Universidade Federal do Ceará

Disciplina de Sistemas Distribuídos – Aula Prática Threads
Professores Windson Viana de Carvalho

Aluno: HUDSON COSTA GONÇALVES DA CRUZ
Matrícula:_____

Preâmbulo

As *threads* correspondem a "linhas de execução independentes" no âmbito de um mesmo processo. No caso da linguagem JAVA, é precisamente o conceito de Thread que é utilizado como modelo para a programação concorrente, permitindo que uma aplicação em execução numa máquina virtual Java possa ter várias linhas de execução concorrentes em que cada uma corresponde a uma thread.

Esta prática, baseada no exercício[1], explora a utilização de threads em JAVA, bem como, de uma forma geral, os conceitos básicos de programação concorrente em JAVA. Como referências de apoio a esta atividade:

- http://www.tutorialspoint.com/java/java_multithreading.htm
- http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
- http://www.caelum.com.br/apostila-java-orientacao-objetos/programacao-concorrente-e-threads/#17-1-threads

1

1) Existem duas formas básicas de criação de Threads em Java, uma utilizando a interface Runnable e outra utilizando a extensão da Classe Thread.

a) Explique quais as diferenças, vantagens e desvantagens das duas abordagens.

Vantagens da interface Runable:

- Uso do polimorfismo.

Vantagens do uso da classe Thread:

- Diversos construtores, bem como diversos métodos.

Desvantagens da extensão da classe Thread:

- Não é possível criar uma subclasse da classe Thread.
- A classe já deriva outra classe, por exemplo a classe Applet. Outras vezes, por questões de pureza de projeto o projetista não deseja derivar a classe Thread simplesmente para poder criar um thread uma vez que isto viola o significado da relação de classe-subclasse.

b) Crie uma classe Racer que possui um "while (true)" e imprime a frase "Racer i – imprimindo" onde i deve ser um parâmetro do seu construtor. Transforme esta classe em uma Thread usando as duas formas de criação e instanciação.

```java
package aulaDeSD13042018;

public class racerThread extends Thread{

    private int i;

    public racerThread(int r){
        this.i = r;
    }

    public void run(){

        while(true) {
        System.out.println("Racer " + this.i + " - imprimindo.");
        }
    }
}
```

```java
package aulaDeSD13042018;

public class implementaRacer {

    public static void mai(String args[]) {

        new racerThread(1).start();
    }

}
```

```java
package aulaDeSD13042018;

public class racerRunnable implements Runnable{

    private int r;

    public racerRunnable(int i) {
        this.r = i;
    }

    public static void main(String args[]) {
        racerRunnable racerUm = new racerRunnable(1);
        Thread t1 = new Thread(racerUm);
        t1.start();
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        while(true) {
            System.out.println("Racer " + this.r + " - imprimindo.");
            }

    }
}
```

c) Crie uma classe Race que cria 10 racers (identificadores de 1 a 10). Como se deu o comportamento dos prints?
R: As impressões acontecem em ordem aleatória a partir de um dado momento.

```
1  package aulaDeSD13042018;
2
3  public class implementaRacer {
4
5      public static void main(String args[]) {
6
7          for (int i=1; i<=10; i++)
8          new racerThread(i).start();
9      }
10
11 }
```

d) Adiciona um tempo de espera (usando o método sleep) nos Racers, o que houve com comportamento do sistema?
R: Há uma parada no processamento de cada thread.

```
1  package aulaDeSD13042018;
2
3  public class implementaRacer {
4
5      public static void main(String args[]) throws InterruptedException {
6
7          for (int i=1; i<=10; i++) {
8                  new racerThread(i).start();
9                  Thread.currentThread().sleep(10000);
10         }
11     }
12
13 }
```

e) Utilize o método setPriority para definir as condições de corrida. Houve mudanças na execução? Se sim, descreva-as.
R: Sim. A thread com prioridade mais alta estava sendo executada primeiro que as demais.

```
1  package aulaDeSD13042018;
2
3  public class implementaRacer {
4
5      public static void main(String args[]) throws InterruptedException {
6
7          for (int i=1; i<=10; i++) {
8              if (i == 5)
9                  Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
10
11             new racerThread(i).start();
12             Thread.currentThread().sleep(3000);
13         }
14     }
15
16 }
```

f) Modifique a classe Racer para que ela imprima apenas 1000 vezes. Em seguida, modifique a classe Race para que os carros pares só iniciem suas corridas quando os ímpares terminarem. Use o método join para tal tarefa

Para as questões seguintes, a classe Depósito será utilizada

```
public class Deposito {
	private int items = 0;
	private final int capacidade = 100;

	public int getNumItens(){
		return items;

	}

	public boolean retirar() {
			items=getNumItens() - 1;
			return true;
	}

	public boolean colocar() {
			items=getNumItens() +1;
			return true;
			}

	public static void main(String[] args) {
		Deposito dep = new Deposito();
		Produtor p = new Produtor(dep, 50);
		Consumidor c1 = new Consumidor(dep, 150);
		Consumidor c2 = new Consumidor(dep, 100);
		Consumidor c3 = new Consumidor(dep, 150);
		Consumidor c4 = new Consumidor(dep, 100);
		Consumidor c5 = new Consumidor(dep, 150);

		//Startar o produtor
		//...
		p.start();
		c1.start();
		c2.start();
		c3.start();
		c4.start();
		c5.start();

		//Startar o consumidor
		//...
		  System.out.println("Execucao do main da classe Deposito terminada");
		}

}
```

