5 THREADS

Como foi exposto anteriormente, os processos podem ter mais de um fluxo de execução. Cada fluxo de execução é chamado de *thread*.

5.1 VISÃO GERAL

Uma definição mais abrangente para *threads* é considerá-lo como uma unidade básica de uso do processador. Ele contém uma mesma estrutura dos processos com: ID de *thread*, contador de programa, conjunto de registradores e uma pilha. Processos com múltiplos *threads* podem ter mais de uma tarefa realizada ao mesmo tempo. A Figura 5.1 ilustra o processo de *multithread*.

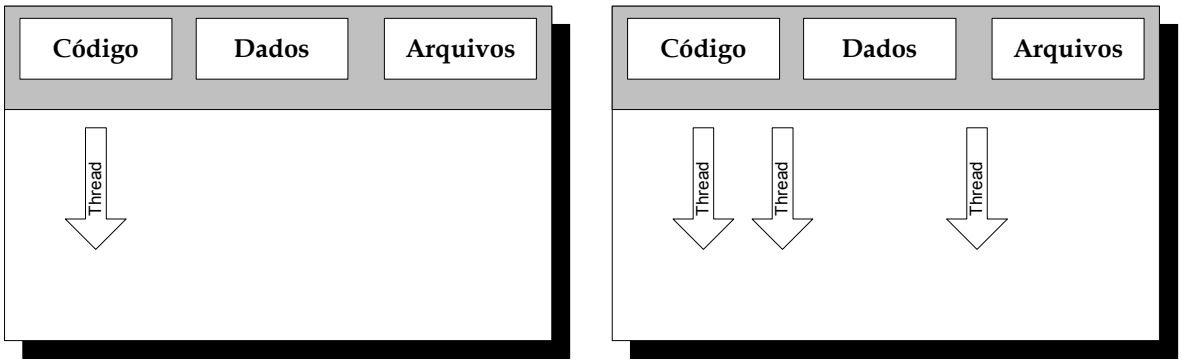


Figura 5-1 - Multithread

Muitos softwares implementam o conceito de *multithread*. Um *browser* de Internet, por exemplo, realiza um *thread* para exibir os textos e imagens e outro para carregar informações da rede. Um processador de texto pode ter um *thread* para auto-salvamento do arquivo e outro para fazer análise ortográfica.

Existem situações onde uma única aplicação pode ser solicitada a realizar várias tarefas semelhantes. Um exemplo interessante é um servidor *Web* que processa várias requisições de diversos clientes. Sem o conceito de *threads*, o servidor iria executar apenas uma requisição por vez. Para solucionar essa limitação, o processo do servidor é dividido em *threads* conforme a Figura 5.2.

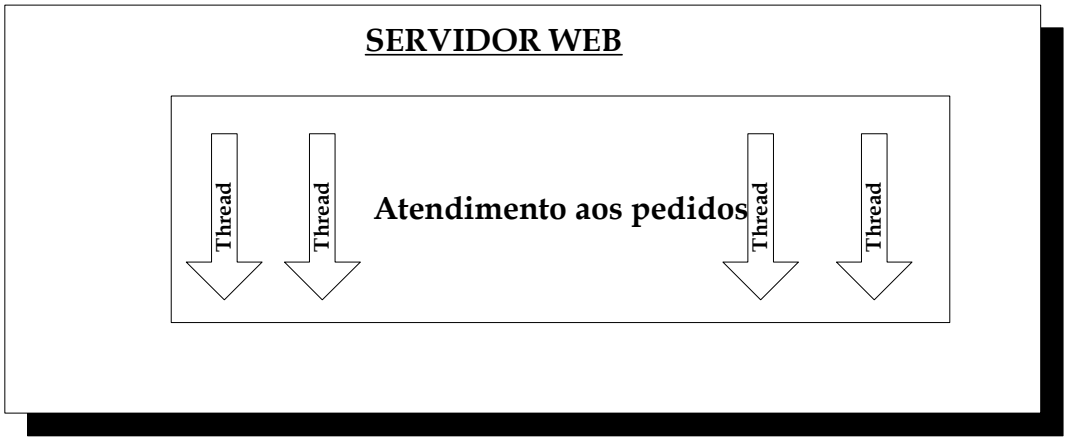


Figura 5-2 - Exemplo de multithreading em servidores WEB

5.2 BENEFÍCIOS

Os benefícios do uso de threads são:

- **Capacidade de resposta**: ela permite que um processo fique executando, mesmo que uma parte desse processo esteja bloqueada. Isto é possível desde que haja *threads* independentes. Isso aumenta a capacidade de resposta dos processos;
- **Compartilhamento de recursos** – os *threads* de um processo podem compartilhar os recursos do mesmo, incluindo memória;
- **Economia** – a alocação de recursos e memória a diversos processos possui um custo computacional muito alto. Então é mais atrativa a implementação de multithreads, pois seu custo computacional é menor;
- **Utilização de arquiteturas multiprocessador** – em uma arquitetura multiprocessador, onde cada thread pode ser executado em um processador diferente;

5.3 THREADS DE USUÁRIOS E DE KERNEL

5.3.1 THREADS DE USUÁRIO

Esses *threads* estão localizados em um nível superior ao *kernel* e são implementados através de uma biblioteca de *threads* no nível de usuários. Esta biblioteca fornece suporte à criação, escalonamento e gerência de *threads* sem a intervenção do *kernel*. Eles são mais rápidos e fáceis de gerenciar.

Entretanto, há uma séria desvantagem em usar *threads* de usuário. Se um *thread* fizer uma chamada bloqueante, todo o processo estará bloqueado, pois o *kernel* não tem acesso ao mesmo.

5.3.2 THREADS DE KERNEL

Estes *threads* são suportados diretamente pelo sistema operacional. Todo o gerenciamento dos *threads* é feito pelo *kernel*. Se um *thread* fizer uma chamada bloqueante, o *kernel* pode chamar outro *thread* para execução.

5.4 MODELOS DE MULTITHREADING

5.4.1 MUITOS PARA UM

No modelo muitos para um, vários *threads* de usuário são mapeados em um único *thread* de *kernel*. A gerência de *threads* fica no nível de usuário, o que o torna mais rápido, porém o processo inteiro será bloqueado em caso de um *thread* fazer uma chamada bloqueante. Além disso, mesmo em sistemas multiprocessadores não será possível executar *multithreads* porque o *kernel* só pode ser acessado por um *thread* por vez. A Figura 5.3 ilustra um exemplo desse modelo.

