Introdução à Programação Concorrente em Java

José Orlando Pereira Paulo Sérgio Almeida
Ana Nunes Alonso

29 de outubro de 2022

"Introdução à Programação Concorrente em Java"
Edição de 29 de outubro de 2022.
©2022 José Orlando Pereira, Paulo Sérgio Almeida, Ana Nunes Alonso

Este trabalho está licenciado sob a Licença Creative Commons Atribuição-NãoComercial-SemDerivações 4.0 Internacional. Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-nd/4.0/.

Prefácio

Motivação

A programação concorrente, em que diversas atividades ou fios de execução num mesmo programa colaboram na resolução de um problema comum, tem uma importância crescente no contexto de sistemas distribuídos e da exploração do paralelismo existente *em todos os sistemas de computação*.

A programação concorrente com memória partilhada é, no entanto, reconhecidamente difícil, uma vez que os programas podem apresentar comportamentos que violam a sua especificação sequencial e escapam à intuição do programador.

Objetivo

Este texto tem como objetivo dar a conhecer, de uma forma prática, os problemas fundamentais da programação concorrente num modelo de memória partilhada, bem como as soluções baseadas em primitivas de monitores – trincos e variáveis de condição – no contexto da linguagem Java. É dada ênfase a um conjunto de idiomas e padrões que permitem a construção de programas corretos e com algumas preocupações quanto ao seu desempenho.

Como tal, pode ser usado como suporte inicial para unidades curriculares introdutórias de Programação Concorrente, Sistemas Operativos ou de Sistemas Distribuídos ao nível do Primeiro Ciclo com uma duração aproximada de 2 ECTS (56 horas de estudo) e uma componente prática laboratorial significativa.

Estrutura

Este texto está dividido em duas partes distintas:

- Os Capítulos 1 a 3 introduzem os problemas e as soluções de forma direta e com pequenos exemplos.

- Os Capítulos 4 e 5 demonstram a utilização conjunta de várias técnicas na resolução de problemas concretos, partindo de problemas clássicos cujas soluções podem ser reutilizadas.

Em seguida

Após este texto introdutório, podem seguir-se dois percursos distintos. Um percurso possível segue a aplicação dos conhecimentos de programação concorrente num curso introdutório de sistemas distribuídos, em particular, na construção de sistemas cliente/servidor. A mais longo prazo neste percurso, é importante conhecer e tirar partido de utilitários de mais alto nível que evitam, em muitos casos, o recurso à programação utilizando diretamente as primitivas, e proporcionam soluções corretas e com bom desempenho [PGB+05].

A alternativa é aprofundar os fundamentos de programação concorrente, desde os critérios de correção e soluções clássicas para o problema da exclusão mútua à construção das próprias primitivas e utilitários [HS12]. Este percurso deve ser completado com a exploração de modelos alternativos de programação concorrente, nomeadamente com a eliminação de estado partilhado pela utilização de modelos de passagem de mensagens.

Qualquer um destes percursos pode ser usado para completar uma Unidade Curricular semestral típica de 5 ECTS. Na Universidade do Minho, correspondem respetivamente a Sistemas Distribuídos na Lic. em Eng. Informática e a Programação Concorrente na Lic. em Ciências da Computação.

Agradecimentos

Agradecemos a todos os que fizeram parte das equipas docentes de Sistemas Distribuídos e Programação Concorrente na Universidade do Minho: Carlos Baquero; João Paulo; Ricardo Macedo; e Francisco Neves.

Convenções tipográficas

Este texto inclui frequentemente fragmentos de código Java que são apresentados da seguinte forma:

```
3 public class Hello {
4     public static void main(String[] args) {
5         new Thread(()->{ ①
6             System.out.println("Hello world!"); // comentário
7         });
8     }
9 }
```

- As primitivas de programação concorrente são destacadas com uma cor diferente das restantes classes e métodos. Por exemplo, veja-se a distinção entre `Thread` e `Hello`.
- As referências a passos importantes no código são feitas com círculos numerados ①, que são comentados no texto seguinte.
- Os exemplos incompletos e que contêm erros estão assinalados com a imagem da bomba, na margem.
- A imagem do lápis, na margem, permite abrir o ficheiro completo que contém o fragmento num editor.

Definições e resumos

As definições de conceitos, regras e idiomas de programação principais são destacados desta forma.

Atenção!

Situações que dão origem a erros frequentes ou graves, bem como simplificações que são usadas para ilustrar um conceito *mas que não devem*

, *ser generalizadas*, são destacadas desta forma.

Conteúdo

Capítulo 1

Concorrência e problemas

1.1 Atividades concorrentes

Um programa em Java pode executar de forma simultânea e independente várias sequências de operações. Chamamos a estas sequências *atividades concorrentes* ou *fios de execução – threads* em inglês. Embora estas atividades possam tirar partido da existência de vários núcleos do processador, não são limitadas por estes, sendo que o sistema operativo troca rapidamente entre cada uma das atividades de forma a criar a ilusão de que todas executam ao mesmo tempo. Deste ponto de vista, não se distinguem de processos do sistema operativo, utilizados para poder executar várias aplicações ao mesmo tempo.

Além da atividade inicial que executa o método `main()`, uma aplicação pode lançar novas atividades criando objetos da classe `Thread` e fornecendo um método a executar sob a forma de uma instância da interface `Runnable`.

```
6     public static void main(String[] args) throws Exception {
7         var t = new Thread(()->{
8             try {
9                 while (true) {
10                    TimeUnit.SECONDS.sleep(1);
11                    System.out.print(".");
12                    System.out.flush();
13                }
14            } catch (InterruptedException ignored) {}
15        });
16        t.setDaemon(true);
17        t.start(); ①
18
19        System.out.println("Press␣RETURN␣to␣quit...");
20        System.in.read(); ②
21    }
```

Neste caso, a atividade inicial está dedicada à leitura do teclado ②. A atividade adicional imprime `"."` a cada segundo, depois de iniciada com a invocação de

`start()` ①. Neste caso, usamos `setDaemon(true)` para que a terminação do programa como um todo não dependa da terminação desta atividade adicional e, assim, que a terminação da atividade inicial provoque a terminação da atividade adicional.

Este é um exemplo da utilização de atividades concorrentes para atender mais do que uma entrada – o teclado e o relógio. Nestes casos, as vários atividades estão na maior parte do tempo inativas, trabalhando apenas quando chega um estímulo. Em sistemas distribuídos, este padrão é particularmente útil para atender canais de comunicação estabelecidos com várias outras máquinas ligadas em rede.

Podemos também usar várias atividades para tirar partido de processadores com vários núcleos, dividindo uma tarefa em partes e executando-as em paralelo.

```
16      public static void main(String[] args) throws Exception {
17          final int N = 10, M = 1000;
18          final var len = M/N;
19
20          var t = new Thread[N];
21          for(var i=0; i<N; i++) {
22              final var slice = i*len;
23              t[i] = new Thread(()->{
24                  subtask(slice, len);
25              });
26              t[i].start();
27          }
28
29          for(var i=0; i<N; i++)
30              t[i].join();
31      }
```

Neste exemplo, dividimos uma tarefa de tamanho `M` em `N` sub-tarefas de tamanho `length` que executamos em atividades concorrentes. O sistema operativo irá assegurar que essas atividades são atribuídas aos núcleos disponíveis. Neste caso, pretendemos esperar que todas estas atividades terminem, correspondendo à conclusão da tarefa como um todo. Para isso usamos o método `join()` que permite esperar pela conclusão de uma atividade. Neste caso, a atividade inicial ficará bloqueada sucessivamente na invocação do método `join()`, sendo libertada pela terminação de cada uma das atividades adicionais. É de notar que a ordem pela qual as atividades adicionais terminam não é pré-determinada, mas tal não é necessário para a utilização deste método para esperar pela conclusão das atividades adicionais.

Reutilização de um objecto *Thread*

A criação e inicio de novas atividades tem um custo não negligenciável quando usamos muitas atividades de curta duração. Nestes casos, é conveniente não criar um novo objeto `Thread` em cada um dos casos.

> Em vez disso, reutilizamos um objeto `Thread` para executar diferentes tarefas. Isto pode ser conseguido utilizando classes que implementam a interface `java.util.concurrent.Executor`.

1.2 Estado partilhado

A caraterística marcante deste modelo de programação concorrente é a possibilidade de partilha de estado – membros (ou seja, campos, ou variáveis de instância) de objetos – de forma simples, ao aceder às referências correspondentes. Torna-se assim possível comunicar dados entre atividades que colaboram numa mesma tarefa. Como exemplo, consideremos um objeto simples que armazena uma variável inteira `i` e oferece métodos para a manipular. Este é, pois, um objeto com estado mutável, que usamos então num programa concorrente, de várias formas, para ilustrar diferentes condições de acesso.

```
3  public class State {
4      private int i;
5
6      public void set(int i) {
7          this.i = i;
8      }
9
10     public int get() {
11         return i;
12     }
13 }
```

```
5      public static void main(String[] args) throws Exception {
6          final int N = 10;
7
8          var shared = new State(); ①
9
10         var t = new Thread[N];
11         for(var i=0; i<N; i++) {
12
13             var local = new State(); ②
14
15             t[i] = new Thread(()->{
16                 var v = shared.get(); ③
17                 local.set(v);
18                 shared.set(v+1);
19
20                 System.out.println(local.get());
21             });
22             t[i].start();
23         }
24
```

```
25          for(var i=0; i<N; i++)
26              t[i].join();
27      }
```

A utilização que fazemos desta classe que armazena estado mutável pode ser:

- Criamos um objeto único ①, cuja referência entregamos a cada uma das atividades. Ou seja, há mais do que uma atividade com acesso à mesma instância do objeto sempre que usem a variável `shared`. Este objeto tem por isso *estado partilhado*.
- Criamos um objeto ② cuja referência está disponível apenas para a atividade inicial (apenas durante a iteração do ciclo em que é criada) e para a atividade adicional instanciada nessa iteração do ciclo `for`. Neste caso, cada vez que uma atividade adicional usar a variável `local` estará a aceder uma instância própria. Este objeto tem por isso *estado local*.

