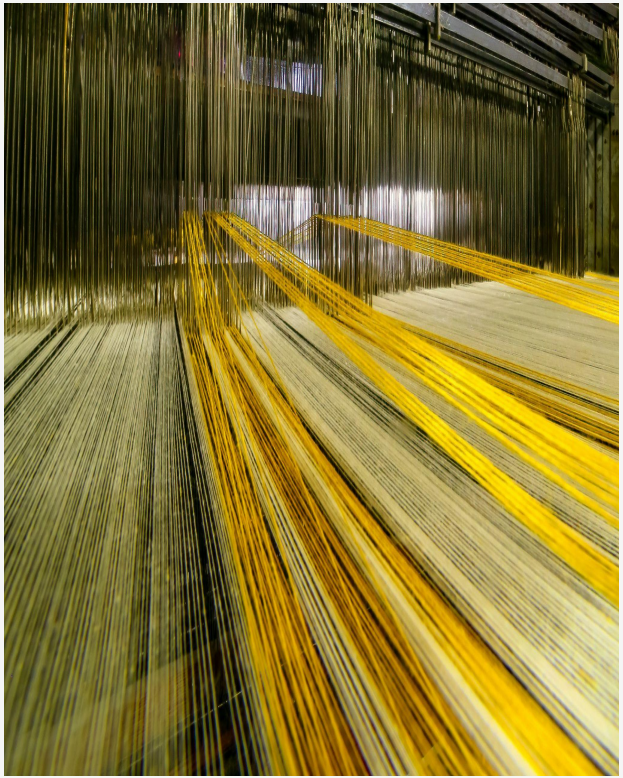


Photo by iggii on Unsplash

Programação concorrente e distribuída

Aula 6 - Locks

Prof. João Robson

Exclusão mútua

- Na aula passada, foi mostrado que para resolver o problema de *threads* ou processos acessando regiões críticas é necessário utilizar o conceito de **exclusão mútua** por meio de monitores. Hoje será mostrado como fazer o mesmo com ***locks*** (“travas”);

Lock

- Também chamados de *mutex*;
  - Forma muito comum de fazer exclusão mútua (**mut**ual **ex**clusion, em inglês);
- Para acessar uma seção crítica, uma *thread*:
  - Adquire a trava/mutex;
  - Acessa a seção crítica e
  - Libera o mutex no fim.
- Enquanto essa *thread* possui acesso exclusivo, todas as outras threads ficam em estado de bloqueio/espera até a liberação do mutex;
  - Ou seja, assim que uma *thread* sai da seção crítica, outra thread pode entrar.

1:1 ; igual ao monitor

Lock vs Monitores (*synchronized*)

- Se existe exclusão mútua por meio de monitores, qual a necessidade de *locks*?
  - *Locks* possuem algumas vantagens:
    - Permitem a implementação de códigos mais sofisticados e flexíveis;
    - Dão suporte a recursos inexistentes em monitores:
      - Bloqueio em um método e desbloqueio em outro;
      - Listagem de *thread* em estado de espera;
      - Modelo de bloqueio justo, garantindo que a *thread* que espera mais tempo obtenha o bloqueio primeiro, permitindo acesso justo ao recurso compartilhado;
      - Método *tryLock*, que permite uma *thread* tentar acessar e bloquear recurso se estiver disponível. Caso contrário, ele retornará falso, fazendo com que a *thread* não seja bloqueada, ou seja, continue sua execução.

Locks em Java - Interface *Lock*

- lock(): primitiva de bloqueio (bloqueia o acesso);
- unlock(): primitiva de desbloqueio (libera o acesso);
- tryLock(): Se o bloqueio não for mantido por nenhuma outra *thread*, a chamada para tryLock() retornará true e a "contagem de bloqueios" (*hold count*) será incrementada em um. Se o bloqueio não estiver livre, o método retornará falso e a *thread* não será bloqueado, mas sairá;
- getHoldCount(): retorna o número de threads que tentaram obter o lock;
- isHeldByCurrentThread(): retorna true se bloqueio é mantido pela *thread* atual;
- isLocked(): checa se *lock* está bloqueada por alguma *thread*;
- getQueueLength(): retorna o número de threads que aguardam pela liberação do *lock*;

Locks em Java - Classe *ReentrantLock*

- A classe *ReentrantLock* implementa a interface *Lock*, criando uma fila de *threads;*
- *Reentrant* -> mesma *thread* pode adquirir *lock* mais de uma vez;
- Construtor: ReentrantLock(boolean fair);
  - fair == true -> bloqueios favorecem a concessão de acesso ao thread de maior espera;
  - fair == false (padrão) -> bloqueio não garante nenhuma ordem de acesso específica, ou seja, *threads* são retiradas da fila sem ordem específica.
- **Observação:** Programas que usam bloqueios justos acessados por muitas *threads* podem exibir menor performance/maior lentidão do que aqueles que usam a configuração padrão, mas têm variações menores nos tempos para obter bloqueios e garantir **ausência de inanição**.

```
public class Classe {
    private Lock lock;

    public Classe {
        this.lock = new ReentrantLock();
    }

    public void metodo() {
        this.lock.lock();
        try {
            // Seção crítica...
        } finally {
            this.lock.unlock();
        }
    }
}
```

Locks em Java - *tryLock*

- Mecanismo usado para fazer com que *thread* só consiga adquirir *lock* se o mesmo estiver disponível e não sendo usado por nenhuma outra *thread*;
- Aceita parâmetro que representa *timeout*: *thread* aguarda determinada quantidade de tempo antes de desistir de adquirir *lock*

```
// ...
boolean isLockAcquired = false;
try {
    isLockAcquired = this.lock.tryLock(5,
TimeUnit.SECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();

}
if (isLockAcquired) {
    // seção crítica
    lock.unlock();
}
// ...
```

Locks em Java - Exercícios

- Implementar exemplo do contador usando *locks* no lugar de um monitor;
- Implementar exemplo do contador usando método *tryLock*, fazendo com que uma *thread* desista de acessar o *lock* após 5 segundos e mostre uma mensagem de erro caso não consiga.

