Threads e Semáforos em Java

Uma Revisão de Sistemas Operacionais

Threads

- Thread
  - É uma atividade, ou linha de execução

- Por que utilizar Threads?
  - Maior desempenho em ambientes multiprocessados;
  - Responsividade em interfaces gráficas;
  - Simplificação na modelagem de algumas aplicações.

Implementação

- Há duas formas de criarmos uma **thread** em Java:
  - Ou usamos **herança** e criamos uma classe que **estende** a classe **Thread**;
  - Ou criamos uma classe que **implementa** a **interface Runnable**

Implementação Usando Herança

Usando herança, nossa classe deve sobrescrever o método **public void run()**

```
public class Tarefa1 extends Thread {
    public void run() {
        for(int i=0; i<1000; i++) {
            System.out.println("Usando herança");
        }
    }
}
```

Implementando Usando Interface

A interface **Runnable** nos obriga a implementar o método **public void run()**

```
public class Tarefa2 implements Runnable {
    public void run() {
        for(int i=0; i<1000; i++){
            System.out.println("Usando Runnable");
        }
    }
}
```

Exemplo de Uso

Para usar as classes **Tarefa1** e **Tarefa2** devemos fazer:

```
public class Main {
    public static void main(String[] args) {
        Thread threadComHeranca = new Tarefa1();
        Thread threadComRunnable = new Thread(new Tarefa2());
        threadComHeranca.start();
        threadComRunnable.start();
    }
}
```

Saída do Exemplo

A saída de nosso exemplo é mais ou menos essa:

```
Usando Runnable
Usando Herança
Usando Herança
Usando Runnable
Usando Herança
Usando Runnable
(...)
```

Estados de uma Thread

Uma **thread** pode estar em um dos seguintes estados:

- criada
- em execução
- suspensa
- morta

Alguns Conceitos

- **Exclusão mútua**
  - uma thread está executando sozinha um determinado código, enquanto as outras esperam para poder executar
- **Sessão crítica**
  - parte do programa que é executada por somente uma thread de cada vez (em exclusão mútua)

Sincronização

Seja o exemplo abaixo:

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

Sincronização

Suponha que duas threads chamem depositar(100)

Thread 1	Thread 2	Saldo
s= getSaldo(); (s= 100)		100
	s= getSaldo() ; (s= 100)	
	s= s+ valor; (s= 200)	
s= s+ valor; (s= 200)		
	setSaldo(s);	200
setSaldo(s);		200

Sincronização

- “Condição de corrida”
  - quando duas ou mais threads modificam de forma intercalada dados de um mesmo objeto
- Condição indesejada
  - pode corromper o estado desse objeto
- Solução
  - sincronizar as modificações realizadas pelas threads
- Sincronização
  - criação de trechos de código que executam de forma atômica, isto é, não são intercalados com outros códigos
- Esses trechos são chamados de seções críticas

Primitivas de Sincronização

- Mutexes
- Semáforos
- Troca de mensagens
- Monitores (Java)

Sincronização

- Como criar seções críticas?
  - Por meio da palavra **synchronized**
- Os seguintes usos do comando synchronized são equivalentes:

```
synchronized public void teste(){
    façaAlgo();
}
```

```
public void teste() {
    synchronized(this) {
    façaAlgo();
    }
}
```

Utilizando synchronized

```
class ContaBancaria {
  private double saldo;  ........
  public synchronized doube depositar(double valor)
  { saldo= saldo + valor;  }
  public synchronized doube retirar(double valor)
  { saldo= saldo - valor; }
}
```

Método Sincronizados

- Todo objeto em Java possui um lock
  - Informa se o objeto foi ou não “travado” por alguma thread
- Seja a chamada **p.m()**: (m é **synchronized**)
  - Antes executar **m**, **thread** deve adquirir lock de **p** (i.e., travar p)
  - Se **p** já estiver travado, **thread** espera até ser destravado
  - Quando **thread** terminar execução de **m**, ela libera o lock de **p**
  - Durante execução de **m**, **thread** pode chamar outros métodos **synchronized** desse objeto (sem esperar)

