

Aplicações com Múltiplas Threads em Java

Programação Distribuída / José Marinho

1

Introdução

- Grande parte das aplicações, e em particular as de rede, requerem que sejam efectuadas várias tarefas em simultâneo (e.g., aguardar pela recepção de dados em várias ligações TCP, processar a entrada standard, actualizar componentes gráficos, etc.)
- Em Java, a solução passa pelo recurso a múltiplas *threads* (linhas de execução), também designadas por *processos leves*
- Todas as *threads* de uma aplicação pertencem ao mesmo processo (i.e., partilham o mesmo código e espaço de endereçamento)
- Processo \neq thread

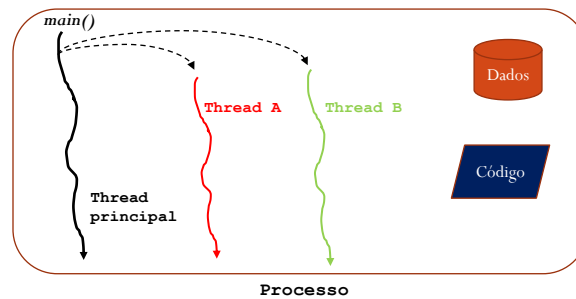
2

Programação Distribuída / José Marinho

2

Introdução

- A gestão de acessos concorrentes aos dados é um aspecto crucial em ambientes *multi-threaded* (e.g., exclusão mútua)



3

Programação Distribuída / José Marinho

3

Criação de *threads* adicionais

- As *threads* encontram-se encapsulada pela classe ***java.lang.Thread***
- O código que deve ser executado por uma *thread* tem como ponto de entrada o método ***run()*** da interface ***java.lang.Runnable***
- Uma *thread* apenas começa a ser executada depois de ter sido criada e invocado o seu método ***start()***
- É possível criar várias *threads* e lançá-las mais tarde

4

Programação Distribuída / José Marinho

4

Criação de *threads* adicionais

- Uma *thread* termina normalmente ao sair do método *run()*
- Dois tipos de *threads*: **utilizador** (*normal*) e **daemon**

```
t.setDaemon(true); t.start();
```

- Uma aplicação em Java termina quando todas as *threads* do tipo utilizador terminam ou quando o método *System.exit()* é invocado, mesmo que ainda existam *threads* do tipo *daemon* activas
- **Primeiro método de criação de *threads***
 - Subclasse da classe *Thread* + *override* do método *run()*
 - Como o Java não suporta herança múltipla, não é possível a *thread* estender outros tipos de classes

5

Programação Distribuída / José Marinho

5

Criação de *threads* adicionais

```
public class ExtendThreadDemo extends java.lang.Thread
{
    int threadNumber;

    public ExtendThreadDemo(int num){threadNumber = num;}

    // RUN METHOD IS EXECUTED WHEN THREAD FIRST STARTED
    public void run()
    {
        System.out.println ("I am thread number " + threadNumber);

        try{
            // SLEEP FOR FIVE THOUSAND MILLISECONDS (5 SECS) ,
            // TO SIMULATE WORK BEING DONE
            Thread.sleep(5000);
        }catch (InterruptedException e) {}

        System.out.println (threadNumber + " is finished!");
    }
}
```

Um método estático que apenas afecta a *thread* que se encontra actualmente escalonada.

6

Programação Distribuída / José Marinho

6

Criação de *threads* adicionais

```
// MAIN METHOD TO CREATE AND START THREADS
public static void main(String args[])
{
    // CREATE FIRST THREAD INSTANCE
    Thread t1 = new ExtendThreadDemo(1);

    // CREATE SECOND THREAD INSTANCE
    Thread t2 = new ExtendThreadDemo(2);

    // MAKE THEM DAEMON THREADS (THEY END BEFORE DISPLAYING THE SECOND MSG)
    t1.setDaemon(true); t2.setDaemon(true);

    // START BOTH THREADS
    t1.start(); t2.start();

    try{
        // SLEEP FOR 1s TO ALLOW THREADS TO DISPLAY FIRST MESSAGE
        Thread.sleep(1000);
    }catch (InterruptedException e) {}
}
```

