Além disso, o exemplo permitiu consolidar conceitos e destacar os principais cuidados ao se usar uma abordagem paralela de programação.

PODCAST

PODCAST

Ouça o podcast com um resumo sobre programação paralela em Java e o uso de threads para esse fim.

AVALIAÇÃO DO TEMA:

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

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!

CONTEUDISTA

Marlos de Mendonça Corrêa

🔗 CURRÍCULO LATTES