Capítulo 2

Processos e Threads

2.1 Processos
2.2 **Threads**
2.3 Comunicação interprocesso
2.4 Problemas clássicos de IPC
2.5 Escalonamento

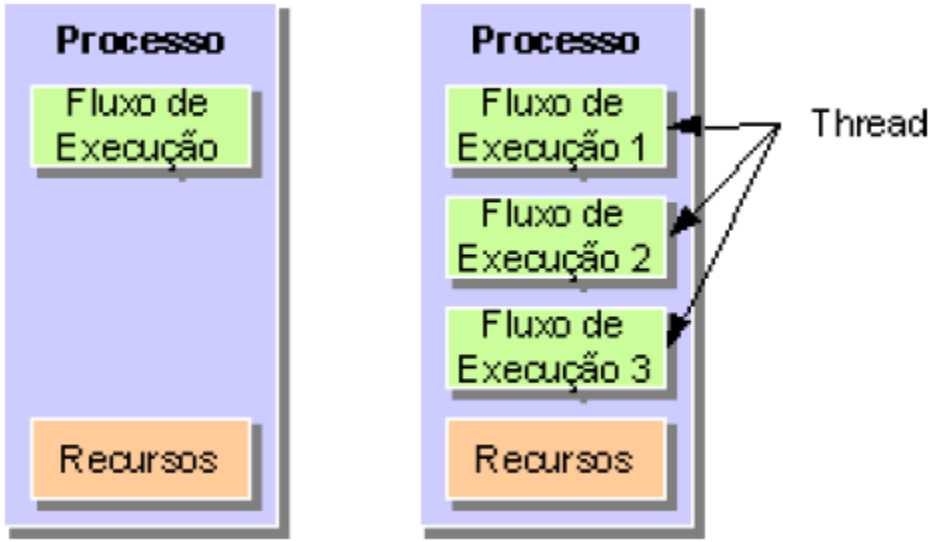
Threads

- Como visto, cada processo conta com uma estrutura de controle razoavelmente sofisticada. Nos casos onde se deseja realizar duas ou mais tarefas simultaneamente, a solução trivial para o desenvolvedor é dividir as tarefas a serem realizadas em dois ou mais processos. Isso implica na criação de duas estruturas de controles distintas para tais processos, onerando o sistema, e complicando, também, o compartilhamento de recursos, por serem processos distintos.

- Uma alternativa ao uso de processos comuns é o emprego de threads. Enquanto cada processo tem um único fluxo de execução, ou seja, só recebe a atenção do processador de forma individual, quando um processo é dividido em threads, cada uma delas recebe a atenção da CPU como um processo, no entanto só existe uma estrutura de controle de processo para tal grupo, o espaço de memória é o mesmo e todos os recursos associados ao processos podem ser compartilhados de maneira bastante simples entre as threads componentes.

Threads

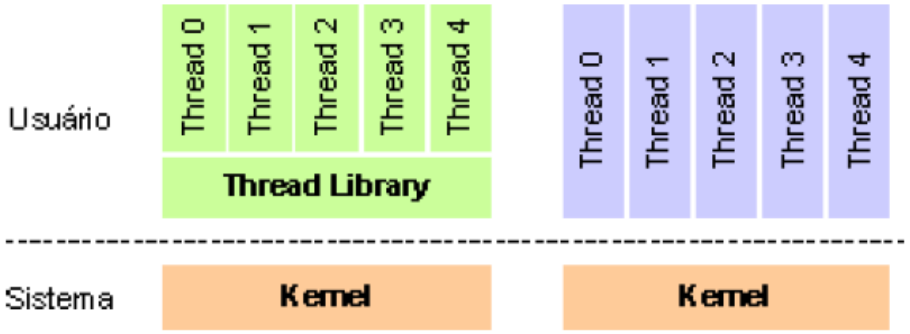
- As threads foram inventadas para permitir a combinação de paralelismo com execução sequencial e chamadas de sistemas bloqueantes. A imagem abaixo mostra um processo comum (observado por Sistemas Operacionais modernos como single thread) e um processo multithread.