Utilizando synchronized

```
class ContaBancaria {
  private double saldo;  ........
  public synchronized doube depositar(double valor)
  { saldo= saldo + valor;  }
  public synchronized doube retirar(double valor)
  { saldo= saldo - valor; }
}
```

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);	

Sincronização: wait() e notify()

- Como colocar uma thread para dormir: **por meio de wait()**
- Chamado por thread de posse do lock de um objeto **p**
- Libera-se o lock deste objeto
- Execução desta thread é suspensa até uma execução de **notify()** ou **notifyAll()** sobre **p**
- Ao ser “**acordada**”, thread volta a disputar o lock de **p**
    - Execução reinicia na linha seguinte ao **wait()**

Sincronização: wait() e notify()

- Como “acordar” uma thread? por meio de **notify()** ou **notifyAll()**
- **notify()**: chamado por thread de posse do lock de um objeto p
  - Acorda (arbitrariamente) uma das threads que estejam “dormindo” sobre este objeto
  - Thread acordada volta a disputar lock de p
- **notifyAll()**: acorda todas as threads que estejam dormindo sobre um determinado objeto

Semáforos

- Um semáforo é uma estrutura de dados que controle o acesso de aplicações aos recursos, baseando-se em um número inteiro, que representa a quantidade de acessos que podem ser feitos
- Assim utilizamos semáforos para controlar a quantidade de acesso a determinado recurso

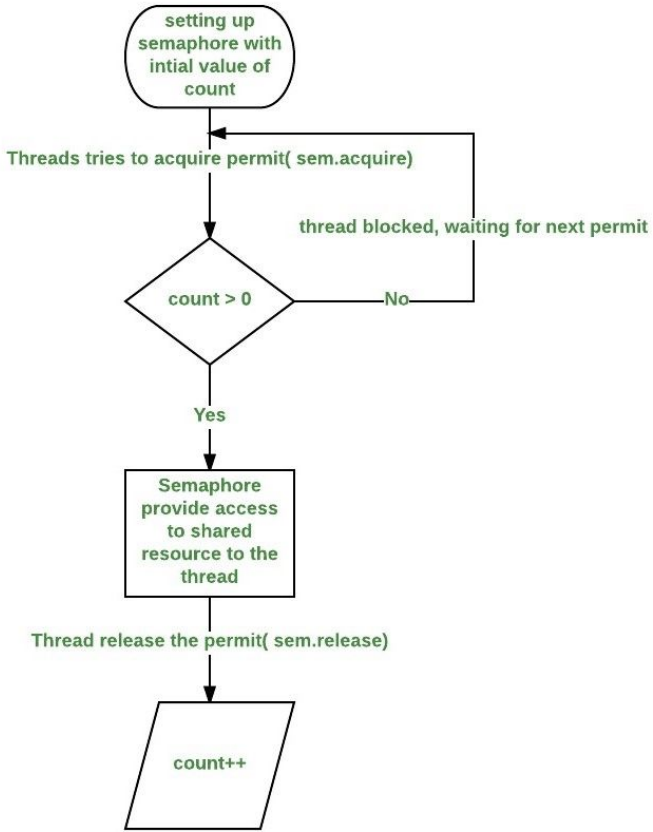
Semáforos

- Introduzidos por Dijkstra em 1968
- São inteiros não negativos com as operações:
- **P(s)**
  - se s for maior do que zero, s é decrementado em uma operação atômica ; caso contrário, atrasa o processo que a executa até que s seja maior que 0;
- **V(s)**
  - incrementa s em uma operação atômica.
- **P** e **V** são também chamadas **wait** e **signal** , ou **up** e **down** ;
- existem implementações tanto com espera-ocupada como com liberação do processador

Semáforos

- Semáforos que só assumem os valores 0 e 1 são conhecidos como semáforos binários
- Os outros são chamados de semáforos gerais, ou contadores
- Semáforo=0 -> recurso livre
  - Nenhum wake-up está armazenado
- Semáforo>0 -> recurso livre
  - Um ou mais wake-ups estão pendentes

