



Turbinando suas aplicações com Multithreading

do básico ao avançado

Professor J

Aplicações, normalmente, executam diversas tarefas ao mesmo tempo. Um programa pode, ao mesmo tempo, disponibilizar telas aos usuários, acessar uma base de dados, implementar controles transacionais e manter um pool de conexões ativas. Sem dúvida, manter o sincronismo de todas essas atividades e completá-las de forma bem-sucedida é uma tarefa ambiciosa. Porém, com as novas potencialidades da programação multithreaded (java.util.concurrent) isso não só é possível, como atinge o estado da arte. Aprenda a utilizar esses poderosíssimos recursos, turbinando suas aplicações Java, independentemente de quais sejam. Para isto, apresentamos este artigo, que cobre do básico ao avançado, tudo que você precisa saber para turbinar seus programas.



Roberto Rubinstein Serson

(roberto@sersoft.com.br)

Possui as certificações Sun SCJP 6, SCWCD 5, SCBCD 5, SCEA 5 e Oracle OCP 11G. Foi instrutor oficial Oracle por quatro anos e instrutor oficial Sun por três anos. É autor dos livros: Certificação Java 6 Teoria – Programador (Brasport 2009), Certificação Java 6 Prática – Programador (Brasport 2009) – Guia de Certificação Java 5 – Brasport 2006 e Oracle Database 10G – Guia do DBA – Novatec 2004. Atualmente é sócio-diretor da Sersoft – Tecnologia em Desenvolvimento de Software.

Uamos supor que você seja responsável por um programa cujo principal objetivo seja a impressão de relatórios. Há necessidade de se produzir três relatórios e são disponibilizadas três impressoras. Dado este cenário podemos visualizar duas possibilidades: a primeira é que todos os relatórios sejam impressos utilizando-se a mesma impressora. A segunda hipótese é que cada relatório seja impresso em uma impressora diferente. Qual cenário mostra-se mais otimizado? Obviamente o segundo. O primeiro cenário, além de subutilizar impressoras, ainda coloca os relatórios em uma fila desnecessária. Em situações como esta, utilizamos a programação paralela, também conhecida como: concorrente, concomitante ou multithreaded, no jargão.

Principal aplicação multithreaded: o modelo produtor/consumidor

Imagine uma prateleira de supermercado. Na medida em que os produtos são consumidos, eles precisam ser repostos. Este modelo pode ser transposto para uma infinidade de situações. Por exemplo: um professor produz conhecimento, o qual é consumido pelos alunos. A revista Mundoj é produzida e consumida por uma infinidade de leitores, ávidos por conhecimento. Uma

usina geradora de energia produz energia que é consumida por fábricas e usuários domésticos. Tente colocar estas ideias nos seus programas, na sua forma de pensar e de agir. Você perceberá como este modelo se encaixa bem em diversas situações.

Em um relacionamento produtor/consumidor de múltiplas threads, várias threads produtoras geram os dados e os colocam em um objeto compartilhado – chamado buffer. Diversas threads consumidoras lêem os dados que estão no buffer.

Agora, façamos um esforço para compreendermos os problemas que podem ser originados a partir deste modelo. Primeiramente podemos citar que o buffer possui um tamanho fixo. Tal tamanho não pode ser excedido. Assim, em uma situação extrema, em que haja muito mais produtores, ou que eles atuem de forma mais rápida do que os consumidores, o buffer enche e não há mais local para colocar produtos.

Outra situação corriqueira é a do buffer vazio. Ela é absolutamente análoga à situação anteriormente descrita, só que agora, ao invés de muitos produtores, temos mais consumidores. Isto faz com que o buffer fique rapidamente vazio e os consumidores continuem a tentar retirar elementos do buffer. Temos que ser capazes de evitar ambos extremos.

No momento em que a thread produtora colocar novos dados no buffer, ela deve invocar o método `signal()`, para permitir que a thread consumidora prossiga consumindo novos elementos.

Vamos considerar os erros de lógica que podem surgir, caso o acesso entre as diversas threads não seja sincronizado. O próximo exemplo implementa um relacionamento produtor/consumidor em que uma thread-produtor grava números de um a dez, em um buffer compartilhado.

Um buffer é uma posição compartilhada de memória, entre duas threads (nesse caso). A thread-consumidor lê os dados do buffer compartilhado e os exibe. A saída do programa mostra os valores que a produtora grava e os valores que a consumidora lê.

Cada valor que a produtora grava no buffer compartilhado deve ser consumido exatamente uma vez pela consumidora. No entanto, as threads, nesse exemplo, não são sincronizadas. Dessa forma, os dados podem ser perdidos se a produtora colocar novos dados no buffer, antes da consumidora consumir os dados anteriores.

Além disso, os dados podem ser duplicados incorretamente, se a consumidora consumir os dados outra vez, antes de a produtora produzir o próximo valor. Para mostrar essas possibilidades, a consumidora, no exemplo, mantém um total de todos os valores que ela lê. A produtora, conforme já dito, produz valores de um a dez.