7

Programação Distribuída / José Marinho

7

Criação de *threads* adicionais

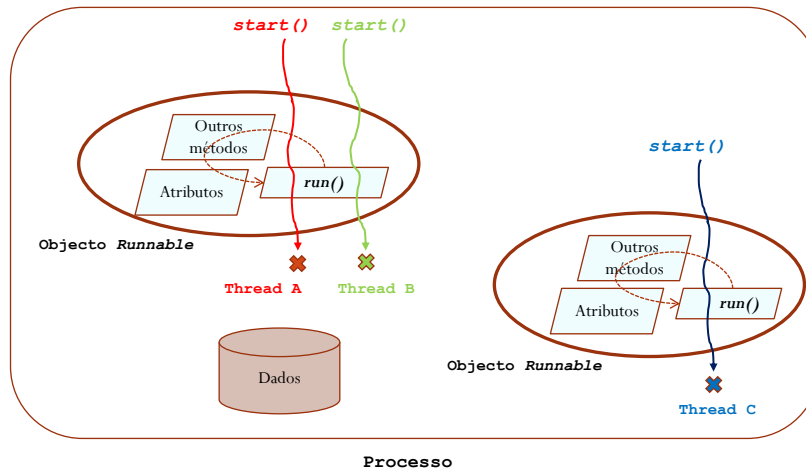
- Segundo método de criação de *threads*
 - Recurso à interface *java.lang.Runnable*
 - Define um único método: *public void run()*
 - Um objecto que implementa a interface *Runnable* é passado como argumento no construtor de uma instância de *Thread*
 - Ao invocar o método *start()* do objecto do tipo *Thread*, este irá chamar o método *run()* do objecto *Runnable*
 - É possível passar o mesmo objecto *Runnable* a várias *threads* que, desta forma, passam a partilhar o mesmo código e actuam sobre os mesmos dados

8

Programação Distribuída / José Marinho

8

Criação de *threads* adicionais



9

Programação Distribuída / José Marinho

9

Criação de *threads* adicionais

```
public class RunnableThreadDemo implements java.lang.Runnable
{
    // RUN METHOD IS EXECUTED WHEN THREAD FIRST STARTED
    public void run()
    {
        System.out.println ("I am an instance of the java.lang.Runnable
                                interface");
    }

    // MAIN METHOD TO CREATE AND START THREADS
    public static void main(String args[])
    {
        System.out.println ("Creating runnable object");

        // CREATE RUNNABLE OBJECT
        Runnable run = new RunnableThreadDemo();

        // CREATE A THREAD, AND PASS THE RUNNABLE OBJECT
        System.out.println ("Creating first thread");
        Thread t1 = new Thread (run , "Thread 1");
    }
}
```

10

Programação Distribuída / José Marinho

10

Criação de *threads* adicionais

```
// CREATE A SECOND THREAD, AND PASS THE RUNNABLE OBJECT
System.out.println ("Creating second thread");
Thread t2 = new Thread (run , "Thread 2");

// START BOTH THREADS
System.out.println ("Starting both threads");
t1.start(); t2.start();
}
}
```

```
Thread t = new Thread (new RunnableThreadDemo()).start();
```

11

Programação Distribuída / José Marinho

11

Interromper *Threads*

```
public class SleepyHead extends Thread
{
    // RUN METHOD IS EXECUTED WHEN THREAD FIRST STARTED
    public void run()
    {
        System.out.println ("I feel sleepy. Wake me in eight hours");

        try{
            // SLEEP FOR EIGHT HOURS
            Thread.sleep( 1000 * 60 * 60 * 8 );
            System.out.println ("That was a nice nap");
        }catch (InterruptedException e){
            System.err.println ("Just five more minutes....");
        }
    }
}
```

12

Programação Distribuída / José Marinho

12

Interromper *Threads*

```
// MAIN METHOD TO CREATE AND START THREADS
public static void main(String args[]) throws java.io.IOException
{
    // CREATE A 'SLEEPY' THREAD
    Thread sleepy = new SleepyHead();

    // START THREAD SLEEPING
    sleepy.start();

    // PROMPT USER AND WAIT FOR INPUT
    System.out.println ("Press enter to interrupt the thread");
    System.in.read();

    // INTERRUPT THE THREAD (DEPRECATED: TO SAFELY TERMINATE A THREAD LET IT DIE
    // OR MAKE IT EXIT ITS METHOD RUN())
    sleepy.interrupt(); //RESUME() METHOD TO RESUME THE THREAD
}
```

