Aplicações multithread

Programação para dispositivos móveis

Prof. Allan Rodrigo Leite

Aplicações concorrentes

- São aplicações compostas por vários fluxos de execução executando concorrentemente em um mesmo processo
  - Distribuídos
    - Fluxos de execução em diferentes dispositivos físicos conectados em rede
    - Objetiva o compartilhamento de recursos computacionais
  - Multithread
    - Fluxos de execução em um mesmo processo
    - Objetiva execução concorrente ou paralela de uma rotina em uma aplicação, melhorando seu desempenho em vários aspectos

Definição de thread

- São unidades de execução independente criada dentro de um contexto de um processo
  - Neste caso não é necessário manter em memória múltiplas cópias do código do programa
  - Thread é um recurso muito mais leve do que um processo para o sistema operacional
  - Podem compartilhar recursos dos seus respectivos processos e/ou outras threads em execução

Gerenciamento de threads

- Pacote `java.lang.threads`
  - Dispõe de classes para programação multithreads em Java
    - Fornece duas técnicas de implementação: `Thread` e `Runnable`
    - Devem implementar um método `run`
  - Classe `Thread`
    - Threads criadas a partir de subclasses de `Thread`
    - Implementação mais prática
  - Interface `Runnable`
    - Threads criadas a partir da interface `Runnable`
    - Útil para contornar herança simples
    - Permite que a classe herde de uma outra classe

Gerenciamento de threads

```
public class PingPong extends Thread {
  private String palavra;
  public PingPong(String palavra) { this.palavra = palavra; }

  public void run() { //implementa o método run
    for (int i = 0; i < 1000; i++) {
      System.out.print(palavra + " ");
      Thread.sleep(1000);
    }
  }

  public static void main(String[] args) {
    new PingPong("ping").start();
    new PingPong("pong").start();
  }
}
```

Gerenciamento de threads

```
public class PingPong implements Runnable {
  private String palavra;
  public PingPong(String palavra) { this.palavra = palavra; }

  public void run() { //implementa o método run
    for (int i = 0; i < 1000; i++) {
      System.out.print(palavra + " ");
      Thread.sleep(1000);
    }
  }

  public static void main(String[] args) {
    new Thread(new Runnable("ping")).start();
    new Thread(new Runnable("pong")).start();
  }
}
```

Sincronização de threads

- Técnica para controle de acesso à múltiplas threads em execução
  - Permite que apenas uma thread acesse um recurso compartilhado ou que sua execução seja controlada externamente

- Geralmente usada para lidar com:
  - Cooperação de múltiplas threads
    - Melhorar o desempenho do sistema por meio de computação assíncrona
    - Aumentar a capacidade de vazão (throughput)
  - Técnica para exclusão mútua
    - Lidar com concorrência ou interferência de múltiplas threads
    - Prevenir problemas de consistência de estado

Cooperação de threads

- Método `join`
  - Aguarda a finalização de uma thread em um outro fluxo de execução

- Métodos `wait`
  - Suspende a execução de uma thread

- Métodos `notify` e `notifyAll`
  - Notifica uma thread suspensa para voltar a executar

Cooperação de threads

```
public class PingPong extends Thread {
  //...

  public static void main(String[] args) {
    var t1 = new PingPong("ping");
    var t2 = new PingPong("pong");

    t1.start();
    t2.start();

    t1.join();
    t2.join();

    System.out.println("Threads finalizadas com sucesso");
  }
}
```

Exclusão mútua

- Técnica para criação de trechos de código que executam de forma atômica, não são intercalados com outros códigos concorrentes
  - Esses trechos são chamados de seções críticas
  - Tais seções críticas são criadas por meio da palavra `synchronized`

- Prevenção de problema de condição de disputa em regiões críticas
  - Este problema é conhecido como condição de corrida
  - Ocorre quando duas ou mais threads tentam acessar o mesmo recurso ao mesmo tempo e causam algum efeito indesejável no sistema

Exclusão mútua

```
public class ContaBancaria {
  private double saldo;
  public void depositar(double valor) {
    var s = getSaldo();
    s += valor;
    setSaldo(s);
  }
  public void sacar(double valor) {
    var s = getSaldo();
    s -= valor;
    setSaldo(s);
  }
  public double getSaldo() { return saldo; };
  public void setSaldo(double saldo) { this.saldo = saldo; }
}
```

Exclusão mútua

- Suponha que duas threads compartilham o mesmo objeto de conta bancária e realizem operações concorrentes

Thread 1	Thread 2
s= getSaldo();	
s= s+ valor;	
setSaldo(s);	
	s= getSaldo() ;
	s= s+ valor;
	setSaldo(s);