Semáforos


setting up semaphore with intial value of count
Threads tries to acquire permit(sem.acquire)
thread blocked, waiting for next permit
count > 0
No
Yes
Semaphore provide access to shared resource to the thread
Thread release the permit(sem.release)
count++

Semáforos em Java

```
public class Semaphore {
  int value;
  public  Semaphore(int initialValue){
    value = initialValue;
  }
  public synchronized void P() {
    while (value <= 0 ) {
      try {
        wait();
      catch(InterruptedException e){}
    }
    value--;
  }
  public synchronized void V() {
    p++;
    notify();
  }
}
```

Utilizando Semáforos

Vamos definir uma implementação de Thread que vai utilizar o semáforo

```
public class ProcessadorThread extends Thread {
    private int idThread;
    private Semaphore semaforo;

    public ProcessadorThread(int id, Semaphore semaphore) {
        this.idThread = id;
        this.semaforo = semaphore;
    }
}
```

Definimos inicialmente um identificador para a nossa Thread e uma referência a um semáforo que irá controlar o acesso a essas variáveis.

Utilizando Semáforos

Agora vamos definir os métodos da nossa Thread, dentro da classe ProcessadorThread

```
private void processar() {
    try {
        System.out.println("Thread #" + idThread + " processando");
        Thread.sleep((long) (Math.random() * 10000));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Este método processar() apenas faz a thread dormir por algum tempo, simulando o efeito de um processamento longo

Utilizando Semáforos

Vamos criar um método que simula o acesso da Thread em uma região não crítica, ou seja, uma região ao qual não é necessário pedir uma trava

- Exibindo o atual estado da Thread, para facilitar o entendimento do progama, e realizamos um processamento qualquer

```
private void entrarRegiaoNaoCritica() {
    System.out.println("Thread #" + idThread + " em região não crítica");
    processar();
}
```

Utilizando Semáforos

Agora criamos um método que será utilizado para simular o acesso da Thread em uma região crítica

- Ele será chamado logo após conseguir a trava do semáforo

```
private void entrarRegiaoCritica() {
    System.out.println("Thread #" + idThread
            + " entrando em região crítica");
    processar();
    System.out.println("Thread #" + idThread + " saindo da região crítica"
}
```

Utilizando Semáforos

Como nossa classe extende o comportamento de uma Thread, nós sobrecarregamos o método run() que será chamado quando a Thread iniciar.

- Neste método nós realizamos um processamento não crítico, depois requisitamos o acesso ao semáforo (com o semaforo.acquire())
- em seguida realizamos o processamento de uma região crítica
- Por fim, liberamos o recurso do semáforo (com o semaforo.release()).

```
public void run() {
    entrarRegiaoNaoCritica();
    try {
        semaforo.acquire();
        entrarRegiaoCritica();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        semaforo.release();
    }
}
```

Entendendo um Pouco Mais Sobre Semáforos

- Todo semáforo deve possuir dois métodos: P e V
  - que têm sua origem das palavras **parsen** (passar) e e **vrygeren** (liberar)
  - Esta definição de semáforo foi proposta por Dijkstra para evitar o tão temido DeadLock.
- Quando se quer requisitar o recurso, faz-se uma chamada ao método P, que verifica se é possível liberar o recurso
- Ao terminar, faz-se uma chamada ao método V, que notifica as outras Thread que o recurso foi liberado
- Na implementação do Java, o método “acquire()” faz o papel do método P e o método “release()” faz o papel do método V

Entendendo um Pouco Mais Sobre Semáforos

- Para construir um semáforo precisamos informar o número máximo de Thread que podem acessar o recurso ao mesmo tempo
  - No nosso exemplo, apenas duas Thread poderão entrar na região crítica