Se a consumidora ler cada valor produzido uma – e apenas uma vez – o total será 55. Entretanto, se executar esse programa várias vezes, você verá que o total nem sempre é 55. Para enfatizar a questão, as threads produtoras e consumidoras, no exemplo, dormem por intervalos regulares, de até três segundos, entre a execução de suas tarefas. Dessa forma, não sabemos exatamente quando a produtora tentará gravar um novo valor, nem quando a thread consumidora tentará ler um valor.

Vamos à montagem do exemplo. O programa consiste na interface `Buffer` e em quatro classes: `Produtor`, `Consumidor`, `BufferExemplo` e `BufferCompartilhadoTeste`. A interface `Buffer` (cujo código é mostrado na Listagem 1) declara

os métodos `get()` e `set()`, que um `Buffer` deve implementar, no sentido de permitir à thread `Produtor` colocar um valor no `Buffer` e à thread `Consumidor` recuperar um valor desse mesmo `Buffer`.

Listagem 1. Interface `Buffer`, que contém os métodos `set` e `get`.

```
public interface Buffer {
    public void set( int valor );
    public int get();
}
```

A classe `Produtor` implementa a interface `Runnable`, permitindo que ela seja executada em uma thread separada, o construtor inicializa a referência `Buffer`, o que, de fato, será realizado no método `main(String args[])`. Como veremos, esse é um objeto `BufferExemplo`, que implementa a interface `Buffer`, sem sincronizar o acesso ao objeto compartilhado.

A thread `Produtor` executa as tarefas especificadas no método `run()`. Cada iteração do loop invoca o método `Thread.sleep()`, para colocar a `Produtora` no estado de espera cronometrada, por um intervalo randômico de zero a três segundos. Observe o código da classe `Produtor` na Listagem 2.

Listagem 2. Classe `Produtor`, que compartilha o `buffer` e executa o método `set`.

```
import java.util.Random;

public class Produtor implements Runnable {
    private static Random gerador = new Random();
    private Buffer localizacaoCompartilhada;
    public Produtor( Buffer compartilhado ) {
        localizacaoCompartilhada = compartilhado;
    }
    public void run() {
        for ( int contador = 1; contador <= 10; contador++ ) {
            try {
                Thread.sleep( gerador.nextInt( 3000 ) );
                localizacaoCompartilhada .set( contador );
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }
        System.out.printf( "\n%s\n%s\n", "Produtor produz!!!!",
            "Fim do Produtor." );
    }
}
```

Quando a thread acordar, o `Buffer` é atualizado com o valor do contador. Um total de todos os valores produzidos é mantido na variável local soma. Quando as iterações terminarem, é exibida uma mensagem indicando que a thread `Produtor` foi concluída, ou seja, completou sua tarefa.

Observe que qualquer método chamado a partir do método `run()` de uma thread é executado como parte dessa thread. De fato, cada thread tem sua

própria pilha de chamadas de método. Isso se tornará relevante quando adicionarmos sincronização ao relacionamento produtor/consumidor.

A classe Consumidor também implementa a interface Runnable, permitindo que o consumidor seja executado concorrentemente com o produtor. O construtor inicializa a referência Buffer compartilhada, com um objeto que implementa a interface Buffer, criado no método main(String args[]). Esse é o mesmo objeto BufferExemplo que é utilizado para inicializar o objeto produtor, portanto, as duas threads compartilham o mesmo objeto. O código da classe Consumidor está na Listagem 3.

A thread Consumidor, nesse programa, realiza as tarefas especificadas pelo método run(). O loop itera dez vezes. A cada iteração, o loop invoca o método Thread.sleep(), que coloca a thread Consumidor em estado de espera sincronizada, randomicamente, entre zero e três segundos.

Em seguida, o método get(), de Buffer, é utilizado para recuperar o valor no buffer compartilhado, então adiciona o valor à variável soma. Na sequência, o total de valores consumidos até o momento é exibido. Quando o loop estiver concluído, a soma é exibida e o método run() termina, indicando que o Consumidor completou sua tarefa. Depois que as threads entram no estado terminado (dead), o programa é encerrado.

O método Thread.sleep() foi usado no corpo do método run(), das classes Produtor e Consumidor, para ilustrar o fato de que, em aplicações com múltiplas threads, é imprevisível o momento em que cada thread realizará sua tarefa e, por quanto tempo ela realizará a tarefa, quando estiver de 'posse' do processador.

Listagem 3. Classe Consumidor, que compartilha o buffer e executa o método get.

```
import java.util.Random;

public class Consumidor implements Runnable {
    private static Random gerador = new Random();
    private Buffer localizacaoCompartilhada;
    public Consumidor( Buffer compartilhado ) {
        localizacaoCompartilhada = compartilhado;
    }
    public void run() {
        int soma = 0;
        for ( int contador = 1; contador <= 10; contador++ ) {
            try {
                Thread.sleep( gerador.nextInt( 3000 ) );
                soma += localizacaoCompartilhada.get();
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }
        System.out.printf( "\n%s%d\n ", "Fim do Consumidor.
        Valor da soma: ", soma );
    }
}
```

Usualmente, questões envolvendo agendamento de threads são de responsabilidade do sistema operacional e, portanto, estão além do controle do desenvolvedor Java. Nesse programa, as tarefas de nossa thread são bem simples: o produtor grava os valores de um a dez no buffer e adiciona cada valor adicionado à variável soma e o consumidor lê dez valores e adiciona cada valor adicionado à variável soma. Na medida em que sofisticarmos nosso exemplo, esta imprevisibilidade se reduzirá a zero.

