MULTITHREADING EM JAVA PARTE III

Prof. Dr. Rodrigo Palácios

MULTITHREADING – PARTE III

- Comunicação entre threads
  - Uma thread chamada T está sendo executada dentro de um método sincronizado e precisa de acesso a um recurso chamado R que por enquanto está indisponível. O que T deve fazer? Se entrar em algum tipo de laço de sondagem à espera de R, T bloqueará o objeto, impedindo que outras threads o acessem. Essa não é uma solução ótima, porque invalida parcialmente as vantagens de programar em um ambiente com várias threads.
  - Uma solução melhor é fazer T abandonar temporariamente o controle do objeto, permitindo que outra thread seja executada. Quando R estiver disponível, T pode ser notificado e retomar a execução. Essa abordagem se baseia em alguma forma de comunicação entre threads em que uma thread pode notificar outra que está bloqueada e ser notificada para retomar a execução.
  - Java dá suporte à comunicação entre threads com os métodos wait( ), notify( ) e notifyAll( ).
  - Tais métodos fazem parte de todos os objetos porque são implementados pela classe Object.
  - Esses métodos só devem ser chamados de dentro de um contexto synchronized.
  - Funcionamento:
    - Quando a execução de uma thread é bloqueada temporariamente, ela chama wait( ).
    - Isso faz a thread entrar em suspensão e o monitor desse objeto ser liberado, permitindo que outra thread use o objeto.
    - A thread em suspensão poderá ser ativada posteriormente quando outra thread entrar no mesmo monitor e chamar notify( ) ou notifyAll( ).

MULTITHREADING – PARTE III

- Comunicação entre threads
  - Temos as diversas formas de wait( ) definidas por Object:
    - final void wait( ) throws InterruptedException
    - final void wait(long milis) throws InterruptedException
    - final void wait(long milis, int nanos) throws InterruptedException
    - A primeira forma espera até haver uma notificação.
    - A segunda espera até haver uma notificação ou o período especificado em milissegundos expirar.
    - A terceira forma permite a especificação do período de espera em nanosegundos.
  - As formas gerais de notify( ) e notifyAll( ):
    - final void notify( )
    - final void notifyAll( )
    - Uma chamada a notify( ) retoma a execução de uma thread que estava esperando.
    - Uma chamada a notifyAll( ) notifica todas as threads, com a de prioridade mais alta ganhando acesso ao objeto.

MULTITHREADING – PARTE III

- Exemplo de wait( ) e notify( )
  - Para entender a necessidade e a aplicação de wait( ) e notify( ), criaremos um programa que simula o tique-taque de um relógio exibindo as palavras Tick e Tock na tela.
  - Para fazê-lo, criaremos uma classe chamada TickTock contendo dois métodos: tick( ) e tock( ).
  - O método tick( ) exibe a palavra "Tick" e tock( ) exibe "Tock".
  - Para o relógio ser executado, duas threads são criadas, uma que chama tick( ) e outra que chama tock( ).
  - O objetivo é fazer as duas threads serem executadas de maneira que a saída do programa exiba um "tique-taque" coerente – isto é, um padrão repetido de um tique seguido por um taque.