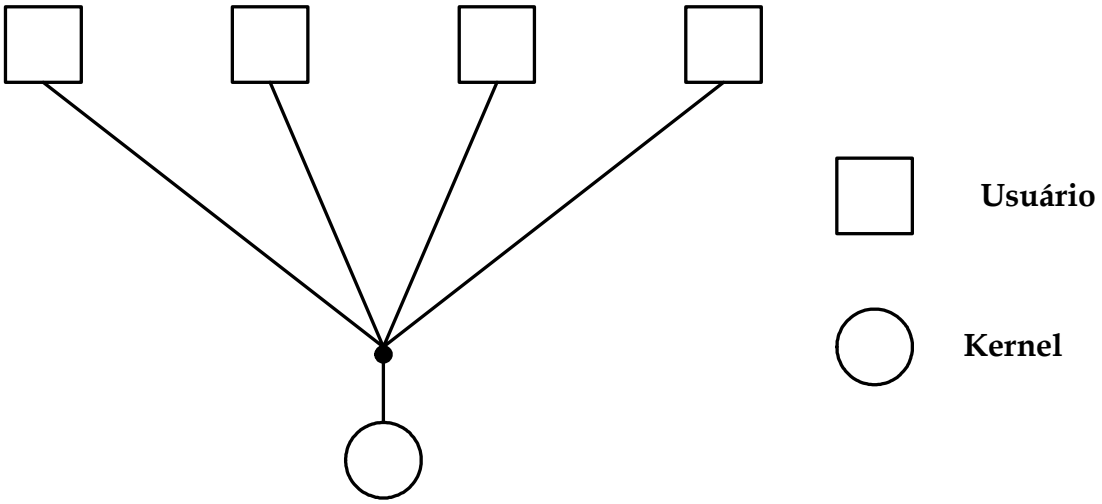


Figura 5-3 - Muitos para um

5.4.2 UM PARA UM

Neste modelo, cada thread de usuário é mapeado em um thread de kernel. Há uma maior concorrência que no modelo muitos para um, pois permite que assim que thread execute uma tarefa bloqueante, um outro seja chamado à execução. Entretanto, para cada thread de usuário criado é preciso criar um thread de kernel, o que implica em uma queda de desempenho do sistema. Este modelo é aplicado nos sistemas Windows NT e OS/2. A Figura 5.4 ilustra um exemplo de modelo um para um.

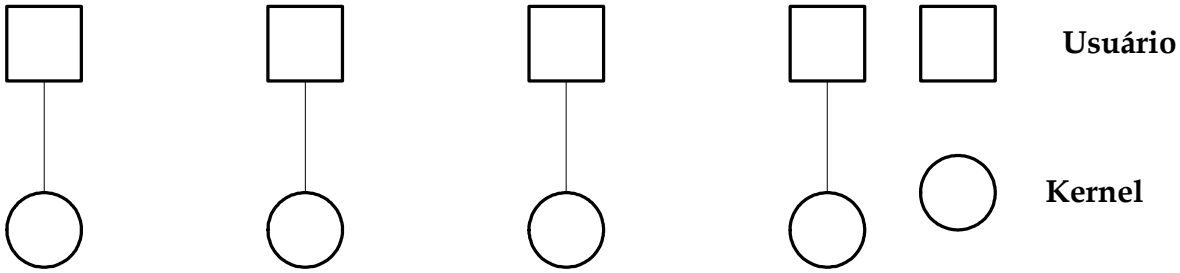


Figura 5-4 - Um para Um

5.4.3 MUITOS PARA MUITOS

Neste modelo, os threads de usuários são multiplexados em um número menor de threads de kernel. O número de threads de kernel pode ser específico para cada aplicação. No modelo "Muitos para Um" não é permitida a verdadeira concorrência, pois uma chamada bloqueante não permite um outro thread ser chamado. Já no modelo um para um, apesar de permitir a concorrência, é preciso ter muito cuidado, pois não é recomendado criar muitos threads em uma única aplicação, implicando em muitos threads de kernel. O modelo "Muitos para Muitos" não apresenta estas desvantagens. O Sistema Solaris, IRIX e Digital UNIX utilizam esse modelo multithreading.