Threads

- Desta forma, as threads podem ser entendidas como fluxos independentes de execução pertencentes a um mesmo processo, que requerem menos recursos de controle por parte do Sistema Operacional. Assim, as threads são os que se considera como processos leves (lightweight processes) e constituem uma unidade básica de utilização do processador.

- Sistemas Operacionais que oferecem suporte para execução de threads são conhecidos como Sistemas Multithreading. Existem dois modelos diferentes de suporte em Sistemas Multithreading.



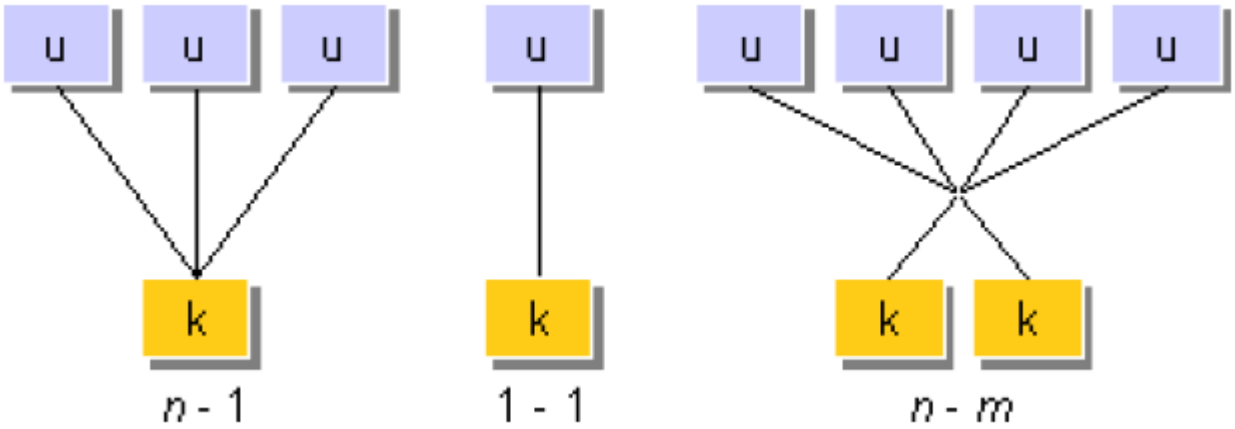
Threads

- **User Threads**:
  - As threads de usuário são aquelas oferecidas através de bibliotecas específicas e adicionais ao S.O., ou seja, são implementadas acima do Kernel (Núcleo do Sistema Operacional) utilizando um modelo de controle que pode ser distinto do sistema operacional, pois não são nativas.
- **Kernel Threads**:
  - As threads do sistema são aquelas suportadas diretamente pelo S.O. e, portanto, nativas.

- Em sistemas não dotados de suporte a threads nativamente, o uso de bibliotecas de extensão permite a utilização de pseudo-threads. Através de tais bibliotecas são oferecidos todos os recursos necessários para a criação e controle de threads. Usualmente, os mecanismos de criação de threads de usuário são bastante rápidos e simples, no entanto, ao ter uma thread bloqueada, possivelmente, todas as outras threads do conjunto também serão por se tratar de um controle diferente do aplicado pelo S.O. Quando o suporte é nativo, a criação de threads é um pouco mais demorada, mas não há o problemas de bloqueio que acontece nos sistemas de controle não nativo de threads.

Modelos de Multithreading

- A forma com que as threads são disponibilizadas para a camada de usuário é o que se denomina modelos de multithreading. Conforme Padrão POSIX, são comuns três modelos distintos



Modelos de Multithreading

- **Modelo n para um**:
  - Este modelo é empregado, geralmente, pelas bibliotecas de suporte de threads de usuário, onde as várias threads do usuário (n) são associadas a um único processo suportado diretamente pelo Sistema Operacional.

- **Modelo um para um**:
  - Modelo simplificado de multithreading verdadeiro, onde cada thread de usuário é associada a uma thread nativa do sistema. Esse modelo é empregado em Sistemas Operacionais tais como o Microsoft Windows.

