

Threads, mecanismos

Este tutorial complementa a discussão teórica sobre programação concorrente vista em sala de aula. Vimos que uma thread define um fluxo de execução de instruções em um processo. Processos com múltiplos threads têm portanto múltiplos fluxos de execução.

Threads de um mesmo processo compartilham a memória do processo. Por sua vez, alguns recursos são criados para uso exclusivo das threads, a pilha de execução, por exemplo. Assim, a imagem de uma thread em memória é menor do que um processo. Ainda, a troca de contexto entre threads pode não envolver o sistema operacional (diferente de troca de contexto entre processos). Por esses motivos, é "mais leve" construir sistemas concorrentes com múltiplas threads ao invés de múltiplos processos.

Neste tutorial, mostraremos mecanismos de algumas linguagens de programação (Java e Python) que permitem criar threads, definir o código que as threads executarão bem como coordenar e sincronizar a execução de threads.

Java

A plataforma Java tem grande suporte para threads desde suas versões iniciais. Neste tutorial, discutiremos mecanismos mais recentes (lançados na versão 21 da plataforma), incluindo tanto threads *virtuais* (ou *green threads*) e threads de plataforma. A diferença fundamental entre essas duas implementações é que cada thread de plataforma corresponde à uma thread do sistema operacional (*kernel thread*, a unidade básica de escalonamento do sistema operacional) enquanto que várias threads virtuais podem ser mapeadas para a mesma thread do sistema operacional.

Considere um cenário em que as threads de um programa Java bloqueiam com frequência, realizando, por exemplo, operações de entrada e saída. Neste cenário, usar *threads* virtuais é uma boa opção pois permite que a *runtime* Java escalone uma outra *thread* para executar no lugar da que está bloqueada sem a necessidade de chamar o escalonador do sistema operacional; ou seja, sem a necessidade da custo adicional de realizar uma *system call*.

O mecanismo de gerenciamento das threads, seja de plataforma ou virtuais, tem três etapas iniciais: 1) a definição do código a ser executado pela thread; 2) a criação da thread; e, por fim, 3) a inicialização da thread. Estes mecanismos, na maior parte das vezes, são os mesmos para ambos os tipos de threads. Ainda, boa parte das diferenças, principalmente na criação das threads, são simplificadas pela abstração [ThreadBuilder](#). Como sempre, a [documentação da plataforma](#) detalha estes e outros aspectos.

Definição do código a ser executado

Em Java, usamos o tipo [Runnable](#) para definir o fluxo que será executado por uma [Thread](#). Em outras palavras, escrevemos o código a ser executado pela Thread através do tipo [Runnable](#). Este tipo define um único método, run, que pode executar código arbitrário definido pelo programador:

```
void run()
```

A thread, ao ser iniciada pelo escalonador, chamará o método run. Este método está limitado a não receber parâmetros e não ter retorno e portanto outras estratégias precisam ser usadas para compartilhar memória entre threads, por exemplo, através de referências passadas por construtores e outros métodos.

De maneira concreta, é possível definir o comportamento de uma thread tanto criando uma classe que implementa a interface definida por [Runnable](#) quanto, por herança, criando uma subclasse de [Thread](#). Abaixo, temos duas implementações, uma para cada modo, em que as threads criadas escrevem na saída padrão a mesma mensagem indefinidamente. Na maior parte dos casos, usa-se a primeiro modo (uma numa implementação da interface Runnable) a menos que se queira mudar o comportamento dos métodos implementados na classe Thread (o que é bastante inusual).

```
import java.util.concurrent.*;

public class HelloRunnable implements Runnable {

    public static void main(String[] args) {
        HelloRunnable helloRunnable = new HelloRunnable();
        helloRunnable.run();

        HelloThread helloThread = new HelloThread();
        helloThread.start();
    }

    @Override
    public void run() {
        while (true) {
            System.out.println("Hello world!");
        }
    }

    static class HelloThread extends Thread {
        @Override
        public void run() {
            while (true) {
                System.out.println("Hello world!");
            }
        }
    }
}
```

```
}  
}
```

Criação da Thread (de plataforma)

Uma vez implementado o método **run** para definir o comportamento que a thread executará, a thread precisa ser de fato criada. Isso pode ser feito de várias maneiras. Por exemplo, criando um objeto [Thread](#) e, por composição, passando uma instância que implementa *Runnable*:

```
Thread myThread = new Thread(new HelloRunnable(), "myThread-HelloRunnable");
```

Ou criando uma subclasse de thread:

```
Thread myThread = new HelloThread("myThread-HelloThread");
```

Veja que usei um argumento String para o construtor. Esse parâmetro define o nome para thread criada e pode ser acessado através do método [getName](#). Atribuir nomes únicos para thread é uma excelente prática que ajuda bastante a depurar defeitos em sistemas concorrentes.

