Introdução à Programação Concorrente em Java

José Orlando Pereira Paulo Sérgio Almeida
Ana Nunes Alonso

20 de novembro de 2023

"Introdução à Programação Concorrente em Java"
Edição de 20 de novembro de 2023.

Prefácio

Motivação

A programação concorrente, em que diversas atividades concorrentes ou *fios de execução* num mesmo programa colaboram na resolução de um problema, 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 três 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 enunciados clássicos cujas soluções podem ser reutilizadas.
- O Capítulo 6 inclui enunciados de problemas de complexidade crescente, que podem ser usados como suporte a uma componente laboratorial e para estudo autónomo.

Embora o texto foque apenas as primitivas `java.util.concurrent` da linguagem Java atual, incluimos breves apêndices que fazem a correspondência com Java antigo (Apêndice A) e com as linguagens C (Apêndice B) e C++ (Apêndice C).

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

Os fragmentos de código Java 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 atividades concorrentes *fios de execução* – de *threads* em inglês. Embora possam tirar partido da existência de vários núcleos do processador, não são limitadas por estes, sendo que o sistema operativo troca rapidamente entre os fios de execução ativos de forma a criar a ilusão de que todos 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 do inicial que executa o método `main()`, uma aplicação pode lançar novos fios de execução 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, o fio de execução inicial está dedicado à leitura do teclado ②. O fio adicional imprime `"."` a cada segundo, depois de iniciado com a invocação

de `start()` ①. Neste caso, usamos `setDaemon(true)` para que a terminação do programa como um todo não dependa da terminação deste fio de execução concorrente e, assim, que a terminação da `main()` provoque a terminação do processo como um todo.

Este é um exemplo da utilização de vários fios de execução para atender mais do que uma entrada – o teclado e o relógio. Nestes casos, os vários fios de execução estão na maior parte do tempo inativos, 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 vários fios de execução 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 concorrentemente. O sistema operativo irá assegurar que esses fios de execução são atribuídas aos núcleos disponíveis. Neste caso, pretendemos esperar que todas estas sub-tarefas terminem, correspondendo à conclusão da tarefa como um todo. Para isso usamos o método `join()` que permite esperar pela conclusão de um fio de execução. Neste caso, o fio inicial ficará bloqueado sucessivamente na invocação do método `join()`, sendo libertado pela terminação de cada um dos fios adicionais. É de notar que a ordem pela qual os fios de execução terminam não é pré-determinada, mas tal não é necessário para a utilização deste método para esperar pela sua conclusão.

Reutilização de um objecto *Thread*

A criação e início de fios de execução tem um custo não negligenciável quando pretendemos realizar 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 mesmo fio 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 fios de execução 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 usamos em cada um dos fios de execução. Ou seja, há mais do que um fio 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 fio de execução inicial (apenas durante a iteração do ciclo em que é criada) e para o fio adicional iniciado nessa iteração do ciclo `for`. Neste caso, cada vez que um fio adicional usar a variável `local` estará a aceder uma instância própria. Este objeto tem por isso *estado local*.
- Qualquer variável automática criada por invocações num fio de execução, como acontece com a variável `v` ③, constitui também *estado local*.

Cada fio de execução 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 fio 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 os fios executam de forma concorrente e assim a ordem pela qual executam (e imprimem) não é necessariamente a ordem pela qual os fios de execução foram iniciados.

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 fios de execução 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 fios de execução 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 um fio de execução 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 vários fios de execução.

```
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 um dos `N` fios de execução 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 concorrentes 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 fios de execução concorrentes, é possível (provável!) que entre os passos 1 e 3 de um fio a haja outro fio b (ou até várias) a efetuar todo o processo (várias vezes). Quando a a executa então o passo 3, irá sobrescrever o valor escrito por 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 dos fios de execução que o constituem, são erros e chamam-se *corridas*.

1.4 Corridas na leitura

Consideramos agora um outro caso, em que temos apenas um fio de execução a escrever e, como tal, não surgem os problemas descritos na secção anterior. No entanto, temos agora duas variáveis distintas que são escritas por um dos fios de execução e lidas por outro.

```
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 um fio de execução, `c1 >= c2`. Consideremos agora a possibilidade de ler estas variáveis a partir de outro fio de execução. 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 fios 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 um fio de execução invoca repetidamente o método `inc()`, para incrementar os dois contadores, enquanto outro 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` pode ser observada pela ordem inversa (i.e. `c1 < c2`) a partir de outro fio de execução! 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 fios de execução concorrentes escrevem e leem 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 um fio suspenda a sua execução até um acontecimento específico num outro. 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 o fio de execução 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 fios de execução 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 uma simulação de leis da física. 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 diversos fios de execução da seguinte forma:

- Uma primeiro por cada cliente, para atender as mensagens que trazem a informação relativa às ações do jogador;
- uma segundo por cada cliente, que periodicamente obtém o estado de simulação e o transmite através do canal de comunicação ao jogador;
- fios 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 estes fios de execução 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 múltiplos fios 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 um fio de execução 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 um fio de execução 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 elas sejam executadas concorrentemente. 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, um fio de execução pode entrar e fechar a porta com o trinco, impedindo outros de entrar. Essas outros ficam à espera do lado de fora da secção crítica.
- Ao sair da secção crítica ocupada, volta a abrir o trinco, dando a vez ao seguinte.

