Sincronização

Aula 7

Introdução

- As threads não requerem acesso a recursos externos ou a chamadas de métodos de outros objetos
- Geralmente, threads devem compartilhar dados e são obrigadas a se preocupar com o que as outras estão fazendo, pois possuem área de dados em comum

Exemplo sem sincronização

- Veja o exemplo de um Produtor/Consumidor
- O produtor e consumidor compartilham dados de um mesmo objeto (UnsynchronizedBuffer)
- Quais problemas poderiam acontecer?

```
Consumer reads  2
Producer writes  3
Producer writes  4
Producer writes  5
Producer writes  6
Consumer reads  6
```

```
Producer writes  1
Consumer reads  1
Consumer reads  1
Producer writes  2
Consumer reads  2
```

Solução

- A solução é fazer a sincronização dentro do objeto compartilhado
- As atividades do produtor/consumidor devem ser sincronizadas de duas formas diferentes
- Primeiro
  - As threads não podem acessar simultaneamente o objeto compartilhado
  - A thread trava o objeto
  - Quando o objeto está travado e uma outra thread tentar chamar um método sincronizado no mesmo objeto, a thread ficará bloqueada até o objeto ser destravado

Solução

- Segundo
  - As threads devem coordenar o seu trabalho
  - O produtor deve ter uma forma de dizer ao consumidor que um novo número está disponível para consumo
  - O consumidor deve ter uma forma de dizer ao produtor que o número já foi consumido, liberando a produção de outro número

Sincronização

- Wait()
- Notify()
- NotifyAll()

Sincronização – Travando um Objeto

- Toda região crítica deve ser sincronizada
  - Pode ser um bloco de código ou um método inteiro
- Devem ser identificado com a palavra
  - **Syncronized**

Sincronização – Travando um Objeto

- Java suporta sincronização de threads através do uso de monitores
- A JVM fornece um lock para cada objeto e o objeto é travado
- No exemplo anterior os métodos set e get são as regiões críticas

Sincronização

- Observe que a Thread trava o objeto
- Se outra thread chamar outro método synchronized do mesmo objeto
- Ela vai esperar até que a primeira libere o lock
- A aquisição e liberação de lock é feito de forma atômica pela JVM

Sincronização

- Pode ser aplicado a um bloco

```
synchronized{
        qualquer conjunto de statement
}
```

- Synchronized sozinho é equivalente
  - synchronized(this);
- É possível adquirir o lock de outro objeto, não só do this

```
synchronized(obj) {
        qualquer conjunto de statement
}
```

Observações

- Só se pode usar wait(), notify() e notifyAll()
- Quando se está de posse do lock do objeto
- Wait() espera uma condição para o objeto corrente
- E está condição ocorre com notify() no mesmo objeto
- “Ter posse do lock” também se chama
  - “Possuir o monitor do objeto”

Sincronização

- Há três versões do método wait
- Wait()
  - Espera indefinidamente por uma condição
- Wait(long timeout)
  - Espera por uma notificação, mas, depois de timeout milissegundos, volta mesmo sem notificação
- Wait (long timeout, int nanos)
  - Idem o anterior, porém com mais tempo

Barreira de Sincronização

- Permite sincronizar várias threads em ponto específico do código
- Uma thread que alcança a barreira automaticamente wait()
- Quando a última thread alcança a barreira ela notifyAll() todas as outras threads

```
import java.util.*;
        class Barreira {
                private int ThreadsParticipante;
                private int EsperandoNaBarreira;
                public Barreira(int num) {
                        ThreadsParticipante = num;
                        EsperandoNaBarreira = 0; }
                public synchronized void Alcançado() {
                        EsperandoNaBarreira++;
                        If(ThreadsParticipante != EsperandoNaBarreira) {
                                try {
                                        wait();
                                catch (InterruptedException e) {}
                        } else {
                                notifyAll();
                                EsperandoNaBarreira=0; }
                }
         }
```

Exercício

- Refaça o exercício do cruzamento
- Porém agora deve haver sincronização entre os semáforos

Exercício – Leitor - Escritor

- Implementar um sistema Leitor/Escritor usando sincronização através de monitor e eventos
- Implementação
  - Existem 240 tarefas, sendo 120 de escrita e 120 de leitura, além de um único objeto buffer de tamanho 1 com dois métodos
  - class Buffer{
    - synchronized void Escrever (int i) {...}
    - synchronized int ler(){...}