Qualquer variável automática criada por invocações numa atividade, como acontece com a variável `v` ③, constitui também *estado local* dessa atividade.

Cada atividade adicional guarda na sua variável local `v` o valor de `shared` no momento em que executa; atualiza a sua variável `local` com o valor lido; incrementa o valor lido e atualiza o valor da variável partilhada. Como resultado da execução deste programa, cada atividade imprimirá o valor da sua variável `local`. Numa primeira abordagem, poderia esperar-se que como resultado se teria a impressão de uma sequência ordenada. Tal pode não acontecer dado que as atividades executam de forma concorrente, e assim a ordem pela qual executam (e imprimem) não é necessariamente a ordem pela qual as atividades foram iniciadas.

A linguagem Java não fornece mecanismos sintáticos ou em tempo de compilação para controlar esta partilha. Por um lado, isso torna fácil a utilização de classes existentes como parte do estado partilhado. Por outro lado, como irá ficar claro no resto deste capítulo, a partilha de estado entre atividades tem implicações importantes na correção do programa.[1] É por isso útil separar claramente o estado local do estado partilhado, devendo este último ser encapsulado, de forma a que a partilha possa ser controlada devidamente.

Atenção!

Convém lembrar que os membros `static` (variáveis de classe) são em Java partilhados entre todas as instâncias da classe. Isto significa que, sendo essas instâncias usadas por atividades diferentes, estes membros constituem assim *estado partilhado*. É pois (ainda mais) importante evitar o uso de `static` em programas concorrentes.

[1] De facto, o programa apresentado tem um erro que deverá ficar claro na Secção 1.3.

Variáveis *ThreadLocal*

A linguagem Java suporta ainda o conceito de variáveis locais a uma atividade acedidas através de uma referência partilhada com a classe `ThreadLocal`. Embora possa ser um recurso interessante, especialmente na conversão de código pré-existente que fazia uso de variáveis globais, não deve ser uma opção quando se desenha um sistema. De forma geral, a melhor estratégia é fazer chegar o estado local relevante ao código onde se pretende usar através da passagem explicita de referências.

1.3 Corridas na escrita

De forma a explorar os problemas que surgem ao utilizar estado mutável partilhado consideremos um contador que é incrementado por várias atividades.

```
3  public class Counter {
4      private long c = 0;
5
6      public void inc() {
7          c = c+1;
8      }
9
10     public long get() {
11         return c;
12     }
13 }
```

```
6      public static void main(String[] args) throws Exception {
7          var c = new Counter();
8
9          Thread[] t = new Thread[N];
10         for(var i=0; i<N; i++) {
11             t[i] = new Thread(()->{
12                 for(var j=0; j<M; j++)
13                     c.inc();
14             });
15         }
16
17         for(var i=0; i<N; i++)
18             t[i].start();
19         for(var i=0; i<N; i++)
20             t[i].join();
21
22         System.out.println(c.get()+" vs. "+(N*M));
23     }
```

Cada uma das `N` atividades irá incrementar este contador `M` vezes. Esperamos por isso que o valor obtido com `c.get()` e impresso na linha 22, depois de esperar que todas as atividades adicionais tenham terminado, seja igual a `M*N`. Como podemos facilmente observar ao executar este programa, isso pode não acontecer! De facto, de cada vez que o executamos podemos obter um valor diferente.

Como explicamos que isto aconteça? Lembrando que as variáveis são armazenadas em memória central mas que a adição é feita pelo processador, a operação da linha 7 implica os seguintes passos:

1. o valor de `c` é lido da memória e trazido para um registo no processador;
2. soma-se 1 ao registo;
3. o registo, já incrementado, é armazenado de volta na variável `c` na memória central.

Ao executar várias destas operações ao mesmo tempo em atividades concorrentes, é possível (provável!) que entre os passos 1 e 3 de uma atividade a haja outra atividade b (ou até várias) a efetuar todo o processo (várias vezes). Quando a atividade a executa então o passo 3, irá sobrescrever o valor escrito pela atividade b, perdendo-se assim o efeito das operações de b que lhe deram origem. Estes problemas podem ser resolvidos pela aplicação de primitivas de exclusão mútua descritas no Capítulo 2.

Corridas

Estas situações, em que o resultado de um programa concorrente depende de forma não determinística da velocidade relativa das atividades que o constituem, são erros e chamam-se *corridas*.

1.4 Corridas na leitura

Consideramos agora um outro caso, em que temos apenas uma atividade a escrever e, como tal, não surgem os problemas descritos na secção anterior. No entanto, temos agora duas variáveis distintas que são escritas por uma das atividades e lidas por outra.

```
3  public class TwoCounters {
4      private long c1 = 0, c2 = 0;
5
6      public void inc() {
7          c1 = c1+1;
8          c2 = c2+1;
9      }
10
```

```
11      public long get() {
12          var v2 = c2;
13          var v1 = c1;
14
15          return v1-v2;
16      }
17  }
```

Vemos que os dois contadores `c1` e `c2` são sempre incrementados em conjunto e que o contador `c1` é sempre incrementado primeiro. Concluímos daqui que, num programa sequencial, em que o método `inc()` é invocado apenas por uma atividade, `c1 >= c2`. Consideremos agora a possibilidade de ler estas variáveis a partir de outra atividade concorrente. Se fizermos a leitura de `c1` antes de `c2`, será possível que:

- `c1 == c2`, no caso em que cada uma das modificações correspondentes ao mesmo incremento é feita antes da leitura respetiva;
- `c1 > c2`, no caso em que as leituras são feitas entre as duas operações de modificação;
- `c1 < c2`, no caso em que uma ou mais invocações completas de `inc()` ocorrem depois de ler `c1` mas antes de ler `c2`.

Se invertermos a ordem das leituras, observando `c2` antes de `c1` como no método `get()`, esperamos obter:

- `c1 == c2`, no caso em que não há sobreposição da execução de `get()` com `inc()` em atividades diferentes;
- `c1 > c2`, no caso uma ou mais invocações, parciais ou completas, de `inc()` ocorrem entre as duas leituras feitas no `get()`.

No entanto, não esperamos observar `c1 < c2`, uma vez que sendo `c2` lido primeiro, a observação de um valor n em `c2` deveria implicar que `c1` terá também já sido incrementado pelo menos n vezes.

Podemos testar esta hipótese com um programa em que uma atividade invoca repetidamente o método `inc()`, para incrementar os dois contadores, enquanto outra atividade invoca também repetidamente o método `get()` para fazer a sua leitura, verificando se o resultado é não negativo.

```
6       public static void main(String[] args) throws Exception {
7           var c = new TwoCounters();
8
9           var t = new Thread(()->{
10              for(var j=0; j<M; j++)
11                  c.inc();
12          });
13          t.start();
14
```

```
15          var i=0;
16          for(var j=0; j<M; j++) {
17              if (c.get()>=0)
18                  i++;
19          }
20
21          t.join();
22
23          System.out.println(i+" vs. "+M);
24      }
```

Ao executarmos este programa podemos observar que, em alguns casos, a atualização das variáveis `c1` e `c2` por uma atividade pode ser observada pela ordem inversa (i.e. `c1 < c2`) a partir de outra atividade! Como estamos perante uma situação de corrida, dependendo do sistema onde se testa, poderá ser necessário repetir várias vezes a experiência e/ou aumentar `M` para que a situação ocorra.

Como explicamos que isto aconteça? De facto, há múltiplas causas, todas elas relacionadas com a complexidade dos sistemas atuais e a procura de maior desempenho. A ordem pela qual as operações têm efeito sobre as variáveis correspondentes em memória depende das decisões do compilador, da execução pelo processador, e da organização e política de escrita de cada um dos níveis de *cache*. Por exemplo, os valores escritos por `inc()` na memória *cache* de um dos núcleos do processador podem ser propagados para a memória central por ordem inversa, dando oportunidade a outro núcleo do processador de observar `c2 > c1`.

As técnicas descritas no Capítulo 2 servirão também para resolver estes problemas, sem que tenhamos que nos preocupar em fazer esta análise complexa para cada caso em particular.

Atenção!

A especificação de que comportamentos podem ser esperados quando atividades concorrentes escrevem e lêem variáveis em programas Java constitui o seu *modelo de memória*, cuja compreensão é necessária para desenvolver compiladores, a máquina virtual e primitivas para controlo de concorrência. Este é um tema de reconhecida complexidade[a] e é conveniente aderir a padrões de programação concorrente que evitem estes problemas de forma simples e geral.

[a] https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

1.5 Espera ativa

As corridas surgem quando temos estado partilhado, mesmo que as operações a efetuar sejam fundamentalmente as mesmas que em programas sequenciais.

Ou seja, quando uma atividade não tem em conta explicitamente o progresso de outras.

