Introdução à Programação Concorrente em Java

Sistemas multitarefas

- Sabe-se que a CPU executa apenas um programa por vez.
- A princípio, se esta regra for seguida de maneira estrita, é impossível utilizar os computadores do modo como é feito hoje, com diversos programas sendo executados ao mesmo tempo.
- Torna-se necessário, então, a criação de sistemas multitarefas.

Sistemas multitarefas

- Um sistema multitarefas executa diversos programas de maneira alternada.
- A troca de programa na CPU (troca de contexto) é realizada tão rapidamente que os programas parecem rodas em paralelo (pseudoparalelismo).

Sistemas multitarefas

- Mesmo em sistemas com diversas CPUs, o número de programas tende a ser maior que o número de CPUs.
- Então, a troca de contexto ainda é presente em ambientes multiprocessados.

Processos

- A unidade de armazenamento (estrutura de dados) de um programa é o processo.
- O processo armazena todos os dados e instruções da execução do programa.

Processos

De maneira simplória, os processos podem realizar dois tipos de instruções fundamentais:

- processamento de dados;
- entrada e saída de dados;

Processos

- A entrada e saída de dados é muito custosa, pois depende de dispositivos muito lentos em relação à CPU.
- De modo geral, a espera pelas operações de entrada e saída de dados torna o programa inerte durante este período.
- Para evitar o desperdício de recursos, pode-se ter um processo dividido em subprocessos, cada um com seu espaço de armazenamento.

Subprocessos

- O problema dos subprocessos está na carestia de tempo de processamento e memória para criá-los.
- Além disso, a comunicação entre processos exige estruturas de dados e técnicas específicas.

Threads

- Para mitigar alguns problemas da criação de subprocessos, tem-se o conceito de *threads*.
- Grosso modo, uma *thread* é um programa que compartilha o espaço de endereçamento com outras *threads*.

Threads

- A comunicação entre *threads* é mais simples que entre processos, pois as *threads* dentro de um mesmo processo compartilham o mesmo espaço de armazenamento.
- A criação de *threads* também é bastante mais rápida que a de subprocessos.
- Apesar disto, o uso de *threads* não elimina completamente o *overhead*, as condições de corrida e de impasse e possíveis situações de inconsistência.

Threads de SO

Para os sistemas operacionais modernos há, comumente, dois tipos de *threads*:

- *thread* de sistema;
- *thread* de usuário.

Threads Java

- As *threads* em Java são todas *threads* de usuário.
- Apesar de criadas pela JVM, as *threads* são criadas como nativas do SO através do uso de bibliotecas.
- A implementação e o mapeamento destas *threads* dependerá da implementação da JVM.

Threads Java

Internamente, a JVM possui dois tipos de *threads*:

- *threads* de usuário;
- *threads daemon*.

Threads Java

- Para a JVM, o método *main* é uma *thread* de usuário chamada *main*.
- Mesmo que *thread main* seja completada, enquanto existirem *thread* de usuário ativas, o programação não será encerrado.
- Isto não é válido para *threads daemon*.

Threads Java

- Se todas as *threads* de usuário forem encerradas, a JVM pode encerrar arbitrariamente as *threads daemon*.
- Por isso, *threads daemon* devem executar tarefas de suporte em *background*, como *garbage collection*.

Criação de *Threads* em Java

As *threads*, em Java, podem ser criadas através de dois modos:

- herança da classe *Thread*;
- implementação da *interface Runnable*.

Herança da classe *Thread*

Exemplo de criação de *thread* em Java pela herança da classe *Thread*:

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("======= Starting " +
            this.getName() + " thread =======");
        for (int i = 0; i < 5; i++) {
            System.out.println("The thread " +
                this.getName() + " prints " + i);
        }
        System.out.println("======= Finishing " +
            this.getName() + " thread =======");
    }
}
```

Herança da classe *Thread*

Exemplo de criação de *thread* em Java pela herança da classe *Thread* (continuação):

```
public static void main(String[] args) {
    System.out.println("======= Starting main thread =======");
    MyThread t1 = new MyThread();
    MyThread t2 = new MyThread();

    t1.start();
    t2.start();
    for (int i = 0; i < 5; i++) {
        System.out.println("Main thread prints " + i);
    }
    System.out.println("======= Finishing main thread =======");
}
```

Herança da classe *Thread*

- A saída do programa pode variar porque, neste caso, não há como especificar a ordem de execução das *threads*.
- O método *start* é reponsável por iniciar a *thread*.
- Toda *thread* em Java deve sobrescrever o método *run*.

Implementação da *interface Runnable*

Outro modo de criar uma *thread* em Java é implementar a *interface Runnable*:

