MULTITHREADING EM JAVA PARTE II

Prof. Dr. Rodrigo Palácios

MULTITHREADING – PARTE II

- Determinando quando uma thread termina
  - Eventualmente é necessário saber quando uma thread terminou.
  - Pode-se fazer a thread principal entrar em suspensão por mais tempo do que as threads filhas que ela gerou.
  - A Thread fornece dois meios para determinar se uma thread terminou.
  - O primeiro é chamar o método isAlive( ) na thread.
    - final boolean isAlive( )
  - O método isAlive( ) retorna true se a thread em que foi chamado ainda estiver sendo executada, caso contrário, retorna false.

MULTITHREADING – PARTE II

- Exemplo do uso do método isAlive( )

```
// Usa isAlive().
class MoreThreads {
  public static void main(String args[]) {
    System.out.println("Main thread starting.");

    MyThread mt1 = new MyThread("Child #1");
    MyThread mt2 = new MyThread("Child #2");
    MyThread mt3 = new MyThread("Child #3");

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Main thread interrupted.");
      }
    } while (mt1.thrd.isAlive() ||
             mt2.thrd.isAlive() ||
             mt3.thrd.isAlive());

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

Espera até todas as threads terminarem.

- Essa versão produz uma saída semelhante a da versão sem o uso do isAlive( ), porém main( ) termina assim que as outras threads terminam.
- A diferença é que a thread principal aguarda as threads filhas terminarem.

MULTITHREADING – PARTE II

- Uso do método join( )
  - Um modo alternativo de esperar uma thread terminar é chamar o método join( ).
    - final void join() throws InterruptedException
  - Esse método espera até que a thread em que foi chamado termine.
  - A thread que fez a chamada ter de esperar até a thread especificada se juntar a ela.
  - Formas adicionais de join( ) nos permitem indicar o período de tempo máximo que queremos esperar até que a thread especificada termine.

MULTITHREADING – PARTE II

- Exemplo do uso do método join( )

```
// Usa join().

class MyThread implements Runnable {
  Thread thrd;

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

  // Começa a execução da nova thread.
  public void run() {
    System.out.println(thrd.getName() + " starting.");
    try {
      for(int count=0; count < 10; count++) {
        Thread.sleep(400);
        System.out.println("In " + thrd.getName() +
                           ", count is " + count);
      }
    }
    catch(InterruptedException exc) {
      System.out.println(thrd.getName() + " interrupted.");
    }
    System.out.println(thrd.getName() + " terminating.");
  }
}
```

```
class JoinThreads {
  public static void main(String args[]) {
    System.out.println("Main thread starting.");

    MyThread mt1 = new MyThread("Child #1");
    MyThread mt2 = new MyThread("Child #2");
    MyThread mt3 = new MyThread("Child #3");

    try {
      mt1.thrd.join();
      System.out.println("Child #1 joined.");
      mt2.thrd.join();
      System.out.println("Child #2 joined.");
      mt3.thrd.join();
      System.out.println("Child #3 joined.");
    }
     catch(InterruptedException exc) {
       System.out.println("Main thread interrupted.");
     }
     System.out.println("Main thread ending.");
  }
}
```

Espera até a thread especificada terminar

MULTITHREADING – PARTE II

- Prioridades das threads
  - Cada thread tem associada a ela uma configuração de prioridade.
  - A prioridade de uma thread determina, em parte, quanto tempo de CPU ela receberá em relação a outras threads ativas.
    - Threads de baixa prioridade recebem pouco tempo.
    - Threads de alta prioridade recebem muito tempo.
  - O tempo de CPU que uma thread recebe tem impacto profundo sobre suas características de execução e sua interação com outras threads concorrentes.
  - É importante entender que outros fatores além da prioridade afetam quanto tempo de CPU a thread receberá.
  - Por exemplo: se uma thread de alta prioridade estiver esperando algum recurso, talvez uma entrada do teclado, ela será bloqueada, e uma thread de prioridade mais baixa será executada. No entanto, quando a thread de alta prioridade ganhar acesso ao recurso, poderá interceptar a thread de baixa prioridade e retomar a execução.
  - Outro fator que afeta o agendamento de threads é a maneira como o sistema operacional implementa a multitarefa. Logo, não é porque você deu prioridade alta a uma thread e prioridade baixa a outra que uma thread será necessariamente executada com mais rapidez ou frequência do que a outra. Simplesmente, a thread de alta prioridade tem mais chances de acessar a CPU.

MULTITHREADING – PARTE II

- Prioridades das threads
  - Quando uma thread filha é iniciada, sua configuração de prioridade é igual à da thread mãe.
  - Pode-se alterar a prioridade de uma thread chamando o método setPriority( ), que é membro de Thread.
    - final void setPriority(int nivel)
  - O valor do parâmetro especifica a nova configuração de prioridade da thread.
  - O valor do parâmetro deve estar dentro do intervalo MIN_PRIORITY (1) e MAX_PRIORITY (10).
  - Para retornar uma thread para a prioridade padrão, especifique NORM_PRIORITY (5).
  - Essas prioridades estão definidas como variáveis estáticas e finais dentro de Thread.
  - Para obter a configuração de prioridade atual deve se utilizar o método getPriority( ) de Thread.
    - final int getPriority( )

MULTITHREADING – PARTE II

- Exemplo de prioridades das threads