Thread 1	Thread 2
	s= getSaldo();
	s= s+ valor;
	setSaldo(s);
s= getSaldo() ;	
s= s+ valor;	
setSaldo(s);	

Exclusão mútua

```
public class ContaBancaria {
  private double saldo;
  public synchronized void depositar(double valor) {
    var s = getSaldo();
    s += valor;
    setSaldo(s);
  }
  public synchronized void sacar(double valor) {
    var s = getSaldo();
    s -= valor;
    setSaldo(s);
  }
  public double getSaldo() { return saldo; };
  public void setSaldo(double saldo) { this.saldo = saldo; }
}
```

Exercícios

- Suponha um buffer com capacidade de armazenamento de um conjunto de valores inteiros, cujo acesso aos dados é realizado por meio de uma fila
  - Este buffer será um recurso compartilhado por duas threads (produtor e consumidor)
    - A primeira thread irá produzir valores e adicioná-los no buffer
    - A segunda thread irá consumir valores existentes no buffer

- Este buffer deve conter os seguintes métodos:
  - `int get()`: recupera o valor inteiro contido no início da fila do buffer
  - `void put(int valor)`: armazena um novo valor inteiro no final da fila do buffer

- Condições
  - O buffer deve possuir uma capacidade máxima de armazenamento
  - Os métodos `get` e `put` devem ser sincronizados
  - O método `put` somente pode ser executado se o buffer não estiver cheio
  - O método `get` somente pode ser executado se o buffer não estiver vazio
  - Os métodos `put` e `get` ficam suspensos dependendo do estado do buffer (cheio ou vazio)

Execuções assíncronas

- A interface `Future` representa um resultado futuro de uma computação assíncrona
  - A partir desta interface é possível:
    - Cancelar a execução de uma tarefa assíncrona
    - Descobrir se a execução já terminou com sucesso ou erro
    - Aguardar a finalização da tarefa, caso ainda não tenha sido finalizada

- Quais cenários são candidatos para usar execuções assíncronas?
  - Métodos de longa execução
  - Métodos que acessam recursos e/ou serviços externos
    - API, banco de de dados, etc.
  - Composição de métodos cuja execução pode ser realizada por tarefas menores e concorrentes

Execuções assíncronas

```
public class PingPong extends Thread {
  //...

  public static void main(String[] args) throws ExecutionException, InterruptedException {
    var executor = Executors.newSingleThreadExecutor();
    List<Future> futures = new ArrayList<>();

    futures.add(executor.submit(new TestFuture("Ping")));
    futures.add(executor.submit(new TestFuture("Pong")));

    System.out.println("Threads submetidas");

    for (Future future: futures)
      future.get();

    executor.shutdown();

    System.out.println("Threads finalizadas");
  }
}
```

Execuções assíncronas

- O `ExecutorService` é uma API do JDK que visa simplificar a execução de tarefas assíncronas
  - Esta API fornece recursos para criação e gerenciamento de um pool de threads para atribuição de tarefas a elas
  - Também é possível estabelecer a maneira de controlar a sequência de execução das tarefas assíncronas

  - Para usar esta API é necessário definir:
    - Qual será o tamanho do pool de threads
    - Qual será a tarefa a ser executada
    - Sequência de execução das tarefas, caso exista alguma restrição

Execuções assíncronas

```
public class PingPong extends Thread {
  //...

  public static void main(String[] args) throws InterruptedException {
    var executor = Executors.newFixedThreadPool(2);

    for (int i = 0; i < 10; i++) {
      executor.execute(new PingPong("ping-" + i));
      executor.execute(new PingPong("pong-" + i));
    }

    System.out.println("Threads submetidas");
    executor.shutdown();

    if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
      executor.shutdownNow();
    }
    System.out.println("Threads finalizadas");
  }
}
```

Execuções assíncronas

- O `ExecutorService` é uma API do JDK que visa simplificar a execução de tarefas assíncronas
  - Esta API fornece recursos para criação e gerenciamento de um pool de threads para atribuição de tarefas a elas
  - Também é possível estabelecer a maneira de controlar a sequência de execução das tarefas assíncronas

  - Para usar esta API é necessário definir:
    - Qual será o tamanho do pool de threads
    - Qual será a tarefa a ser executada
    - Sequência de execução das tarefas, caso exista alguma restrição

Exercícios

- Altere a implementação do exercício anterior sobre produtor e consumidor para adotar:
  - Pool de threads para produtor e consumidor por meio da API `ExecutorService`
  - Utilização de tarefas assíncronas por meio da interface `Future`

- Ao final da execução de uma sequência de tarefas assíncronas (produtor e consumidor), exiba o tempo que as threads consumidor ficaram ociosas aguardando por um novo valor no buffer

Aplicações multithread

Programação para dispositivos móveis

Prof. Allan Rodrigo Leite