No entanto, há situações em que queremos que uma atividade suspenda a sua execução até que outra desencadeie um acontecimento específico. Ou então, que suspenda a sua execução até que um predicado avaliado sobre o estado atual seja verdade. Por exemplo, que um contador partilhado como o da Secção 1.3 atinja um determinado valor.

```
6     public static void main(String[] args) throws Exception {
7         var c = new Counter();
8
9         var t = new Thread(()->{
10            for(var j=0; j<M; j++)
11                c.inc();
12        });
13        t.start();
14
15        while (!(c.get() >= M/2))
16            ;
17
18        System.out.println("metade da tarefa");
19
20        t.join();
21    }
```

Esta solução sofre, em primeiro lugar, de todos os problemas descritos na secção anterior: se a condição depende de múltiplas variáveis, podemos observar valores paradoxais. Além disso, mesmo que na verificação da condição do ciclo se observe que uma determinada variável atingiu um certo valor, não é garantido que o código executado a seguir observe modificações a outras variáveis que até precedem as operações que tornaram a condição verdadeira.

Em segundo lugar, este código faz uma *espera ativa*, uma vez que a atividade principal nas linhas 15 a 16 não está a fazer trabalho útil mas mantém um núcleo do processador ocupado a recalcular repetidamente o valor da condição. Além de competir com atividades que estejam a fazer trabalho útil, na mesma ou em outra aplicação, tem como consequência o consumo de energia e a dissipação de calor. Estes fatores são críticos respetivamente em sistemas móveis, alimentados por bateria, e em sistemas de elevado desempenho, em que a manutenção de uma temperatura segura de funcionamento é muitas vezes o fator limitante. As primitivas e técnicas necessárias para resolver este problema são discutidas no Capítulo 3.

Capítulo 2

Exclusão mútua

2.1 Exemplo

Como exemplo motivador para apresentação das técnicas de programação concorrente usamos um jogo distribuído multi-utilizador, em que um programa servidor mantém a simulação e programas cliente fazem a interação com os jogadores.

Neste jogo, cada jogador tem associado um avatar com uma localização num espaço. Este avatar desloca-se de acordo com os comandos do jogador, mas também de acordo com leis da física da simulação. Cada jogador pode efetuar disparos em direção a outro, que resultam no incremento da sua pontuação e no decremento da saúde do alvo.

Assumimos assim que o programa servidor usa diversas atividades da seguinte forma:

- Uma primeira atividade por cada cliente, para atender as mensagens que trazem a informação relativa às ações do jogador;
- uma segunda atividade por cada cliente, que periodicamente obtém o estado de simulação e o transmite através do canal de comunicação ao jogador;
- atividades adicionais que movimentam os jogadores, por exemplo, periodicamente de acordo com um modelo de física.

Atenção!

A utilização de vários fios de execução para atender e representar um mesmo cliente num servidor não será a melhor opção, sobretudo, se pretendermos atingir um número elevado de jogadores ativos simultaneamente. De facto, nessa situação adequa-se uma estratégia de programação concorrente por eventos como demonstrada pelo *Eve Online*.[a] A

utilização deste exemplo neste texto destina-se apenas a dar semântica a cada um dos conceitos demonstrados para mais fácil compreensão.

[a] https://en.wikipedia.org/wiki/Eve_Online#Development

O estado partilhado entre estas atividades no servidor suporta esta funcionalidade e consiste num mapa que associa a cada nome de jogador um conjunto de informação: a localização no espaço, a saúde e a pontuação atuais.

```
10  private static class Player {
11      int x,y;
12      int health;
13
14      int score;
15  }
16
17  private SortedMap<String,Player> players;
```

Este estado é representativo do que temos em muitas aplicações. Temos elementos do estado de grão fino (neste caso, `Player`), não encapsulados, manipulados no contexto de objetos que os contêm. Podemos então considerar quais as consequências de consultar e modificar esta informação no sentido de realizar as funcionalidades pretendidas para o jogo.

2.2 Corridas e trincos

Começamos por considerar um método para mover o avatar associado a um jogador, modificando as suas coordenadas.

```
19  public void move(String name, int dx, int dy) {
20      var player = players.get(name);
21
22      player.x += dx;
23      player.y += dy;
24  }
```

De acordo com o que vimos no Capítulo 1, podemos esperar dois tipos de problemas:

- Corridas quando temos múltiplas atividades a executar este método, caso em que alguns incrementos às coordenadas podem ser perdidos. Por exemplo, começando com $(1, 1)$ e tentando concorrentemente somar $(+1, +1)$ e $(+1, -1)$ podemos obter no fim $(3, 0)$.
- Corridas quando temos uma atividade a ler as coordenadas concorrentemente com uma alteração. Por exemplo, ao mover de $(1, 1)$ para $(2, 2)$ seria possível observar as coordenadas $(2, 1)$ ou mesmo $(1, 2)$.

Secção crítica

Chamamos *secção crítica* a uma parte do programa que, quando executada concorrentemente em mais do que uma atividade dá origem a corridas. Uma secção crítica pode não ser contígua, ou seja, estar dividida por várias funções ou métodos.

A solução passa pois por delimitar, no programa, as secções críticas e evitar que nelas se encontrem múltiplas atividades. Para o garantir, usamos uma primitiva de exclusão mútua conhecida como *trinco* (*lock* em inglês) devido à seguinte analogia:

- Quando a secção crítica está vazia, uma atividade pode entrar e fechar a porta com o trinco, impedindo outras de entrar. Essas outras atividades ficam em espera do lado de fora da secção crítica.
- Ao sair da secção crítica ocupada, uma atividade volta a abrir o trinco, dando a vez a uma atividade seguinte.

Isto garante que não há corridas de escrita. Além disso, um trinco garante também que uma atividade que feche o trinco irá observar todas as escritas feitas pela atividade anterior, antes de abrir o mesmo trinco. Isto garante que também não há corridas de leitura.

Em Java obtemos um trinco instanciando a classe `ReentrantLock`, tipicamente com o mesmo âmbito (i.e., membro do mesmo objeto) que os dados cuja manipulação justifica a secção crítica. No nosso exemplo, acrescentamos um trinco à classe do jogo.

```
19      private SortedMap<String,Player> players;
20      private Lock l = new ReentrantLock();
```

E usamos então os métodos do trinco para assinalar a entrada e saída da secção crítica.

```
22      public void move(String name, int dx, int dy) {
23          try {
24              l.lock(); ①
25
26              var player = players.get(name);
27
28              player.x += dx;
29              player.y += dy;
30          } finally {
31              l.unlock();  ②
32          }
33      }
```

- A invocação do método `lock()` ① corresponde à entrada na secção crítica com o fecho do trinco. Outras atividades que tentem fazer o mesmo, aqui ou noutra parte do código que use o mesmo trinco `l`, irão esperar.

- A invocação do método `unlock()` ② corresponde à saída da secção crítica com a abertura do trinco. Uma das atividades que esteja em espera da abertura do mesmo trinco `l` poderá então fazer progresso.

Isto significa que as linhas 24 a 31 fazem parte da secção crítica associada ao trinco `l`.

A utilização do par `try/finally` em conjunto com o trinco tem como objetivo garantir que a operação de `unlock()` é sempre efetuada, mesmo que a saída da secção crítica seja feita pelo lançamento de uma excepção. Embora não seja obrigatório, este idioma é bastante conveniente e evita esquecimentos.

2.3 Estado imutável

A utilização de trincos em programas concorrentes é necessária para evitar situações de corridas com estado partilhado. No entanto, tem também um impacto relevante no desempenho e escalabilidade desses programas.

Por um lado, a existência de trincos limita o desempenho quando há contenção. Por exemplo, admitindo que cada uma das atividades num programa precisa de passar uma parte r do seu tempo dentro de uma mesma secção crítica, o programa poderá escalar no máximo até $1/r$ atividades concorrentes. De facto, o impacto no desempenho será visível antes disso, na medida em que, mesmo que a secção crítica não esteja ocupada o tempo todo, haverá contenção e espera para o trinco correspondente. Devemos pois tentar minimizar o tempo que cada atividade passa dentro de uma secção crítica.

Consideremos, no nosso exemplo, um método que recolhe as coordenadas de todos os jogadores para que sejam enviadas ao programa cliente, que as utiliza para desenhar o estado atual no ecrã.

```
51    public void draw(DataOutputStream stream) throws IOException {
52        try {
53            l.lock();
54
55            for(var player: players.values()) {
56                stream.writeInt(player.x);
57                stream.writeInt(player.y);
58            }
59        } finally {
60            l.unlock();
61        }
62    }
```

Este código faz a invocação do método `writeInt()` no objeto que é recebido como parâmetro. O implementador desta classe, admitindo que ela poderá ser usada por terceiros, desconhece qual a implementação concreta da interface `DataOutputStream` e é concebível que possa dar origem a operações de entrada/saída, para ficheiros ou para comunicação em rede. Se assim for, estas operações podem bloquear temporariamente. Como são feitas no contexto de

uma secção crítica associada ao trinco `l`, irão impedir quaisquer outras atividades de o utilizarem durante esse tempo, levando a contenção e limitando o desempenho e escalabilidade.

É pois conveniente robustecer a implementação de uma classe às utilizações que possam dela ser feitas. Aqui, a solução passa por evitar a invocação de métodos externos, que não controlamos no contexto do trinco, mas sem comprometer a correção. Uma possibilidade seria efetuar uma cópia dos dados em questão para uma estrutura temporária na secção crítica, usando então essa cópia, já com o trinco aberto, para invocar o `writeInt()`.

