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 vimos, cada processo conta com uma estrutura de controle razoavelmente sofisticada. Nos casos onde se deseja realizar duas ou mais tarefas *simultaneamente*, o solução trivial a disposição do programador é dividir as tarefas a serem realizadas em dois ou mais processos. Isto implica na criação de manutenção de duas estruturas de controle distintas para tais processos, onerando o sistema, e complicando também o compartilhamento de recursos, dados 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 divido em *threads*, cada uma das *threads* componentes recebe a atenção do processador como um processo comum. 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 processo podem ser compartilhados de maneira bastante mais simples entre as suas *threads* componentes.

Threads

Segundo Tanebaum, "*as* threads *foram inventadas para permitir a combinação de paralelismo com execução seqüencial e chamadas de sistema bloqueantes*" [TAN92, p. 509]. Na Figura 2.19 representamos os fluxos de execução de um processo comum e de outro, divido em *threads*.

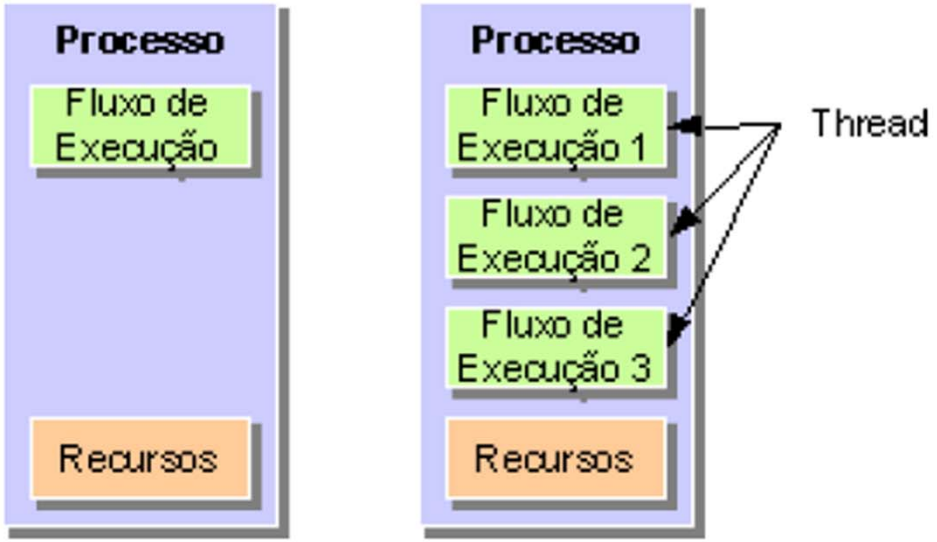


Figura 2.19: Processos e *threads*

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 o que consideramos processos leves (*lightweight processes*) e constituem uma unidade básica de utilização do processador [TAN92, p. 508] [SGG01, p. 82].

Sistemas computacionais que oferecem suporte para as *threads* são usualmente conhecidos como sistemas *multithreading*. Como ilustrado na Figura 2.20, os sistemas *multithreading* podem suportar as *threads* segundo dois modelos diferentes:

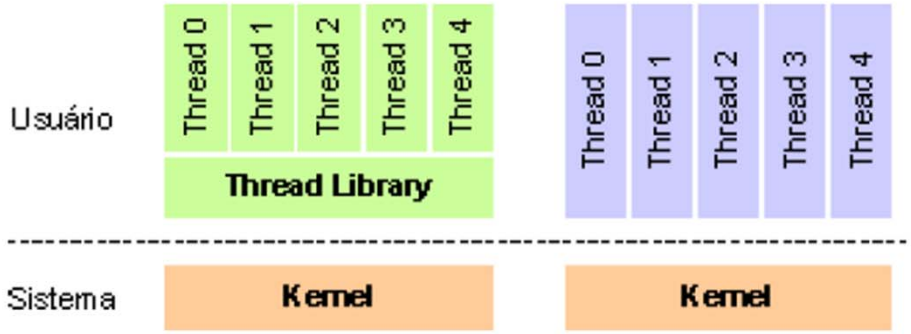


Figura 2.20: *Threads* de usuário e de *kernel*

Threads

User threads As *threads* de usuário são aquelas oferecidas através de bibliotecas específicas e adicionais ao sistema operacional, ou seja, são implementadas acima do *kernel* (núcleo do sistema) utilizando um modelo de controle que pode ser distinto do sistema operacional, pois não são nativas neste sistema.