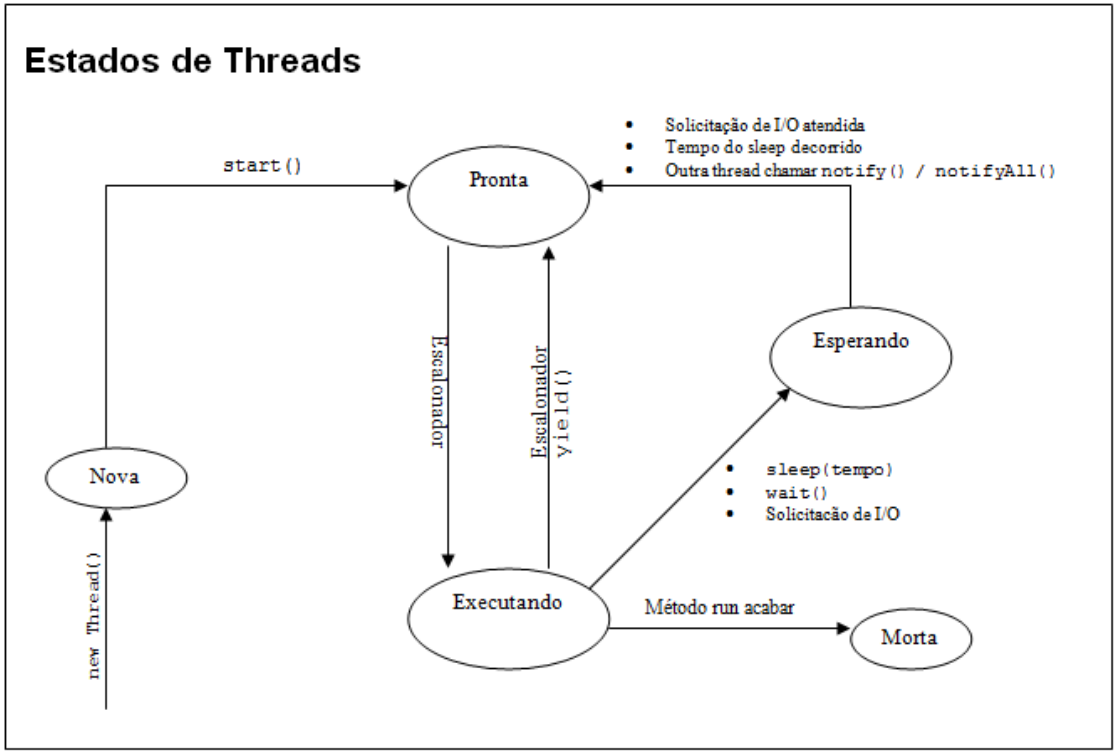
- **Modelo n para m**:
  - Modelo mais sofisticado de multithreading verdadeiro, onde um conjunto de threads do usuário n é associado a um conjunto de threads nativas do sistema, não necessariamente do mesmo tamanho (m). Este model é empregado em Sistemas Operacionais como o Oracle Solaris.

Benefícios do Multithreading

- A utilização de threads pode trazer diversos benefícios para as aplicações e para o próprio S.O.:
  - Melhor capacidade de resposta, pois a criação de uma nova thread é substancialmente mais rápida do que a criação de um novo processo;
  - Compartilhamento de recursos simplificado entre as threads de um mesmo processo, que é a situação mais comum de compartilhamento e comunicação inter-processos (IPC).
  - Economia, pois o uso de estruturas de controle reduzidas, em comparação ao controle típico de processos, desonera o sistema. Além disso, o compartilhamento de recursos simplificado leva à economia de outros recursos
  - Explorar as arquiteturas computacionais multicore.