```
class Priority implements Runnable {
  int count;
  Thread thrd;

  static boolean stop = false;
  static String currentName;

  /* Constrói uma nova thread. Observe que
     esse construtor não inicia realmente
     a execução das threads. */

  Priority(String name) {
    thrd = new Thread(this, name);
    count = 0;
    currentName = name;
  }

  // Começa a execução da nova thread.
  public void run() {
    System.out.println(thrd.getName() + " starting.");
    do {
      count++;

      if(currentName.compareTo(thrd.getName()) != 0) {
        currentName = thrd.getName();
        System.out.println("In " + currentName);
      }

    } while(stop == false && count < 10000000);
    stop = true;

    System.out.println("\n" + thrd.getName() +
                       " terminating.");
  }
}
```

A primeira thread a alcançar 10.000.000 interrompe todas as threads.

```
class PriorityDemo {
  public static void main(String args[]) {
    Priority mt1 = new Priority("High Priority");
    Priority mt2 = new Priority("Low Priority");

    // define as prioridades
    mt1.thrd.setPriority(Thread.NORM_PRIORITY+2);
    mt2.thrd.setPriority(Thread.NORM_PRIORITY-2);

    // inicia as threads
    mt1.thrd.start();
    mt2.thrd.start();

    try {
      mt1.thrd.join();
      mt2.thrd.join();

    }
    catch(InterruptedException exc) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("\nHigh priority thread counted to " +
                       mt1.count);
    System.out.println("Low priority thread counted to " +
                       mt2.count);
  }
}
```

Dá a **mt1** uma prioridade mais alta que a de **mt2**.

- Nessa execução, a thread de alta prioridade obteve grande parte do tempo da CPU. É claro que a saída exata produzida por esse programa dependerá da velocidade da CPU, do número de CPUs do sistema, do sistema operacional que está sendo executado e de quantas tarefas mais estão em execução no sistema.

MULTITHREADING – PARTE II

- Descrição o exemplo anterior
  - O exemplo anterior demonstra duas threads com prioridades diferentes.
    - As threads são criadas como instâncias da classe Priority.
    - O método run( ) contém um laço que conta o número de iterações.
    - O laço para quando a contagem alcança 10.000.000 ou a variável estática stop é igual a true.
    - Inicialmente, stop é configurada com false, mas a primeira thread a chegar ao fim da contagem configura stop com true.
    - Isso faz a segunda thread terminar em sua próxima fração de tempo.
    - A cada passagem do laço o string de currentName é comparado com o nome da thread que está sendo executada. Se não coincidirem é porque ocorreu uma alternância de tarefa.
    - Sempre que ocorre uma alternância de tarefa, o nome da nova thread é exibido e atribuído à currentName.
    - A exibição de cada alternância de threads permite que você saiba (de modo muito preciso) quando a thread obteve acesso à CPU.
    - Quando as duas threads terminam, é exibido o número de iterações de cada laço.

MULTITHREADING – PARTE II

- Prioridades das threads
  - A implementação da multitarefa por parte do sistema operacional afeta o tempo de CPU que uma thread vai receber?
  - Além da configuração de prioridade de uma thread, o fator mais importante que afeta a execução de threads é a maneira como o sistema operacional implementa a multitarefa e o agendamento. Alguns sistemas operacionais usam a multitarefa com preempção em que cada thread recebe uma fração de tempo, pelo menos ocasionalmente. Outros sistemas usam o agendamento sem **preempção** em que uma thread deve abandonar a execução para outra ser executada. Em sistemas sem preempção, é fácil uma thread assumir o controle, impedindo que outras sejam executadas.

  - **Preempção = é o ato de interromper temporariamente uma tarefa sendo executada por um sistema computacional.**

MULTITHREADING – PARTE II

- Sincronização
  - Quando várias threads são usadas, às vezes é necessário coordenar as atividades de duas ou mais.
  - O processo que faz isso se chama sincronização.
  - A razão mais comum para o uso da sincronização é para quando duas ou mais threads precisam de acesso a um recurso compartilhado que só pode ser usado por uma thread de cada vez.
  - Por exemplo:
    - Quando uma thread está gravando em um arquivo, uma segunda thread deve ser impedida de gravar ao mesmo tempo.
    - Outra razão para usarmos a sincronização é quando uma thread está esperando um evento causado por outra thread. Nesse caso, é preciso que haja um meio da primeira thread ser mantida em estado suspenso até o evento ocorrer. Então, a thread em espera deve retomar a execução.

MULTITHREADING – PARTE II

- Sincronização
  - Essencial para a sincronização em Java é o conceito de monitor, que controla o acesso a um objeto.
    - Um monitor funciona implementando o conceito de bloqueio. Quando um objeto é bloqueado por uma thread, nenhuma outra thread pode ganhar acesso a ele. Quando a thread termina, o objeto é desbloqueado e fica disponível para ser usado por outra thread.
  - Todos os objetos em Java têm um monitor.
  - Esse recurso existe dentro da própria linguagem Java. Logo, todos os objetos podem ser sincronizados.
  - A sincronização é suportada pela palavra-chave synchronized e alguns métodos bem definidos que todos os objetos têm.
  - Já que a sincronização foi projetada em Java desde o início, é muito mais fácil de usar do que parece. Na verdade, para muitos programas, a sincronização de objetos é quase transparente.
  - Há duas maneiras de você sincronizar seu código com o uso da palavra-chave synchronized.

MULTITHREADING – PARTE II

- Usando métodos sincronizados
  - Você pode sincronizar o acesso a um método modificando-o com a palavra-chave synchronized.
  - Quando esse método for chamado, a thread chamadora entrará no monitor do objeto, que então será bloqueado.
  - Enquanto ele estiver bloqueado, nenhuma outra thread poderá entrar no método ou em qualquer outro método sincronizado definido pela classe do objeto.
  - Quando a thread retornar do método, o monitor desbloqueará o objeto, permitindo que ele seja usado pela próxima thread. Logo, a sincronização é obtida sem que você faça praticamente nenhum esforço de programação.
  - O programa a seguir demonstra a sincronização controlando o acesso a um método chamado sumArray( ), que soma os elementos de um array de inteiros.

MULTITHREADING – PARTE II

- Exemplo de métodos sincronizados