Kernel threads As *threads* do sistema são aquelas suportadas diretamente pelo sistema operacional e, portanto, nativas.

Em sistemas não dotados de suporte a *threads* nativamente, o uso de bibliotecas de extensão permite a utilização de *pseudo-threads*. Exemplos de bibliotecas de suporte *threads* de usuário são o **PThreads** do POSIX ou o **C-threads** do sistema operacional Mach.

Através de tais biblioteca são oferecidos todos os recursos necessários para a criação e controle das *threads*. Usualmente os mecanimos de criação de *threads* de usuário são bastante rápidos e simples, mas existe uma desvantagem: quando uma *thread* é bloqueada (por exemplo, devido ao uso de recursos de I/O), as demais *threads* freqüentemente também são devido ao suporte não nativo. Quando o suporte é nativo, a criação das *threads* é usualmente mais demorada, mas não ocorrem os inconveniente decorrentes do bloqueio de uma ou mais *threads* em relação às demais.

Modelos de Multithreading

A forma com que as *threads* são disponibilizadas para os usuários é o que denominamos modelos de *multithreading*. Como mostra a Figura 2.21, são comuns três modelos distintos:

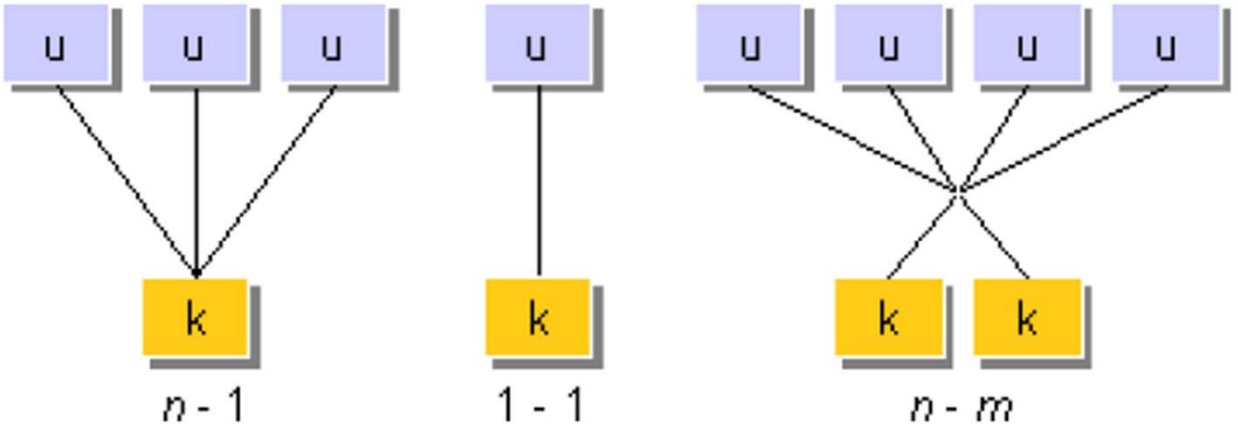


Figura 2.21: Modelos de *multithreading*

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 *threads* do usuário é associada a uma *thread* nativa do sistema. Este modelo é empregado em sistemas operacionais tais como o MS-Windows NT/2000 e no IBM OS/2.

- 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 necessáriamente do mesmo tamanho (`m`). Este modelo é empregado em sistemas operacionais tais como o Sun Solaris, Irix e Digital UNIX.