Ciclo de vida das Threads

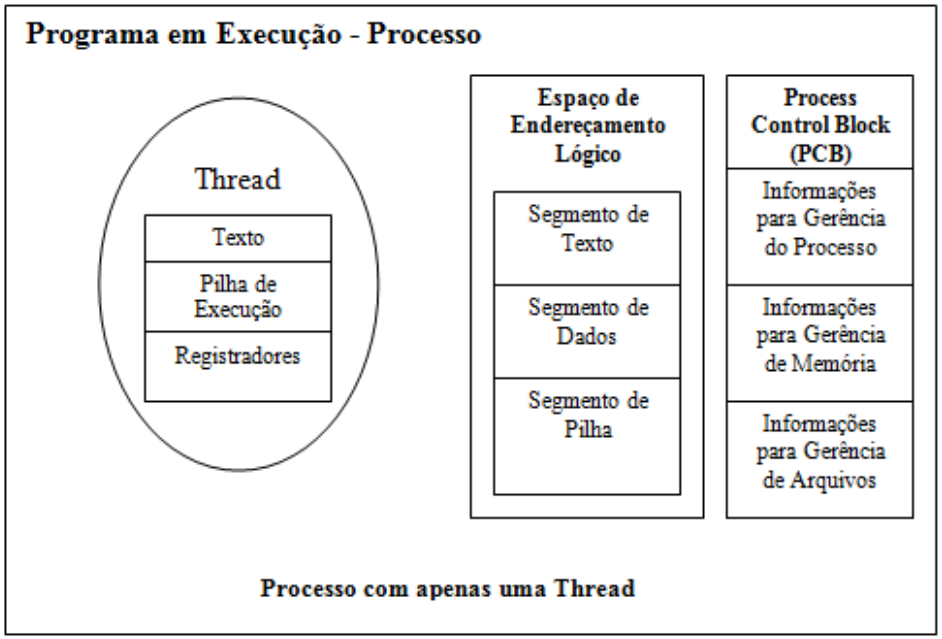
- A melhor forma de analisa o ciclo de vida de uma thread é através das operações que podem ser feitas sobre elas, tais como: criar, iniciar, esperar, parar e encerrar.



Ciclo de vida de uma thread
(Nova/Executando/Pronta/Esperando/Morta)

Aplicação das Threads

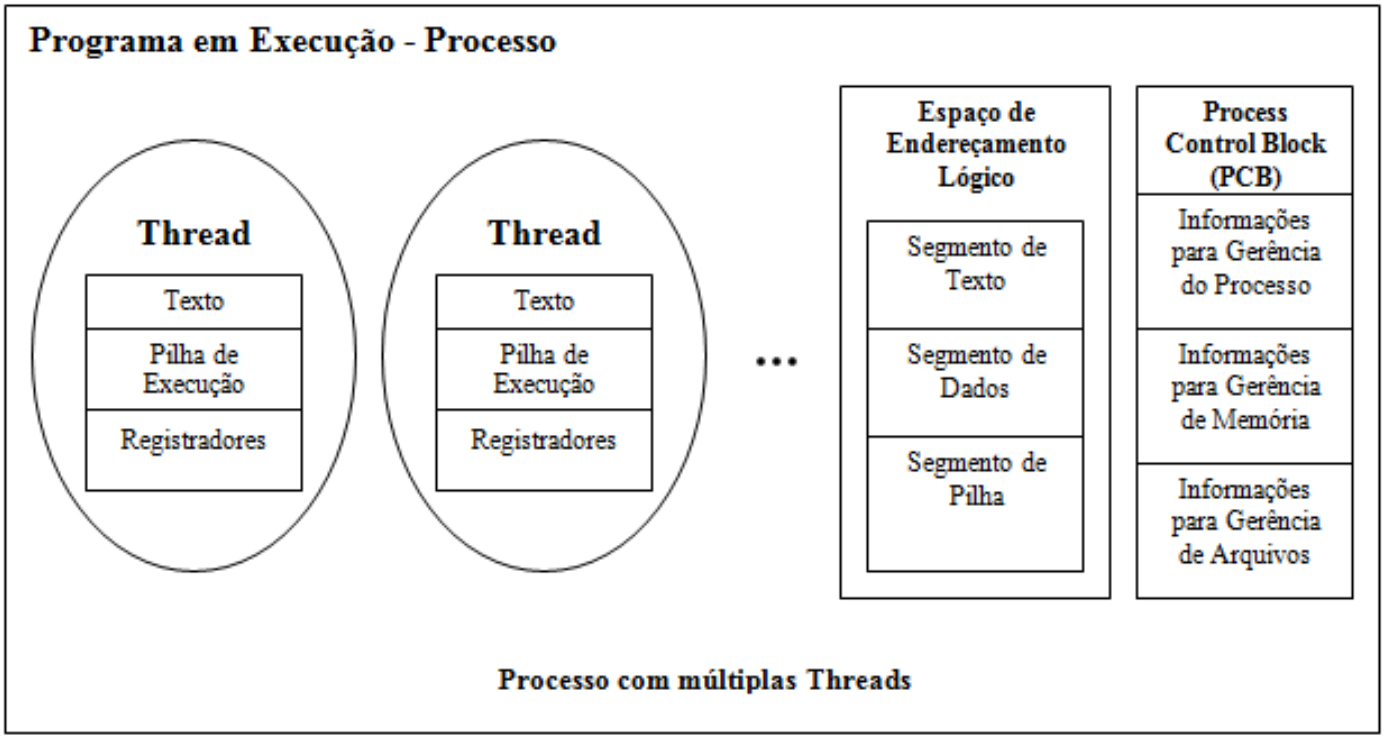
- Desenvolver uma thread é similar ao desenvolvimento de qualquer algoritmo, possuindo um início, uma sequência de execuções e um fim, contudo, uma thread não é um programa, ela não pode ser executada sozinha e sim, inserida no contexto de uma aplicação que possuirá vários pontos de execução distintos, cada um representado por uma thread.