De facto, podemos reduzir ou mesmo eliminar a sobrecarga associada a essa cópia se usarmos objetos imutáveis manipulados de forma funcional. Aplicamos esta estratégia isolando as coordenadas num objeto separado.

```
14      private static final class Coord {
15          private final int x, y; (1)
16
17          public Coord(int x, int y) {
18              this.x = x;
19              this.y = y;
20          }
21      }
22
23      private static class Player {
24          Coord xy;
25          int health;
26
27          int score;
28      }
```

Neste caso, todos os campos da classe `Coord` estão marcados como `final` (1) pelo que são imutáveis e não dão origem a corridas.

Records

Em Java 16 ou posterior é possivel usar a palavra-chave `record` para declarar uma classe imutável e não encapsulada como `Coord` com uma sintaxe concisa:

```
    private record Coord(int x, int y) {}
```

Esta alternativa tem ainda a vantagem de produzir métodos `equals()`, `hashCode()` e `toString()` com o comportamento adequado.

Reescrevemos então o código do método `move()` de forma a criar uma nova instância ds classe `Coord` de cada vez que modificamos as coordenadas associadas ao jogador (2), em vez de alterar o valor dos campos `x` e `y`.

```
33      public void move(String name, int dx, int dy) {
34          try {
```

```
35              l.lock();
36
37              var player = players.get(name);
38              player.xy = new Coord(player.xy.x + dx, player.xy.y+dy); ②
39          } finally {
40              l.unlock();
41          }
42      }
```

Note-se que a alteração das coordenadas ② continua dentro da secção crítica, pois continuamos sujeitos a corridas de escrita. No entanto, deixamos de estar sujeitos a corridas de leitura, uma vez que a visibilidade por outras atividades depende apenas a alteração da variável `player.xy` e que o Java garante que o resultado de um construtor é visto atomicamente.

Modificamos então o métdo `draw()` para evitar a chamada de métodos externos a partir da nossa secção crítica.

```
60      public void draw(DataOutputStream stream) throws IOException {
61          List<Coord> coords;
62
63          try {
64              l.lock();
65              coords = players.values()
66                  .stream().map(p -> p.xy)
67                  .collect(Collectors.toList());  ③
68          } finally {
69              l.unlock();
70          }
71
72          for(var coord: coords) {  ④
73              stream.writeInt(coord.x);
74              stream.writeInt(coord.y);
75          }
76
77      }
```

Começamos por, dentro da secção crítica, recolher os dados relevantes numa estrutura temporária ③. Podemos então, já fora da secção crítica, percorrer esses dados ④ e invocar os métodos externos. De facto, realizamos algum trabalho extra ao criar a lista temporária `coords`, mas tornamos o nosso código mais robusto, especialmente se estamos a fazer uma biblioteca que vai ser utilizada por terceiros.

Note-se que não podíamos obter o mesmo resultado recolhendo os objetos `Player`, porque estes são mutáveis e não os poderiamos usar fora da secção crítica.

Atenção!

Pela mesma razão, é preciso evitar a fuga de referências para estado mutável interno a um objeto encapsulado. Por exemplo, a consulta da coleção de nomes de jogadores feita desta forma irá entregar para fora da secção crítica uma referência indireta ao mapa `players`:

```
public Collection<Player> getNames() {
    try {
        l.lock();
        return players.keys(); // referência para estado mutável
    } finally {
        l.unlock();
    }
}
```

Uma vez que o código externo ao objeto não tem acesso ao trinco `l` para garantir que o mapa `players` não está a ser modificado, isto resultará certamente numa situação de corrida. Esta é mais uma razão para usar estado imutável.

2.4 Impasses e ordenação

Uma solução alternativa, quando a contenção num trinco se revela problemática para o desempenho e escalabilidade de um programa, consiste na partição do estado mutável em questão e na utilização de vários trincos, uma para cada parte.

```
23  private static class Player {
24      Coord xy;
25      int health;
26      int score;
27
28      private Lock l = new ReentrantLock();
29  }
```

No nosso exemplo passamos a usar um trinco por cada jogador e eliminamos o trinco global a todo o jogo. Assumimos, por agora, que o mapa `players` é imutável durante cada jogo e por isso não dá origem a corridas. Temos assim duas possibilidades:

- Quando a secção crítica usa apenas uma das partições resultantes, basta fechar o trinco correspondente. Outras atividades sobre outras partições podem prosseguir concorrentemente. No nosso exemplo, isso acontece com o método `move()`, que poderá fechar apenas o trinco correspondente.
- Caso contrário, quando a operação em causa envolve mais do que uma partição do estado, será necessário fechar todos os trincos relevantes. No

nosso exemplo, isso acontece com o método `shoot()`, que necessitará de fechar os trincos de ambos os jogadores envolvidos.

Na escolha da granularidade com que fazemos a partição do estado é preciso ter em consideração que cada operação com um trinco tem um custo em si mesmo. Uma partição que resulte em centenas ou milhares de trincos a ser adquiridos por uma única operação, irá pois representar um custo significativo, em termos de desempenho, e deve ser evitada.

Além disso, a utilização em simultâneo de vários trincos aumenta a complexidade do código e dá origem a novos problemas. Consideremos pois a implementação do método `shoot()`.

```
43    public void shoot(String sname, String tname) {
44        var splayer = players.get(sname);
45        var tplayer = players.get(tname);
46        try {
47            splayer.l.lock();
48            tplayer.l.lock();
49
50            if (tplayer.health > 0) {
51                splayer.score += 1;
52                tplayer.health -= 1;
53            }
54        } finally {
55            splayer.l.unlock();
56            tplayer.l.unlock();
57        }
58    }
```

Neste exemplo, a secção crítica nas linhas 50 a 53 é executada com ambos os trincos correpondentes a `splayer` e `tplayer` fechados, pelo que outras atividades que manipulem os mesmo objetos não darão origem a corridas.

Consideremos, no entanto, a seguinte sequência de acontecimentos em duas atividades em que t_1 executa `shoot(a,b)` e t_2 executa `shoot(b,a)`:

- t_1 fecha o trinco correspondente ao jogador a;
- t_2 fecha o trinco correspondente ao jogador b;
- t_1 tenta tenta mas não consegue o trinco correspondente a b (porque está fechado por t_2) e fica bloqueado à espera;
- t_2 tenta tenta mas não consegue o trinco correspondente a a (porque está fechado por t_1) e fica bloqueado à espera;

Em resumo, temos t_1 e t_2 à espera um do outro e ambos impossibilitados de fazer progresso!

Impasses

Chamamos *impasse* (ou *deadlock* em inglês) a esta situação em que várias atividades esperam umas pelas outras, formando um ciclo. Um impasse constitui um erro num programa concorrente e não pode ser resolvido facilmente, pois isso implicaria a terminação de pelo menos uma das atividades.

Atenção!

Neste caso é fácil de ver. Mas quando há invocações a métodos externos, pode ser mais complicado de detectar a possibilidade de impasses. Por isso é bom minimizar o que se chama dentro de uma secção crítica, por exemplo, com a técnica da secção anterior.

Como não podemos resolver uma situação de impasse depois de já ter acontecido, resta-nos preveni-la. Uma vez que um impasse surge quando uma atividade tenta obter um par de trincos pela ordem contrária a que outra atividade (ou conjunto de atividades) está também a tentar adquiri-los, a solução consiste em garantir que os trincos que podem ser obtidos ao mesmo tempo o são por uma ordem pré-definida, respeitada por todas as atividades.

```
43    public void shoot(String sname, String tname) {
44        var splayer = players.get(sname);
45        var tplayer = players.get(tname);
46        try {
47            if (sname.compareTo(tname)<0) {
48                splayer.l.lock(); tplayer.l.lock();
49            } else {
50                tplayer.l.lock(); splayer.l.lock();
51            }
52
53            if (tplayer.health > 0) {
54                splayer.score += 1;
55                tplayer.health -= 1;
56            }
57        } finally {
58            splayer.l.unlock();
59            tplayer.l.unlock();
60        }
61    }
```

No nosso exemplo, usamos a ordem dada pelos nomes dos jogadores. Esta solução é conveniente porque é a mesma ordem que usamos já no mapa `players`, cuja ordenação é usada noutros métodos. A abertura dos trincos pode no entanto ser feita por qualquer ordem.

Atenção!

Convém considerar se é mesmo necessário adquirir os dois trincos em simultâneo. Neste caso, bastaria trocar a ordem das duas linhas e admitir que podíamos ver o `health` decrementado sem o `score` correspondente. Além de evitar completamente os impasses, evitava também estar temporariamente à espera do segundo trinco tendo já o primeiro adquirido.

2.5 Trincos em duas fases

A partição do estado mutável e a utilização de vários trincos tem como objetivo reduzir a contenção, diminuindo a probabilidade de que várias atividades necessitem simultaneamente do mesmo trinco. No entanto, há situações em que é necessário fechar vários trincos. No nosso exemplo, consideremos a operação em que o jogador com o maior valor na variável `health` é premiado com pontuação adicional.

```
63      private void bonus() {
64          try {
65              for(var p: players.values()) p.l.lock(); ①
66
67              Player best = null;
68              for(var p: players.values())
69                  if (best == null || p.health > best.health)  ②
70                      best = p;
71
72              best.score += 1;
73          } finally {
74              for(var p: players.values()) p.l.unlock();  ③
75          }
76      }
```

Esta solução começa por fechar os trincos associados a todos os objetos que pretendemos consultar ou modificar ①. Em seguida, observa o estado de todos os objetos e modifica aquele que é escolhido como sendo o melhor ②. Finalmente, abre todos os trincos que tinham sido adquiridos ③.