Caso não houvesse a invocação do método Thread.sleep() e se produtor executasse primeiro, considerando os processadores ultrarrápidos que temos hoje, o produtor possivelmente terminaria sua tarefa antes do consumidor ter uma chance de executar. Caso o consumidor executasse primeiro, ela provavelmente consumiria o valor -1 dez vezes e terminaria antes que produtor pudesse colocar o primeiro valor no Buffer.

A classe BufferExemplo implementa a interface Buffer. Um objeto dessa classe é compartilhado entre produtor e consumidor. O atributo buffer é declarado como int e recebe o valor -1. Isso pode ser observado na Listagem 4.

Esse valor é utilizado para demonstrar o caso em que o consumidor tenta consumir um valor antes que o produtor coloque algum valor em buffer. O método set() simplesmente atribui seu argumento a buffer ao passo que o método get() retorna o valor de buffer.

Listagem 4. Classe BufferExemplo, que cria, de fato, o Buffer e os métodos set e get.

```
public class BufferExemplo implements Buffer {
    private int buffer = -1;
    public void set( int valor ) {
        System.out.printf( "Produtor grava: %t%d\n", valor );
        buffer = valor;
    }
    public int get() {
        System.out.printf( "Consumidor lê: %t%d\n", buffer );
        return buffer;
    }
}
```

A classe BufferExemploTeste (presente na Listagem 5) contém o método main(String args[]), que inicia a aplicação. Um ExecutorService é criado com duas threads, para executar produtor e consumidor. Um objeto BufferExemplo é instanciado e sua referência é atribuída à variável local localizacaoCompartilhada.

Esse objeto armazena os dados que serão compartilhados entre as threads produtor e consumidor. Observe que os construtores de produtor e consumidor recebem exatamente essa mesma referência (localizacaoCompartilhada), assim, cada objeto é inicializado com uma referência ao mesmo Buffer.

Essas linhas também carregam, implicitamente, as threads e invocam o método run(), de cada Runnable. Ao final, o método shutdown() é chamado, de modo que o programa possa terminar, quando as

threads produtor e consumidor completarem suas tarefas. Quando o método `main(String args[])` terminar, a thread principal de execução é finalizada.

Listagem 5. Classe `BufferExemploTeste`, que cria, de fato, as threads produtor e consumidor.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BufferExemploTeste {

    public static void main( String[] args ) {

        ExecutorService teste = Executors.newFixedThreadPool( 2 );

        Buffer localizacaoCompartilhada= new BufferExemplo();

        try {

            teste.execute( new Produtor( localizacaoCompartilhada ) );

            teste.execute( new Consumidor( localizacaoCompartilhada ) );

        } catch ( Exception exception ) {

            exception.printStackTrace();

        }

        teste.shutdown();

    }

}
```

Saída:

```
Consumidor lê:  -1
Produtor grava:  1
Produtor grava:  2
Produtor grava:  3
Produtor grava:  4
Consumidor lê:  4
Produtor grava:  5
Produtor grava:  6
Consumidor lê:  6
Consumidor lê:  6
Produtor grava:  7
Produtor grava:  8
Produtor grava:  9
Consumidor lê:  9
Consumidor lê:  9
Produtor grava:  10