Exercício – Leitor - Escritor

- As threads Escritor devem ser escalonadas usando ScheduleExecutorService, ou seja, todas as 120 tarefas devem ser controladas por esse escalonador. Esse escalonador deve liberar uma thread Escritor a cada 1 milissegundo (scheduleAtFixedRate) que irá escrever um valor inteiro incremental no Buffer
- As 120 restantes devem ser controladas por um ExecutorService usando um pool de tamanho fixo que deve permitir até 4 threads Leitor em estado de pronto durante a execução do código

Exercício – Leitor - Escritor

- Por fim, a primeira thread Leitor que conseguir ler o valor escrito, deve configurar a variável do Buffer para 0. Para que as outras threads Leitor não leiam 0, você deve utilizar monitor juntamente com os eventos de sincronização. Isto é, quando uma thread Leitor verificar que o buffer está vazio, este invoca o método wait(). E quando uma thread Escritor tiver escrito um dado, este invoca o método notifyAll()

Locks Explícitos e Variáveis Condicionais

- Para proteger regiões críticas
- É mais flexível do que usar a palavra synchronized
- Implementação a interface Look
  - Normalmente instancia-se ReentrantLock

```
private ReentrantLock lock;
public void methodA() {
  ...
  lock.lock();
  ...
}

public void methodB() {
  ...
  lock.unlock();
  ...
}
```

Lock

- Para obter o lock, usa-se o método lock()
- Para liberá-lo, usa-se unlock()
- Para esperar por um lock explícito, cria-se uma variável condicional
  - Interface Condition
  - Usar Lock.newCondition()

Lock

- await
  - Para esperar até a condição ser verdadeira
- signal e signalAll
  - Para avisar as threads que a condição ocorreu
- Variantes do método await

Métodos Condition.await	
Método	**Descrição**
`await`	Espera uma condição ocorrer
`awaitUninterruptibly`	Espera uma condição ocorrer. Não pode ser interrompido.
`awaitNanos(long timeout)`	Espera uma condição ocorrer. Espera no máximo timeout nanossegundos
`await(long timeout, TimeUnit unit)`	Espera uma condição ocorrer. Espera no máximo timeout TimeUnit
`await(Date timeout)`	Espera uma condição ocorrer. Espera no máximo até a data especificada

Lock

- O ReentrantLock fornece funcionalidade adicional sobre o uso de métodos e instruções sincronizados
- Fornece uma opção de justiça
- Fornece uma tentativa sem bloqueio de adquirir um bloqueio (tryLock()),
- Fornece uma tentativa de adquirir o bloqueio que pode ser interrompido (lockInterruptably(),)
- Fornece uma tentativa de adquirir o bloqueio que pode expirar (tryLock(long, TimeUnit)).

Semáforo

- Controla o acesso de aplicações a região crítica
- Baseia-se em número inteiros
- Todo semáforo deve possuir 2 métodos
- P (passar) e V(liberar)
- Foi proposto por Dijkstra para evitar o DeadLock

Semáforo

- Em Java há a classe Semaphore que realiza tal papel
- E os métodos
  - Acquire() – papel do P
  - Release() – papel do V

Exercício

- Arquivo do Word
- Lab-III-Trafego-Aereo

Exercício

- Implementar em Java um programa que faça o controle da entrada de pessoas em um banheiro unissex.
  - Suponha que em uma pequena empresa trabalham 20 pessoas - 10 homens e 10 mulheres. A sede desta empresa possui apenas um banheiro, no qual cabem no máximo três pessoas. De acordo com as regras estabelecidas pela empresa, o banheiro só pode ser ocupado por pessoas do mesmo sexo. Uma pessoa, ao tentar entrar no banheiro, ficará esperando se o banheiro estiver ocupado por pessoas do sexo oposto ao seu ou se ele estiver lotado. Sempre que chegar uma mulher na fila, ela terá preferência sobre os homens para entrar no banheiro, se somente se, o banheiro estiver vazio ou lotado por homens ou após X milissegundos, definido pelo programador

Exercício

  - Foram instalados dois leitores de cartão magnético na porta do banheiro - um na entrada e outro na saída - para controlar o acesso das pessoas. Para abrir a porta, o funcionário deve passar o seu crachá pelo leitor de cartão.
- Implementação
  - Você foi incumbido de desenvolver o sistema de controle de acesso ao banheiro. Implemente o sistema em Java usando um monitor. Para testar o sistema, crie 20 processos que representem os funcionários da empresa. Sempre que uma pessoa entrar no banheiro, imprima na tela o número de pessoas de cada sexo no banheiro e o seu nome.
- Podem utilizar PriorityQueue com a implementação da interface Comparable (quando feito internamente) ou Comparator (quando feito externamente). Reescrevam o método compareTo()