Definição : Uma thread representa um fluxo de controle de execução dentro de um programa

Aplicação das Threads

- O grande benefício no uso de threads é quando temos várias threads em um mesmo processo sendo executadas simultaneamente e podendo realizar tarefas diferentes.



Definição : Com múltiplas theads um programa possui múltiplos pontos de execução

Aplicação das Threads

- Percebe-se, portanto, que aplicações multithreads podem realizar tarefas distintas simultaneamente, dando ideia de paralelismo. É possível exemplificar com um aplicativo que pode executar uma animação, tocar uma música, exibir figuras, mesmo rodando em um computador com apenas uma CPU. Cada thread realizará uma tarefa distinta.

- Cada thread, diferente de um processo, precisa apenas das informações necessárias a sua execução, compartilhando todo o contexto de execução do processo com as demais threads do mesmo conjunto de um processo.

- A linguagem Java possui mecanismos e classes voltados para o desenvolvimento multithreads:
  - A classe `java.lang.Thread` utilizada para criar, iniciar e controlar threads
  - As palavras reservadas `synchronized` e `volatile` usadas para controlar a execução de código em objetos compartilhados por múltiplas threads, permitindo exclusão mútua entre estas
  - Os métodos `wait`, `notify` e `notifyAll` definidos em `java.lang.Object` usados para coordenar as atividades das threads, permitindo comunicação entre elas.

Paralelismo x Concorrência

- Threads podem executar suas funções de forma paralela ou concorrente.
- São paralelas quando desempenham seu papel sem nenhuma dependência da execução de outra contida no mesmo conjunto.
- São concorrentes quando atuam sobre objetos compartilhados de forma simbiótica, necessitando de sincronismo no acesso a esses objetos, assim, deve ser garantido o direito de atomicidade e exclusão mútua das operações das threads sobre objetos compartilhados

Aplicação das Threads

- **Criando Threads**:
  - A criação de uma thread é feita através da chamada ao seu construtor colocando a thread no estado **Nova**, o qual representa um thread vazia, ou seja, nenhum recurso do sistema foi alocado a ela ainda. Quando uma thread está nesse estado, a única operação que pode ser realizada é a inicialização desta thread através do método `start()`, se qualquer outo método for chamado, irá acontecer uma exceção (IllegalThreadStateException), assim como quando qualquer método for chamado e sua ação não for condizente com o estado atual da thread.

- **Iniciando Threads**:
  - A inicialização de uma thread é feita através do método `start()` e, nesse momento, os recursos necessários para execução da mesmo são alocados, tais como recursos para execução, escalonamento e chamada do método `run()` da thread. Após a chamada ao método `start()`, a thread está pronta para ser executada e será, assim que for possível, ficando no estado de **Pronta** até chegar sua vez. Essa mudança de estado (Pronta / Executando) será feita pelo escalonador. O importante é que a thread está pronta para executar e a mesma será executada de acordo com os critérios do escalonador.