Benefícios do Multithreading

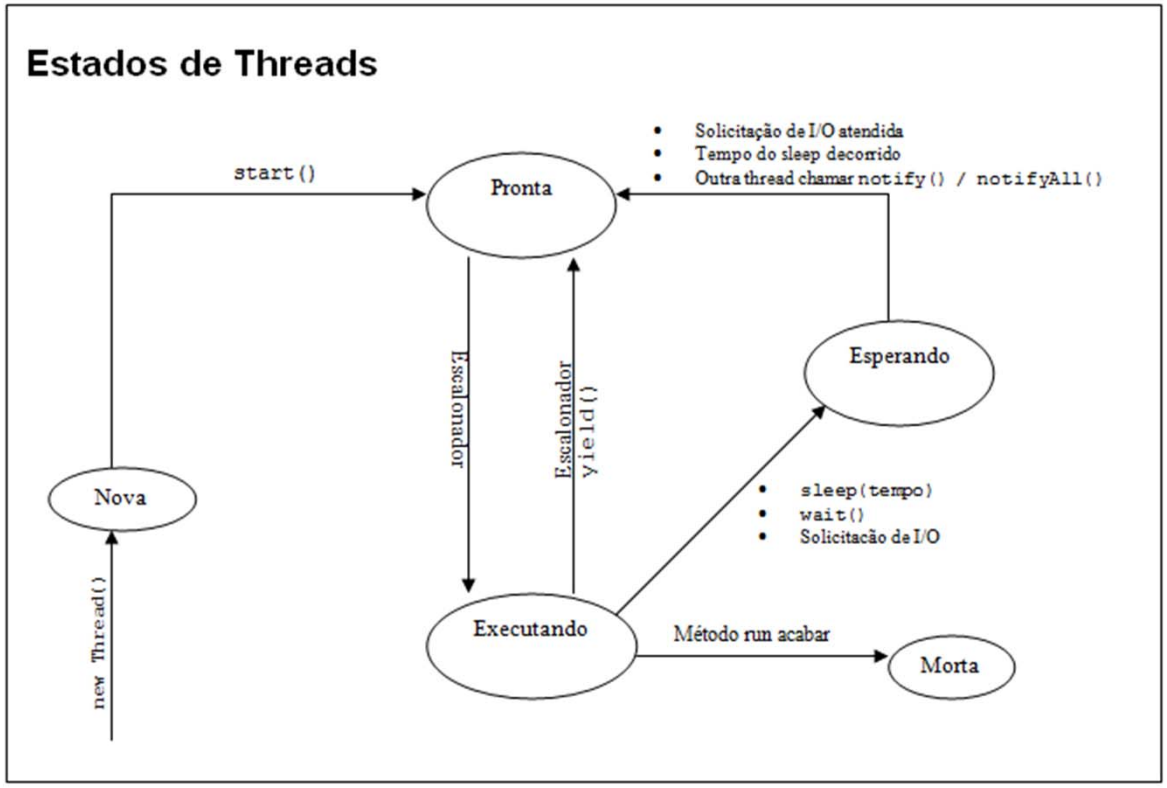
A utilização das *threads* pode trazer diversos benefícios para os programas e para o sistema computacional em si [SGG01, p. 83]:

- Melhor capacidade de resposta, pois a criação de uma nova *thread* e substancialmente mais rápida do 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.
- Economia, pois o uso de estruturas de controle reduzidas em comparação ao controle típico dos processos, desoneramos o sistema. Além disso o compartilhamento de recursos simplificado leva também a economia de outros recursos.
- Permitem explorar mais adequadamente as arquiteturas computacionais que dispõem de múltiplos processadores.

Aplicação das Threads

O ciclo de vida de uma Thread

A melhor forma de analisar o ciclo de vida de uma thread é através das operações que podem ser feitas sobre as mesmas, tais como : criar, iniciar, esperar, parar e encerrar. O diagrama abaixo ilustra os estados os quais uma thread pode assumir durante o seu ciclo de vida e quais métodos ou situações levam a estes estados.