13

Programação Distribuída / José Marinho

13

Parar *Threads*

```
public class StopMe extends Thread
{
    // RUN METHOD IS EXECUTED WHEN THREAD FIRST STARTED
    public void run()
    {
        int count = 1;

        System.out.println ("I can count. Watch me go!");

        for (;;) {
            // PRINT COUNT AND INCREMENT IT
            System.out.print (count++ + " ");

            // SLEEP FOR HALF A SECOND
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}
```

14

Programação Distribuída / José Marinho

14

Parar *Threads*

```
// MAIN METHOD TO CREATE AND START THREADS
public static void main(String args[]) throws java.io.IOException
{
    // CREATE AND START COUNTING THREAD
    Thread counter = new StopMe();
    counter.start();

    // PROMPT USER AND WAIT FOR INPUT
    System.out.println ("Press any enter to stop the thread
                                counting");

    System.in.read();

    // FORCES THE THREAD TO STOP EXECUTING (DEPRECATED: TO SAFELY TERMINATE A
    //THREAD LET IT DIE OR MAKE IT EXIT ITS METHOD RUN())
    counter.stop();
}
```

15

Programação Distribuída / José Marinho

15

Operações diversas sobre *threads*

- Uma *thread* activa pode explicitamente libertar o tempo de processador que ainda lhe resta

```
Thread.yield();
```

- Em determinadas situações, é útil poder testar se uma *thread* já terminou (i.e., se saiu do método *run()*)
 - Método *isAlive()*

```
if(t1.isAlive()){ // Devolve um valor do tipo booleano
    //...        // Apenas permite esperas activas!
}
```

16

Programação Distribuída / José Marinho

16

Operações diversas sobre *threads*

- Método *join()*

```
void join()           → espera até que a thread termine  
void join(long m)      → espera, no máximo, m milissegundos até que a  
                        thread termine  
void join(long m, int n) → espera, no máximo, m milissegundos e n  
                        nanossegundos até que a thread termine
```

17

Programação Distribuída / José Marinho

17

Operações diversas sobre *threads*

```
public static void main(String args[]) throws  
    java.lang.InterruptedException  
{  
    Thread dying = new WaitForDeath();  
    dying.start();  
  
    System.out.println ("Waiting for thread death");  
  
    // WAIT TILL DEATH  
    dying.join();  
  
    System.out.println ("Thread has died");  
}
```

18

Programação Distribuída / José Marinho

18

Sincronização de *Threads*

- Quando múltiplas *threads* manipulam os mesmos dados sem qualquer mecanismo de sincronização, podem surgir problemas de integridade/consistência
- A sincronização deve ser efectuada sempre que não exista garantia de acesso atómico aos dados
- Em Java, a sincronização pode ser feita em dois níveis distintos de granularidade
 - Método
 - Bloco de código

19

Programação Distribuída / José Marinho

19

Sincronização de *Threads*

- Sincronização ao nível do método
 - Previne a possibilidade de várias *threads* executarem em momentos coincidentes qualquer método “sincronizado” num determinado objecto (serialização dos acessos ao objecto)
 - É usada a palavra chave *synchronized*
 - Internamente, são usados *locks* no acesso ao objecto
 - O acesso a métodos “não sincronizados” continua a ser feito de forma concorrente e sem qualquer restrição

```
public class SomeClass // which aims at being thread-safe
{
    public synchronized void changeData( ... ) { ..... }
    public synchronized Object getData ( ... ) { ..... }
}
```

20

Programação Distribuída / José Marinho

20

Sincronização de *Threads*

```
public class Counter
{
    private int countValue;

    public Counter(){ countValue = 0; }
    public Counter(int start){ countValue = start; }

    public synchronized void increaseCount()
    {
        int count = countValue;
        // SIMULATE SLOW DATA PROCESSING AND MODIFICATION.
        // REMOVE THE SYNCHRONIZED KEYWORD AND SEE WHAT HAPPENS...
        try{
            Thread.sleep(5);
        }catch (InterruptedException ie) {}

        count = count + 1; countValue = count;
    }

    public synchronized int getCount() { return countValue; }
}
```