Produtor produz!!!!
Fim do Produtor.
Consumidor lê:  10
Consumidor lê:  10
Consumidor lê:  10
Consumidor lê:  10
```

Fim do Consumidor.Valor da soma:73

✧ Análise

A interface `Buffer` define que quem implementá-la deve implementar os métodos `set()` e `get()`. Note que a classe `BufferExemplo` implementa essa interface. Ela realiza três tarefas: inicializa o atributo `buffer` com `-1`, define o método `set()`, que exibe a mensagem "Produtor grava: ", recebe um valor e o atribui à `Buffer`, além de definir o método `get()`. O método `get()` exibe a mensagem "Consumidor lê: " e retorna o valor de `Buffer`.

Vamos analisar o que o produtor faz. Em primeiro lugar, o produtor implementa `Runnable`, indicando que possuirá o método `run()` e que será encapsulado em uma thread. A classe possui dois atributos: `gerador` e `localizacaoCompartilhada`.

O construtor recebe uma referência `Buffer`. Isso significa que `localizacaoCompartilhada` poderá fazer uso dos métodos `set()` e `get()`, ou seja, colocar e tirar elementos do `Buffer`. Sendo um produtor, a expectativa, que se confirma no corpo do método `main(String args[])` é que o método `set()` seja utilizado.

No método `run()`, uma variável soma é definida. Na sequência, um loop `for`, que conta de um a dez, realiza dez iterações. Um bloco `try` é posicionado logo após o início do loop. Sua função é controlar a chamada `sleep()` de `Thread`, avaliando se ela será ou não bem-sucedida. Assim, todo vez que entra no loop, o produtor dorme de zero a três segundos. A cada iteração o valor em `localizacaoCompartilhada` é configurado.

Considerando a execução desse programa, lembre-se de que gostaríamos que a thread produtor executasse primeiro e que cada valor produzido fosse consumido exatamente uma vez pelo consumidor.

Assim, o exemplo passado demonstra, claramente, que o acesso a um objeto compartilhado por threads concorrentes deve ser cuidadosamente controlado, pois, do contrário, o programa pode – e irá – produzir resultados incorretos.

✧ Conceito de sincronização

Normalmente, diversas threads podem manipular um mesmo objeto em memória. Quando isso acontece, esse objeto é modificado por essas threads e podem ocorrer resultados indeterminados, a menos que o objeto compartilhado seja administrado de forma adequada.

Caso uma thread esteja em um processo de atualização do objeto compartilhado e outra thread tentar modificá-lo, é provável que parte do objeto reflita as informações de uma thread, enquanto outra parte reflita informações relacionadas à outra thread.

Quando isso ocorre, o comportamento do programa não é confiável, podendo produzir resultado incorretos. De qualquer forma, não há mensagem de erro indicando que o objeto compartilhado foi manipulado de forma incorreta.

Uma das soluções é fornecer a uma thread por vez o código de acesso exclusivo que atualiza o objeto compartilhado. Durante esse tempo, outras threads que anseiem por manipular o objeto em questão são mantidas em espera.

Quando a thread com o acesso exclusivo ao objeto terminar de invocá-lo, uma das threads que foi mantida em espera tem a permissão de prosseguir. Dessa forma, toda thread que acessa o objeto compartilhado exclui todas as outras threads de fazerem isso, concomitantemente.

Esse procedimento é chamado de exclusão mútua. Ele permite ao desenvolvedor realizar a sincronização de threads. A sincronização de threads coordena o acesso aos dados compartilhados por múltiplos threads concorrentes.

O Java utiliza locks para realizar a sincronização. Qualquer objeto pode conter um objeto que implementa a interface Lock (pacote `java.util.concurrent.locks`). Trata-se de um exemplo claro de composição. Uma thread invoca o método `lock()` de Lock, para obter o bloqueio.

Uma vez que um Lock foi obtido por uma thread, o objeto Lock não permitirá que outra thread obtenha o recurso até que a primeira thread libere o Lock. A liberação é realizada por meio da chamada ao método `unlock`, também da classe Lock.

Caso haja diversas threads tentando chamar o método `lock()`, no mesmo objeto Lock, concorrentemente, somente uma thread poderá obter o bloqueio por vez. As demais threads, que tentarem obter o Lock, contido no mesmo objeto serão colocadas em estado de espera, em decorrência desse bloqueio.

Quando uma thread invocar o método `unlock`, o bloqueio sobre o objeto será liberado e a thread, em espera, com mais alto nível de prioridade, que tentar desbloquear o objeto, prosseguirá. A classe `ReentrantLock` (pacote `java.util.concurrent.locks`) é uma implementação básica da interface Lock.

O construtor de um `ReentrantLock` aceita um argumento booleano que especifica se o bloqueio tem uma diretiva de imparcialidade. Caso esse argumento seja configurado como `true`, a diretiva de imparcialidade do `ReentrantLock` determina que a thread que aguarda a mais tempo vai obter o bloqueio, quando ele estiver disponível. Se esse argumento for `false`, não é garantido que a thread, em espera há mais tempo irá obter o bloqueio.

Utilizar um Lock com uma diretiva relativamente justa evita o adiamento indefinido, mas também pode reduzir significativamente a eficiência geral de um programa. Por causa da grande diminuição de desempenho, os bloqueios imparciais só são necessários em circunstâncias extremas.

Caso uma thread que possua um bloqueio sobre um objeto determine que não é possível continuar sua tarefa, até que uma condição seja satisfeita, a thread pode aguardar em uma variável de condição. Isso dispensa a thread de disputa pelo processador, colocando-a em uma fila de espera pela variável de condição e liberando o bloqueio.

As variáveis de condição devem estar associadas a um Lock e são criadas por meio da invocação ao método `newCondition`, da classe Lock. Esse método retorna um objeto que implementa a interface Condition (pacote `java.util.concurrent.locks`).

Para esperar uma variável de condição, a thread pode invocar o método `await()`, de Condition. Isso libera imediatamente o Lock associado e coloca a thread, em execução, no estado de espera dessa Condition. Outras threads, então, podem tentar obter o Lock.

Quando uma thread executável completar uma tarefa e determinar que a thread em espera pode continuar, a thread executável pode invocar o método `Condition.signal()`, para permitir que uma thread

no estado de espera dessa Condition retorne ao estado executável.

Nesse ponto, a thread que fez a transição do estado de espera para o estado executável pode tentar readquirir o Lock no objeto. Mesmo se fosse capaz de readquirir o Lock, a thread ainda poderia não ser capaz de realizar sua tarefa, nesse momento. Esse é o caso em que a thread pode chamar o método `await()` para liberar o Lock e entrar novamente no estado de espera.

Caso diversas threads estiverem no estado de espera de uma Condition, quando signal for invocado, a implementação-padrão de Condition sinaliza a thread de espera mais longa, para mudar para o estado executável.