Esta solução está correta, na medida em que não está sujeita a corridas, pois nenhuma outra atividade pode observar ou modificar o estado mutável envolvido na operação durante todo o seu decurso. De facto, é equivalente à utilização de um trinco global a todo o jogo que fosse fechado para realizar a operação.

Note-se que uma solução que feche apenas um trinco de cada vez pode levar a que se observe a seguinte sequência de acontecimentos, em que os jogadores a, b e c começam respetivamente com a variável `health` a 8, 9 e 10:

- é observado o jogador a que tem `health` igual a 8;
- o jogador a modifica `health` para 10;
- é observado o jogador b que tem `health` igual a 9, sendo até agora o maior observado;
- o jogador c modifica `health` para 8.
- é observado o jogador c que tem `health` igual a 8;
- finalmente, é modificado o `score` do jogador b.

Em resumo, sem que em nenhum momento o jogador b fosse aquele com a variável `health` máxima, é essa a conclusão tirada da observação.

No entanto, podemos facilmente verificar que a solução proposta, é suficiente mas não necessária para evitar esta corrida. Por exemplo, poderíamos abrir todos os trincos excepto o de `best` na linha 71, deixando apenas esse para a linha 74, sem que isso fizesse qualquer diferença nas combinações de valores que podem ser observadas por outras atividades. Convém no entanto evitar que seja necessário determinar a ordem correta particular para cada situação, já que obriga a uma análise complexa.

Trincos em duas fases

A regra dos trincos em duas fases (*two phase locking* ou *2PL* em inglês) indica que uma operação sobre várias variáveis precedida do fecho de todos os trincos associados e seguida da sua abertura é equivalente a garantir apenas que:

1. Cada item de dados é acedido com o trinco correspondente fechado.
2. Todas as operações de fecho de trincos (`lock()`) precedem todas as operações de abertura (`unlock()`).

A primeira fase corresponde ao aumento do número de trincos que está fechado, podendo já incluir operações sobre os dados associados a esses trincos. Há então um momento único, em que todos os trincos necessários até ao fim da operação estão já adquiridos e em que pode haver operações sobre qualquer item de dados. Numa segunda fase, os trincos vão sendo libertados, podendo ainda haver em cada momento operações sobre os dados associados aos restantes. Podemos então aplicar esta regra ao método `bonus()` no nosso exemplo.

```
29      private void bonus() {
30          Player best = null;
31          for(var p: players.values()) {  ④
32              p.l.lock();
33              if (best == null || p.health > best.health)
```

```
34                  best = p;
35          }
36
37          for(var p: players.values()) {    ⑤
38              if (p != best)
39                  p.l.unlock();
40          }
41          best.score += 1;
42          best.l.unlock();
43      }
```

Nesta implementação alternativa, na primeira fase ④ é fechado cada um dos trincos e feita a operação de consulta do estado correspondente. Tentamos pois adiar o fecho de cada trinco até ser estritamente necessário.

Na segunda fase ⑤, como não necessitamos de fechar mais trincos, pela segunda cláusula da regra, podemos começar a abrir aqueles de que já não precisamos. Atrasamos apenas a abertura daquele que precisamos para cumprir a primeira cláusula da regra na linha 41.

Finalmente, repare-se que, ao usar esta regra, acontece frequentemente que temos as operações misturadas com a abertura e fecho de trincos, que não são feitas por ordem estritamente inversa. Torna-se assim mais difícil usar o idioma comum com `try/finally` para garantir de forma genérica que todos os trincos são abertos em caso de excepção. Somos assim obrigados a fazer uma análise mais cuidado do código para descobrir exatamente onde pode haver excepções. Neste exemplo, as operações em causa não podem lançar excepções – tendo o cuidado de garantir que `best != null`.

Capítulo 3

Espera por eventos

3.1 Exemplo

Retomando o exemplo do jogo, em que o servidor é implementado usando múltiplas atividades, assumimos agora que cada partida do jogo admite um número máximo de jogadores simultâneos `max`. Isto significa que, conforme os jogadores vão chegando, havendo mais do que `max` jogadores ativos, alguns terão que esperar pela sua vez, ou seja, até que outros abandonem o jogo.

Mantemos os restantes pressupostos da Secção 2.1 e tomamos como ponto de partida a implementação da Secção 2.3 em que utilizamos um único trinco para cada jogo, protegendo o acesso ao mapa dos jogadores bem como ao estado de cada jogador.

3.2 Variáveis de condição

Começamos por considerar métodos para um jogador entrar, acrescentando a sua informação ao jogo e esperando até que haja condições para o jogo começar, e para sair, eliminando a sua informação.

```
31  public void enter(String name) {
32      while (players.size() >= max)
33          ;
34      players.put(name, new Player());
35  }
36
37  public void leave(String name) {
38      players.remove(name);
39  }
```



De acordo com o que vimos nos capítulos anteriores, podemos esperar dois tipos de problemas:

- Corridas, quando temos várias atividades a executar estes métodos, uma vez que ambos os métodos modificam a estrutura de dados `players`.
- O método `enter()` faz uma espera ativa, consumindo recursos enquanto a condição se mantiver verdadeira.

Começamos então por tentar resolver as corridas aplicando trincos, tal como feito no capítulo anterior, para os restantes métodos.

```
31  public void enter(String name) {
32      l.unlock();
33      while (players.size() >= max)
34          ;
35      players.put(name, new Player());
36      l.unlock();
37  }
38
39  public void leave(String name) {
40      l.unlock();
41      players.remove(name);
42      l.unlock();
43  }
```

Esta solução resolve o nosso primeiro problema, uma vez que a manipulação e observação do estado partilhado, que é o mapa `players`, passa a ser feito com o trinco fechado, garantindo que o é por uma atividade de cada vez.

Curiosamente, ao resolver o primeiro problema tornamos o segundo mais complicado. Neste momento, uma atividade que entre no corpo do ciclo, ou seja, enquanto não tem lugar para jogar, ficará à espera com o trinco fechado. Isto significa que será impossível a outras atividades modificar `players`, nomeadamente para efetuar a saída de jogadores, pois para isso precisariam de encontrar o trinco aberto para executar o método `leave()`. Temos assim uma nova situação de impasse, em que uma atividade espera indefinidamente por algo que não pode acontecer, pois depende de um trinco que a própria atividade mantém fechado. Concluímos pois que durante a espera será necessário abrir o trinco que protege o estado necessário para avaliar a condição.

Ao mesmo tempo, para evitar a espera ativa vamos precisar de uma primitiva que permita suspender atividades, de forma a que não consumam recursos, e retomá-las quando a condição pela qual cada uma espera se torna verdadeira. Passamos assim de uma lógica de testar continuamente a condição à espera de observar uma alteração do seu valor para passar a uma lógica de notificação caso a veracidade da condição se possa ter alterado.

De forma a cumprir estes dois requisitos precisamos de algo que corresponda a este pseudo-código:

```
public void enter(String name) {
    l.unlock();
    if (players.size() >= max) {
        l.unlock();
        await event; // suspende atividade
```

```
            l.lock();
        }
        players.put(name, new Player());
        l.unlock();
    }

    public void leave(String name) {
        l.unlock();
        players.remove(name);
        signal event; // acorda atividades suspensas
        l.unlock();
    }
```

Precisamos pois de uma primitiva que nos permita suspender a atividade à espera de um evento, o que deve ser feito quando a observação do estado mostra que esse evento ainda não ocorreu, e com o trinco correspondente aberto. De forma a justificar o funcionamento e utilização dessa primitiva, consideremos uma execução possível de duas atividades concorrentes, numa situação em que `players.size() == max`:

- a atividade a invoca o método `enter()`, fecha o trinco e verifica que a condição é falsa pois `players.size() == max`;
- entretanto a atividade b invoca `leave()` e fica à espera do trinco;
- a atividade a abre o trinco antes de começar a espera;
- neste momento, o trinco é fechado pela atividade b, que modifica `players` e tenta avisar atividades suspensas, que neste momento ainda não existem;
- a atividade a prossegue então para a espera e fica suspensa.

Neste cenário, é possível que a fique suspensa indefinidamente, apesar de `players.size() < max` uma vez que temos uma situação de corrida entre a suspensão e aviso de uma atividade. Isto significa que a primitiva que suspende uma atividade terá que o fazer atomicamente com a abertura do trinco correspondente.

Consideremos ainda outro cenário, desta vez com três atividades concorrentes, também numa situação em que `players.size() == max` e em que a atividade a está já suspensa:

- entretanto, a atividade b invoca `leave()`, fecha o trinco, modifica `players` e avisa atividades suspensas, que neste momento é apenas a;
- a atividade a ao acordar tenta fechar de novo o trinco;
- ao mesmo tempo, uma atividade c invoca `enter()` e tenta também fechar o trinco, pelo que estão duas atividades a competir por `l`;
- a atividade c ganha, fecha o trinco, verifica que a condição é falsa, insere um novo elemento em `players`, fazendo com que `players.size() == max` de novo, e liberta então o trinco;

- a atividade a consegue finalmente fechar o trinco e insere também um novo elemento em `players`.