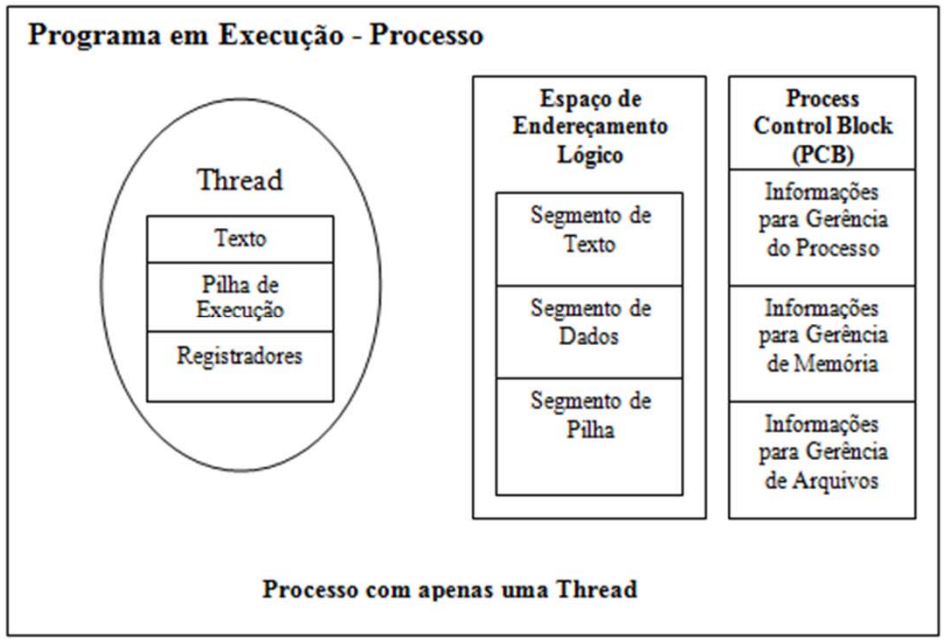
Ciclo de vida de uma thread
(Nova/Executando/Pronta/Esperando/Morta)

Aplicação das Threads

Thread em Java

Todo programador está familiarizado com a programação sequencial, pois sem dúvida até o presente momento esta é a forma de programação mais comum. Programas do tipo "Hello World", ou que ordenam uma lista de nome, ou que gera e imprime uma lista de números primos, etc são programas tipicamente sequenciais, onde cada um possui : seu início, sequência de execução e fim e em qualquer momento da sua execução estes programas possuem apenas um ponto de execução.

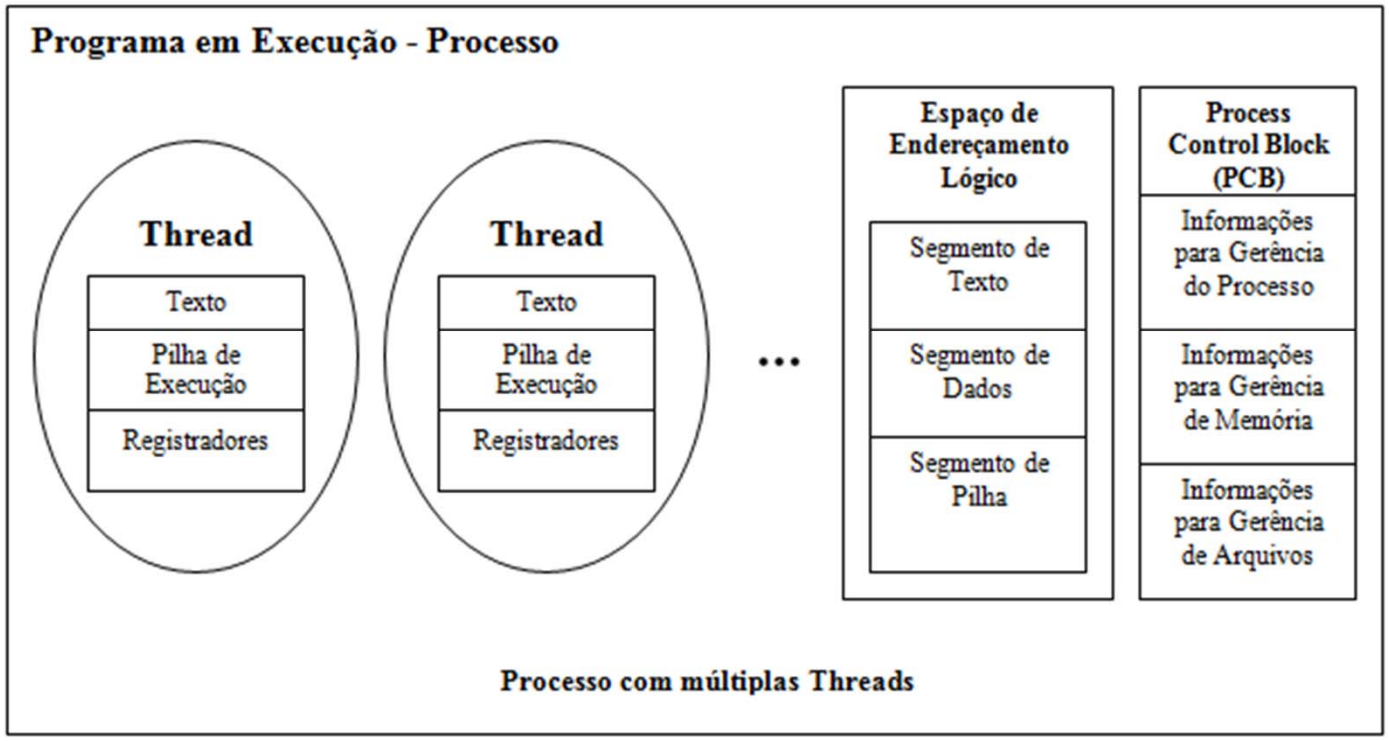
Uma thread é similar ao programas sequenciais, pois possui um início, sequência de execução e um fim e em qualquer momento uma thread possui um único ponto de execução. Contudo, uma thread não é um programa, ela não pode ser executada sozinha e sim inserida no contexto de uma aplicação, onde essa aplicação sim, possuirá vários pontos de execuções distintos, cada um representado por uma thread. A figura abaixo descreve esse mecanismo de funcionamento da thread dentro de um programa em execução.