Aplicação das Threads

- **Fazendo Thread Esperar**:
  - Uma thread irá para estado de **Bloqueada** quando:
    - O método `sleep()` (faz a thread esperar por um determinado tempo) for chamado
    - O método `wiat()` (faz a thread esperar por uma determinada condição) for chamado
    - Quando houver solicitação de E/S

- Quando uma thread for para o estado de **Bloqueada**, ela retornará ao estado de **Pronta** quando a condição que a levou ao estado de **Bloqueada** for atendida, ou seja:
  - Se a thread solicitou dormir por determinado intervalo de tempo (`sleep()`), assim que este intervalo for encerrado
  - Se a thread solicitou esperar por um determinado evento (`wait()`), assim que esse evento ocorrer (outra thread fizer o chamado de `notify` ou `notifyAll`)
  - Se a solicitação de E/S foi atendida

- **Finalizando Threads**:
  - Uma thread é finalizada quando acabar a execução do seu método `run()`, então ela vai para o estado de **Morta**, onde seus recursos podem ser liberados e ela será eliminada.

- **Verificando se threads estão Rodando/Pronta/Bloqueada ou Nova/Morta**:
  - A classe Thread possui o método `isAlive()`, o qual permite verificar se uma thread está no estado **Rodando/Pronta/Bloqueada** ou no estado **Nova/Morta**. Quando o retorno do método for `true` a thread está participando do processo de escalonamento e quando o retorno for `false`, está fora do processo de escalonamento. Não é possível diferenciar **Rodando**, **Pronto** e **Bloqueada**, assim como também não é possível diferenciar entre **Nova** e **Morta**

Aplicação das Threads

- **Threads Daemon**:
  - São threads que rodam em background e realizam tarefas de limpeza ou manutenção, as quais devem rodar enquanto a aplicação estiver em execução. As threads daemon somente morrem quando o processo for finalizado, em outras palavras, um processo não finalizará enquanto suas threads daemon não forem finalizadas. Threads daemon são denominadas **Threads de Serviço** e as não daemon de **Threads de Usuário**.

- Os métodos para manipulação de thread daemon são:

`public final void setDaemon(boolean)`	Torna ou não uma thread daemon
`public final boolean isDaemon()`	Verifica se uma thread é daemon

Aplicação das Threads

- **Escalonamento de Threads**:
  - O escalonamento de threads é fundamental pois, certamente, haverá mais threads rodando que a capacidade de processadores multicore / multithreads de executá-los simultaneamente, assim, a condição de paralelismo é simulada pelo algoritmo do escalonador. As threads escalonáveis são as que estão em estado de Rodando ou Pronta e podem receber um valor inteiro de prioridade no intervalo [`MIN_PRIORITY ... MAX_PRIORITY`], quanto maior o valor inteiro, maior a prioridade da thread. Cada nova thread recebe a mesma da thread que a criou, podendo ser alterada através do método `setPriority(int priority)`.

  - O escalonador trabalha da seguinte forma:
    1. Quando várias threads estiverem **Prontas**, aquela que tiver maior prioridade será executada
    2. Quando existirem várias threads com prioridades iguais, seguirão o algoritmo de escalonamento Round Robin
    3. Uma thread será executada até que:
       - Outra thread de maior prioridade fique **Pronta**
       - Aconteça algum evento que a coloque em estado de **Bloqueada**
       - O método `run()` acabar
       - O Quantum dela esgotar
    4. Threads com prioridades mais baixas terão direito garantido de serem executadas para que não ocorram situações de starvation.