21

Programação Distribuída / José Marinho

21

Sincronização de *Threads*

```
public class CountingThread implements Runnable
{
    Counter myCounter;
    int countAmount;

    // CONSTRUCT A COUNTING THREAD TO USE THE SPECIFIED COUNTER
    public CountingThread (Counter counter, int amount)
    {
        myCounter = counter;
        countAmount = amount;
    }

    public void run()
    {
        // INCREASE THE COUNTER THE SPECIFIED NUMBER OF TIMES
        for (int i = 1; i <= countAmount; i++)
        {
            // INCREASE THE COUNTER
            myCounter.increaseCount(); //SYNCHRONIZED METHOD
        }
    }
}
```

22

Programação Distribuída / José Marinho

22

Sincronização de *Threads*

```
public static void main(String args[]) throws Exception
{
    // CREATE A NEW, THREAD-SAFE COUNTER
    Counter c = new Counter();

    // OUR RUNNABLE INSTANCE WILL INCREASE THE COUNTER
    // TEN TIMES, FOR EACH THREAD THAT RUNS IT
    Runnable runner = new CountingThread( c, 10 );

    System.out.println ("Starting counting threads");
    Thread t1 = new Thread(runner);
    Thread t2 = new Thread(runner);
    Thread t3 = new Thread(runner);
    t1.start(); t2.start(); t3.start();

    // WAIT FOR ALL THREE THREADS TO FINISH
    t1.join(); t2.join(); t3.join();

    System.out.println ("Counter value is " + c.getCount() );
}
```

23

Programação Distribuída / José Marinho

23

Sincronização de *Threads*

- Sincronização ao nível do bloco de código
 - A palavra chave *synchronized* continua a ser usada, mas com a finalidade de colocar *locks* em torno de blocos de código
 - A sincronização é referente a um objecto

```
synchronized (Object o)
{
    //...
}
```

- Não é necessário o objecto ter métodos *synchronized*
- Uma das vantagens é permitir o acesso *thread-safe* (serializado) sem necessidade de ter acesso ou alterar o código fonte

24

Programação Distribuída / José Marinho

24

Sincronização de *Threads*

- Exemplo: variação do exemplo anterior
 - O contador é um simples atributo do tipo *int*
 - Cada *thread* incrementa o contador uma vez e acrescenta o valor numa *StringBuffer*
 - A contagem deve aparecer no *buffer* na ordem correcta

```
public class SynchBlock implements Runnable
{
    StringBuffer buffer; // A STRING WHICH CAN BE MODIFIED.
                        // ACCESS TO BUFFER AIMS AT BEING SYNCHRONIZED
                        // WITHOUT MODIFYING STRINGBUFFER.

    int counter;

    public SynchBlock()
    {
        buffer = new StringBuffer();
        counter = 1;
    }
}
```

25

Programação Distribuída / José Marinho

25

Sincronização de *Threads*

```
public void run()
{
    synchronized (buffer)
    {
        System.out.print ("Starting synchronized block ");
        int tempVariable = counter++;

        // CREATE MESSAGE TO ADD TO BUFFER, INCLUDING LINE FEED
        String message = "Count value is : " + tempVariable +
            System.lineSeparator();

        try {
            Thread.sleep(100); // SIMULATE TIME NEEDED FOR SOME TASK
        } catch (InterruptedException ie) {}

        buffer.append (message);
        System.out.println ("... ending synchronized block");
    }
}
```

26

Programação Distribuída / José Marinho

26

Sincronização de *Threads*

```
public static void main(String args[]) throws Exception
{
    // CREATE A NEW RUNNABLE INSTANCE
    SynchBlock block = new SynchBlock();

    Thread t1 = new Thread (block);
    Thread t2 = new Thread (block);
    Thread t3 = new Thread (block);
    Thread t4 = new Thread (block);
    t1.start(); t2.start(); t3.start(); t4.start();

    // WAIT FOR ALL THREE THREADS TO FINISH
    t1.join(); t2.join(); t3.join(); t4.join();

    // COUNT SHOULD APPEAR IN THE CORRECT ORDER
    System.out.println (block.buffer);
}
```