Inicialização da thread (de plataforma)

Nos passos anteriores, criamos uma thread e atribuímos a ela um código a ser executado. Ainda é necessário um passo adicional, a inicialização. Para isso, chamamos o método **start** da thread. Isso permite que a thread criada esteja pronta para executar. Eventualmente, o escalonador pode escolher a thread para ser executada.

```
myThread.start();
```

Como dissemos, as threads de um programa são, cada uma, um fluxo de execução distinto dos demais. Em muitos casos, é preciso que as threads sincronizem sua execução. Decidir como sincronizar threads para termos sistemas corretos é algo decidido caso a caso. Aqui, mostraremos somente um mecanismo que permite que uma thread espere que outra termine a sua execução, por exemplo para obter um resultado da computação da thread que terminou. Isso, pode ser feito através do método **join**:

```
myThread.join();
```

Neste caso, a thread que executa a linha acima irá bloquear ao chamar o método **join** até que a thread **myThread** termine sua execução.

Threads virtuais

Os mecanismos acima, permitem criar e executar **threads de plataforma**. Para manipular **threads virtuais**, usamos [ThreadBuilder](#).

Usando este Builder, podemos criar uma thread virtual através das seguintes chamadas:

```
Thread thread = Thread.ofVirtual().unstarted(runnable);
```

e, em seguida:

```
thread.start();
```

Note que o objeto runnable passado como argumento é uma implementação de Runnable tal como fizemos para threads de plataforma.

Ainda, é possível iniciar a thread na mesma cadeia de criação:

```
Thread thread = Thread.ofVirtual().start(runnable);
```

Semáforos

Em java, usamos o tipo `java.util.concurrent.Semaphore` para manipular Semáforos. Um objeto do tipo Semaphore é criado indicando o valor inicial do semáforo:

```
new Semaphore(initValue);
```

Os métodos abaixo, implementam a semântica de wait e signal:

```
public void acquire()  
public void release()
```

Em particular, o valor do semáforo em java atinge no mínimo zero.

Exemplo

```
import java.util.concurrent.*;  
  
public class FindMinInArray {  
    public static void main(String[] args) throws InterruptedException {  
        int[] array = {12, 3, 19, 8, 7, 25, 5, 17, 6, 9, 14, 2, 1, 20};  
  
        Task task = new Task(array);  
        Thread myThread = new Thread(task, "myThread-Task");
```

```

        myThread.start();
        myThread.join();

        int min = task.getMinValue();
        System.out.println("O menor número no vetor é: " + min);
    }

    public static class Task implements Runnable {
        private final int[] array;
        private int minValue;

        public Task(int[] array) {
            this.array = array;
            this.minValue = Integer.MAX_VALUE;
        }

        @Override
        public void run() {
            for (int num : array) {
                if (num < minValue) {
                    minValue = num;
                }
            }
        }

        public int getMinValue() {
            return minValue;
        }
    }
}

```

Explicação dos Componentes

- `public class Task implements Runnable`: Define uma classe chamada `Task` que implementa a interface `Runnable`. Implementar `Runnable` significa que a classe `Task` pode ser executada por uma thread.
- `@Override`: É uma anotação que indica que o método `run()` está sobrescrevendo o método da interface `Runnable`.

- `public void run()`: Este é o método que será executado quando a thread iniciar. Dentro deste método, você define o que a thread deve fazer. Neste caso, imprime "Executando a tarefa." no console.
- `public static void main(String[] args)`: É o ponto de entrada do programa. Este método é chamado pela JVM quando o programa é executado.
- `throws InterruptedException`: Declara que o método `main` pode lançar uma `InterruptedException`. Isso é necessário porque o método `join()` pode lançar essa exceção.
- `new Thread(new Task(), "my-task")`: Cria uma nova instância de `Thread`. O construtor `Thread` recebe dois argumentos:
 - `new Task()`: Passa uma instância de `Task` que implementa `Runnable`, definindo o código que a thread deve executar.
 - `"my-task"`: Nomeia a thread como "my-task", o que pode ser útil para depuração e monitoramento.
- `myThread.start()`: Inicia a execução da thread. A JVM cria um novo fluxo de execução e chama o método `run()` da instância `Task`. A partir deste ponto, a thread executa seu código de forma independente.
- `myThread.join()`: Este método faz com que a thread principal (a que está executando o método `main`) espere até que a thread `myThread` termine sua execução. Sem isso, a thread principal poderia terminar antes da thread `myThread`, encerrando o programa antes que `myThread` complete sua execução.

