Universidade Federal do Ceará

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

Aluno:_____
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) 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.
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 instaciação .

1

c) Crie uma classe Race que cria 10 racers (identificadores de 1 a 10). Como se deu o comportamento dos prints?
d) Adiciona um tempo de espera (usando o método sleep) nos Racers, o que houve com comportamento do sistema?
e) Utilize o método setPriority para definir as condições de corrida. Houve mudanças na execução? Se sim, descreva-as.
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?
   b) Qual é última mensagem de execução do código? Porque isso aconteceu?
   c) Modifique a classe Depósito para que somente se possa retirar itens caso o valor deles seja maior que 0.
   d) Qual é última mensagem de execução do código? Porque isso aconteceu?

3) Quando um método é invocado mas a pré-condição necessária à sua execução (por exemplo existirem caixas no depósito) não se verifica, alguns 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?

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