Sistemas de Programação II **INF 02828** (Engenharia de Computação / Elétrica)	**Sistemas Operacionais** **INF 02780** (Ciência da Computação)

2º Trabalho de Programação
Período: 2008/1
Profª Patrícia Dockhorn Costa
Email: pdcosta@inf.ufes.br

Data de Entrega: 26/06/2008 ***Permitidos Grupos de 2-3 pessoas (mínimo 2 pessoas)***
Data de entrega inadiável. Para cada dia de atraso, será tirado um ponto da nota.

Material a entregar

- Impresso:
  - Relatório contendo as respostas das questões.
  - Listagem de todos os arquivos criados

- Por email:
  - Relatório (em formato PDF)
  - Todos os arquivos criados

Introdução Teórica

O suporte para threads na linguagem JAVA é realizado através da classe *java.lang.Thread* cujos métodos definem a API disponível para a gestão de threads. Entre as operações disponíveis incluem-se métodos para iniciar, executar, suspender e gerenciar uma thread. O código que define o que uma thread vai fazer é o que estiver no método *run*. A classe *Thread* é uma implementação genérica de uma thread com um método *run* vazio, cabendo ao programador definir esse código para uma thread em particular.

O ciclo de vida de uma thread começa com a criação do respectivo objeto, continua com a execução do método *run* (iniciada através da invocação do método *start*), e irá terminar quando terminar a execução do método *run*.

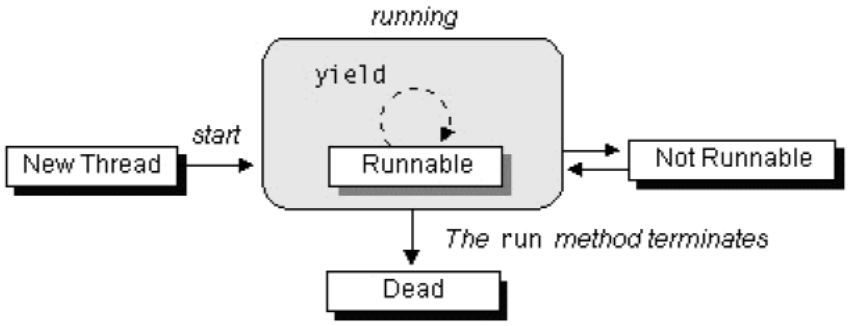

Figura 1: Ciclo de vida de uma thread

Umas das técnicas para criar uma nova thread de execução numa aplicação JAVA é criar uma subclasse de *Thread* e re-escrever o método *run*. A nova classe deverá redefinir o método *run* da classe *Thread* de forma a corresponder ao código que se pretende que a nova thread execute.

Exemplo 1

```
class PrimeThread extends Thread {

long minPrime;
PrimeThread(long minPrime) {
this.minPrime = minPrime;
}

public void run() {
// código a executar pela thread
. . .
}
}
```

A classe que pretendia criar a nova thread teria de executar o seguinte código:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

A outra forma de criar uma *Thread* é declarar uma classe que implementa a interface *Runnable* e que implementa o método *run*.

Exemplo 2

```
class PrimeRun implements Runnable {
long minPrime;
PrimeRun(long minPrime) {
this.minPrime = minPrime;
}

public void run() {
// código a executar pela thread
. . .
}
}
```

A classe que pretendia criar a nova thread teria de executar o seguinte código:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Mais informações: veja Tutorial de Java Threads em
http://java.sun.com/docs/books/tutorial/essential/threads/index.html

Parte I: O Problema do Produtor/Consumidor

Suponha um depósito de caixas, onde um produtor vai armazenando as caixas que vai produzindo. Um consumidor vai ao depósito para buscar essas mesmas caixas. O programa apresentado a seguir corresponde ao código do *Deposito*, permitindo a retirada e a armazenagem de elementos através dos métodos *retirar* e *armazenar*, respectivamente.

```
(Esqueleto da Classe Deposito.java)
public class Deposito {
      private int items=0;
      private final int capacidade=10;

      public int retirar() {
            if (items>0) {
                  items--;
                  System.out.println("Caixa retirada: Sobram "+items+
                  " caixas");
                  return 1; }
            return 0;
      }

      public int armazenar () {
            if (items<capacidade) {
                  items++;
                  System.out.println("Caixa armazenada: Passaram a ser"
                  +items+" caixas");
                  return 1; }
            return 0;
      }

      public static void main(String[] args) {
            Deposito dep = new Deposito();
            Produtor p1 = new Produtor(d, 1);
            Produtor p2 = new Produtor(d, 3);
            Consumidor c1 = new Consumidor(d, 2);
            Consumidor c2 = new Consumidor(d, 3);
            Consumidor c3 = new Consumidor(d, 4);

            // lançar a execução dos consumidores
            //...
            // lançar a execução dos produtores
            //...
            System.out.println("Execucao de Deposito.main() terminada");
      }
}
```

Crie uma classe *Produtor* que funcione como uma thread independente e que vai invocando o método *armazenar* da classe *Deposito* de forma a acrescentar caixas ao depósito. A classe *Produtor* deve receber no construtor uma referência para o objeto *dep* (da classe *Deposito*) onde os métodos vão ser invocados, e um inteiro *tmp* correspondente ao tempo em segundos entre produções de caixas.