```
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("======= Starting runnable =======");
        for (int i = 0; i < 5; i++) {
            System.out.println("A runnable prints " + i);
        }
        System.out.println("======= Finishing runnable =======");
    }
}
```

Implementação da *interface Runnable*

O método *main* permanece quase idêntico:

```
public static void main(String[] args) {
    System.out.println("======= Starting main thread =======");
    Thread t1 = new Thread(new MyRunnable());
    Thread t2 = new Thread(new MyRunnable());

    t1.start();
    t2.start();
    for (int i = 0; i < 5; i++) {
        System.out.println("Main thread prints " + i);
    }
    System.out.println("======= Finishing main thread =======");
}
```

Implementação da *interface Runnable*

- Note que no exemplo, são criadas dois objetos *Thread*.
- No construtor de cada um é passado um objeto que implementa a *interface Runnable*.
- A classe *Thread* tem diversos construtores que podem ser utilizados para dar mais flexibilidade ao programador.

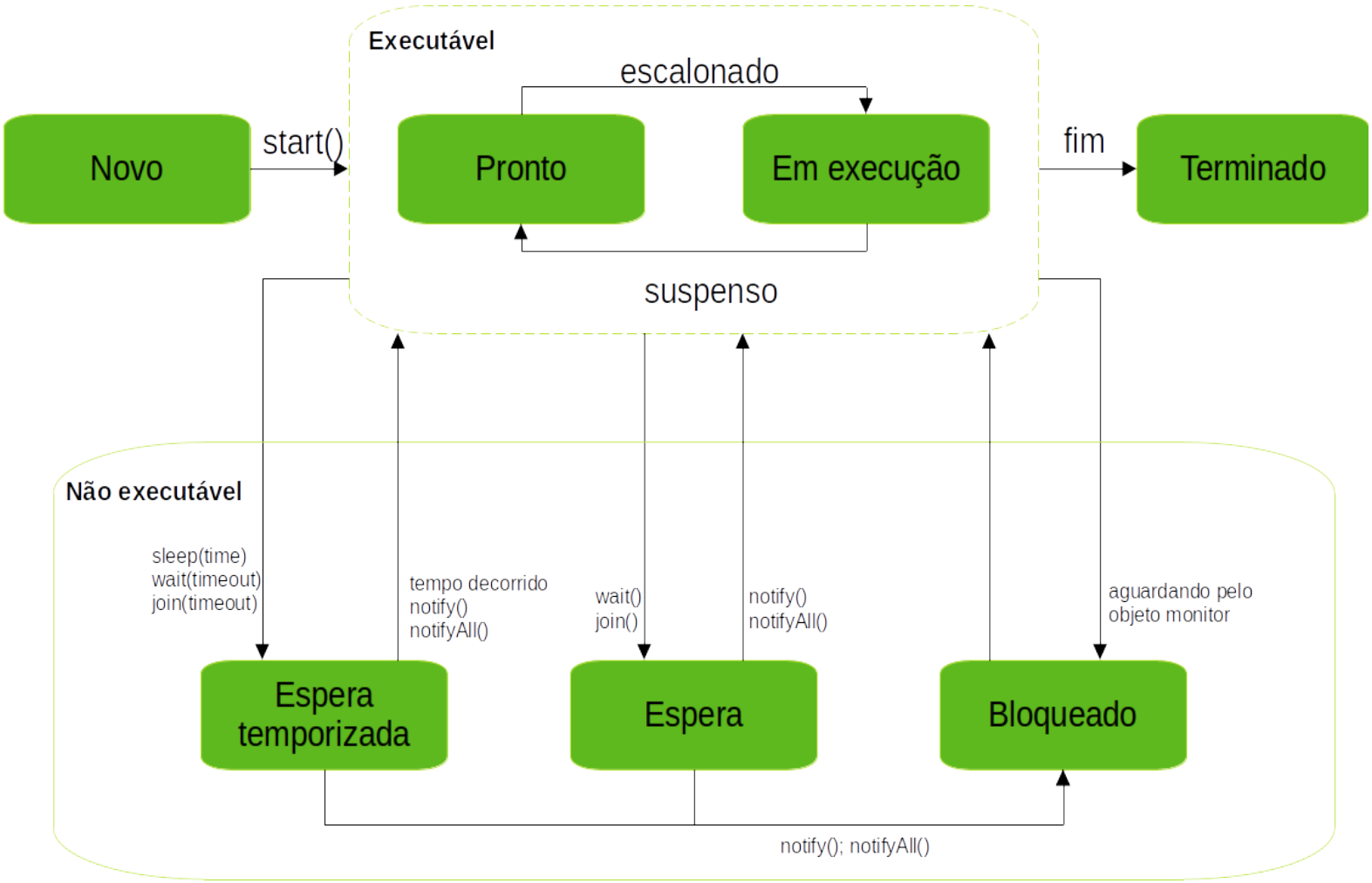
Implementação da *interface Runnable*

- Um ponto interessante é que a implementação de *Runnable* exige também sobrescrever *run*.
- Mais do que isso, *Thread* também implementa *Runnable*.
- Por isso possui um método *run*.

Criação de *threads* em Java

- Implementar a *interface Runnable* é preferencial em relação à herdar a classe *Thread*.
- Java não aceita herança múltipla, mas aceita a implementação de mais de uma *interface*.
- Assim, implementar *Runnable* torna-se mais vantajoso.
- Além disso, implementar *interfaces* é quase sempre melhor do ponto de vista de coesão/acoplamento entre as classes.
- Só se deve herdar de *Thread* se for necessário sobrescrever outros métodos de *Thread* além de *run*.

Estados das *threads* em Java



Método *sleep*

- Um modo de parar temporariamente uma *thread* é o método *sleep* (da classe *Thread*).
- Ao executar este método, a *thread* passa para um estado não-executável (espera temporizada).
- A *thread* volta ao estado de pronto assim que o tempo é decorrido ou caso o método *sleep* levante uma *InterruptionException*.

Método *sleep*