Se uma thread chamar o método `Condition.signalAll()`, todas as threads que esperam essa condição mudam para o estado executável e tornam-se elegíveis para readquirir o Lock. Apenas uma dessas threads pode obter o Lock do objeto por vez. Outras threads que tentarem adquirir o mesmo Lock aguardarão até que o Lock se torne disponível novamente.

Caso o Lock não tenha sido criado com uma diretiva de imparcialidade, a thread de espera mais longa irá adquirir o Lock. Quando uma thread concluir sua tarefa, em um objeto compartilhado, ele deve chamar o método `unlock()` para liberar o Lock.

Se múltiplas threads esperarem pelo valor da variável de condição, uma thread separada pode chamar o método `Condition.signalAll()`, como uma salvaguarda para garantir que todas as threads em espera tenham outra oportunidade de realizar suas tarefas. Se isso não for feito, o adiamento indefinido ou impasse poderá ocorrer.

O bloqueio que acontece com a execução dos métodos `lock()` e `unlock()` poderia levar a um impasse se os bloqueios nunca fossem liberados. As chamadas para o método `unlock` devem ser colocadas em blocos `finally` para assegurar que os bloqueios sejam liberados e evitar esses tipos de impasses. Os bloqueios devem durar o mínimo possível.

A sincronização, para alcançar a precisão em programas de múltiplas threads, pode tornar a execução de programas mais lenta, como resultado de overhead de thread e da transição frequente de threads entre os estados de espera e executável. Não há, entretanto, muito a dizer sobre programas multiencaçados altamente eficientes, mas incorretos.

Vale a pena ressaltar que é um erro se uma thread emitir um `await`, um `signal` ou um `signalAll` em uma variável de condição sem adquirir o bloqueio dessa variável de condição. Isso causa uma `IllegalMonitorStateException`.

✎ Produtor/consumidor – com sincronização

O próximo exemplo demonstra a utilização de um buffer com sincronização. Nesse caso, o consumidor consome corretamente um valor apenas, depois de o produtor ter produzido um valor; esta, por sua vez, produz corretamente um novo valor, apenas depois de o consumidor ter consumido o valor produzido anteriormente.

Vamos alterar somente a classe que implementa a interface Buffer (`BufferExemplo` – presente na Listagem 6). A interface Buffer é a

mesma e as classes Produtor, Consumidor e BufferExemploTeste também são as mesmas. Essa abordagem permite demonstrar que as threads que acessam o objeto compartilhado estão cientes de que estão sendo sincronizadas.

🔍 Análise

Vamos analisar as saídas. Observe que cada inteiro produzido é consumido exatamente uma vez. Além disso, nenhum valor é perdido e nenhum valor é consumido mais de uma vez. A sincronização e as variáveis de condição garantem que o produtor e o consumidor não podem realizar suas tarefas, a não ser que seja o momento certo para isso.

Produtor produz primeiro, o consumidor deve aguardar se o produtor não tiver produzido desde que o consumidor realizou o consumo pela última vez. Analogamente, o produtor deve esperar se o consumidor ainda não tiver consumido o valor que o produtor produziu mais recentemente.

Caso você decida contar o número de execuções, notará que foram ao todo 20:: dez do produtor e dez do consumidor. A soma dos elementos de ambos, em decorrência direta da sincronização, é 55, que é a soma dos valores de um a dez.

🔍 Produtor/consumidor – com sincronização e utilização do Buffer circular

A última aplicação vista utiliza a sincronização de thread, para assegurar que duas threads manipulem corretamente os dados em um buffer compartilhado. No entanto, a aplicação não tem como apresentar um desempenho ótimo.

Caso duas threads operem a diferentes velocidades, uma delas gastará mais tempo na espera. Por exemplo, no último programa uma única variável de inteiro foi manipulada por duas threads. Se o produtor produzir valores mais rapidamente do que o consumidor tem condições de consumir, o produtor terá que esperar pelo consumidor.

De maneira análoga, se o consumidor consumir valores mais rapidamente que o produtor, é ele quem terá que aguardar.

Mesmo quando há threads que operam nas mesmas velocidades relativas, essas threads podem ficar ocasionalmente 'fora de sintonia' por um período de tempo, fazendo com que uma delas espere pela outra. É impossível fazer suposições a respeito da velocidade relativa de threads concorrentes.

As interações que acontecem com o sistema operacional, a rede, o usuário e outros fatores podem fazer com que as threads passem a operar a diferentes velocidades. Quando isso acontece, as threads esperam. Quando threads esperam excessivamente, os programas tornam-se menos eficientes, os programas interativos com usuários tornam-se menos responsivos e as aplicações sofrem esperas maiores.

No sentido de se minimizar a quantidade de tempo de espera por threads que compartilham os mesmos recursos e operam nas mesmas velocidades médias, podemos implementar um buffer circular. O buffer circular fornece espaço de buffer extra, em que a produtora pode colocar valores e a partir da qual a consumidora pode recuperá-los.

Listagem 6. Classe BufferExemplo, que implementa os métodos set e get.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
public class BufferExemplo implements Buffer {
    private Lock lockDeAcesso = new ReentrantLock();
    private Condition podeGravar = lockDeAcesso.newCondition();
    private Condition podeLer = lockDeAcesso.newCondition();
    private int buffer = -1;
    private boolean ocupado = false;
    public void set( int valor ) {
        lockDeAcesso.lock();
        try {
            while ( ocupado ) {
                System.out.println( "Produtor tenta gravar." );
                mostrarEstado( "Buffer Cheio. Produtor aguarda." );
                podeGravar.await();
            }
            buffer = valor;
            ocupado = true;
            mostrarEstado( "Produtor grava: " + buffer );
            podeLer.signal();
        } catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } finally {
            lockDeAcesso.unlock();
        }
    }
    public int get() {
        int valorLido = 0;
        lockDeAcesso.lock();
        try {
            while ( !ocupado ) {
                System.out.println( "Consumidor tenta ler." );
                mostrarEstado( "Buffer vazio. Consumidor aguarda." );
                podeLer.await();
            }
            ocupado = false;
            valorLido = buffer;
            mostrarEstado( "Consumidor lê: " + valorLido );
            podeGravar.signal();
        } catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } finally {
            lockDeAcesso.unlock();
        }
        return valorLido;
    }
    public void mostrarEstado( String operacao ) {
        System.out.printf( "%-40s%d\t\t\t\t\t", operacao, buffer, ocupado );
    }
}
```