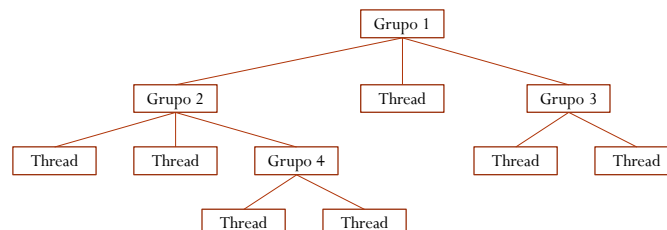
27

Programação Distribuída / José Marinho

27

Grupos de *Threads*

- É possível criar grupos de *threads*
- Permite que uma operação seja aplicada a várias *threads*
- Classe: ***ThreadGroup***
- Não é possível eliminar *threads* de um grupo, apenas adicionar
- Grupos de *threads* podem incluir subgrupos de *threads*



28

Programação Distribuída / José Marinho

28

Grupos de *Threads*

- Criar um grupo em que o grupo de base é aquele a que pertence a *thread* activa actual

```
public ThreadGroup(String name) throws java.lang.SecurityException
```

- Criar um grupo fornecendo o grupo de base a que deve pertence

```
public ThreadGroup(ThreadGroup parentGroup, String name) throws java.lang.SecurityException
```

```
ThreadGroup parent = new ThreadGroup("parent");  
ThreadGroup subgroup = new ThreadGroup(parent, "subgroup");
```

29

Programação Distribuída / José Marinho

29

Grupos de *Threads*

- Uma *thread* apenas pode ser adicionada a um grupo no momento da sua criação

- **Thread** (**ThreadGroup** group, Runnable runnable)
- **Thread** (**ThreadGroup** group, String name)
- **Thread** (**ThreadGroup** group, Runnable runnable, String name)

Método	Objectivo
int activeCount()	Devolve o número de <i>threads</i> activas no grupo e subgrupos.
int activeGroupCount()	Devolve o número de subgrupos com <i>threads</i> activas.
void destroy()	Destrói o grupo e subgrupos desde que não haja qualquer <i>thread</i> activa. Caso contrário, gera <i>java.lang.IllegalThreadStateException</i> .
boolean isDestroyed()	Devolve verdadeiro se o grupo foi destruído.
void list()	Escreve informação sobre o grupo em System.out
int enumerate(Thread[] threadList)	Copia para o <i>array</i> as referências das <i>threads</i> activas do grupo (i.e., até ao tamanho limite). Devolve o número de referências copiadas.

30

Programação Distribuída / José Marinho

30

Grupos de *Threads*

Método	Objectivo
int enumerate (Thread[] threadList, boolean subgroupFlag)	Semelhante ao anterior quando a <i>flag</i> é verdadeira.
int enumerate (ThreadGroup[] groupList)	Copia para o <i>array</i> as referências dos grupos e subgrupos activos. Devolve o número de referências copiadas.
int enumerate (ThreadGroup[] groupList, boolean subgroupFlag)	Semelhante ao anterior quando a <i>flag</i> é verdadeira.
String getName ()	Devolve o nome associado ao grupo.
ThreadGroup getParent ()	Devolve o grupo de base.
void interrupt ()	Invoca o método <i>Thread.interrupt()</i> de todas as <i>threads</i> do grupo e subgrupos.
boolean parentOf (ThreadGroup otherGroup)	Verifica se o grupo indicado é um subgrupo (<i>filho</i>).
int getMaxPriority ()	Devolve o nível máximo de prioridade permitido para a <i>threads</i> do grupo.
void setMaxPriority (int priority)	Define o nível máximo de prioridade para o grupo.