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

Aplicação das Threads

Não há nada de novo nesse conceito de processo com uma única thread, pois o mesmo é idêntico ao conceito tradicional de processo. O grande benefício no uso de thread é quando temos várias thread num mesmo processo sendo executadas simultaneamente e podendo realizar tarefas diferentes. A figura abaixo representa um processo com múltiplas threads (Programação Multi-Thread.)



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

Aplicação das Threads

Dessa forma podemos perceber facilmente que aplicações multithreads podem realizar tarefas distintas ao "mesmo tempo", dando idéia de paralelismo. Veja um exemplo do navegador web HotJava, o qual consegue carregar e executar applets, executar uma animação, tocar um som, exibir diversas figuras, permitir rolagem da tela, carregar uma nova página, etc e para o usuário todas essas atividades são simultâneas, mesmo possuindo um único processador. Isso é possível, por que dentro da aplicação do navegador HotJava várias threads foram executadas, provavelmente, uma para cada tarefa a ser realizada.

Alguns autores tratam threads como processos leves. Uma thread é similar a um processo no sentido que ambos representam um único fluxo de controle de execução, sendo considerada um processo leve por ser executada dentro do contexto de um processo e usufruir dos recursos alocados pelo processo. Cada thread necessita possuir apenas as informações (contador de programa e pilha de execução) necessárias a sua execução, compartilhando todo o contexto de execução do processo com todas a demais threads do mesmo processo.

A linguagem Java possui apenas alguns mecanismos e classes desenhadas com a finalidade de permitir a programação Multi-Thread, o que torna extremamente fácil implementar aplicações Multi-Threads, sendo esses :

- 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` and `notifyAll` definidos em `java.lang.Object` usados para coordenar as atividades das threads, permitindo comunicação entre estas.

Aplicação das Threads

Paralelismo x Concorrência

Threads podem executar suas funções de forma paralela ou concorrente, onde quando as threads são paralelas elas desempenham o seus papeis independente uma das outras. Já na execução concorrente, as threads 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 nas operações das threads sobre objetos compartilhados.

Aplicação das Threads

Criando Threads

A criação de uma thread é feita através da chamado ao seu construtor colocando a thread no estado **Nova**, o qual representa uma thread vazia, ou seja, nenhum recurso do sistema foi alocado para ela ainda. Quando uma thread está nesse estado a única operação que pode ser realizada é a inicialização dessa thread através do método `start()`, se qualquer outro método for chamado com a thread no estado **Nova** 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 mesma 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, até lá ficará no estado **Pronta**. Essa mudança de estado (Pronta/Executando) será feito pelo escalonador do Sistema de Execução Java. O importante é saber que a thread está pronta para executar e a mesma será executada, mais cedo ou mais tarde de acordo com os critérios, algoritmo, de escalonamento do Sistema de Execução Java.

Aplicação das Threads

Fazendo Thread Esperar

Uma thread irá para o estado **Esperando** quando :

- O método `sleep` (faz a thread esperar por um determinada tempo) for chamado;
- O método `wait` (faz a thread esperar por uma determinada condição) for chamado;
- Quando realizar solicitação de I/O.