Isto garante que não há corridas de escrita. Além disso, um trinco garante também que um fio de execução que feche o trinco irá observar todas as escritas feitas pelo 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. Outros fios de execução 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. Um dos fios 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 exceção. 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 fio de execução 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 outros fios de execução 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 outros fios de execução 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 método `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 concorrentes 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 correspondentes a `splayer` e `tplayer` fechados, pelo que outros fios de execução que manipulem os mesmo objetos não darão origem a corridas.

Consideremos, no entanto, a seguinte sequência de acontecimentos em dois fios de execução em que a executa `shoot(a,b)` e b executa `shoot(b,a)`:

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

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

Impasses

Chamamos *impasse* (ou *deadlock* em inglês) a esta situação em que fios de execução esperam uns pelos outros, 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 um dos fios de execução.

Atenção!

Neste caso é fácil de ver. Mas quando há invocações a métodos externos, pode ser mais complicado de detetar a possibilidade de impasses. Por isso é bom minimizar a invocação de outros métodos 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 um fio de execução tenta obter um par de trincos pela ordem contrária à que outro (ou outros) 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 todos.

```
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 exceto 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 outros fios de execução. 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, 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 fios 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.lock();
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.lock();
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 um fio de execução de cada vez.

Curiosamente, ao resolver o primeiro problema tornamos o segundo mais complicado. Neste momento, uma fio de execução 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 outros 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 um fio de execução espera indefinidamente por algo que não pode acontecer, pois depende de um trinco que o próprio 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 fios de execução, 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.lock();
    if (players.size() >= max) {
        l.unlock();
        await event; // suspende fio de execução
```

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

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

Precisamos pois de uma primitiva que nos permita suspender o fio de execução 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 dois fios concorrentes, numa situação em que `players.size() == max`:

- a invoca o método `enter()`, fecha o trinco e verifica que a condição é verdadeira pois `players.size() == max`;
- entretanto b invoca `leave()` e fica à espera do trinco;
- a abre o trinco antes de começar a espera;
- neste momento, o trinco é fechado por b, que modifica `players` e tenta avisar fios de execução suspensos, que neste momento ainda não existem;
- a prossegue então para a espera e fica suspenso.

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

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

- entretanto, b invoca `leave()`, fecha o trinco, modifica `players` e avisa os suspensos, que neste momento é apenas a;
- a ao acordar tenta fechar de novo o trinco;
- ao mesmo tempo, c invoca `enter()` e tenta também fechar o trinco, pelo que estão dois fios de execução a competir por `l`;
- 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 consegue finalmente fechar o trinco e insere também um novo elemento em `players`.

Neste cenário, vemos que chegamos a um estado em que violamos o invariante `players.size() <= max`. Isto significa que, independentemente do funcionamento da primitiva que permite suspender e acordar fios de execução, 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, que não seria um problema se o método `enter()` fosse usado apenas por um único fio de execução, a especificação destas primitivas permite acordar 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 fios de execução, 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 um fio de execução, seja capaz de atomicamente abrir o trinco com o início 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 os fios de execução suspensos com ③.

Espera por um evento

O idioma genérico para permtir esperar 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, executa-se 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 os fios suspensos de forma a que o possam reavaliar:

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

3.3 Minimização dos fios de execução acordados

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, todos os fios em espera são acordados, mesmo nos casos em que uma análise semântica do programa indica que apenas um – ou até nenhum! – dos em espera conseguirá fazer progresso. No caso de serem numerosos, isto fará com que todos 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 otimização 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 todos os que estejam à espera, das quais