31

Programação Distribuída / José Marinho

31

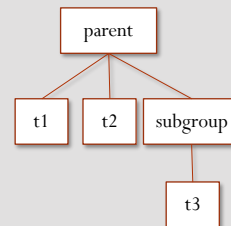
Grupos de *Threads*

```
public class GroupDemo implements Runnable
{
    public static void main(String args[]) throws Exception
    {
        ThreadGroup parent = new ThreadGroup("parent");
        ThreadGroup subgroup = new ThreadGroup(parent, "subgroup");

        Thread t1 = new Thread ( parent, new GroupDemo() );
        Thread t2 = new Thread ( parent, new GroupDemo() );
        Thread t3 = new Thread ( subgroup, new GroupDemo() );
        t1.start(); t2.start(); t3.start();
        parent.list();

        // WAIT FOR USER, THEN TERMINATE
        System.out.println ("Press enter to continue");
        System.in.read(); System.exit(0);
    }

    public void run()
    {
        for(;;){ Thread.yield(); }    // DO NOTHING
    }
}
```



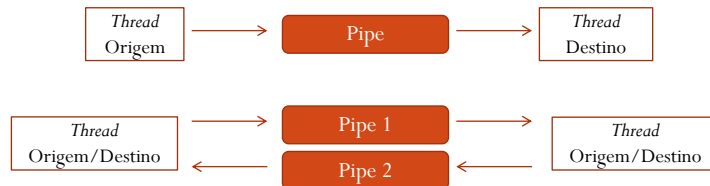
32

Programação Distribuída / José Marinho

32

Comunicação entre *Threads*

- Através de atributos públicos e métodos em objectos
- Através de *pipes* de comunicação
 - Troca directa de dados entre *threads*
 - Comunicação unidireccional
 - Semelhante à utilização de *streams* de entrada e saída



33

Programação Distribuída / José Marinho

33

Comunicação entre *Threads*

```
public class PipeDemo extends Thread
{
    PipedOutputStream output;

    public PipeDemo(PipedOutputStream out) { output = out; }

    public void run()
    {
        try {
            // CREATE A PRINTSTREAM FOR CONVENIENT WRITING
            PrintStream p = new PrintStream( output );

            p.println("Hello from another thread, via pipes!");
            p.close();
        } catch (Exception e) {}
    }
}
```

34

Programação Distribuída / José Marinho

34

Comunicação entre *Threads*

```
public static void main (String args[])
{
    try{
        // CREATE A PIPE FOR WRITING
        PipedOutputStream pout = new PipedOutputStream();
        // CREATE A PIPE FOR READING, AND CONNECT IT TO OUTPUT PIPE
        PipedInputStream pin = new PipedInputStream(pout);

        // CREATE A NEW PIPE DEMO THREAD, TO WRITE TO OUR PIPE
        PipeDemo pipedemo = new PipeDemo(pout);
        pipedemo.start();

        // READ THREAD DATA A BYTE AT A TIME UNTIL THE INPUT PIPE IS CLOSED
        int input = pin.read();
        while (input != -1){ //EOF
            System.out.print ( (char) input);
            input = pin.read();
        }
    }catch(Exception e){ System.err.println ("Pipe error " + e); }
}
```

35

Programação Distribuída / José Marinho

35

Comunicação entre *Threads*

- Notificações
 - Existem situações em que uma *thread* deve esperar, antes de poder prosseguir, que outra conclua uma determinada tarefa
 - Os métodos *join()* permitem esperar pela conclusão de uma *thread*
 - Outra solução passa pelo recurso a **notificações**
 - Métodos: *Object.wait()*, *Object.notify()* e *Object.notifyAll()* (nota: em Java, todos os objectos descendem da super-classe *Object*)
 - Antes de se poder invocar os métodos, é necessário adquirir o monitor do respectivo objecto (i.e., através de blocos *synchronized*)

36

Programação Distribuída / José Marinho

36

Comunicação entre *Threads*

```
public class WaitNotify extends Thread
{
    public static void main(String args[]) throws Exception
    {
        // START THE ADDITIONAL NEW THREAD THE MAIN THREAD WILL BE AINTING FOR.
        Thread notificationThread = new WaitNotify();
        notificationThread.start();

        // WAIT FOR THE NOTIFICATION THREAD TO TRIGGER EVENT
        synchronized (notificationThread){
            notificationThread.wait();
        }

        // NOTIFY USER THAT THE WAIT() METHOD HAS RETURNED
        System.out.println ("The wait is over");
    }
}
```