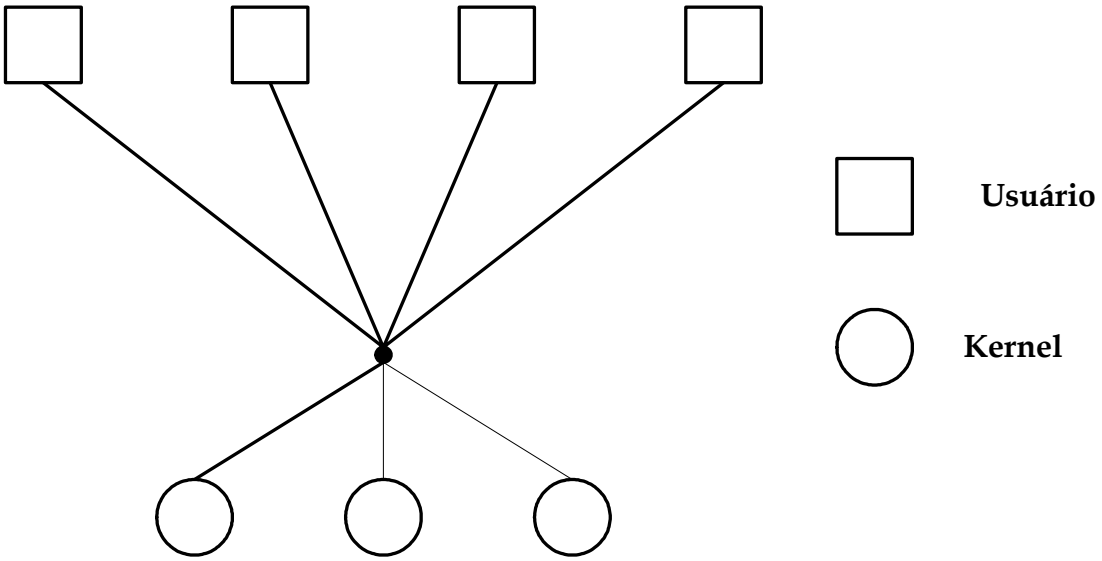


Figura 5-5 - Muitos para Muitos

5.5 THREADS DE JAVA

O suporte a *multithreading* é fornecido pelo sistema operacional ou por uma biblioteca. A biblioteca Win32 fornece um conjunto de APIs para efetuar

multithreading em aplicações nativas do Windows. A linguagem Java fornece o suporte a *threads* ao nível de linguagem de programação.

Todos os programas escritos em Java possuem pelo menos um thread de controle. A plataforma Java permite a criação e manipulação de threads graças a sua arquitetura de máquina virtual.

5.5.1 CRIAÇÃO DE THREADS

Uma forma de criar um *thread* em Java é criar uma nova classe derivada da classe Thread e redefinir o método run() dessa classe.

Um objeto dessa classe derivada executará como um *thread* de controle separado na máquina virtual Java. A criação de um objeto derivado da classe Java só especifica o *thread*. A sua criação fica por conta do método start(). Este método aloca memória e inicializa um novo *thread* na máquina virtual. Quando o método run() é executado o *thread* é passível de execução. A Figura 5.6 ilustra um exemplo de programação com threads em Java usando uma derivada da classe Thread.

```
Class Worker1 extends Thread
{
        public void run(){
                System.out.println("Trabalho em um thread");
        }
}

public class First
{
        public static void main(String args[])
        {
                Worker1 runner = new Worker1();
                runner.start();
                System.out.println("Trabalhando no thread principal");
        }
}
```

Figura 5-6 – Uso da classe thread

Outra forma de implementar um *thread* é definir uma classe com a interface Runnable, conforme ilustrado na Figura 5.7.

```
public interface Runnable
{
        public abstract void run();
}
```

Quando uma classe implementa Runnable, deverá definir um método run() e implementá-la é bastante similar a derivar a classe Thread, conforme está ilustrado na Figura 5.9.

```
class Worker2 implements Runnable
{
        public void run()
        {
                System.out.println("Trabalhando em um thread");
        }
}
```

Entretanto, criar um thread a partir de uma classe com interface Runnable é um pouco diferente de criar através da derivação da classe Thread com o método start(), conforme é ilustrado na Figura 5.10.

```
public class Second
{
        public static void main(String args[])
        {
                Runnable runner = new Worker2();
                Thread thrd = new Thread(runner)
                thrd.start();
                System.out.println("Trabalhando no thread principal");
        }
}
```

Na classe Second, um novo objeto Thread é criado, sendo passado um objeto Runnable no seu construtor. Quando o thread é criado com o método start(), o novo thread começa a execução no método run() do objeto Runnable.

Todos esses métodos são bastante utilizados e não há fatores comprovados de qual é a melhor abordagem ou em quais situações. O fato de haver duas abordagens de threads em Java é explicado pela característica do Java não suportar múltiplas heranças. Ou seja, se uma classe é derivada de uma outra, não poderá adquirir as características de outra classe.

5.5.2 GERÊNCIA DE THREADS

Para gerenciar e manipular os *threads*, a linguagem Java fornece uma série de métodos:

- suspend() – suspende a execução de um thread que estiver em processamento naquele momento;
- sleep() – suspende a execução de um thread por um determinado período de tempo;
- resume() – retorna a execução de um thread que estava suspenso;
- stop() – interrompe permanentemente a execução de um thread.

