UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
Bacharelado em Sistemas de Informação

Processamento Paralelo
Threads

Aluno:
Wagner Palacio

Turma:
3º período

Professor:
Giordano Cabral

Recife, 29 de maio de 2012

Sumário

1. O que é um Processo?

Um processo é basicamente um programa sendo executado dentro de um espaço de endereçamento individual na memória do computador. Ele é executado passo após passo, formando uma **linha de execução**. Quando uma execução é finalizada, a outra é iniciada, e assim por diante, e essa é a forma que os sistemas mais tradicionais funcionam.

2. O que é um Processamento Paralelo (*Threads*)?

Pode-se afirmar que um programa está sendo executado em Processamento Paralelo quando há várias linhas de execução num mesmo programa, ou seja, há dois ou mais comandos sendo executados paralelamente, sem que haja a interdependência deles, onde um só poderia ser iniciado quando o outro terminasse, e essas execuções dentro de um mesmo processo são chamadas de *Threads*.

O suporte a *Threads* pode ser fornecido pelo próprio SO, no caso da Kernel-Level Thread (KLT), ou implementada através de uma biblioteca de uma determinada linguagem, no caso de uma User-Level Thread (ULT).

Threads também são chamados de processos leves, pois se deve ao fato de o tempo de execução com atividades de criação e escalonamento de *threads* serem menores, se forem comparadas a processos. Cada thread tem o mesmo contexto de software e compartilha o mesmo endereçamento de memória de um mesmo processo pai, fato que se deve ter cuidado na implementação pelo usuário, pois as *threads* compartilham as mesmas variáveis globais e podem fazer alterações em informações executadas por outra thread.

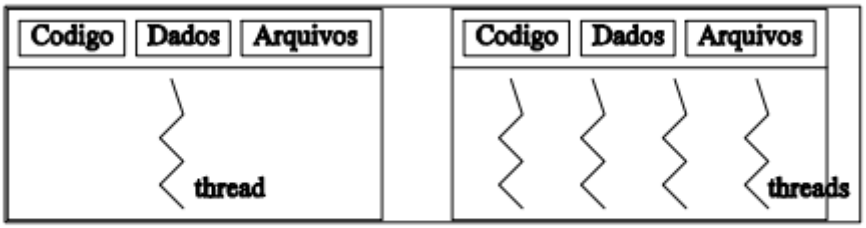
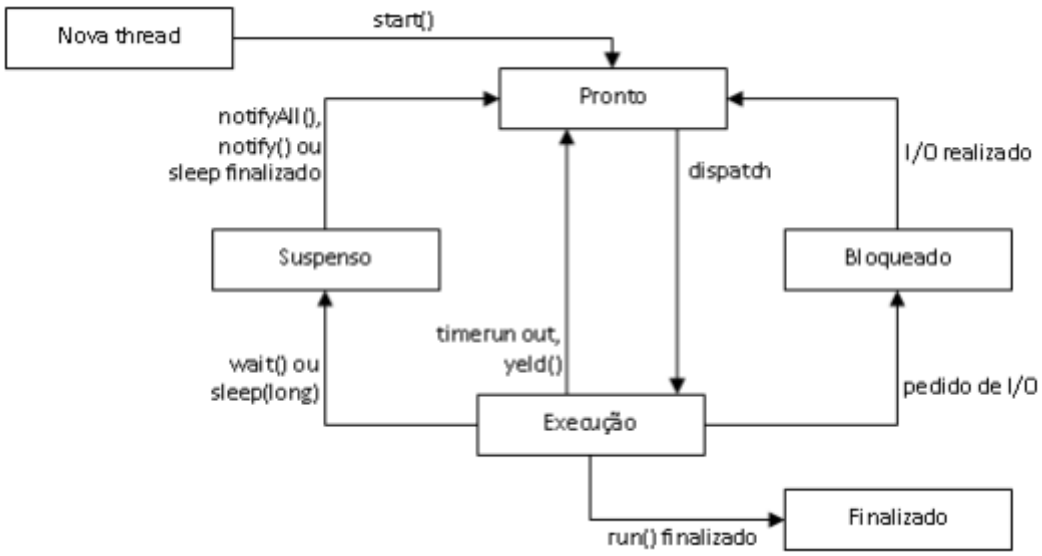


Figura 1. Processos com um e com múltiplos threads.

