Conceitos da classe Thread em Java e comparativo com Pthread em C/C++

Pedro Antônio de Souza (201810557)

A linguagem de programação Java foi desenvolvida pela empresa Sun Microsystems na década de 1990. Os líderes de seu desenvolvimento acreditavam que em pouco tempo haveria uma compatibilização entre computadores e eletrodomésticos. Baseando-se no lema "escreva uma vez, execute em qualquer lugar", a nova linguagem teria como objetivo realizar a comunicação entre essas máquinas de forma harmônica e sem conversões. Assim, foi desenvolvida uma linguagem orientada a objetos que é compilada para bytecodes pelo compilador *javac*. Por sua vez, os bytecodes são interpretados pela Java Virtual Machine (JVM). A portabilidade de programas Java é estabelecida na interpretação, já que um mesmo programa pode ser executado em qualquer equipamento que possua uma JVM instalada (daí o lema citado acima).

Desde o início, a linguagem Java foi projetada para suportar programação concorrente. Porém, somente a partir da versão 5.0 a plataforma Java passou a incluir APIs concorrentes de alto nível. Para criar processos adicionais em Java, deve-se utilizar um objeto `ProcessBuilder`. Porém, nesse documento iremos focar na criação de threads. Então, para criar uma aplicação concorrente em Java, utiliza-se a classe `Thread`. Assim, basta instanciar um objeto `Thread` toda vez que for necessário iniciar uma atividade assíncrona. Dessa forma, cada `Thread` terá uma atividade específica a ser executada. Há duas formas de criar instancias `Thread`:

- Criar um objeto `Runnable` e passá-lo ao construtor da `Thread`. A interface `Runnable` define um único método, chamado `run`, onde está contido o código a ser executado pela `Thread`:

```
public class OlaRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Olá de uma thread!");
    }
```

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

}
```

- Criar uma subclasse de `Thread` implementando seu método `run`, como visto no exemplo abaixo:

```
public class OlaThread extends Thread {

    @Override
    public void run() {
        System.out.println("Olá de uma thread!");
    }

    public static void main(String args[]) {
        (new OlaThread()).start();
    }

}
```

Como pode-se observar nos dois exemplos acima, para iniciar a nova thread é necessário invocar o método `start`.

Assim como na biblioteca `Pthread` (POSIX thread library) estudada no curso, em Java é possível obter o identificador da thread utilizando a combinação de métodos estáticos `Thread.currentThread().getId()`. Também é possível unir os fluxos de controle utilizando o método `join`. Supondo que `t` é uma `Thread`, a seguinte chamada irá pausar a execução da thread atual até que a `t` termine sua execução:

```
t.join();
```

Diferentemente da `Pthread`, as threads em Java possuem um bloco de código específico a ser executado, ou seja, cada thread executa sua atividade e não todo o código do programa.

Para realizar comunicação entre duas threads, é possível utilizar o conceito de pipes. O exemplo abaixo mostra como utilizar esse conceito através da instanciação das classes `PipedInputStream` e `PipedOutputStream`. Além dos pipes, fluxos de dados devem ser criados com as classes `DataInputStream` e `DataOutputStream`.

```
public class Remetente extends Thread {
    private DataOutputStream out;

    public Remetente(PipedOutputStream os) {
        out = new DataOutputStream(os);
    }

    public void run() {
        try {
            out.writeUTF("Olá mundo!");
            out.flush();
        } catch(Exception e) { e.printStackTrace(); }
    }
}
```

```
public class Destinatario extends Thread {
    private DataIntputStream in;

    public Destinatario(PipedIntputStream is) {
        in = new DataIntputStream(is);
    }

    public void run() {
        try {
            String mensagem = in.readUTF();
            System.out.println(mensagem);
        } catch(IOException e) { e.printStackTrace(); }
    }
}
```

```
public class ExemploPipe {
    public static void main(String args[]) {
        try {
            PipedOutputStream out = new PipedOutputStream();
            PipedInputStream in = new PipedInputStream(out);
```

```
            Remetente remetente = new Remetente(out);
            Destinatario destinatario = new Destinatario(in);

            remetente.start();
            destinatario.start();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Para realizar a sincronização de um método, basta inserir a palavra-chave `synchronized` em sua declaração:

```
public class ContadorSincronizado {
    private int c = 0;

    public synchronized void incrementar() {
        c++;
    }

    public synchronized void decrementar() {
        c--;
    }

    public synchronized int valor() {
        return c;
    }
}
```

O uso do `sychronized` funciona de forma similar ao semáforo com variável de execução existente na `Pthread`. Em Java, os métodos `wait`, `notify` e `notifyAll`, podem ser comparados com os métodos `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast` respectivamente.

Abaixo é apresentado um exemplo de código que utiliza os conceitos apresentados nesse documento.

```
import java.util.Scanner;
public class Exemplo {
    public static void main(String[] args) throws InterruptedException {
        final PC pc = new PC();
```

```
        // Cria uma thread produtora
        Thread produtor = new Thread(new Runnable() {
            @Override
            public void run() {
                try { pc.produzir(); }
                catch(InterruptedException e) { e.printStackTrace(); }
            }
        });

        // Cria uma thread consumidora
        Thread consumidor = new Thread(new Runnable() {
            @Override
            public void run() {
                try { pc.consumir(); }
                catch(InterruptedException e) { e.printStackTrace(); }
            }
        });

        // Inicia as threads
        produtor.start();
        consumidor.start();

        // produtor finaliza antes do consumidor
        produtor.join();
        consumidor.join();
    }

    // Classe PC (Produtor-Consumidor) com métodos
    // produzir() e consumir().
    public static class PC {
        // Imprime produção e espera por consumir()
        public void produzir() throws InterruptedException {
            // bloco synchronized assegura que apenas uma thread
            // roda o seu escopo.
            synchronized(this) {
                System.out.println("Produzido");

                // libera o bloqueio do recurso
                wait();

                // e espera que outro método invoque notify().
                System.out.println("Retomado");
            }
        }

        // Espera um segundo e aguarda que uma tecla seja pressionada.
        // Após pressionar a tecla, produzir() é notificado.
```

```
        public void consumir() throws InterruptedException {
            // isso faz com que produzir() execute primeiro.
            Thread.sleep(1000);
            Scanner s = new Scanner(System.in);

            // bloco synchronized assegura que apenas uma thread
            // roda o seu escopo.
            synchronized(this) {
                System.out.println("Esperando que uma tecla seja
pressionada.");
                s.nextLine();
                System.out.println("Tecla pressionada");

                // Notifica a thread produtora que ela pode ser retomada
                notify();

                Thread.sleep(2000);
            }
        }
    }
}
```

Como já dito anteriormente, as threads em Java possuem atividades específicas a serem executadas. Assim, elas são ideais para realizar atividades assíncronas. Por exemplo, um programa de bate-papo pode possuir uma thread responsável pela interface gráfica e outra que fica sempre escutando o servidor a espera de mensagens. Também é possível que isso seja feito utilizando `Pthread` em C/C++. Contudo, é nítida a diferença de organização do código em Java, já que as atividades de cada thread utilizando `Pthread` deve ser feita através de condicionais utilizando seu identificador.