Locks em Java - Interface *ReadWriteLock*

- Possui dois *locks*:
  - readLock(): acesso **compartilhado** com acesso de leitura.
    - **Observação:** o bloqueio de leitura pode ser mantido simultaneamente por vários threads, desde que não haja escrita.
  - writeLock(): acesso **exclusivo** com direito de escrita (modificação). Ou seja, se nenhuma *thread* estiver ~~lendo ou~~ gravando, então apenas uma *thread* pode adquirir o bloqueio de escrita.
- Implementada por *ReentrantReadWriteLock;*
- Por padrão, não garante a ordem de liberação nem preferência entre leitores e escritores;
- Ordenação FIFO é garantida passando **true** para o construtor.

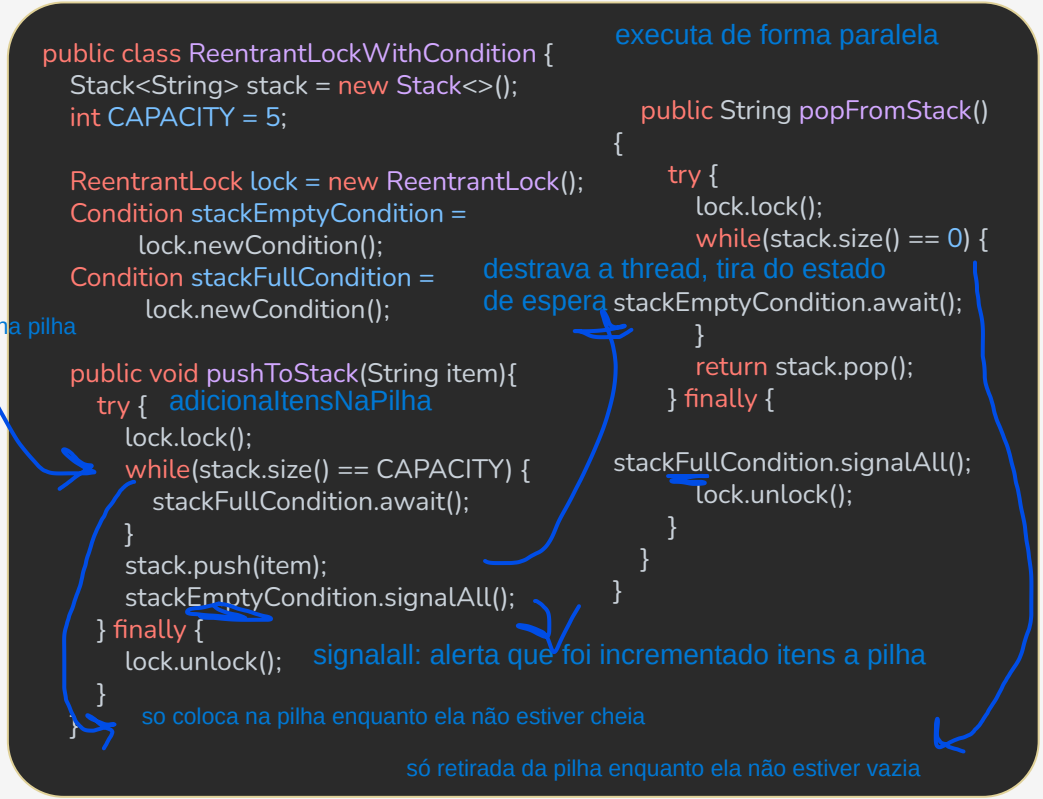
```
public class Classe {
   private ReadWriteLock lockRW;

   public Classe {
      this.lockRW = new
ReentrantReadWriteLock();
   }

   public void metodo() {
      this.lockRW.readLock().lock(); // OU
.writeLock()
      try {
         // Seção crítica...
      } finally {
         this.lockRW.readLock().unlock(); // OU
.writeLock()
      }
   }
}
```

Locks em Java - Classe *Condition*

- Fornece a capacidade de um thread esperar que alguma condição ocorra durante a execução da seção crítica;
- Pode ocorrer quando *thread* adquire acesso à seção crítica, mas não possui condições necessárias para realizar sua operação;
  - Exemplo: *thread* de leitura pode obter acesso ao *lock* de uma pilha compartilhada que ainda não possui dados.
- Tradicionalmente, Java fornece métodos wait(), notify() e notifyAll() para intercomunicação de *threads* (**vistos aula passada**).



Exercícios

- Implemente um código para simular o problema do **produtor-consumidor**:
  - Família de problemas criada descrita por Dijkstra desde 1965;
  - Um ou mais produtores criam produto e colocam em um *buffer*;
  - Um ou mais consumidores consomem produto do *buffer*;
  - O produtor precisa esperar por espaço no buffer para produzir;
  - O consumidor precisa esperar por produtos no buffer para consumir;
  - Dificuldade: sincronizar o acesso ao recurso.
- Desafios:
  - Adapte o código de forma que o consumidor some todos os valores que consumiu e mostre ao final o total;
  - Adapte o código para que o consumidor desista de tentar o lock do buffer se tiver mais de uma thread na fila. Mas a quantidade de consumos deve ser mantida.

Referências

- Guide to java.util.concurrent.Locks. Disponível em: https://www.baeldung.com/java-concurrent-locks
- FONSECA, Edson Francisco da. Memória Compartilhada – Lock e Semáforo.