```
// Usa a sincronização para controlar o acesso.

class SumArray {
  private int sum;

  synchronized int sumArray(int nums[]) {   <-- sumArray é sincronizado
    sum = 0; // redefine sum

    for(int i=0; i<nums.length; i++) {
      sum += nums[i];
      System.out.println("Running total for " +
             Thread.currentThread().getName() +
             " is " + sum);
      try {
        Thread.sleep(10); // permite a alternância de tarefas
      }
      catch(InterruptedException exc) {
        System.out.println("Thread interrupted.");
      }
    }
    return sum;
  }
}
```

```
class MyThread implements Runnable {
  Thread thrd;
  static SumArray sa = new SumArray();
  int a[];
  int answer;

  // Constrói uma nova thread.
  MyThread(String name, int nums[]) {
    thrd = new Thread(this, name);
    a = nums;
    thrd.start(); // inicia a thread
  }

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

    System.out.println(thrd.getName() + " starting.");

    answer = sa.sumArray(a);
    System.out.println("Sum for " + thrd.getName() +
                       " is " + answer);

    System.out.println(thrd.getName() + " terminating.");
  }
}
```

MULTITHREADING – PARTE II

- Exemplo de métodos sincronizados (continuação)

```
class Sync {
  public static void main(String args[]) {
    int a[] = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Child #1", a);
    MyThread mt2 = new MyThread("Child #2", a);