Links úteis:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/join.html>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

Python

Criação de Threads em Python

O estilo adotado para uso de threads em python é bastante parecido com java (embora mais simples, como de costume).

Uma diferença marcante é que não é necessário criar um tipo para definir o comportamento da thread. Para isso, basta implementar uma função regular (thread_function abaixo).

Em seguida, a thread é criada usando o construtor `threading.Thread` (<https://docs.python.org/3/library/threading.html#threading.Thread>) e por fim, a thread é inicializada com a função `start`

```
import threading
import time

def thread_function():
    time.sleep(2)

if __name__ == "__main__":
    x = threading.Thread(target=thread_function)
    x.start()
```

Passando Argumentos para Funções de Thread

Se a função que a thread vai executar precisar de argumentos, você pode passar esses argumentos utilizando o parâmetro `args` ao criar a thread:

```
import threading
import time

def thread_function(arg0, arg1):
    print(arg0, arg1)
    time.sleep(2)

if __name__ == "__main__":
    x = threading.Thread(target=thread_function, args=(1,2,))
    x.start()
```

Esperando a Thread Terminar

Por fim, a thread pode demorar mais para finalizar do que a thread mãe. Para isso, utiliza-se o método `join()`:

```
import threading
import time

def thread_function():
    time.sleep(2)
    print("Thread finished")

if __name__ == "__main__":
    x = threading.Thread(target=thread_function)
    x.start()
    x.join() # Aguarda a conclusão da thread
    print("Main thread continues")
```

Explicação dos Componentes

- `import threading`: Importa o módulo `threading`, que fornece ferramentas para trabalhar com threads.
- `def thread_function()`:: Define a função que a thread executará. Esta função pode realizar qualquer tarefa necessária.
- `threading.Thread(target=thread_function)`: Cria uma nova thread, especificando a função `thread_function` como a tarefa a ser executada pela thread.
- `x.start()`: Inicia a execução da thread.
- `x.join()`: Bloqueia a thread principal até que a thread `x` termine a execução.

Links úteis:

<https://docs.python.org/3/library/threading.html#>

Semáforos em Python

O semáforo é uma abstração básica para sincronização de threads no modelo de concorrência por compartilhamento de memória. A seguir, mostramos como usar semáforos em python. É importante ter em mente que a API da linguagem python é **ligeiramente diferente** da abstração discutida em sala.

Semáforo Explícitos

Em python, o tipo semáforo mantém um contador que é **decrementado através da operação acquire**. Por sua vez, é incrementado **através da operação release**.

Quando o semáforo estiver com valor zero, qualquer thread que chamar **acquire** será bloqueada. Uma thread bloqueada no semáforo será desbloqueada por outra thread que chame **release** no semáforo.

```
import threading
import time
import random

semaphore = threading.Semaphore(2)

def thread_function(name):
    semaphore.acquire()
    try:
        print(f"{name} acquired the semaphore")
        time.sleep(random.randint(1, 10))
    finally:
        print(f"{name} releasing the semaphore")
        semaphore.release()

if __name__ == "__main__":
    threads = []
    for i in range(6):
        thread = threading.Thread(target=thread_function,
args=(f"Thread-{i}",))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

Semáforos com with

Além do modo anterior, no qual era necessário chamar **release** e **acquire** explicitamente, é possível também utilizar um semáforo com a declaração **with** (context manager). Isso simplifica o código e garante que o semáforo será liberado corretamente, mesmo que ocorra uma exceção dentro do bloco **with**.

```
import threading
import time
import random

semaphore = threading.Semaphore(2)

def thread_function(name):
    with semaphore:
        print(f"{name} acquired the semaphore")
        time.sleep(random.randint(1, 10))
        print(f"{name} releasing the semaphore")

if __name__ == "__main__":
    threads = []
    for i in range(6):
        thread = threading.Thread(target=thread_function,
args=(f"Thread-{i}",))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

Barreira

A classe **Barrier** do módulo **threading** é usada para sincronizar um ponto em que todas as threads devem esperar umas pelas outras para continuar. Da mesma forma que vimos em sala, uma barreira em python é construída com um tamanho (chamado de **parties** na definição de python) e tem um método **wait** para sinalizar a chegada no ponto da barreira.

```
import threading
import time
import random

barrier = threading.Barrier(6)

def thread_function(name):
    print(f"{name} is waiting at the barrier")
    time.sleep(random.randint(1, 10))
    barrier.wait()
    print(f"{name} passed the barrier")

if __name__ == "__main__":
    threads = []
    for i in range(6):
        thread = threading.Thread(target=thread_function,
args=(f"Thread-{i}",))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```