Neste cenário, vemos que chegamos a um estado inválido em que violamos o invariante `players.size() <= max`. Isto significa que, independentemente do funcionamento da primitiva que permite suspender e acordar atividades, precisamos de verificar de novo a condição após a espera. Por esta razão, a espera por uma condição deve usar sempre um ciclo.

Atenção!

Além desta possibilidade de corrida com outras atividades, que não seria um problema se o método `enter()` fosse usado apenas por uma, a especificação destas primitivas permite que uma atividade seja acordade de forma espúria, ou seja, sem que tenha havido qualquer aviso explícito. Esta é mais uma razão para que a avaliação da condição seja sempre feita com um ciclo.

A concretização da primitiva com estas caraterísticas, que permite suspender e retomar atividades, é a *variável de condição*, que corresponde em Java à interface `Condition` e que podemos aplicar para completar o exemplo.

```
30    private Lock l = new ReentrantLock();
31    private Condition c = l.newCondition(); (1)
32
33    public void enter(String name) throws InterruptedException {
34        l.lock();
35        while (players.size() >= max)
36            c.await(); (2)
37        players.put(name, new Player());
38        l.unlock();
39    }
40
41    public void leave(String name) {
42        l.lock();
43        players.remove(name);
44        c.signalAll(); (3)
45        l.unlock();
46    }
```

Cada variável de condição `c` está associada a um trinco `l`, sendo criada com o método `l.newCondition()` (1), de forma a que o método `c.await()` (2), que permite suspender a atividade, seja capaz de atomicamente abrir o trinco com o ínicio do período de espera. Este método fecha também o trinco de novo, automaticamente, ao acordar, para o que terá que esperar que o trinco esteja aberto. Quando é feita uma alteração ao estado observado pela condição, acordam-se as atividades suspensas com (3).

Espera por um evento

O idioma genérico para permtir que uma atividade espere por um evento que é atestado por um predicado $p(v_1, \ldots, v_n)$ sobre um conjunto de variáveis v_i usa um trinco, correspondente a estas variáveis do estado, e uma variável de condição:

```
Lock l = new ReentrantLock();
Condition c = l.newCondition();
```

Para esperar pelo evento, uma atividade executa o seguinte fragmento de código:

```
l.lock();
while (¬p(v1,...,vn)) // avaliação da condição
    c.await();
l.unlock();
```

Sempre que é alterada qualquer das variáveis que fazem parte do predicado que atesta o evento acordam-se as atividades suspensas de forma a que o possam reavaliar:

```
l.lock();
vi ← ... // modificação do estado
c.signalAll();
l.unlock();
```

3.3 Minimização das atividades acordadas

O idioma genérico sugerido permite obter uma solução correta de uma forma mecânica, analisando apenas sintaticamente o programa para descobrir os sítios no código que modificam as variáveis em causa. Não é, no entanto, ótimo em termos de desempenho, uma vez que sempre que uma das variáveis em causa é alterada, todas as atividades em espera são acordadas, mesmo nos casos em que uma análise semântica do programa indica que apenas uma – ou até nenhuma! – das atividades em espera conseguirá fazer progresso. No caso das atividades em espera poderem ser numerosas, isto fará com que todas elas tenham que acordar, testar a condição e voltar a suspender-se, competindo momentaneamente pelos recursos de processamento. Este efeito é conhecido como *tropel* (*thundering herd* em inglês).

De facto, a proposta de solução para o exemplo inclui já uma optimização em relação ao idioma genérico. A alteração ao estado na linha 37 não está acompanhada de uma invocação correspondente a `c.signalAll()` pois é evidente que a alteração feita, que acrescenta um novo elemento a `players` nunca fará com que a condição da linha 35 se torne falsa. Antes pelo contrário!

Uma análise deste exemplo mostra também que existe um efeito de tropel

sempre que um jogador sai usando o método `leave()`, uma vez que deixa apenas uma vaga disponível mas acorda todas as atividades que estejam à espera, das quais apenas uma terá oportunidade de prosseguir e iniciar o jogo. Podemos assim fazer uma segunda optimização.

```
33    public void enter(String name) throws InterruptedException {
34        l.unlock();
35        while (players.size() >= max)
36            c.await();
37        players.put(name, new Player());
38        l.unlock();
39    }
40
41    public void leave(String name) {
42        l.unlock();
43        players.remove(name);
44        c.signal(); (1)
45        l.unlock();
46    }
```

Neste caso, em vez de `signalAll()`, usamos apenas `signal()` (1), que acorda no máximo uma das atividades em espera, escolhendo aquela que está suspensa há mais tempo.

signal() vs. signalAll()

De uma forma genérica usamos `signalAll()` em qualquer uma destas situações:

1. A modificação ao estado que acabamos de efetuar pode levar a que mais do que uma atividade em espera possa vir a fazer progresso.
2. A variável de condição que estamos a sinalizar pode ter à espera atividades cujo progresso não é afetado pela modificação que acabamos de fazer, além da(s) que possa(m) fazer progresso. Caso contrário, poderá acontecer que a atividade acordada não possa fazer progresso enquanto que outras que pudessem progredir continuem suspensas indefinidamente.

Devemos usar `signal()` sempre que nenhuma destas condições for verdade, uma vez que é mais eficiente e justo.

3.4 Justiça relativa e ordenação

As variáveis de condição garantem que a atividade acordada pela invocação de `signal()` é a que invocou o `await()` há mais tempo. Ou seja, garantem que as

atividades são acordadas por ordem de chegada. Ao evitar acordar actividades em vão, usando as técnicas descritas na secção anterior, está já a contribuir-se para uma justiça relativa entre as várias atividades que competem por um recurso através de uma variável de condição.

Porém, depois de acordadas as atividades vão ainda competir pelo fecho do trinco correspondente e, por omissão, este não garante uma ordenação. Mesmo quando o trinco é configurado para garantir também a ordem, há situações em que são acordadas mais atividades do que as que podem fazer progresso. Nestes casos, as que não podem fazer progresso são novamente suspensas, ficam por isso em último lugar na fila. No caso de existir uma elevada contenção, é possível que uma atividade seja ultrapassada repetidamente no acesso a um recurso. Consideremos como exemplo a seguinte sequência de acontecimentos, em que `players.size() == max` e há diversas atividades já suspensas, das quais a mais antiga e como tal na cabeça da fila é a:

- uma atividade b invoca `leave()` para sair do jogo, removendo um elemento do mapa e acordando uma atividade, que será a;
- a atividade a tenta então fechar o trinco `l` para testar de novo a condição;
- simultaneamente, uma outra atividade c invoca `enter()`, tentando também fechar o trinco;
- se for a atividade c a conseguir fechar o trinco, vai conseguir ultrapassar a (e todas as restantes em espera);
- quando a testa de novo a condição, esta já é novamente falsa pelo que a re-inicia a espera mas é colocada no fim da fila.

Em resumo, não só permitimos que uma atividade ultrapasse todas as outras que estão à espera, mas fazemos também com que a que estava na cabeça da fila perca a vez e vá para o fim da fila. Em alguns casos, é possível que a atividade c que consegue passar à frente, seja de facto a mesma que b, que acabou de sair, tornando a situação ainda mais injusta.

Podemos, no entanto, incluir explicitamente a ordem de chegada na condição. Para o conseguir, começamos por assumir a existência de uma variável `nextTicket`, observada e incrementada à entrada do método `enter()`. Incluimos então, na condição, a necessidade do número de ordem assim obtido ser menor ou igual que o valor de uma segunda variável `nextTurn`, incrementada à saída do método `enter()`. Embora isto garanta uma ordenação estrita à entrada, para qualquer que seja o evento por que esperamos, tem duas consequências importantes:

- A condição passa a ser diferente para cada atividade em espera, pelo que passa a ser válida a segunda cláusula da Secção 3.3 e temos que usar `signalAll()`.
- A modificação da variável `nextTurn` à saída do método `enter()` pode levar a que uma atividade em espera possa fazer progresso, pelo que precisa também de ser acompanhada de `signalAll()`.

Em conjunto, estas duas alterações fazem com que estejamos a acordar atividades desnecessariamente com mais frequência, o que não é desejável.

Se assumirmos que é aceitável que uma atividade seja ultrapassada, desde que não perca a possibilidade de fazer progresso, podemos obter uma solução mais simples.

```
33      private int nextTicket, nextTurn = max;
34
35      public void enter(String name) throws InterruptedException {
36          l.unlock();
37          var ticket = nextTicket++;
38          while (ticket>nextTurn) ①
39              c.await();
40          players.put(name, new Player());
41          l.unlock();
42      }
43
44      public void leave(String name) {
45          l.unlock();
46          players.remove(name);
47          nextTurn++; ②
48          c.signalAll();
49          l.unlock();
50      }
```

Esta solução incrementa `nextTurn` no método `leave()` ②. Ou seja, damos autorização a mais uma atividade para entrar no jogo, não quando tivermos a certeza que todos os predecessores já entraram, mas sim quando temos a certeza que há espaço para essa atividade e todas as predecessoras. De facto, como passamos a contar explicitamente os lugares livres dentro do jogo, podemos simplificar a condição ① que não precisa de observar o tamanho de `players`.