Saída:

```

Produtor grava: 1      1      true
Consumidor lê: 1      1      false
Produtor grava: 2      2      true
Consumidor lê: 2      2      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  2      false
Produtor grava: 3      3      true
Consumidor lê: 3      3      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  3      false
Produtor grava: 4      4      true
Consumidor lê: 4      4      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  4      false
Produtor grava: 5      5      true
Consumidor lê: 5      5      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  5      false
Produtor grava: 6      6      true
Consumidor lê: 6      6      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  6      false
Produtor grava: 7      7      true
Consumidor lê: 7      7      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  7      false
Produtor grava: 8      8      true
Consumidor lê: 8      8      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  8      false
Produtor grava: 9      9      true
Consumidor lê: 9      9      false
Consumidor tenta ler.
Buffer vazio. Consumidor aguarda.  9      false
Produtor grava: 10     10     true
Produtor produziu!!!!
Fim do Produtor.
Consumidor lê: 10     10     false
Valores lidos por consumidor totalizam: 55.
Fim do Consumidor.

```

Vamos partir do pressuposto que o buffer é implementado como um array e que o produtor e o consumidor funcionem bem, desde o início do array. Quando qualquer thread alcançar o fim do array, ela simplesmente retorna ao primeiro elemento do array, para realizar sua próxima tarefa.

Caso o produtor, temporariamente, produza valores mais rápido do que o consumidor consegue consumi-los, o produtor pode escrever valores adicionais, no espaço extra do buffer (caso haja algum disponível). Essa potencialidade permite ao produtor realizar sua tarefa, mesmo que o consumidor não esteja pronto para receber o valor atual sendo produzido.

A “grande sacada” para utilizar um buffer circular com um produtor e um consumidor, que operam quase na mesma velocidade é fornecer ao buffer posições suficientes para tratar a produção extra antecipadamente. Se, por um determinado período de tempo, determinarmos que o produtor produz recorrentemente até três valores a mais do que o consumidor consegue consumir, podemos fornecer um buffer de pelo menos três posições, para tratar a produção extra.

Vamos alterar somente a classe que implementa a interface Buffer (BufferExemplo – presente na Listagem 7). A interface Buffer é a mesma e as classes produtor, consumidor, além de BufferExemplo-Teste também são as mesmas.

Análise

Toda vez que o produtor grava um valor, ou o consumidor lê um valor, o programa gera a saída da ação realizada (uma leitura ou uma gravação), em conjunto com o conteúdo do buffer e a localização dos índices de gravação e leitura.

Na primeira execução, o produtor grava primeiro o valor 1. O buffer então contém o valor 1 na primeira posição e o valor -1 (valor-padrão) nas outras duas posições. O índice de gravação é atualizado para a segunda posição, enquanto o índice de leitura permanece na primeira posição.

Na sequência, o consumidor lê 1. O buffer contém os mesmos valores, no entanto, o índice de leitura foi atualizado na segunda posição. O consumidor então tenta ler novamente, mas o buffer está vazio e ele tem que aguardar. Observe que durante essa execução foi necessário que a thread aguardasse apenas uma vez.

Produtor/consumidor – com sincronização e utilização do buffer circular com a classe ArrayBlockingQueue – O estado da arte

O Java 5 inclui uma classe de buffer circular completamente implementada chamada ArrayBlockingQueue, que pertence ao pacote java.util.concurrent. Essa classe implementa a interface BlockingQueue. Essa interface, por sua vez, implementa a interface Queue, que é uma Coleção. BlockingQueue declara os métodos put() e take(), os quais são análogos aos métodos offer() e pool() de Queue.