Crie uma classe *Consumidor* que funcione como uma thread independente e que vai invocando o método *retirar* da classe depósito de forma a retirar caixas do depósito. A classe *Consumidor* deve receber no construtor uma referência para o objeto *dep* onde os métodos vão ser invocados e um inteiro correspondente ao tempo em segundos entre consumos de caixas.

IMPORTANTE:

- Como discutido em sala de aula, a existência de threads concorrentes levanta a necessidade de sincronização. Em Java, cada objeto tem associado um mecanismo similar ao de um monitor através do qual é possível garantir o acesso exclusivo às seções críticas desse mesmo objeto. O controle de acesso às secções críticas de um objeto é feito de forma automática pelo sistema de run-time do Java. O que o programador precisa fazer é apenas assinalar as secções críticas usando a primitiva *synchronized.* Um bloco de código sincronizado é uma região de código que apenas pode ser executado por uma thread de cada vez. Pode-se declarar um método como sendo *synchronized*:

  ```
  synchronized int pull () {... }
  ```

  Ou pode-se declarar apenas uma parte de um método como sendo synchronized:

  ```
  synchronized (this) {... }
  ```

- É importante permitir a coordenação entre threads *Produtor* e *Consumidor*. Em Java esta coordenação faz-se usando os seguintes métodos da classe Object: *wait*, *notify*, e *notifyAll* . No seu conjunto estes métodos permitem às threads bloquear à espera de determinada condição ou notificar outras threads que

se encontrem bloqueadas de que a condição pela qual estão à espera pode já se ter realizado. Utilize os métodos citados de forma a permitir uma coordenação adequada entre os Produtores e Consumidores. Observe que em Java a implementação do monitor é baseada na abordagem de Hansen: a condição lógica pela qual a thread estava esperando **pode não ser mais verdadeira** quando ela voltar a executar após ter sido "acordada" por um *notify* ou *notifyAll*. Explique sob a forma de comentário no código da classe *Consumidor* se isso pode causar algum problema. Caso afirmativo, resolva...

Parte II: O Problema dos Fumantes

O problema dos fumantes é um problema clássico de sincronização. Considere a situação com três fumantes (três threads) e um representante de cigarros (uma thread). Cada fumante continuamente faz os próprios cigarros e os fuma. Para fazer os cigarros, cada fumante precisa de três ingredientes: tabaco, papel e fósforos. Uma das thread de fumante tem somente papel, a outra tem somente tabaco e a terceira tem somente fósforos. O representante de cigarros tem um estoque infinito dos três materiais. As três threads fumantes estão inicialmente bloqueadas. O representante de cigarros escolhe randomicamente dois ingredientes diferentes, os coloca na mesa e desbloqueia o fumante que tiver o ingrediente restante. O representante então bloqueia. O fumante que está desbloqueado remove os dois ingredientes da mesa, faz o cigarro e o fuma por um tempo aleatório, desbloqueando o representante quando terminar de fumar o cigarro. O representante então coloca outros dois ingredientes na mesa, e o ciclo se repete.

Escreva um programa Java que usa um monitor para sincronizar as três threads dos fumantes e a thread do representante de cigarros. A thread do representante executa em um objeto instanciado de uma classe representante. Cada thread fumante executa em um objeto fumante. Todos os objetos fumantes são instanciados de uma classe fumante cujo construtor é usado para especificar os ingredientes que o fumante possui. Uma classe que chamaremos de "driver" implementa o método *main*, instancia os objetos e inicia as threads.

Para estabelecer a sincronização, use um único objeto monitor instanciado de uma classe *Controle*. Para manter a sincronização, as quatro threads invocam métodos `synchronized` do monitor. Não use semáforos nem blocos de sincronização (somente métodos `synchronized`). Não use *busy waiting* em sua solução.

O tempo (em segundos) que a simulação deve rodar e o tempo máximo em segundos que um fumante fuma o cigarro devem ser definidos como constantes no seu programa. O tempo máximo que um fumante pode fumar um cigarro é usado para delimitar valores aleatórios (e.x.: int fumando = 1 + (int) random(1000*maxFumando)).

BOM TRABALHO!