Continuamos porém a precisar de usar `signalAll()` de forma a garantir que acordamos a atividade seguinte entre as várias que estão suspensas, o que ainda não é ideal de um ponto de vista do desempenho, se admitirmos que temos uma grande número de atividades em espera. Podemos, finalmente, ultrapassar também esta limitação, tirando partido da possibilidade de criar várias variáveis de condição associadas a cada trinco.

```
32      private Lock l = new ReentrantLock();
33      private Queue<Condition> q = new LinkedList<>(); ③
34
35      private int nextTicket, nextTurn = max;
36
37      public void enter(String name) throws InterruptedException {
38          l.unlock();
39          var ticket = nextTicket++;
40          if (ticket>nextTurn) {
41              var c = l.newCondition(); ④
42              q.add(c);
43              do
```

```
44                  c.await();
45              while (ticket>nextTurn);
46          }
47          players.put(name, new Player());
48          l.unlock();
49      }
50
51      public void leave(String name) {
52          l.unlock();
53          players.remove(name);
54          nextTurn++;
55          var c = q.remove();
56          if (c != null)
57              c.signal(); ⑤
58          l.unlock();
59      }
```

Começamos por substituir a declaração de uma variável de condição por uma fila para armazenar várias, uma por cada atividade suspensa ③. Sempre que percebemos que é necessário suspender uma atividade, criamos uma nova variável de condição ④ que inserimos na lista, por ordem de chegada. Finalmente, quando verificamos que é necessário acordar uma atividade suspensa – quando a fila não está vazia – podemos então usar `signal()` para acordar especificamente a atividade que nos interessa. Curiosamente, neste caso `signalAll()` teria exatamente o mesmo resultado, porque temos sempre no máximo uma atividade na variável de condição (e só não haverá nenhuma, se ela tiver acordado de forma espúria).

Semáforos

A funcionalidade que construímos, de controlar o acesso a um recurso de forma a que não seja usado por mais do que `max` atividades, corresponde de facto a uma primitiva clássica de programação concorrente, os *semáforos*, que encontramos também no Java na classe `j.u.c.Semaphore`. Essa implementação garante também a ordem de acesso ao recurso por ordem de chegada.

Atenção!

As estratégias para melhoria do desempenho e justiça relativa na espera por condições não devem no entanto ser a primeira escolha, pois é provável que a solução base seja adequada, excepto em casos de grande contenção e com número elevado de atividades suspensas. De facto, a utilização de variáveis de condição para ter atividades em espera por longos períodos de tempo não é aconselhável, pois cada atividade reserva recursos de memória significativos e assim se limita a escalabili-

dade do programa. Será aconselhável nestes casos dividir a tarefa em várias mais pequenas, que possam ser individualmente atribuídas a atividades quando não precisarem de ficar bloqueadas.

Capítulo 4

Trincos partilhados

O objetivo deste capítulo é implementar uma conhecida ferramenta de exclusão mútua: o *trinco partilhado*. Distingue-se de um trinco normal ao permitir que múltiplas atividades, desde que identificadas como leitoras, possam aceder concorrentemente a uma secção crítica. Em Java encontramos uma implementação desta primitiva na classe `java.util.concurrent.ReadWriteLock`. Encontramos esta primitiva normalmente em sistemas de gestão de bases de dados.

4.1 Solução genérica

Começamos por procurar uma solução correta, sem preocupações de desempenho, utilizando o método genérico do Capítulo 3, em que deduzimos a utilização das variáveis de condição mecanicamente das condições e manipulação do estado partilhado.

Para implementar um trinco partilhado, partimos da interface que queremos expôr:

- Métodos `lockSh()` e `lockEx()`, que permitem fechar o trinco respetivamente nos modos partilhado e exclusivo. Poderiamos também usar apenas um método e indicar o modo pretendido como parâmetro, mas optamos por esta interface por ser mais clara e corresponder ao que encontramos também na implementação existente na plataforma Java.
- Métodos correspondentes `unlockSh()` e `unlockEx()`, que permitem abrir o trinco em ambos os casos. Mais uma vez, seria possível usar só um método, que determinava qual o modo a usar dependendo de como tinha sido fechado.

Podemos assim esquematizar as condições pelas quais cada atividade deverá esperar em cada uma das situações.

```
public void lockEx() {
    while (fechado por algum leitor ou escritor)
```

```
            espera;
        regista a presença de um escritor;
    }
    public void lockSh() {
        while (fechado por algum escritor)
            espera;
        regista a presença de mais um leitor;
    }
    public void unlockEx() {
        regista a saída do escritor;
    }
    public void unlockSh() {
        regista a saída de um leitor;
    }
```

Neste caso, não temos no programa ainda qualquer estado partilhado que sirva para avaliar estas condições. Devemos pois adicionar as variáveis necessárias para indicar qual o estado do trinco partilhado que estamos a construir.

```
9     private int readers;
10    private boolean writer;
```

Em relação à avaliação da presença de leitores, precisamos de contar quantos entram e quantos saem da secção crítica guardada por este trinco. Usamos para isso o inteiro `readers`.

No caso dos escritores, embora pudessemos usar exatamente a mesma estratégia, é fácil de ver que esta variável nunca teria um valor superior a 1. Caso contrário, teríamos uma situação de erro. Por isso usamos apenas um booleano `writer` para indicar a presença de um escritor.

Tirando então partido de um trinco e de uma variável de condição, obtemos uma solução genérica.

```
12    private Lock l = new ReentrantLock();
13    private Condition c = l.newCondition();
14
15    public void lockEx() throws InterruptedException {
16        l.lock();
17        while (writer || readers>0) (1)
18            c.await();
19        writer = true;
20        l.unlock();
21    }
22
23    public void lockSh() throws InterruptedException {
24        l.lock();
25        while (writer) (2)
26            c.await();
27        readers++;
28        l.unlock();
29    }
30
```

```
31      public void unlockEx() {
32          l.lock();
33          writer = false; ③
34          c.signalAll();
35          l.unlock();
36      }
37
38      public void unlockSh() {
39          l.lock();
40          readers--; ④
41          c.signalAll();
42          l.unlock();
43      }
```

A condição de entrada de um escritor ① observa ambas as variáveis `readers` e `writer`. As alterações a estas variáveis ③④ que podem resultar na condição mudar para falsa são acompanhadas da sinalização da variável de condição. As restantes alterações a estas variáveis, nas linhas 19 e 27, mudam ou mantêm a condição verdadeira, pelo que não precisam de sinalizar a variável de condição. Do mesmo modo, a condição de entrada de um leitor ② observa apenas a variável `writer`. A alteração a esta variável ③, que pode resultar na condição mudar para falsa, é acompanhada da sinalização da variável de condição.

Em ambos os casos, usamos `signalAll()` e não `signal()`, pois não estamos ainda preocupados com o desempenho. Aliás, podemos até ver que há situações em que cumprimos as cláusulas da Secção 3.3. Por exemplo, a saída de um escritor pode dar acesso tanto a outro escritor como a vários leitores.

4.2 Simplificação e optimização

Embora correta, uma vez que cumpre o pedido sem corridas ou impasses, a solução genérica não é a mais simples nem obtém um desempenho ideal. Por exemplo, podemos facilmente observar que a abertura do trinco à saída do método `lockEx()` (linha 20) e o seu fecho de novo à entrada de `unlockEx()` (linha 24) não são úteis, uma vez que qualquer outra atividade que aproveite para o fechar não conseguirá fazer progresso.

Sendo assim, podemos omitir estas linhas deixando o trinco `l` fechado durante todo o período em que um escritor está ativo na secção crítica. Intuitivamente, isto faz sentido, uma vez que um trinco partilhado usado apenas no modo exclusivo é equivalente a um trinco normal. Isto tem uma consequência interessante: quando a variável `writer` assume o valor verdadeiro, volta ao valor falso sem que o trinco tenha sido aberto. Isto significa que nunca é possível observar esta variável a verdadeiro!

Podemos então eliminar essa variável e simplificar todo o código assumindo que essa variável seria sempre falsa.

```
14      public void lockEx() throws InterruptedException {
15          l.lock();
```

```
16          while (readers > 0)
17              c.await();
18          // omitimos a abertura do trinco (1)
19      }
20
21      public void unlockEx() {
22          // omitimos o fecho do trinco (2)
23          l.unlock();
24      }
```

O código resultante omite assim a abertura do trinco (1) e o fecho correspondente no método seguinte (2), bem como as modificações correspondentes à variável `writer` que nunca poderiam ser observadas por outras atividades.

A eliminação da variável `writer` tem impacto no método que controla a entrada de leitores.

```
26      public void lockSh() throws InterruptedException {
27          l.lock();
28          // omitimos a espera (3)
29          readers++;
30          l.unlock();
31      }
```

Neste caso, admitindo que `writer` seria sempre falso, o ciclo de espera nunca seria usado. Logo podemos omiti-lo completamente (3).

Finalmente, observando que só faz sentido acordar os escritores depois de o último leitor ter saído, reduzimos as situações em que sinalizamos a variável de condição.

```
33      public void unlockSh() {
34          l.lock();
35          readers--;
36          if (readers == 0) (4)
37              c.signalAll();
38          l.unlock();
39      }
```

É importante justificar porque continuamos a usar `signalAll()` (4) quando, das atividades que acordam – necessariamente escritoras – apenas uma poderá fazer progresso. De facto, se usarmos aqui `signal()` corremos o risco de deixar alguns escritores suspensos. Quando o escritor que avança em seguida sai da secção crítica, não iria acordar esses outros escritores, deixando o trinco aberto com atividades candidatas à espera indefinidamente.