Isso significa que o método put() colocará um elemento final de BlockingQueue e aguardará, caso a fila esteja cheia. O método take(), de forma contrária, retira um elemento do topo de BlockingQueue e o armazenará, caso a fila esteja vazia.

A classe ArrayBlockingQueue utiliza um array. Isso faz com que a estrutura de dados tenha um tamanho fixo e isso significa que a fila não poderá ser aumentada, para acumular elementos extras. A classe ArrayBlockingQueue encapsula todas as funcionalidades da classe BufferCircular, criada no exemplo anterior.

O próximo programa mostra um produtor e um consumidor acessando um buffer circular (nesse caso um ArrayBlockingQueue), com sincronização. A classe BufferBlocking (presente na Listagem 8) implementa a interface Buffer e contém um atributo ArrayBlockingQueue, que armazena objetos Integer. Lembre-se de que uma coleção somente armazena referências. Escolhendo implementar Buffer, nossa aplicação pode reutilizar as classes produtor e consumidor, sem a necessidade de realizar nenhuma alteração.

Primeiramente, perceba a simplicidade da classe BufferBlocking. Nossa única necessidade é definir o tamanho do buffer, no sentido de se minimizar as esperas. Observe também que ArrayBlockingQueue é uma classe genérica. Isso significa que a fila pode ser composta por referências de quaisquer tipos. Nosso exemplo utiliza uma fila

Listagem 7. Classe BufferExemplo, que implementa os métodos set e get – Buffer Circular.

```

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
public class BufferExemplo implements Buffer {
    private Lock accessLock = new ReentrantLock();
    private Condition podeGravar = accessLock.newCondition();
    private Condition podeLer = accessLock.newCondition();
    private int[] buffer = { -1, -1, -1 };
    private int buffersOcupados = 0;
    private int gravarIndice = 0;
    private int lerIndice = 0;
    public void set( int valor ) {
        accessLock.lock();
        try {
            while ( buffersOcupados == buffer.length ) {
                System.out.printf( "Todos os buffers cheios. Produtor aguarda.\n" );
                podeGravar.await();
            }
            buffer[ gravarIndice ] = valor;
            gravarIndice = ( gravarIndice + 1 ) % buffer.length;
            buffersOcupados++;
            mostrarEstado( "Produtor grava: " + valor );
            podeLer.signal();
        } catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } finally {
            accessLock.unlock();
        }
    }
    public int get() {
        int valorLido = 0;
        accessLock.lock();
        try {
            while ( buffersOcupados == 0 ) {
                System.out.printf( "Todos os buffers estão vazios. Consumidor aguarda.\n" );
                podeLer.await();
            }
            valorLido = buffer[ lerIndice ];
            lerIndice = ( lerIndice + 1 ) % buffer.length;
            buffersOcupados--;
            mostrarEstado( "Consumidor lê: " + valorLido );
            podeGravar.signal();
        } catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } finally {
            accessLock.unlock();
        }
        return valorLido;
    }
    public void mostrarEstado( String operacao ) {
        System.out.printf( "%s%s%d\n", operacao, " Buffers ocupados: ", buffersOcupados );
    }
}

```

Saída:

```

Produtor grava: 1 Buffers ocupados: 1
Consumidor lê: 1 Buffers ocupados: 0
Todos os buffers estão vazios. Consumidor aguarda.
Produtor grava: 2 Buffers ocupados: 1
Consumidor lê: 2 Buffers ocupados: 0
Produtor grava: 3 Buffers ocupados: 1
Consumidor lê: 3 Buffers ocupados: 0
Produtor grava: 4 Buffers ocupados: 1
Consumidor lê: 4 Buffers ocupados: 0
Produtor grava: 5 Buffers ocupados: 1
Consumidor lê: 5 Buffers ocupados: 0
Produtor grava: 6 Buffers ocupados: 1
Consumidor lê: 6 Buffers ocupados: 0
Todos os buffers estão vazios. Consumidor aguarda.
Produtor grava: 7 Buffers ocupados: 1
Consumidor lê: 7 Buffers ocupados: 0
Produtor grava: 8 Buffers ocupados: 1
Produtor grava: 9 Buffers ocupados: 2
Consumidor lê: 8 Buffers ocupados: 1
Produtor grava: 10 Buffers ocupados: 2

Produtor produz!!!!
Fim do Produtor.
Consumidor lê: 9 Buffers ocupados: 1
Consumidor lê: 10 Buffers ocupados: 0

Valores lidos por consumidor totalizam: 55.
Fim do Consumidor.

```


de objetos Integer. Outro fato marcante é que a classe de teste é a mesma do exemplo anterior.

A presença dos métodos set e get também deve ser notada. O método set invoca o método put de ArrayBlockingQueue. Essa invocação cria um bloqueio até que haja espaço no buffer, para colocar o valor. Quando o valor é colocado na fila o tipo primitivo int é convertido, implicitamente, em uma referência Integer. Esse processo, também novidade do Java 5, é denominado autoboxing.