apenas um terá oportunidade de prosseguir e iniciar o jogo. Podemos assim fazer uma segunda otimização.

```
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 um dos fios de execução 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 um fio de execução em espera possa vir a fazer progresso.
2. A variável de condição que estamos a sinalizar pode ter à espera fios de execução cujo progresso não é afetado pela modificação que acabamos de fazer, além do(s) que possa(m) fazer progresso. Caso contrário, poderá acontecer que o escolhido para acordar 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 o fio de execução acordado pela invocação de `signal()` é o que invocou o `await()` há mais tempo. Ou seja, garantem que são acordados por ordem de chegada. Ao evitar acordar fios de execução em vão, usando as técnicas descritas na secção anterior, contribui-se para

uma justiça relativa entre as várias atividades que competem por um recurso através de uma variável de condição.

Porém, depois de acordados 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 fios de execução do que os que podem fazer progresso. Nestes casos, os que não podem fazer progresso são novamente suspensos, ficam por isso em último lugar na fila. No caso de existir uma elevada contenção, é possível que um fio de execução seja ultrapassada repetidamente no acesso a um recurso. Consideremos como exemplo a seguinte sequência de acontecimentos, em que `players.size() == max` e há diversas fios de execução já suspensos, das quais a mais antiga e como tal na cabeça da fila é a:

- b invoca `leave()` para sair do jogo, removendo um elemento do mapa e acordando a que estava à espera;
- a tenta então fechar o trinco `l` para testar de novo a condição;
- simultaneamente, c invoca `enter()`, tentando também fechar o trinco;
- se for 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 colocado no fim da fila.

Em resumo, não só permitimos que um fio de execução ultrapasse todos os outros que estão à espera, mas fazemos também com que o que estava na cabeça da fila perca a vez e vá para o fim da fila. Em alguns casos, é possível que o recém-chegado c que consegue passar à frente de c, seja de facto b, que acabou de sair, a regressar, 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 um 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 um fio de execução 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 fios de execução desnecessariamente com mais frequência, o que não é desejável.

Se assumirmos que é aceitável a ultrapassagem, 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 uma autorização 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 todos os predecessores e mais um. 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 o seguinte entre os vários que estão suspensos, o que ainda não é ideal de um ponto de vista do desempenho, se admitirmos que temos uma grande número 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 fio de execução suspenso ③. Sempre que percebemos que é necessário suspender mais um, criamos uma nova variável de condição ④ que inserimos na lista, por ordem de chegada. Finalmente, quando verificamos que é necessário acordar um fio de execução suspenso – quando a fila não está vazia – podemos então usar `signal()` para acordar especificamente o que nos interessa. Curiosamente, neste caso `signalAll()` teria exatamente o mesmo resultado, porque temos sempre no máximo um utilizador de cada variável de condição (e só não haverá nenhum, se 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, exceto em casos de grande contenção e com número elevado de fios de execução suspensos. A utilização de variáveis de condição esperar por longos períodos de tempo não é aconselhável, pois cada fio de execução reserva recursos de memória significativos e assim se limita a escalabilidade do programa. Será aconselhável nestes casos dividir a tarefa em várias mais pequenas, que

possam ser individualmente atribuídas a fios de execução quando não precisarem de ficar bloqueados.

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 múltiplos fios de execução, desde que identificados como leitores, 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 fio de execução 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 outro 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 outros.

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, dos fios de execução que acordam – necessariamente escritores – apenas um 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 candidatos 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 estes trabalhem de forma assíncrona, isto é, sem que cada um tenha que esperar em cada passo pelo outro, 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 fios de execução:

- 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 um fio de execução 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 fios de execução 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 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 um fio de execução em espera numa variável de condição não é exatamente a mesma, não é fácil garantir que uma invocação de `signal()` irá acordar o que queremos. No caso de ser acordada outro, para a qual a condição ainda é verdadeira, o aviso 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 muitos em espera, pois irão sempre acordar todos quando apenas um deles de facto poderá fazer progresso.

5.4 Uma variável para cada condição

Neste caso, podemos ainda reconhecer que temos fios de execução 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 um dos que está à 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.

Capítulo 6

Exercícios

Os exercícios neste capítulo devem ser resolvidos implementando uma classe com a interface indicada que possa ser usada por múltiplos fios de execução concorrentes. Esta implementação deverá recorrer a primitivas de monitores – trincos e variáveis de condição – para garantir a correção e obter a semântica de sincronização descrita.

Sempre que um enunciado se refere a utilizadores finais (como clientes ou jogadores), admitimos que cada um é representado por um fio de execução único. As constantes referidas em cada enunciado devem ser fornecidas como parâmetros ao construtor da classe a desenvolver. Finalmente, pretende-se sempre minimizar os fios de execução que são acordadas sem necessidade como indicado na Secção 3.3.

6.1 Tutorial

Exercício 6.1.1. Pretende-se uma classe para fazer a gestão de contas bancárias, com as seguintes oprações:

```
int createAccount(int balance);
int closeAccount(int id);
int balance(int id);
boolean deposit(int id, int value);
boolean withdraw(int id, int value);
boolean transfer(int from, int to, int value);
int totalBalance(int[] ids);
```

A criação de uma conta devolve um identificador de conta, para ser usado em outras operações; o fecho de uma conta devolve o saldo desta; é possível obter a soma do saldo de um conjunto de contas. Algumas operações devolvem `false` ou 0 se um identificador de conta não existir ou não houver saldo suficiente.

A implementação deve permitir concorrência, mas garantir que os resultados sejam equivalentes a ter acontecido uma operação de cada vez. Por exem-

plo, ao somar os saldos de um conjunto de contas, faça com que a operação seja equivalente a uma fotografia instantânea, não permitindo que sejam usados montantes a meio de uma transferência (depois de retirar da conta origem e antes de somar à conta destino).

Exercício 6.1.2. Reconsidere o exercício anterior (6.1.1) utilizando os trincos partilhados do Capítulo 4 ou da classe `ReentrantReadWriteLock` para permitir mais concorrência, nomeadamente, não bloqueando o banco todo sempre que diferentes fios de execução tentam manipular as mesmas contas. Dado o custo maior destes trincos, tente limitar o seu uso, tendo em conta a expectativa de não haver grande probabilidade de contenção nas contas (pensando o caso de um banco com muitas contas).

Exercício 6.1.3. Pretende-se escrever uma classe para permitir que N atividades se sincronizem, ou seja, que esperem todas umas pelas outras, com o método:

```
void await();
```

A operação `await` deverá bloquear até que N fios de execução o tenham invocado; nesse momento o método deverá retornar em cada um deles. Assuma que cada fio de execução apenas vai invocar `await` uma vez sobre o objeto.

Exercício 6.1.4. Modifique a resposta ao exercício anterior (6.1.3) para permitir que a operação possa ser usada várias vezes por cada fio de execução (barreira reutilizável), de modo a suportar a sincronização no fim de cada uma de várias fases de computação.

Exercício 6.1.5. Implemente uma classe que permitia a gestão de um armazém acedido concorrentemente. Deverão ser disponibilizados os métodos:

```
void supply(String item, int quantity);
void consume(String... items);
```

A operação `supply()` abastece o armazem com uma dada quantidade de um item; a operação `consume()` obtém do armazem um conjunto de itens, bloqueando enquanto tal não for possível.

Assuma que cada consumidor é egoísta, tentando apropriar-se de cada item pretendido o mais cedo possível, mesmo que ainda não estejam disponíveis todos os itens de que necessita.

Exercício 6.1.6. Reconsidere o exercício anterior (6.1.5), assumindo agora que a implementação tenta otimizar o uso do armazém como um todo, não reservando itens para clientes que não possam ser satisfeitos no momento (porque faltam alguns itens pretendidos), mesmo que tal possa fazer com que alguns clientes sejam constantemente ultrapassados e nunca mais sejam satisfeitos (*starvation*).

Exercício 6.1.7. Reconsidere o exercício anterior (6.1.6), e modifique a implementação de modo a resolver o problema identificado (*starvation*).

6.2 Espera por eventos

Exercício 6.2.1. Pretende-se uma classe para monitorização de eventos com os seguintes métodos:

```
void espera(int tipo1, int n1, int tipo2, int n2);
void sinaliza(int tipo);
```

O método `espera` deverá bloquear até serem `sinalizados`, depois do momento em que é invocado, `n1` vezes um evento `tipo1` e `n2` vezes um evento `tipo2`. Considere que o tipo de um evento é um inteiro entre 1 e E.

Exercício 6.2.2. Pretende-se uma classe para monitorização de eventos com os seguintes métodos:

```
void sinaliza(String evento);
void espera(String... eventos);
```

A operação `espera` deverá bloquear até terem sido `sinalizados`, a partir desse instante, cada um dos eventos passados como parâmetro, pela ordem indicada. Por exemplo, `espera("olá", "mundo", "mundano")`, deverá esperar que seja sinalizado `"olá"`, depois `"mundo"` (não interessando se entretanto são sinalizados outros eventos), e finalmente `"mundano"`. Evite consumo crescente e desnecessário de memória.

Exercício 6.2.3. Pretende-se uma classe para monitorização de eventos com os seguintes métodos:

```
void event(String evento);
void waitDouble(String evento);
```

A operação `waitDouble` deverá bloquear até uma dupla ocorrência (a partir do momento da invocação) do evento com tipo passado como argumento. Esta é definida como a ocorrência de um evento (sinalizada com a operação `event`) precedida de uma outra ocorrência (desse evento) com um intervalo de menos de 100 milisegundos. Utilize a função `System.currentTimeMillis()` para obter o tempo corrente em milisegundos.

6.3 Controlo do grau de concorrência

Exercício 6.3.1. Pretende-se controlar o acesso a dois recursos, identificados pelos números 0 e 1, em que apenas um deles pode estar a ser acedido em cada momento e no máximo por T atividades concorrentes. A solução deve implementar os métodos:

```
int requestResource(int i);
void releaseResource(int i);
```

O método `requestResource()` é usado para solicitar o acesso a um recurso, tem como parâmetro o identificador do recurso pretendido e deverá bloquear até o recurso poder ser acedido; o método `releaseResource()` é invocado quando um

fio de execução completou o uso do recurso correspondente. Ou seja, consideramos que o recurso está a ser acedido a partir do momento em que o método `requestResource()` termina e até ao momento em que o `releaseResource()` é invocado. Tente garantir que nenhum fio de execução fique bloqueado para sempre (*starvation*).

Exercício 6.3.2. Pretende-se implementar um serviço de execução de tarefas que limite a concorrência. A solução deve implementar o método:

```
void executa(Runnable tarefa, int prioridade);
```

que invoca diretamente, no mesmo fio de execução, o método `run()` do objeto `tarefa` passado como argumento, respeitando as restrições de concorrência de acordo com a prioridade `prioridade` pedida. Assumindo que existem tarefas de prioridade 1, 2, ou 3, o serviço deverá, até C tarefas concorrentes, permitir a execução sem restrições. Para mais de C tarefas concorrentes: não permitir começar novas tarefas de prioridade 1 ou 2 caso estejam a executar tarefas de prioridade 3; só permitir começar novas tarefas de prioridade 1 se não for ultrapassado o rácio de uma tarefa de prioridade 1 por cada duas tarefas de prioridade 2 em execução.

Exercício 6.3.3. Reconsidere o exercício anterior (6.3.2) mas agora, caso o limite de concorrência C seja atingido, as tarefas em espera devem ser executadas de acordo com a prioridade. Para evitar que alguma tarefa fique à espera para sempre (*starvation*), deve ser aumentada a prioridade em uma unidade às tarefas à espera para executar, até ao máximo de P, por cada C tarefas que entretanto terminem.

Exercício 6.3.4. Pretende-se executar tarefas em pano de fundo reutilizando um conjunto fixo de NT fios de execução, garantindo também que não se armazenam mais do que NS tarefas à espera de vez para ser executadas. A submissão de uma tarefa deve ser feita com o seguinte método:

```
void execute(Runnable task);
```

Este método deve terminar imediatamente, no caso de haver menos do que NS tarefas em espera, ou bloquear em caso contrário, até a tarefa poder ser colocada em espera. A tarefa é executada, por um dos fios de execução chamando o seu método `run()`. As tarefas em espera devem ser executadas por ordem de chegada. Assuma $NS > NT$.

Exercício 6.3.5. Pretende-se um sistema para controlo de acesso a uma ponte pedonal, utilizada para visitar um navio ancorado. Podem circular pessoas em ambos os sentidos ao mesmo tempo, mas só podem estar no máximo 10 pessoas em cima da ponte e 100 no navio. A solução deve implementar os seguintes métodos:

```
void inicioIda();
void inicioRegresso();
void entrarNavio();
void fimRegresso();
```

Os dois primeiros deverão bloquear até ser possível iniciar a travessia da ponte, no sentido respetivo. Os dois últimos são usados quando uma pessoa termina a travessia da ponte, no sentido respetivo. Destes, o método `entrarNavio()` também poderá ter necessidade de bloquear. Caso estejam pessoas a querer atravessar em ambos os sentidos, deverá ser dada prioridade a quem quer regressar da visita.

Assuma que apenas uma pessoa de cada vez entra ou sai do navio e tenha cuidado em evitar situações de impasse (*deadlock*).

Exercício 6.3.6. Reconsidere o exercício anterior mas agora permitindo trocas, em que uma pessoa que está na ponte junto ao navio e quer entrar, pode fazê-lo, mesmo que a ponte e o navio estejam com a capacidade máxima atingida, se também estiver uma pessoa no navio a querer sair.

6.4 Espera simétrica

Exercício 6.4.1. Escreva uma generalização de uma barreira reutilizável (Exercício 6.1.4) de forma a obter uma abstração de acordo:

```
int propose(int choice);
```

Esta deve permitir que N threads se sincronizem para chegar a acordo num valor. Cada thread propõe um valor, ficando o método `propose` bloqueado até todas a N o terem feito; nesse momento o método deverá retornar em cada thread o máximo dos valores propostos. Tal como na barreira reutilizável, deverá ser possível uma sucessão de acordos.

Exercício 6.4.2. Pretende-se sincronizar jogadores para formar grupos para raides. Cada jogador, avaliando a dificuldade do raide, diz qual o número de jogadores que o grupo terá que ter, aceitando também um grupo com um jogador a mais do que o pedido, que é feito com o método:

```
void startRaid(int n);
```

O método `startRaid` deve bloquear até se poder formar um grupo, com n ou $n + 1$ jogadores, que o tenham invocado.

Exercício 6.4.3. Pretende-se emparelhar atividades produtora e consumidora sobre uma fila criada para o efeito. Para tal, implemente os seguintes métodos:

```
BoundedBuffer waitForConsumer();
BoundedBuffer waitForProducer();
```

A operação `waitForConsumer()`, para ser utilizada por um produtor, deverá bloquear até ele ser emparelhado com um consumidor (que tenha invocado ou vá invocar o `waitForProducer`), devolvendo um `BoundedBuffer`, que deverá ser criado para uso exclusivo deste par. De igual modo, `waitForProducer`, para ser usada por um consumidor, deverá bloquear até este poder ser emparelhado com um produtor.

Assuma um número arbitrário de produtores e consumidores que possam querer ser emparelhados, faça o emparelhamento por ordem de chegada e evite acordar fios de execução sem necessidade. Assuma a existência da classe `BoundedBuffer`, como descrita no Capítulo 5.

Exercício 6.4.4. Pretende-se reunir jogadores para formar equipas para partidas de futebol. Cada jogador invoca um método:

```
int obterEquipa(int preferencia);
```

Este método deve bloquear até haver jogadores suficientes (22), retornando o número de equipa atribuído ao jogador (1 ou 2); tem como parâmetro a equipa preferida, ou 0, caso seja indiferente e deve satisfazer, caso seja possível, tal preferência. Deve ser possível ir continuando a formar equipas para sucessivas partidas, mal uma tenha jogadores suficientes.

6.5 Outros exercícios

Exercício 6.5.1. Pretende-se um jogo de adivinha em que cada partida envolve 4 jogadores, que competem para ver quem adivinha primeiro um número gerado aleatoriamente entre 1 e 100. Cada partida é limitada a um minuto e a 100 tentativas de resposta (total para todos os jogadores). Devem ser suportadas várias partidas a decorrer em simultâneo. As interfaces a implementar são:

```
interface Jogo {
    Partida participa();
}

interface Partida {
    String adivinha(int n);
}
```

A operação `participa()` deverá bloquear até poder começar uma partida (4 jogadores a quererem participar), devolvendo o objeto que representa a partida. Sobre este objeto, a operação `adivinha(n)`, usada para jogar, devolve um de: `"GANHOU"` se esta tentativa foi a primeira a acertar (dentro dos limites de tempo e tentativas de resposta); `"PERDEU"` se algum jogador já ganhou; `"TEMPO"` se esgotou o limite de tempo da partida (um minuto); `TENTATIVAS` se foi excedido o limite de tentativas; `"MAIOR"` / `"MENOR"` se o número escondido está acima/abaixo de `n`.

Exercício 6.5.2. Pretende-se um jogo em que cada partida envolve um número variável de jogadores, entre um mínimo de 3 e um máximo de 9. Podem estar várias partidas a decorrer em simultâneo, começando uma partida quando já estão à espera 9 jogadores, ou quando estão pelo menos 3 jogadores à espera e já passaram 5 minutos desde que o primeiro desses jogadores chegou. Implemente as interfaces:

```
interface Jogo {
    Partida participa();
```

```
}

interface Partida {
    boolean aposta(int n, int soma);
}
```

A operação `participa()` deverá bloquear até poder começar uma partida, devolvendo o objeto que a representa. Este objeto é usado pelos jogadores dessa partida, que consiste em cada jogador escolher um número `n` entre 1 e 100 e tentarem adivinhar qual é a soma dos números escolhidos por todos. Para tal, cada jogador faz uma única invocação da operação `aposta(n, soma)`, que deverá devolver `true` se o jogador acertou na soma, ou `false` caso contrário.

Exercício 6.5.3. Considere um sistema de controlo de acesso a R recursos. O sistema deve limitar a T o número máximo de fios de execução a utilizar o conjunto dos recursos, e deve tentar manter os números de utilização em curso equilibrados, quando o limite T estiver atingido (considere $T \gg R$). O sistema deve ainda rejeitar pedidos de uso quando em manutenção. Paro este fim, implemente os seguintes métodos:

```
    int requestResource(int i);
    void releaseResource(int i);
    void startMaintenance();
    void endMaintenance();
```

O `requestResource` tem como parâmetro o número de recurso pretendido (entre 1 e R), e deverá bloquear até o uso ser concedido, devolvendo 0 em caso de sucesso, ou -1 se o sistema estiver/entrar em manutenção; `releaseResource` é invocado quando uma thread completou o uso do recurso correspondente; `startMaintenance` indica início do período de manutenção, devendo bloquear até toda a utilização em curso terminar, e fazendo com que todos os restantes pedidos sejam rejeitados, até ao fim da manutenção, indicada por `endMaintenance`.

Exercício 6.5.4. Pretende-se gerir uma fila de espera para vacinação, assumindo que cada frasco de vacina serve para N utentes e que como tal só pode ser usado quando N utentes estiverem prontos, implementando os seguintes métodos:

```
    void pedirParaVacinar();
    void fornecerFrascos(int frascos)
```

Quando um utente chega e pretende ser vacinado deve invocar `pedirParaVacinar`, que irá bloquear até estarem reunidas as condições de início de vacinação. O método `fornecerFrascos(int frascos)` sinaliza a entrada de mais frascos na unidade de saúde. Considere que o sistema arranca com 0 frascos de vacinas disponíveis. Garanta que os utentes não são ultrapassados no acesso à vacina tendo em conta a ordem de chegada.

Apêndice A

Monitores nativos em Java

Além das primitivas que utilizamos neste texto e que existem desde a versão 5 da linguagem Java em `java.util.concurrent`, abreviado por `j.u.c`, a linguagem em si suporta desde o inicio a programação concorrente com primitivas e sintaxe própria. Em resumo, nesta alternativa existe um trinco e uma variável de condição implícitos em cada objeto, em que:

- O trinco é usado através da palavra-chave `synchronized`, que faz com que o bloco de código indicado, delimitado por `{` e `}`, feche o trinco à entrada e o abra de novo à saída, seja pelo fim do bloco ou através de `return` ou de uma exceção. É por isso equivalente ao idioma usado para os trincos usando `try` e `finally` descrito na Secção 2.2.
- A variável de condição é usada através dos métodos `wait()`, para suspender o fio de execução, e de `notify()` e `notifyAll()`, para sinalizar um ou todos os suspensos.

Com estas primitivas, podemos reescrever o exemplo da Secção 5.3:

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3
4  public class BoundedBufferMon<T> {
5      private Queue<T> q = new LinkedList<>();
6      private final int MAX = 10;
7
8      public synchronized T get() throws InterruptedException {
9          while (q.isEmpty())
10             wait();
11         notifyAll();
12         return q.remove();
13     }
14
15     public synchronized void put(T s) throws InterruptedException {
16         while (q.size() >= MAX)
```

```
17              wait();
18          q.add(s);
19          notifyAll();
20      }
21  }
```

Apesar de ter uma sintaxe mais concisa, não se prefere esta alternativa porque tem várias desvantagens:

- Obriga a que os trincos sejam abertos pelo ordem contrária à que são fechados, uma vez que estão associados a blocos aninhados. Isso limita a possibilidade de utilização de ordenação das operações, como descrito na Secção 2.5.
- Limita cada trinco a uma única variável de condição associada, o que impede a obtenção de soluções que minimizam o efeito tropel, descritivos na Secção 5.4.
- Estas primitivas não dão garantias de ordenação que contribuam para a justiça relativa no acesso à secção crítica.

Apêndice B

Concorrência em C

Os conceitos e técnicas de programação concorrente descritos neste texto podem também ser implementados em C, usando as primitivas de sincronização fornecidas pela biblioteca POSIX. Apresentamos por isso a tradução de alguns exemplos simples para C, que podem ser usados como ponto de partida para o estudo da programação concorrente nesta linguagem.

B.1 Fios de execução

A criação de fios de execução faz-se através da função `pthread_create()`, que recebe como argumentos um apontador para depositar o identificador do fio criado, atributos (que não vamos detalhar aqui), uma função a executar e o respetivo argumento. Uma vez que é recebido apenas um argumento, é prática comum criar uma estrutura de dados que contenha todos os argumentos necessários à função a executar e uma função adicional que lida com essa estrutura, executando a função desejada e assegurando que a memória usada é libertada.

A função `ptrhead_join` permite esperar pela conclusão da fios de execução e, ao contrário do que acontece em Java, permite recolher explicitamente um resultado. Seria possível pois na função `subtask_thread` devolver um apontador que seria depositado na variável indicada como segundo parâmetro de `pthread_join`. Mais uma vez, é importante garantir que a alocação da memória continua válida depois da função terminar e que é depois libertada pelo fio de execução principal. Claro que seria também possível ignorar esta possibilidade e utilizar um parâmetro adicional para apontar para um local onde depositar este resultado.

Podemos assim reescrever em C o segundo exemplo da Secção 1.1:

```
1 #include <stdlib.h>
2 #include <pthread.h>
3
4 static int subtask(int base, int len) {
```

```
5       int c = len;
6       for(int i=base; i<base+len; i++) {
7           for(int j=2; j<i; j++)
8               if (i%j == 0) {
9                   c = c-1;
10                  break;
11              }
12      }
13      return c;
14  }
15
16  struct subtask_params {
17      int base, len;
18  };
19
20  static void* subtask_thread(void* p) {
21      struct subtask_params* params = (struct subtask_params*) p;
22      subtask(params->base, params->len);
23      free(p);
24      return NULL;
25  }
26
27  int main(int argc, char* argv[]) {
28      int N = 10, M = 1000;
29      int len = M/N;
30
31      pthread_t t[N];
32      for(int i=0; i<N; i++) {
33          struct subtask_params* p = malloc(sizeof(*p));
34          p->base = i*len;
35          p->len = len;
36          pthread_create(&t[i], NULL, subtask_thread, (void*)p);
37      }
38
39      for(int i=0; i<N; i++)
40          pthread_join(t[i], NULL);
41
42      return 0;
43  }
```

B.2 Controlo de concorrência

Os trincos e variáveis de condição estão disponíveis através de variáveis dos tipos `pthread_mutex_t` e `pthread_cond_t`, respectivamente, que devem ser inicializadas de forma estática ou então através das funções `pthread_mutex_init` e `pthread_cond_init`. As funções disponíveis são na generalidade idênticas aos métodos correspondentes em Java.