Capítulo 5

Produtor/Consumidor

O objetivo é agora construir uma fila que possa ser usada para transferir objetos entre um produtor e um consumidor. De forma a permitir que estas atividades trabalhem de forma assíncrona, isto é, sem que cada atividade tenha que esperar em cada passo pela outra, a fila permite o armazenamento temporário de vários objetos.

5.1 Bloqueio do consumidor

Começamos por considerar apenas o caso em que, estando a fila vazia, o consumidor tem que esperar que o produtor insira um novo elemento. Assumimos que o estado da fila pode ser representado por um objeto `Queue q` e, de forma a poder cumprir os passos para a sincronização, declaramos também um `Lock l` e uma `Condition c`:

```
11    private Queue<T> q = new LinkedList<>();
12    private Lock l = new ReentrantLock();
13    private Condition c = l.newCondition();
```

Identificamos o evento pelo qual queremos esperar como sendo a existência de pelo menos um elemento na fila `q`, ou seja, que a fila não está vazia.

```
15    public T get() throws InterruptedException {
16        try {
17            l.lock();
18            while (q.isEmpty()) (1)
19                c.await();
20            return q.remove();
21        } finally {
22            l.unlock();
23        }
24    }
25
26    public void put(T s) {
```

```
27          try {
28              l.lock();
29              q.add(s); ②
30              c.signal();
31          } finally {
32              l.unlock();
33          }
34      }
```

Identificamos então qual o estado que é observado pela condição associada à espera e quais as modificações a esse estado que podem levar a que a condição se torne falsa:

- A condição ① associada à espera pelo evento observa apenas o estado correspondente à fila `q`. A espera propriamente dita é feita sobre a variável de condição `c` na linha 19.
- A modificação ② do estado da fila `q` é relevante para a avaliação da condição, pois a adição de um elemento à fila pode fazer a condição ① passar de verdadeira a falsa. É pois necessário fazer a sinalização da variável de condição `c` na linha 30 para interromper a espera.
- A fila `q` é também modificada na linha 20. No entanto, não é necessário sinalizar a variável de condição `c` para acordar outros consumidores, pois a remoção de um elemento da fila nunca fará com que a condição ① se torne falsa. Antes pelo contrário!

Note-se que aqui é evidente que basta utilizar o método `signal()` em vez do `signalAll()`, uma vez que a adição de um único elemento será relevante no máximo para um consumidor.

5.2 Bloqueio do produtor

Assumindo um limite superior `MAX` para o número de elementos na fila `q`, consideramos agora apenas o caso em que, estando a fila cheia, o produtor tem que esperar que o consumidor remova um elemento.

```
16      public T get() {
17          try {
18              l.lock();
19              c.signal();
20              return q.remove(); ①
21          } finally {
22              l.unlock();
23          }
24      }
25
26      public void put(T s) throws InterruptedException {
27          try {
```

```
28              l.lock();
29              while (q.size() >= MAX) ②
30                  c.await();
31              q.add(s);
32          } finally {
33              l.unlock();
34          }
35      }
```

Repetindo então o processo de identificação da condição de espera e do estado relevante para a sua avaliação:

- A condição ② associada à espera pelo evento observa apenas o estado correspondente à fila `q`. A espera propriamente dita é feita sobre a variável de condição `c` na linha 30.
- A modificação ① do estado da fila `q` é relevante para a avaliação da condição, pois a remoção de um elemento à fila pode fazer a condição ② passar de verdadeira a falsa. É pois necessário fazer a sinalização da variável de condição `c` na linha 19 para interromper a espera.
- A fila `q` é também modificada na linha 31. No entanto, não é necessário sinalizar a variável de condição `c` para acordar outros produtores, pois a remoção de um elemento da fila nunca fará com que a condição ② se torne falsa. Antes pelo contrário!

Ordem dentro da secção crítica

É interessante observar que a ordem relativa da modificação do estado ① e a sinalização da variável de condição correspondente dentro de uma secção crítica não é relevante. Neste caso, é mais conveniente fazer a sinalização na linha 19 antes de remover o elemento da fila `q`, mas isso não incorre no risco do produtor ser acordado antes de haver espaço disponível, pois para que o produtor observe ou modifique a fila, terá que adquirir `l`.

5.3 Bloqueio de ambos

Podemos agora combinar o código que escrevemos para fazer os consumidores esperar quando a fila está vazia, com o código necessário para fazer os produtores esperar quando a fila está cheia.

```
16      public T get() throws InterruptedException {
17          try {
18              l.lock();
19              while (q.isEmpty()) ①
```

```
20                  c.await();
21              c.signal();
22              return q.remove(); ②
23          } finally {
24              l.unlock();
25          }
26      }
27
28      public void put(T s) throws InterruptedException {
29          try {
30              l.lock();
31              while (q.size() >= MAX) ③
32                  c.await();
33              q.add(s);  ④
34              c.signal();
35          } finally {
36              l.unlock();
37          }
38      }
```



Neste caso, temos duas condições que podem levar ao bloqueio de atividades:

- A condição ① irá esperar pela modificação do estado ④.
- A condição ③ irá esperar pela modificação do estado ②.

Surge então a questão: será a utilização do método `signal()` ainda suficiente?

1. De acordo com o primeiro critério, sim, pois a modificação do estado em ② ou ④ nunca permite o progresso de mais do que uma atividade em espera, que consome o novo elemento inserido ou ocupa o espaço livre deixado pelo elemento removido.
2. De acordo com o segundo critério, aparentemente, também sim, pois para ter atividades em condições diferentes à espera na mesma variável de condição parece ser necessário que a fila `q` esteja vazia – para a condição ① ser verdadeira – e cheia – para a condição ③ ser verdadeira – simultaneamente.

Consideremos no entanto um caso em que temos uma fila vazia e `MAX`+1 consumidores bloqueados à espera no método `get()`. É possível que um produtor faça em rápida sucessão `MAX` invocações do método `put()`, levando a que:

- `MAX` consumidores tenham sido sinalizados e estejam a tentar obter `l` para poder testar a condição e fazer progresso;
- a fila `q` contenha `MAX` elementos, uma vez que ainda nenhum dos consumidores conseguiu fazer progresso e remover um elemento;
- na variável de condição `c` esteja ainda à espera um consumidor.

Isto é possivel uma vez que os consumidores, ao acordarem, estão em competição por `l` com o produtor, podendo `l` ser atribuído a qualquer um deles.

Se o produtor tentar agora inserir o elemento `MAX`+1, a condição ③ é verdadeira – a fila está cheia – e a atividade irá esperar também na condição `c`. Nesta situação temos então, simultaneamente, um consumidor e um produtor à espera na mesma variável de condição, contradizendo a nossa intuição inicial de que isto não seria possível.

Atenção!

Quando a condição usada para colocar uma atividade em espera numa variável de condição não é exatamente a mesma, não é fácil garantir que uma invocação de `signal()` irá acordar a atividade que queremos. No caso de ser acordada outra atividade, para a qual a condição ainda é verdadeira, a sinalização irá perder-se e podemos dar origem a um impasse.

Note-se que mesmo que as condições sejam sintaticamente iguais, ou até exatamente a mesma linha de código, as variáveis em causa podem referir-se a objetos diferentes e fazer com que na prática, estejamos perante condições diferentes!

Podemos obter um programa correto se modificarmos as linhas 21 e 34 para usar `signalAll()`. Esta opção não é, no entanto, a mais eficiente se admitirmos como provável que, em cada momento, possam estar muitas atividades em espera, pois irão sempre acordar todas quando apenas uma delas de facto poderá fazer progresso.

5.4 Uma variável para cada condição

Neste caso, podemos ainda reconhecer que temos atividades à espera de apenas duas condições distintas, uma vez que a variável `q` refere sempre o mesmo objeto. Podemos por isso trocar a variável de condição única `c` por duas outras, uma para cada caso:

```
13      private Condition notEmpty = l.newCondition();
14      private Condition notFull = l.newCondition();
```

É conveniente usar nomes descritivos para estas variáveis, que tornem a sua utilização fácil de entender. Por exemplo, `notEmpty.await()` indica claramente que estamos à espera que a fila deixa de estar vazia e `notEmpty.signal()` que isso mesmo pode ter acontecido.

```
17      public T get() throws InterruptedException {
18          try {
19              l.lock();
20              while (q.isEmpty()) ①
```

```
21                  notEmpty.await();
22              notFull.signal();
23              return q.remove(); ②
24          } finally {
25              l.unlock();
26          }
27      }
28
29      public void put(T s) throws InterruptedException {
30          try {
31              l.lock();
32              while (q.size() >= MAX) ③
33                  notFull.await();
34              q.add(s);  ④
35              notEmpty.signal();
36          } finally {
37              l.unlock();
38          }
39      }
```

Tirando partido de duas variáveis de condição, temos agora duas aplicações distintas do padrão genérico para esperar por um evento:

- A condição ① irá esperar pela modificação do estado ④, que é o único relevante para uma transição da condição de verdadeira para falsa. É assim suficiente a sinalização feita na linha 35, uma vez que a adição de um elemento à fila permite o progresso de apenas uma das atividades em espera.
- De forma análoga, a condição ③ irá esperar pela modificação do estado ② e é suficiente a sinalização da linha 22.

Esta solução é assim uma forma correta e eficiente de assegurar um mecanismo de comunicação entre atividades concorrentes, permitindo o seu progresso assíncrono. Pode também ser usada como ponto de partida para outros problemas semelhantes, que imponham semântica adicional às operações.

Bibliografia

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, revised 1st edition, 2012.

[PGB+05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.