O método get() invoca o método take(). Mais uma vez, essa chamada cria um bloqueio até que haja um elemento no buffer a ser removido. Nenhum desses métodos requer um objeto Lock ou Condition. ArrayBlockingQueue trata de toda a sincronização para nós. Obviamente, e conforme já citamos antes, a quantidade de código nesse programa diminui significativamente. Então podemos observar um exemplo maravilhoso de orientação a objetos, que envolve encapsulamento e reutilização de software, a um só tempo.

❖ Considerações finais

O foco do artigo é a apresentação de multithreading, em uma perspectiva evolutiva. Para tanto, foram apresentados quatro modelos produtor/consumidor. O primeiro, sem nenhum compromisso com sincronização, faz com que o produtor atue de forma totalmente independente do consumidor. Vimos que, na prática, não funciona. O segundo modelo, que implementa sincronização, faz com que o consumidor só consuma se o produtor já produziu e vice-versa. O terceiro modelo implementa o modelo produtor/consumidor utilizando o conceito de buffer circular, o que diminui significativamente as esperas. O quarto modelo encapsula toda lógica vista anteriormente, por meio da utili-

zação da classe `ArrayBlockingQueue` (pacote `java.util.concurrent`), facilitando sobremaneira o nosso trabalho. Teste cada um dos modelos diversas vezes e tente compreendê-los analisando as diferentes saídas. 

Listagem 8. Classe `BufferExemplo`, que implementa os métodos `set` e `get`, com `ArrayBlockingQueue`.

```
import java.util.concurrent.ArrayBlockingQueue;

public class BufferBlocking implements Buffer {
    private ArrayBlockingQueue<Integer> buffer;

    public BufferBlocking() {
        buffer = new ArrayBlockingQueue<Integer>( 3 );
    }

    public void set( int value ) {
        try {
            buffer.put( value );
            System.out.printf( "%s%2d\t%s%\n", "Produtor grava: ", value,
                               "Buffers ocupados: ", buffer.size() );
        } catch ( Exception exception ) {
            exception.printStackTrace();
        }
    }

    public int get() {
        int readValue = 0;
        try {
            readValue = buffer.take();
            System.out.printf( "%s %2d\t%s%\n", "Consumidor lê: ", readValue,
                               "Buffers ocupados: ", buffer.size() );
        } catch ( Exception exception ) {
            exception.printStackTrace();
        }
        return readValue;
    }
}
```

Saída:

```
Produtor grava: 1 Buffers ocupados: 1
Produtor grava: 2 Buffers ocupados: 2
Consumidor lê: 1 Buffers ocupados: 1
Consumidor lê: 2 Buffers ocupados: 0
Produtor grava: 3 Buffers ocupados: 1
Produtor grava: 4 Buffers ocupados: 2
Consumidor lê: 3 Buffers ocupados: 1
Consumidor lê: 4 Buffers ocupados: 0
Produtor grava: 5 Buffers ocupados: 1
Produtor grava: 6 Buffers ocupados: 2
Produtor grava: 7 Buffers ocupados: 3
Consumidor lê: 5 Buffers ocupados: 2
Produtor grava: 8 Buffers ocupados: 3
Consumidor lê: 6 Buffers ocupados: 2
Consumidor lê: 7 Buffers ocupados: 1
Produtor grava: 9 Buffers ocupados: 2
Consumidor lê: 8 Buffers ocupados: 1
Consumidor lê: 9 Buffers ocupados: 0
Consumidor lê: 10 Buffers ocupados: 0
```

Valores lidos por consumidor totalizam: 55.
Fim do Consumidor.
Produtor grava: 10 Buffers ocupados: 0

Produtor produz!!!!
Fim do Produtor.

Referências

- **Artigo** – Programação Concorrente – Mundoj – número 23
- *Java Concurrency in Practice* Brian Goetz, et al Addison-Wesley, Publicado em 2006, ISBN 0321349601
- **Concurrency: State Models and Java Programs, 2nd Edition** Jeff Magee, et al Wiley, Publicado em 2006, 2nd edition, ISBN 0470093552
- Brian Goetz – **Definição de programação concorrente** <http://www-128.ibm.com/developerworks/java/library/j-jtp10264/> – Publicado em 2005
- **O que é programação multithreading?** – Wikipedia – Disponível em: < <http://en.wikipedia.org/wiki/Multithreading> >
- **Java – The Complete Reference – Seventh Edition** – Herbert Schildt 2007 – Editora Osborne
- **Java Programming Language** – Arnold, Ken, James Gosling e David Holmes. The, Terceira edição. Addison-Wesley, Boston 2000 ISBN: 0201704331
- **A Arte do Java** – Autor: Schildt, Herbert / Holmes, James Editora: Campus ISBN: 8535213317
- **Beyond Java** – Autor: Tate, Bruce A. Editora: O'Reilly ISBN: 0596100949
- [DL1] Doug Lea, *Concurrent Programming in Java, Second Edition*, Addison-Wesley, 1996
- [DL2] Doug Lea, *JSR-166 Components*, PDF slides. gee.cs.oswego.edu
- [IBM1] IBM DeveloperWorks. *Java 5.0 Concurrency Tutorial* www.ibm.com/developerworks/edu/j-dw-java-concur-i.html