As diferenças principais em relação ao código Java são duas:

- Em vez da associação entre a variável de condição e o trinco ser feita na sua inicialização, é feita apenas no momento da utilização com a função `pthread_cond_wait`, que recebe como argumento o trinco a associar.
- A notificação com as funções `pthread_cond_signal` e `pthread_cond_broadcast` pode ser feita sem o trinco correspondente. De facto, até é recomendado que assim seja para evitar acordar um fio de execução que não pode ainda prosseguir por não dispor do trinco.

Podemos assim reescrever em C o exemplo da Secção 3.2:

```
1
2  #include <string.h>
3  #include <pthread.h>
4
5  #define max 10
6
7  static char* players[max];
8  static int size;
9  static pthread_mutex_t l = PTHREAD_MUTEX_INITIALIZER;
10 static pthread_cond_t c = PTHREAD_COND_INITIALIZER;
11
12 void enter(char* name) {
13     pthread_mutex_lock(&l);
14     while (size >= max)
15         pthread_cond_wait(&c, &l);
16     for(int i=0;i<max;i++)
17         if (players[i]==NULL) {
18             players[i] = name;
19             break;
20         }
21     size++;
22     pthread_mutex_unlock(&l);
23 }
24
25 void leave(char* name) {
26     pthread_mutex_lock(&l);
27     for(int i=0;i<max;i++)
28         if (strcmp(players[i],name)==0) {
29             players[i] = NULL;
30             break;
31         }
32     size--;
33     pthread_mutex_unlock(&l);
34     pthread_cond_signal(&c);
35 }
```

Apêndice C

Concorrência em C++

Embora seja possível usar também a biblioteca POSIX para programação concorrente, a lingagem C++ oferece uma biblioteca própria de mais alto nível que facilita a programação e reduz a possibilidade de erros.

C.1 Fios de execução

Novos fios de execução concorrentes são criados através da classe `std::thread` que recebe como argumento a função a executar e um número variável de parâmetros.

O método `join()` permite esperar pela sua conclusão e, tal como acontece em Java, não permite recolher explicitamente um resultado. É no entanto possível passar por referência uma variável no construtor para recolher esse resultado.

Podemos assim reescrever em C++ o segundo exemplo da Secção 1.1:

```
1  #include <thread>
2
3  static int subtask(int base, int len) {
4      int c = len;
5      for(int i=base; i<base+len; i++) {
6          for(int j=2; j<i; j++)
7              if (i%j == 0) {
8                  c = c-1;
9                  break;
10             }
11     }
12     return c;
13 }
14
15 int main(int argc, char* argv[]) {
16     int N = 10, M = 1000;
17     int len = M/N;
```

```
18
19      std::thread t[N];
20      for(int i=0; i<N; i++) {
21          t[i] = std::thread(subtask, i*len, len);
22
23      for(int i=0; i<N; i++)
24          t[i].join();
25
26      return 0;
27  }
```

C.2 Controlo de concorrência

Os trincos e variáveis de condição estão disponíveis através de instâncias das classes `std::mutex` e `std::condition_variable`, respectivamente, com utilização e comportamento semelhantes às primitivas em Java. Tal como em C, a associação entre a variável de condição e o trinco é feita no momento da utilização com o método `wait()`.

A diferença principal em relação ao código Java ou C é que os trincos não são usados explicitamente através de operações `lock` e `unlock` mas através da definição de um objeto `std::unique_lock` que implicitamente fecha o trinco durante a construção e o liberta ao ser destruído. Garante assim que o trinco está fechado na secção do programa em que o objeto existe e corresponde assim ao idioma descrito na Secção 2.2 ou à palavra-chave `synchronized` do Apêndice A.

Podemos assim reescrever em C++ o exemplo da Secção 3.2:

```
1
2   #include <string>
3   #include <list>
4   #include <mutex>
5   #include <condition_variable>
6
7   class GameCondition {
8       int max;
9
10      std::list<std::string> players;
11      std::mutex l;
12      std::condition_variable c;
13
14   public:
15      void enter(std::string name) {
16          std::unique_lock guard(l);
17          while (players.size() >= max)
18              c.wait(guard);
19          players.push_back(name);
20      }
21
22      void leave(std::string name) {
```

```
23          std::unique_lock guard(l);
24          players.remove(name);
25          guard.unlock();
26          c.notify_all();
27      }
28
29      GameCondition(int max) : max(max) {}
30  };
```

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.