Exemplo de Threads (Java)

```java
package ex1;

public class Ex1ThreadDiv extends Thread {

    private int a, b, res;

    public Ex1ThreadDiv (int a, int b){
        this.a = a;
        this.b = b;
    }

    public void run(){
        res = a / b;
        System.out.println(a + " / " + b + " = "+ res);
    }

}
```

```java
package ex1;

public class Ex1ThreadMult extends Thread {

    private int a, b, res;

    public Ex1ThreadMult (int a, int b){
        this.a = a;
        this.b = b;
    }

    public void run(){
        res = a * b;
        System.out.println(a + " * " + b + " = "+ res);
    }

}
```

```java
package ex1;

public class Ex1ThreadSub extends Thread {

    private int a, b, res;

    public Ex1ThreadSub (int a, int b){
        this.a = a;
        this.b = b;
    }

    public void run(){
        res = a - b;
        System.out.println(a + " - " + b + " = "+ res);
    }

}
```

```java
package ex1;

public class Ex1ThreadSoma extends Thread {

    private int a, b, res;

    public Ex1ThreadSoma (int a, int b){
        this.a = a;
        this.b = b;
    }

    public void run(){
        res = a + b;
        System.out.println(a + " + " + b + " = "+ res);
    }

}
```

Exemplo de Threads (Java)

```
package ex1;

public class Exemplo1 {

    static int a = 8;
    static int b = 3;

    public static void main(String[] args) {

        Ex1ThreadSoma ex1ThreadSoma = new Ex1ThreadSoma(a, b);
        Ex1ThreadDiv ex1ThreadDiv = new Ex1ThreadDiv(a, b);
        Ex1ThreadMult ex1ThreadMult = new Ex1ThreadMult(a, b);
        Ex1ThreadSub ex1ThreadSub = new Ex1ThreadSub(a, b);
        ex1ThreadSoma.start();
        ex1ThreadSub.start();
        ex1ThreadMult.start();
        ex1ThreadDiv.start();
    }

}
```

Exemplo de Threads (Java)

- Resultados Possíveis :

```
8 + 3 = 11
8 * 3 = 24
8 - 3 = 5
8 / 3 = 2
```

```
8 + 3 = 11
8 / 3 = 2
8 * 3 = 24
8 - 3 = 5
```

ETC ...

```
8 - 3 = 5
8 / 3 = 2
8 + 3 = 11
8 * 3 = 24
```

Exemplo de Threads (Java)

```
public class Exemplo2 {

    static int a = 10;
    static int b = 2;

    public static void calc(int a, int b, int tipoOperacao){
        int res = 0;
        String op = "";
        switch(tipoOperacao){
        case 0:
            res = a + b;
            op = "+";
            break;
        case 1:
            res = a - b;
            op = "-";
            break;
        case 2:
            res = a * b;
            op = "*";
            break;
        case 3:
            res = a / b;
            op = "/";
            break;
        }
        System.out.println(a + " " + op + " " + b + " = " + res);
    }

    static Thread tSoma = new Thread(){
        public void run(){
            calc(a, b, 0);
        }
    };

    static Thread tSub = new Thread(){
        public void run(){
            calc(a, b, 1);
        }
    };

    static Thread tMult = new Thread(){
        public void run(){
            calc(a, b, 2);
        }
    };

    static Thread tDiv = new Thread(){
        public void run(){
            calc(a, b, 3);
        }
    };

    public static void main(String[] args) {
        tSoma.start();
        tSub.start();
        tMult.start();
        tDiv.start();
    }

}
```

Exemplo de Threads (Java)

```java
package ex2;

public class Ex2ThreadCalc extends Thread {

    private int a, b, operacao;

    public Ex2ThreadCalc(int a, int b, int operacao){
        this.a = a;
        this.b = b;
        this.operacao = operacao;
    }

    public void run(){
        calc(a,b,operacao);
    }


    public static void calc(int a, int b, int operacao){
        int res = 0;
        String op = "";
        switch(operacao){
        case 0:
            res = a + b;
            op = "+";
            break;
        case 1:
            res = a - b;
            op = "-";
            break;
        case 2:
            res = a * b;
            op = "*";
            break;
        case 3:
            res = a / b;
            op = "/";
            break;
        }
        System.out.println(a + " " + op + " " + b + " = " + res);
    }
```

```java
package ex2;

public class Exemplo2 {

    static int a = 8;
    static int b = 3;

    public static void main(String[] args) {

        for (int i = 0 ; i < 4 ; i++){
            int tempoEspera = (int) (Math.random() * 1000);
            try {
                Thread.sleep(tempoEspera);
                Ex2ThreadCalc ex2ThreadCalc = new Ex2ThreadCalc(a, b, i);
                ex2ThreadCalc.start();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```