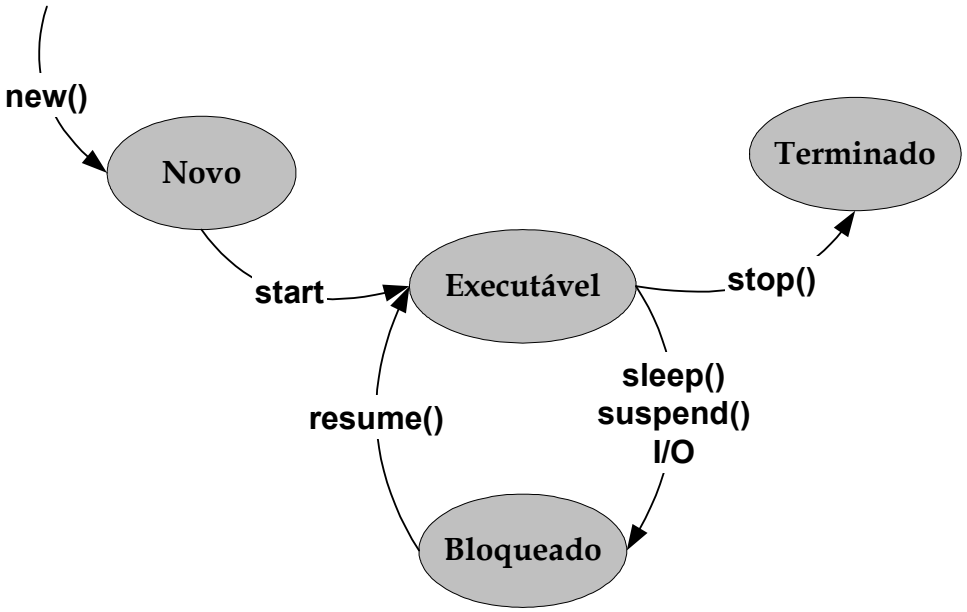
Cada um desses métodos de controle é útil em diferentes situações. A Figura 5.11

```
import java.applet.*;
import java.awt.*;
public class ClockApplet extends Applet imlements Runnable
{
	public void run()
	{
		while (true)
		{
			try
			{
				Thread.sleep(1000)
			}
			catch (InterruptedException e) { }
			repaint();
		}
	}
	public void start()
	{
		if (clockThread == null)
		{
			clockThread = new Thread(this);
			clockThread.start();
		}
		else
		{
			clockThread.resume();
		}
	}
	public void stop()
	{
		if(clockThread != null)
		{
			clockThread.suspend();
		}
	}
	public void destroy()
	{
		if(clockThread != null)
		{
			clockThread.stop();
			clockThread = null;
		}
	}
	public vois paint(Graphics g)
	{
		g.drawString(new java.util.Date().toString(),10,30);
	}
	private Thread clockThread;
}
```

5.5.3 ESTADOS DE UM THREAD

Assim como os processos, um thread pode estar em uma série de estados que são descritos a seguir e ilustrados na Figura 5.11:

- Novo – um thread está nesse estado quando um objeto para o thread é criado: método new();
- Executável – chamar os método start() aloca memória para o novo thread na Máquina Virtual Java e chama o método run() para o objeto thread. Quando o método run() de um thread é chamado, o thread passa do estado Novo para Executável, onde pode ser executado pela Máquina Virtual Java. Não existe distinção entre um thread passível de ser executado e um em execução;
- Bloqueado – para entrar nesse estado, o thread deve executar alguma instrução bloqueante como operações de I/O ou usar métodos suspend() ou sleep();
- Terminado – o método run() do thread termina ou é executado o método stop().



5.5.4 THREADS E A MÁQUINA VIRTUAL JAVA

Além dos threads dos programas, há outros que são executados de forma assíncrona com a Máquina Virtual Java. Que gerenciam a memória e gráficos. Por exemplo, verificam quais posições de memória não estão mais em uso.