```
public class SleepingThread implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                System.out.println("A runnable prints " + i);
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Método *sleep*

```
public static void main(String[] args) {
    Thread t1 = new Thread(new SleepingThread());
    t1.start();
}
```

Threads daemon

Agora, com o método *sleep*, pode-se exemplificar a diferença entre *threads* de usuário e *threads daemon*:

```
public class UserOrDaemonThread extends Thread {
    public int sleepingTime = 0;
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(this.sleepingTime);
                System.out.println("The thread " +
                    this.getName() + " prints " + i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Threads daemon

```
public static void main(String[] args) {
    UserOrDaemonThread t1 = new UserOrDaemonThread();
    UserOrDaemonThread t2 = new UserOrDaemonThread();
    t1.sleepingTime = 1000;
    t2.sleepingTime = 2000;
    t2.setDaemon(true);
    t1.start();
    t2.start();
}
```

Na saída do programa, será possível observar que a *thread* **t2** não é executada até o fim.

Problema de produtor/consumidor

- Um problema clássico da computação concorrente é o problema de produtor/consumidor.
- Neste problema, há um ou mais produtores que inserem o "produto"em um *buffer* e um ou mais consumidores que consomem o conteúdo deste mesmo *buffer*.

Problema de produtor/consumidor

- O desafio é garantir que produtores e consumidores trabalhem de maneira sincronizada.
- O *buffer* só pode ser acessado por um agente por vez.
- O consumidor só pode consumir se existir algum produto ainda não consumido no *buffer*.

Problema de produtor/consumidor

- Morelli e Walde apresentam um exemplo para ilustrar o problema de produtor/consumidor na vida real.
- Tenha-se uma padaria onde os clientes pegam uma senha e são atendidos em ordem.
- Esta situação, assim como várias onde há fila, pode ser vista como um problema de produtor/consumidor.
- O cliente é o produtor (pois pede uma senha).
- O atendente é o consumidor (pois consome uma senha gerada).
- A máquina de senha é o *buffer*.

Problema de produtor/consumidor

- A máquina de senha deve manter a contagem e não pode pular números e/ou clientes.
- Dois clientes não podem ter o mesmo número.
- O atendente não pode atender um cliente inexistente.

Problema de produtor/consumidor

Para resolver este problema, deve-se decompo-lo em partes menores:

- um objeto para a máquina de senha;
- uma classe representando o(s) atendente(s);
- uma classe representando o(s) cliente(s);
- o cliente pede um número à máquina;
- o atendente atende o próximo número da fila;
- uma classe para padaria que terá o *main* e criará as *threads*.

Máquina de senha (classe *TakeANumber*)

```
public class TakeANumber {
    private int next = 0;
    private int serving = 0;
    public int nextNumber() {
        next = next + 1; return next;
    }
    public int nextCustomer() {
        ++serving; return serving;
    }
    public boolean thereIsCustomerWaiting() {
        return next > serving;
    }
}
```

- *nextNumber* e *nextCustomer* são chamados por clientes e atendentes respectivamente.
- *thereIsCustomerWaiting* é para garantir que não serão atendidas senhas inexistentes.

Adaptado de Morelli e Walde.

Cliente (classe *Customer*)

A classe *Customer* é bastante trivial.