MULTITHREADING – PARTE III

- Exemplo de wait( ) e notify( )

```
class TickTock {

  String state; // contém o estado do relógio

  synchronized void tick(boolean running) {
    if(!running) { // interrompe o relógio
      state = "ticked";
      notify(); // notifica qualquer thread que estiver esperando
      return;
    }

    System.out.print("Tick ");

    state = "ticked"; // define o estado atual com ticked

    notify(); // permite que tock() seja executado <-- tick( ) notifica tock( ).
    try {
      while(!state.equals("tocked"))
        wait(); // espera tock() terminar <-- tick( ) espera tock( ).
    }
    catch(InterruptedException exc) {
      System.out.println("Thread interrupted.");
    }
  }

  synchronized void tock(boolean running) {
    if(!running) { // interrompe o relógio
      state = "tocked";
      notify(); // notifica qualquer thread que estiver esperando
      return;
    }

    System.out.println("Tock");

    state = "tocked"; // define o estado atual com tocked

    notify(); // permite que tick() seja executado <-- tock( ) notifica tick( ).
    try {
      while(!state.equals("ticked"))
        wait(); // espera tick() terminar <-- tock( ) espera tick( ).
    }
    catch(InterruptedException exc) {
      System.out.println("Thread interrupted.");
    }
  }
}
```

```
class MyThread implements Runnable {
  Thread thrd;
  TickTock ttOb;

  // Constrói uma nova thread.
  MyThread(String name, TickTock tt) {
    thrd = new Thread(this, name);
    ttOb = tt;
    thrd.start(); // inicia a thread
  }

  // Começa a execução da nova thread.
  public void run() {

    if(thrd.getName().compareTo("Tick") == 0) {
      for(int i=0; i<5; i++) ttOb.tick(true);
      ttOb.tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.tock(true);
      ttOb.tock(false);
    }
  }
}
```

```
class ThreadCom {
  public static void main(String args[]) {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Tick", tt);
    MyThread mt2 = new MyThread("Tock", tt);

    try {
      mt1.thrd.join();
      mt2.thrd.join();
    } catch(InterruptedException exc) {
      System.out.println("Main thread interrupted.");
    }
  }
}
```

MULTITHREADING – PARTE III

- Suspendendo, retomando e encerrando threads
  - Às vezes é necessário a suspensão da execução de uma thread.
    - Por exemplo, uma thread separada pode ser usada para exibir a hora do dia. Se o usuário não quiser um relógio, sua thread pode ser suspensa. Seja qual for o caso, é uma simples questão de suspender uma thread. Uma vez suspensa, também só temos de reiniciá-la.
    - O mecanismo de suspensão, encerramento e retomada de threads difere entre as versões antigas de Java e as versões mais modernas, a partir de Java 2. Antes de Java 2, os programas usavam suspend( ), resume( ) e stop( ), que são métodos definidos por Thread, para pausar, reiniciar e encerrar a execução de uma thread. Eles têm as formas a seguir:
      - final void resume( )
      - final void suspend( )
      - final void stop( )

MULTITHREADING – PARTE III

- Suspendendo, retomando e encerrando threads
  - Embora esses métodos pareçam uma abordagem perfeitamente sensata e conveniente para o gerenciamento da execução de threads, eles não devem mais ser usados.
    - O método suspend( ) da classe Thread foi substituído em Java 2. Isso foi feito porque às vezes suspend( ) pode causar problemas sérios que envolvem deadlock.
    - O método resume( ) também foi substituído; ele não causa problemas, mas não pode ser usado sem o método suspend( ) como complemento.
    - O método stop( ) da classe Thread também foi substituído em Java 2. A razão é que esse método às vezes também pode causar problemas sérios.
  - A thread deve ser projetada de modo que o método run( ) verifique periodicamente se ela deve suspender, retomar ou encerrar sua própria execução. Normalmente, isso pode ser feito com o estabelecimento de duas variáveis flag: uma para suspender e retomar e outra para encerrar.
  - Para a suspensão e retomada, se o flag estiver configurado com "em execução", o método run( ) deve continuar permitindo que a thread seja executada. Se essa variável for configurada com "suspender", a thread deve pausar. Quanto ao flag de encerramento, se ele for configurado com "encerrar", a thread deve terminar.