2) Crie duas classes, Produtor e Consumidor, que aumentem e diminuem o valor da variável itens do Depósito seguindo as seguintes regras:
   - Utilize threads para que o processamento seja simultâneo.
   - Para produzir, uma instância da classe Produtor deve invocar o método colocar da classe Depósito de forma a acrescentar caixas ao depósito. A produção não deve ser contínua. Um inteiro correspondente ao tempo em milissegundos entre produções é passado como parâmetro em seu construtor. Ela deve se encerrar após produzir 100 caixas.
   - Para consumir, uma instância da classe Consumidor invoca o método retirar da classe Depósito de forma a retirar caixas do depósito. Esse consumo ocorre seguindo um inteiro, passado no construtor, que correspondente ao tempo em milissegundos entre consumos de caixas. Um consumidor consome 20 caixas.

a) Como ficou o código das suas classes?

```java
package aula20042018;

public class Produtor extends Thread{

    Deposito dep;
    int i;
    boolean flag;

    public Produtor(Deposito dep, int i) {
        this.dep = dep;
        this.i = i;
    }

    public void run() {
        for (int j=1; j<=100;j++) {
            flag = dep.colocar();
            Thread.currentThread();
            try {
                Thread.sleep(i);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println("Produzido: " + dep.getNumItens());
    }

}
```

```java
package aula20042018;
public class Consumidor extends Thread{

    Deposito dep;
    int i;

    public Consumidor (Deposito dep, int i){
        this.dep = dep;
        this.i = i;
    }

    public void run() {
        for (int j=1; j<=20;j++) {

            while (!dep.retirar()) {

            try {
                Thread.currentThread();
                Thread.sleep(i);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            }
        }
        System.out.println("Consumido: " + dep.getNumItens());
    }
}
```

b) Qual é última mensagem de execução do código? Porque isso aconteceu?

```
Execucao do main da classe Deposito terminada
Consumido: -39
Consumido: -39
Consumido: -59
Consumido: -99
Consumido: -99
Produzido: 0
```

R: Estava sendo possível retirar antes de produzir.

c) Modifique a classe Depósito para que somente se possa retirar itens caso o valor deles seja maior que 0.

```
public boolean retirar() {
    if (this.getNumItens() <=0)
        return false;
        else {
        items=getNumItens() - 1;
        return true;
        }
}
```

d) Qual é última mensagem de execução do código? Porque isso aconteceu?

```
Execucao do main da classe Deposito terminada
Consumido: 0
Consumido: 0
Consumido: 0
Consumido: 0
Consumido: 0
Produzido: 22
```

R: Agora sim, a produção precede a retirada, ou seja, o consumo.

3) Quando um método é invocado mas a pré-condição necessária à sua execução (por exemplo existirem caixas no depósito) não se verifica, algumas atitudes devem ser tomadas. Nesses casos podem ser seguidas essencialmente três abordagens:

− Uma abordagem optimista (liveness first) é assumir que mais tarde ou mais cedo a pré-condição há-de ser verdadeira e por isso espera-se.

− Uma abordagem conservadora (safety first) admite que a pré-condição pode nunca vir a ser verdadeira (pelo menos em tempo útil) e por isso se num determinado momento não é possível executar o método então mais vale devolver uma indicação de erro.

− Uma abordagem intermédia consiste em esperar mas definir um tempo limite.

a) Modifique a Classe Consumidor para que caso não possa retirar um objeto do depósito, ela espere 200 ms e tente novamente.

b) Qual é última mensagem de execução do código? Porque isso aconteceu?

```
Execucao do main da classe Deposito terminada
Consumido: 1
Consumido: 0
Consumido: 0
Consumido: 0
Consumido: 0
Produzido: 17
```

Execução antes da alteração

```
Execucao do main da classe Deposito terminada
Consumido: 1
Consumido: 0
Consumido: 0
Consumido: 0
Consumido: 1
Produzido: 19
```

Execução após a alteração

R: É curioso notar que a cada execução do programa Depósito os valores do Produzido e do Consumido mudam.

4) A aplicação continua apresentando inconsistências pois a checagem do valor de itens e a operação de retirada ou de inserção podem não ocorrer de forma atômica devida ao *interleaving*. Modifique o código da questão anterior, use a sincronização de threads para controlar o acesso aos itens. Para isso leia o material abaixo e escolha o objeto e o método que deve ser sincronizado:

https://www.caelum.com.br/apostila-java-orientacao-objetos/apendice-problemas-com-concorrencia/#exerccios-avanados-de-programao-concorrente-e-locks

a) Após a implementação, observando os resultados é possível notar alguma diferença? Se sim, qual?

R: Sim. O método retirar da classe Depósito foi quem recebeu o `synchronized.` Antes disso a execução estava completamente imprevisível pois os valores que eram apresentados estavam mudando a cada execução.

Depois que o `synchronized` foi adicionado o comportamento dos valores tornou-se estável e previsível.

b) Que garantias o método *synchronized* trouxe para a execução?

R: Uma execução previsível.

c) Implemente o Produtor-Consumidor completo diminuindo também a capacidade do Depósito para 10 e checando na hora da produção se é possível ou não produzir novos itens. Implemente o padrão *Guarded Suspension* com notify() e wait() para resolver esse problema.