2.1 *Ciclo de vida de uma Thread:*

Uma thread desde a sua criação até sua finalização, passa por diversos estados e transições. O conjunto de estados e transições de uma thread pode ser visualizado na figura abaixo:



Estes métodos da classe *Object* implementam o conceito de monitores sem utilizar espera ativa. Ao invés disso, notificam as *threads* indicando se estas devem ser suspensas ou se devem voltar a ficar em execução.

- **wait()** - suspende a thread que chamou o método até que outra thread a acorde ou até que o tempo especificado como argumento tenha passado;
- **notify()** - acorda, se existir, alguma thread que esteja esperando um evento neste objeto;
- **notifyAll()** - acorda todas as *threads* que estejam esperando neste objeto.

2.2 *Sincronismo das Threads:*

Assíncronas: São *threads* que trabalham independentemente uma da outra, sem necessidade de comunicação entre elas;

Síncronas: Nesse caso, elas se comunicam de alguma forma ou utilizam dados uma da outra. Geralmente é necessário que a informação que está sendo compartilhada seja consistente, evitando que duas *threads* mexam nesta ao mesmo tempo, gerando um resultado inconsistente. Para isso, existe mecanismos de sincronização, como as funções vistas anteriormente, que garantem que essa sincronização seja consistente.

3. Exemplo

Uso de *threads* em Java:

Neste exemplo duas threads são criadas com igual prioridade. Cada uma imprime os 100 primeiros inteiros seguidos da frase TERMINOU SANTOS! ou TERMINOU SÃO PAULO! Uma thread se chama Santos, e a outra, São Paulo.
Após cada linha impressa, a thread "dorme" um número aleatório de milissegundos - entre 0 e 399 - para evitar que o laço de impressão termine antes de se esgotar a fatia de tempo da thread (delta t). O nome é dado pelo argumento passado ao construtor da thread. O método getName() da classe Thread retorna a string com o nome:

```
public class ThreadSimples extends Thread {
public ThreadSimples(String str) {
  super(str);
}
public void run() {
for (int i = 0; i < 100; i++) {
System.out.println(i + " " + getName());
  try {
    sleep((long)(Math.random() * 400));
  }
  catch (InterruptedException e) {}
  }
  System.out.println("TERMINOU " + getName()+"!");
}
}
public class TesteDuasThreads {
public static void main (String[] args) {
  ThreadSimples santo = new ThreadSimples("SANTOS");
  santo.setPriority(4);
ThreadSimples sampa =new ThreadSimples("SÃO PAULO");
  sampa.setPriority(6);
Thread.currentThread().setPriority(1);
  santo.start();
  sampa.start();
  System.out.println("Main terminado!");
}
}
```

4. Benefícios

4.1 *Velocidade da criação das threads:* As threads são mais fáceis de criar e destruir do que os processos

4.2 *Capacidade de resposta:* a utilização do multithreading pode permitir que um programa continue executando e respondendo ao usuário mesmo se parte dele está bloqueada ou executando uma tarefa demorada. Por exemplo, enquanto um navegador Web carrega uma figura ele permite a interação com o usuário.

4.3 *Compartilhamento de recursos:* todos os recursos alocados e utilizados pelo processo aos quais pertencem são compartilhados pelos threads.

4.4 *Economia:* como os threads compartilham recursos dos processos aos quais pertencem, é mais econômico criar e realizar a troca de contexto de threads.

4.5 *Utilização de arquiteturas multiprocessadas:* é possível executar cada uma das threads criadas para um mesmo processo em paralelo (usando processadores diferentes). Isso aumenta bastante os benefícios do esquema multithreading.

4.6 *Desempenho:* obtido quando há grande quantidade de computação e E/S, os threads permitem que essas atividades se sobreponham e, logo, melhore o desempenho da aplicação.