Quando um thread for para estado Esperando ela retornará ao estado **Pronta** quando a condição que a levou ao estado **Esperando** for atendida, ou seja :

- Se a thread solicitou dormir por determinado intervalo de tempo (`sleep`), assim que este intervalo de tempo for decorrido;
- Se a thread solicitou esperar por determinado evento (`wait`), assim que esse evento ocorrer (outra thread chamar `notify` ou `notifyAll`);
- Se a thread realizou solicitação de I/O, assim que essa solicitação for atendida.

Finalizando Threads

Uma Thread é finalizada quando acabar a execução do seu método `run`, e então ela vai para o estado Morta, onde o Sistema de Execução Java poderá liberar seus recursos e eliminá-la .

Verificando se Threads estão Executando/Pronta/Esperando ou Novas/Mortas

A classe `Thread` possui o método `isAlive`, o qual permite verificar se uma thread está nos estado **Executando/Pronta/Esperando** ou nos estados **Nova/Morta**. Quando o retorno do método for `true` a thread esta participando do processo de escalonamento e o retorno for `false` a thread está fora do processo de escalonamento. Não é possível diferenciar entre **Executando**, **Pronta** ou **Esperando**, assim também como não é possível diferenciar entre **Nova** ou **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 daemons somente morrerem quando a aplicação for encerrada. Sendo que o Sistema de Execução Java identifica que uma aplicação acabou quando todas as suas threads estão morta, assim para que seja possível que uma aplicação seja encerrada ainda possuindo threads, tais threads devem ser daemon. Em outras palavras, o Sistema de Execução Java irá terminar uma aplicação quando todas as suas threads **não** daemon morrerem. Threads daemons são denominadas de **Threads de Serviço** e as não daemon são denominadas de **Threads de Usuário**.

Os métodos para manipulação de threads 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 é fundamental quando é possível a execução paralela de threads, pois, certamente existirão mais threads a serem executadas que processadores, assim a execução paralela de threads é simulada através de mecanismos do escalonamento dessas threads, onde os processadores disponíveis são alternados pelas diversas threads em execução. O mecanismo de escalonamento utilizado pelo Sistema de Execução Java é bastante simples e determinístico, e utiliza um algoritmo conhecido como **Escalonamento com Prioridades Fixas**, o qual escalona threads baseado na sua prioridade. As threads escalonáveis são aquelas que estão nos estados **Executando** ou **Pronta**, para isso toda thread possui uma prioridade, a qual pode ser um valor inteiro no intervalo [MIN_PRIORITY ... MAX_PRIORITY], (estas são constantes definidas na classe Thread), e quanto maior o valor do inteiro maior a prioridade da thread. Cada thread Nova recebe a mesma prioridade da thread que a criou e a prioridade de uma thread pode ser alterada através do método `setPriority(int priority)`.

O algoritmo de escalonamento com Prioridades Fixas utilizado pela Sistema de Execução Java funcionada da seguinte forma :

1. Quando várias threads estiverem Prontas, aquela que tiver a maior prioridade será executada.
2. Quando existir várias threads com prioridades iguais, as mesmas serão escalonadas segundo o algoritmo Round-Robin de escalonamento.
3. Uma thread será executada até que : uma outra thread de maior prioridade fique Pronta; acontecer um dos eventos que a faça ir para o estado Esperando; o método `run` acabar; ou em sistema que possuam fatias de tempo a sua fatia de tempo se esgotar.
4. Threads com prioridades mais baixas terão direito garantido de serem executadas para que situações de starvation não ocorram.

Exemplo de Threads (Java)

```
public class Exemplo1 {

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

    static Thread tSoma = new Thread(){
        public void run(){
            int res = a + b;
            System.out.println(a + " + " + b + " = " + res);
        }
    };

    static Thread tSub = new Thread(){
        public void run(){
            int res = a - b;
            System.out.println(a + " - " + b + " = " + res);
        }
    };

    static Thread tMult = new Thread(){
        public void run(){
            int res = a * b;
            System.out.println(a + " * " + b + " = " + res);
        }
    };

    static Thread tDiv = new Thread(){
        public void run(){
            int res = a / b;
            System.out.println(a + " / " + b + " = " + res);
        }
    };

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

```
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;
        }
        int tempoEspera = (int) (Math.random() * 1000);
        try {
            Thread.sleep(tempoEspera);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        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();
    }

}
```