```
public class Customer extends Thread {
    private static int number = 10000;
    private int id;
    private TakeANumber takeANumber;

    public Customer( TakeANumber gadget ) {
      id = ++number; takeANumber = gadget;
    }
    public void run() {
      try {
        sleep( (int)(Math.random() * 1000 ) );
        System.out.println("Customer " + id +
         " takes ticket " + takeANumber.nextNumber());
      } catch (InterruptedException e) {
        System.out.println("Exception " + e.getMessage());
      }
    }
}
```

Adaptado de Morelli e Walde.

Atendente (classe *Clerk*)

A classe *Clerk* é também bastante trivial.

```
public class Clerk extends Thread {
    private TakeANumber takeANumber;

    public Clerk(TakeANumber gadget) {
        takeANumber = gadget;
    }
    public void run() {
        while (true) {
            try {
                sleep((int)(Math.random() * 50));
                if (takeANumber.thereIsCustomerWaiting())
                    System.out.println("Clerk serving ticket " +
                        takeANumber.nextCustomer());
            } catch (InterruptedException e) {
                System.out.println("Exception " + e.getMessage() );
            }
        }
    }
}
```

Adaptado de Morelli e Walde.

Padaria (classe *Bakery*)

```
public class Bakery {
    public static void main(String args[]) {
        System.out.println("Starting clerk and customer threads");
        TakeANumber numberGadget = new TakeANumber();
        Clerk clerk = new Clerk(numberGadget);
        clerk.start();
        for (int k = 0; k < 5; k++) {
            Customer customer = new Customer(numberGadget);
            customer.start();
        }
    }
}
```

Adaptado de Morelli e Walde.

Problema de produtor/consumidor

- A solução apresentada pode, a princípio, parecer correta.
- Entretanto, o código não é seguro e apresentará problemas quando houver situações de concorrência.

Sincronização de *threads*

Tenha-se o código do método *nextNumber*:

```
public int nextNumber() {
    next = next + 1;
    return next;
}
```

- Nada garante que a *thread* executando este código não será retirada da CPU entre a primeira e segunda instrução.
- Caso isso ocorra, outra *thread* poderá também chamar *nextNumber*, o que poderá causar que dois clientes peguem a mesma senha e que algumas senhas sejam "puladas".

Adaptado de Morelli e Walde.

Sincronização de *threads*

Para simular este problema, basta chamar o método *sleep* entre as duas instruções:

```
public int nextNumber() {
    next = next + 1;
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {}
    return next;
}
```

Ao executar o código, será possível notar que mais de um cliente terá a mesma senha.

Algumas vezes a mensagem do atendente atendendo antes da mensagem do cliente pegar a senha pode aparecer, mas isso se dá porque o foi chamado o métodos *sleep*, atrasando a *thread* do cliente.

Região crítica

- As duas linhas originais de *nextNumber* formam a chamada região crítica.
- A região crítica é uma região do código que deve ser executada de maneira atômica, para evitar possíveis inconsistências em condições de concorrência.

Bloco *synchronized*

- Java permite que um bloco de código seja declarado como uma região crítica.
- Para isto, usa-se a palavra-chave *synchronized*, além de um objeto que servirá como monitor.
- Este monitor pode ser todo e qualquer objeto.
- Ao entrar em uma região crítica, a *thread* executora deve conseguir o *lock* deste objeto.
- Se outra *thread* tentar entrar na mesma região crítica ou que tenha o mesmo objeto monitor, esta deverá aguardar até que a *thread* que possui o *lock* o largue.

Bloco *synchronized*

Além de modificar *nextNumber*, adicionou-se um atributo *o* do tipo *Object*:

```
private Object o = new Object();
public int nextNumber() {
    synchronized (o) {
        next = next + 1;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        return next;
    }
}
```

1

[1] O mesmo se aplica ao método *nextCustomer* caso haja mais de um atendente.

Método *synchronized*

- Algumas vezes, teremos a necessidade que o método inteiro seja região crítica.
- Para isto, Java permite que um método seja declarado como *synchronized*.

Método *synchronized*

Retira-se o atributo *o* e modifica-se *nextNumber*:

```
public synchronized int nextNumber() {
    next = next + 1;
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {}
    return next;
}
```

2

[2] Para garantir a segurança do código como um todo, é interessante declarar como sincronizados os outros dois métodos de *TakeANumber*.

Método *synchronized*

- Em métodos *synchronized* o monitor é o próprio objeto.
- Neste caso, a instância de *TakeANumber*.
- Seria como:

```
public int nextNumber() {
    synchronized(this) {
        next = next + 1;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        return next;
    }
}
```

Problema da espera-ocupada

- Um problema do exemplo atual é que a *thread Clerk*, por estar executando em um laço infinito, continua tomando tempo da CPU mesmo quando não há clientes com senhas.
- A esta situação é dado nome de espera-ocupada.
- Para evitar isso, Java possui um mecanismo de espera/notificação, que coloca a *thread* em estado de espera, evitando que entre na CPU.

Mecanismo de espera/notificação

```
public synchronized int nextCustomer() {
    try {
      while (next <= serving)
        wait();
    } catch(InterruptedException e) {
    } finally {
      ++serving;
      System.out.println("Clerk serving ticket " + serving);
      return serving;
    }
  }
```

Quando não houver clientes desassistidos (next <= serving), a *thread* que invocar *nextCustomer* ficará aguardando (método *wait*).

Não há mais a necessidade de *thereIsCustomerWaiting*.

Mecanismo de espera/notificação

- Caso não seja passado um tempo limite para *wait*, a *thread* ficará esperando indefinidamente.
- Por isso, é preciso que outra *thread* notifique a *thread* que está aguardando.
- No caso aqui descrito, o método *nextNumber* será o responsável por notificar as *thread* aguardando.

Mecanismo de espera/notificação

```
public synchronized int nextNumber(int custId) {
    next = next + 1;
    System.out.println("Customer " + custId +
        " takes ticket " + next);
    notify();
    return next;
}
```

notify e *notifyAll*

- O método *notify* notifica a *thread* mais a frente na fila de *threads* em estado de aguardo.
- Já o método *notifyAll* notifica todas as *threads* que estão aguardando.
- Por isso, o método *wait* é chamado dentro de um laço, pois, caso contrário, todas as *threads Clerk* executariam os códigos da sua cláusula *finally*.

classe *TakeANumber* - versão final

```
public class TakeANumber {
    private int next = 0; private int serving = 0;
    public synchronized int nextNumber(int custId) {
        next = next + 1;
        System.out.println("Customer:" + custId +
            " takes ticket " + this.next);
        notify();
        return next;
    }
    public synchronized int nextCustomer() {
        try {
            while (next <= serving) {
                System.out.println("Clerk waiting ");
                wait();
            }
        } catch (InterruptedException e) {
        } finally {
            ++serving;
            System.out.println("Clerk serving ticket " + serving);
            return serving;
        }
    }
}
```

classe *Clerk* - versão final

```
public class Clerk extends Thread {
    private TakeANumber takeANumber;
    public Clerk(TakeANumber gadget) {
        this.takeANumber = gadget;
    }
    public void run() {
        while (true) {
            try {
                sleep((int) (Math.random() * 50));
                takeANumber.nextCustomer();
            } catch (InterruptedException e) {}
        }
    }
}
```

Adaptado de Morelli e Walde.

classe *Customer* - versão final

```
public class Customer extends Thread {
    private static int number = 10000;
    private int id;
    private TakeANumber takeANumber;
    public Customer(TakeANumber gadget) {
        this.id = ++number;
        this.takeANumber = gadget;
    }
    public void run() {
        try {
            sleep((int) (Math.random() * 1000));
            this.takeANumber.nextNumber(this.id);
        } catch (InterruptedException e) {
        }
    }
}
```

Adaptado de Morelli e Walde.

Outros tópicos

- A programação concorrente em Java possui diversos outros tópicos que não serão abordados aqui.
- A linguagem ainda permite operações de *join* e *fork*, variáveis atômicas e objetos imutáveis.

Referências

- Morelli, Ralph; Walde, Ralph. Java, Java, Java: Object-Oriented Problem Solving. 3ed. Hatford, EUA. 2017.
- Baeldung. Life Cycle of a Thread in Java. 2023. Disponível em: https://www.baeldung.com/java-thread-lifecycle
- Baeldung. Daemon Threads in Java. Disponível em: https://www.baeldung.com/java-daemon-thread
- Oracle. The Java Tutorials. Disponível em: https://docs.oracle.com/javase/tutorial/essential/concurrency
- Dutta, Raddhi. Java Concurrency & Multithreading Complete Course. Disponível em: https://www.youtube.com/watch?v=WldMTtUWqTg