MULTITHREADING – PARTE III

- Exemplo de suspensão, retomada e encerramento de threads

```
// Suspendendo, retomando e encerrando uma thread.

class MyThread implements Runnable {
  Thread thrd;

  boolean suspended;  <-- Suspende a thread quando igual a true.
  boolean stopped;    <-- Encerra a thread quando igual a true.

  MyThread(String name) {
    thrd = new Thread(this, name);
    suspended = false;
    stopped = false;
    thrd.start();
  }
```

```
  // Este é o ponto de entrada da thread.
  public void run() {
    System.out.println(thrd.getName() + " starting.");
    try {
      for(int i = 1; i < 1000; i++) {
        System.out.print(i + " ");
        if((i%10)==0) {
          System.out.println();
          Thread.sleep(250);
        }

        // Usa um bloco sincronizado para verificar suspended e stopped.
        synchronized(this) {
          while(suspended) {
            wait();
          }
          if(stopped) break;
        }
      }
    } catch (InterruptedException exc) {
      System.out.println(thrd.getName() + " interrupted.");
    }
    System.out.println(thrd.getName() + " exiting.");
  }
  // Encerra a thread.
  synchronized void mystop() {
    stopped = true;

    // O código a seguir assegura que uma thread suspensa possa ser encerrada.
    suspended = false;
    notify();
  }

  // Suspende a thread.
  synchronized void mysuspend() {
    suspended = true;
  }

  // Retoma a thread.
  synchronized void myresume() {
    suspended = false;
    notify();
  }
}
```

Esse bloco sincronizado verifica **suspended** e **stopped**.

MULTITHREADING – PARTE III

- Exemplo de suspensão, retomada e encerramento de threads (continuação)

Um exemplo da saída desse programa é mostrado abaixo. (Você pode obter uma saída um pouco diferente).

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Stopping thread.
My Thread exiting.
Main thread exiting.
```

```
class Suspend {
  public static void main(String args[]) {
    MyThread ob1 = new MyThread("My Thread");

    try {
      Thread.sleep(1000); // permite que a thread ob1 comece a ser executada

      ob1.mysuspend();
      System.out.println("Suspending thread.");
      Thread.sleep(1000);

      ob1.myresume();
      System.out.println("Resuming thread.");
      Thread.sleep(1000);

      ob1.mysuspend();
      System.out.println("Suspending thread.");
      Thread.sleep(1000);

      ob1.myresume();
      System.out.println("Resuming thread.");
      Thread.sleep(1000);

      ob1.mysuspend();
      System.out.println("Stopping thread.");
      ob1.mystop();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    // espera a thread terminar
    try {
      ob1.thrd.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

MULTITHREADING – PARTE III

```
class UseMain {
  public static void main(String args[]) {
    Thread thrd;

    // Acessa a thread principal.
    thrd = Thread.currentThread();

    // Exibe o nome da thread principal.
    System.out.println("Main thread is called: " +
                       thrd.getName());

    // Exibe a prioridade da thread principal.
    System.out.println("Priority: " +
                       thrd.getPriority());

    System.out.println();

    // Define nome e prioridade.
    System.out.println("Setting name and priority.\n");
    thrd.setName("Thread #1");
    thrd.setPriority(Thread.NORM_PRIORITY+3);

    System.out.println("Main thread is now called: " +
                       thrd.getName());

    System.out.println("Priority is now: " +
                       thrd.getPriority());
  }
}
```

- Usando a thread principal
  - Todos os programas Java têm pelo menos uma thread de execução, chamada thread principal
  - Esta thread é fornecida ao programa automaticamente quando ele começa a ser executado.
  - Exemplo:

A saída do programa é mostrada abaixo:

```
Main thread is called: main
Priority: 5

Setting name and priority.

Main thread is now called: Thread #1
Priority is now: 8
```

MULTITHREADING – PARTE III

- Considerações
  - O segredo para o uso eficiente de várias threads é pensar ao mesmo tempo em vez de sequencialmente.
  - Por exemplo, se você tiver dois subsistemas totalmente independentes dentro de um programa, considere transformá-los em threads individuais.
  - No entanto, é preciso tomar cuidado. Se você criar threads demais, pode piorar o desempenho de seu programa em vez de melhorá-lo.
  - Lembre-se, a sobrecarga está associada a mudança de contexto.
  - Se você criar threads demais, mais tempo da CPU será gasto com mudanças de contexto do que na execução de seu programa!

MULTITHREADING – PARTE III

- Referência
  - Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015.