```
public static void main(String[] args) {
    int numeroDePermicoes = 2;
    int numeroDeProcessos = 6;
    Semaphore semaphore = new Semaphore(numeroDePermicoes);
    ProcessadorThread[] processos = new ProcessadorThread[numeroDeProcessos];
    for (int i = 0; i < numeroDeProcessos; i++) {
        processos[i] = new ProcessadorThread(i, semaphore);
        processos[i].start();
    }
}
```

Entendendo um Pouco Mais Sobre Semáforos

Executando o programa a saída no console será algo do tipo:

```
Thread #1 em região não crítica
Thread #1 processando
Thread #0 em região não crítica
Thread #0 processando
Thread #5 em região não crítica
Thread #5 processando
Thread #3 em região não crítica
Thread #3 processando
Thread #2 em região não crítica
Thread #2 processando
Thread #4 em região não crítica
Thread #4 processando
Thread #0 entrando em região crítica
Thread #0 processando
```

```
Thread #4 entrando em região crítica
Thread #4 processando
Thread #4 saindo da região crítica
Thread #5 entrando em região crítica
Thread #5 processando
Thread #5 saindo da região crítica
Thread #3 entrando em região crítica
Thread #3 processando
Thread #0 saindo da região crítica
Thread #1 entrando em região crítica
Thread #1 processando
Thread #1 saindo da região crítica
Thread #2 entrando em região crítica
Thread #2 processando
Thread #3 saindo da região crítica
Thread #2 saindo da região crítica
```

Entendendo um Pouco Mais Sobre Semáforos

- Em Java os Thread são acordados aleatoriamente
  - então a saída não será a mesma
- O que é realmente importante notar é que nunca temos mais que duas Thread na região crítica
- A Thread 0 e 4 entram na área crítica e, somente quando a Thread 4 libera o recurso, a Thread 5 entra na região

```
Thread #1 em região não crítica
Thread #1 processando
Thread #0 em região não crítica
Thread #0 processando
Thread #5 em região não crítica
Thread #5 processando
Thread #3 em região não crítica
Thread #3 processando
Thread #2 em região não crítica
Thread #2 processando
Thread #4 em região não crítica
Thread #4 processando
Thread #0 entrando em região crítica
Thread #0 processando
```

```
Thread #4 entrando em região crítica
Thread #4 processando
Thread #4 saindo da região crítica
Thread #5 entrando em região crítica
Thread #5 processando
Thread #5 saindo da região crítica
Thread #3 entrando em região crítica
Thread #3 processando
Thread #0 saindo da região crítica
Thread #1 entrando em região crítica
Thread #1 processando
Thread #1 saindo da região crítica
Thread #2 entrando em região crítica
Thread #2 processando
Thread #3 saindo da região crítica
Thread #2 saindo da região crítica
```

Referências

- Introdução ao uso de Threads em Java. Daniel de Angelis Cordeiro. Disponível em: https://www.ime.usp.br/~gold/cursos/2004/mac438/aulaJava.pdf
- Java. Marco Túlio de Oliveira Valente. Disponível em: https://homepages.dcc.ufmg.br/~mtov/poo/01%20-%20Java.pdf
- Programando com Threads em C. Bruno Diniz de Paula. Disponível em: https://homepages.dcc.ufmg.br/~coutinho/pthreads/ProgramandoComThreads.pdf
- Praticando concorrência em Java! – Semáforos. Marcos Brizeno. Disponível em: https://brizeno.wordpress.com/2011/09/25/praticando-concorrencia-em-java-semaforos/
- Semáforos. Osvaldo Carvalho. Disponível em: https://homepages.dcc.ufmg.br/~vado/cursos/progpar972/curso/node103.html, https://homepages.dcc.ufmg.br/~vado/cursos/progpar972/curso/node104.html
- TANENBAUM, A. S. Sistemas operacionais modernos / Andrew S. Tanenbaum, Herbert Bos ; tradução: Daniel Vieira e Jorge Ritter ; revisão técnica: Prof. Dr. Raphael Y. de Camargo. [s. l.]: Pearson Education do Brasil, 2016. Disponível em: https://research.ebsco.com/linkprocessor/plink?id=392054ea-04eb-3160-a16b-ee7ab1e67278. Acesso em: 11 abr. 2024.