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

  }
}
```

A saída do programa (podendo ser diferente no seu computador.)

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 1
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

MULTITHREADING – PARTE II

- Exemplo de métodos sincronizados (continuação)
  - Detalhes do exemplo anterior:
    - O programa cria três classes:
      - A classe SumArray contém o método sumArray( ), que soma um array de inteiros.
      - A segunda classe é MyThread, que usa um objeto static de tipo SumArray para obter a soma de um array de inteiros. Esse objeto se chama “sa" e, já que é static, há apenas uma cópia dele compartilhada por todas as instâncias de MyThread.
      - A classe Sync cria duas threads e as faz calcular a soma de um array de inteiros.
    - Dentro de sumArray( ), sleep( ) é chamado para permitir que ocorra uma alternância intencional de tarefas, se puder ocorrer uma, mas não pode. Já que sumArray( ) é sincronizado, só pode ser usado por uma thread de cada vez. Logo, quando a segunda thread filha começa a ser executada, ela não entra em sumArray( ) até que a primeira thread filha tenha acabado de usá-lo. Isso assegura que o resultado correto seja produzido.

MULTITHREADING – PARTE II

- Exemplo de métodos sincronizados (continuação)
  - Para entender melhor os efeitos de synchronized, tente removê-la da declaração de sumArray( ). Após fazê-lo, sumArray não será mais sincronizado e um número ilimitado de threads poderá usá-lo ao mesmo tempo.
  - O problema é que o total atual é armazenado em sum, que será alterada por cada thread que chamar sumArray( ) por intermédio do objeto estático “sa".
  - Logo, quando duas threads chamam sa.sumArray( ) ao mesmo tempo, resultados incorretos são produzidos porque sum reflete a soma feita pelas duas threads juntas.
  - Por exemplo, a seguir o exemplo da saída do programa após synchronized ser removida da declaração de sumArray( ). (A saída exata pode ser diferente no seu computador.)

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #2 is 8
Running total for Child #1 is 11
Running total for Child #2 is 15
Running total for Child #1 is 19
Running total for Child #2 is 24
Sum for Child #2 is 24
Child #2 terminating.
Running total for Child #1 is 29
Sum for Child #1 is 29
Child #1 terminating.
```

MULTITHREADING – PARTE II

- Exemplo de métodos sincronizados (continuação)
  - Como a saída mostra, as duas threads filhas estão chamando sa.sumArray( ) ao mesmo tempo e o valor de sum foi corrompido. Antes de prosseguir, examinemos os postos-chave de um método sincronizado:
    - Um método sincronizado é criado quando precedemos sua declaração com synchronized.
    - Para qualquer objeto dado, uma vez que um método sincronizado tiver sido chamado, o objeto será bloqueado e nenhum método sincronizado no mesmo objeto poderá ser usado por outra thread de execução.
    - Outras threads que tentarem chamar um objeto sincronizado em uso entrarão em estado de espera até o objeto ser desbloqueado.
    - Quando uma thread deixa o método sincronizado, o objeto é desbloqueado.

MULTITHREADING – PARTE II

- A instrução synchronized
  - Embora a criação de métodos synchronized dentro das classes que criamos seja um meio fácil e eficaz de obter sincronização, ele não funciona em todos os casos.
  - Por exemplo, podemos querer sincronizar o acesso a algum método que não seja modificado por synchronized. Isso pode ocorrer por querermos usar uma classe que não foi criada por nós, e sim por terceiros, e não termos acesso ao código-fonte. Logo, não é possível adicionar synchronized aos métodos apropriados dentro da classe. Como o acesso a um objeto dessa classe pode ser sincronizado? É indicado inserir as chamadas aos métodos definidos por essa classe dentro de um bloco synchronized.

```
synchronized(refobj) {
        // instruções a serem sincronizadas
}
```

MULTITHREADING – PARTE II

- A instrução synchronized
  - Uma outra maneira de sincronizar as chamadas a sumArray( ) é chamá-lo de dentro de um bloco sincronizado, como mostrado na versão seguinte do programa:

```
// Usa um bloco sincronizado para controlar o acesso a sumArray.
class SumArray {
  private int sum;

  int sumArray(int nums[]) {
    sum = 0; // redefine sum

    for(int i=0; i<nums.length; i++) {
      sum += nums[i];
      System.out.println("Running total for " +
            Thread.currentThread().getName() +
            " is " + sum);
      try {
        Thread.sleep(10); // permite a alternância de tarefas
      }
      catch(InterruptedException exc) {
        System.out.println("Thread interrupted.");
      }
    }
    return sum;
  }
}
```

Aqui, **sumArray()** não é sincronizado.

```
class MyThread implements Runnable {
  Thread thrd;
  static SumArray sa = new SumArray();
  int a[];
  int answer;

  // Constrói uma nova thread.
  MyThread(String name, int nums[]) {
    thrd = new Thread(this, name);
    a = nums;
    thrd.start(); // inicia a thread
  }

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

    System.out.println(thrd.getName() + " starting.");

    // sincroniza as chamadas a sumArray()
    synchronized(sa) {
      answer = sa.sumArray(a);
    }
    System.out.println("Sum for " + thrd.getName() +
                       " is " + answer);

    System.out.println(thrd.getName() + " terminating.");
  }
}
```

Aqui, as chamadas a **sumArray( )** em **sa** são sincronizadas.

```
class Sync {
  public static void main(String args[]) {
    int a[] = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Child #1", a);
    MyThread mt2 = new MyThread("Child #2", a);

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

MULTITHREADING – PARTE II

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