37

Programação Distribuída / José Marinho

37

Comunicação entre *Threads*

```
public void run()
{
    //THIS IS THE ADDITONAL THREAD WHICH WAS CREATED+STARTED BY THE MAIN THREAD
    System.out.println ("Hit enter to stop waiting thread");
    try{
        System.in.read();
    }catch (java.io.IOException e){}

    // NOTIFY ANY THREADS WAITING ON THIS THREAD
    synchronized (this){
        this.notifyAll();
    }
}
```

38

Programação Distribuída / José Marinho

38

Prioridades de *Threads*

- Em ambientes que não sejam multi-processador, existe um escalonamento alternado das *threads* que cria a ilusão de concorrência (e.g., abordagem *Round-robin*)
- A ordem de execução das *threads* não é previsível
- No entanto, é possível atribuir níveis distintos de prioridade (i.e., *Round-robin* com múltiplas filas de prioridade)
- As prioridades variam de 1 (a mais baixa) a 10 (a mais alta)

```
Thread.MAX_PRIORITY    (10)
Thread.NORM_PRIORITY   (5)
Thread.MIN_PRIORITY    (1)
```

39

Programação Distribuída / José Marinho

39

Prioridades de *Threads*

```
Thread t = new Thread (runnable);
t.setPriority ( Thread.MIN_PRIORITY );
t.start();
```

```
Thread t = Thread.currentThread();
System.out.println ("Priority : " + t.getPriority());
```

```
ThreadGroup group = new ThreadGroup ( "mygroup" );
group.setMaximumPriority(8);
```

40

Programação Distribuída / José Marinho

40

Servidores TCP concorrentes

```
...
while(true){

    toClientSocket = socket.accept();

    t = new Thread(new ServerChild(toClientSocket, ++threadNum));

    t.start();

    System.out.println( "Lancada thread n.º " + threadNum +
        " para atender o cliente " +
        toClientSocket.getInetAddress().getHostAddress()
        + ":" + toClientSocket.getPort() );

}
...
```

41

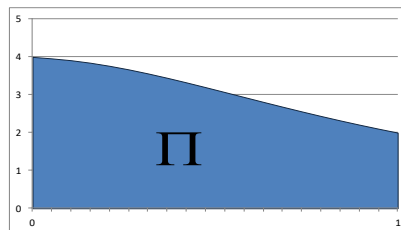
Programação Distribuída / José Marinho

41

Computação Paralela Baseada em Múltiplas Threads

- Em arquiteturas multiprocessador, em que *threads* do mesmo processo possam ser escalonadas em processadores distintos, o recurso a múltiplas threads permite a realização de computação paralela
- Exemplo: o cálculo do valor do Π por integração numérica

$$\Pi = \int_0^1 \frac{4}{1+x^2} dx$$



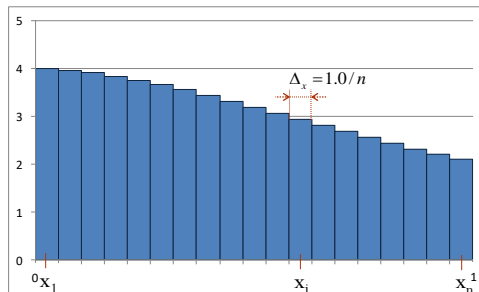
42

Programação Distribuída / José Marinho

42

Computação Paralela Baseada em Múltiplas Threads

- O valor aproximado de um integral pode ser obtido através do somatório das áreas de n intervalos em que se encontra dividida a área total
- Quanto maior é o número de intervalos n , maior é a precisão do resultado obtido



$$\Pi = \sum_{i=1}^n \left(\frac{4}{1+x_i^2} \Delta_x \right) = \Delta_x \sum_{i=1}^n \frac{4}{1+x_i^2}$$

43

Programação Distribuída / José Marinho

43

Computação Paralela Baseada em Múltiplas Threads

```
public class ParallelPi extends Thread
{
    private int myId, nThreads, nIntervals, myIntervals;
    private double dX;
    private double myResult;

    public ParallelPi(int myId, int nThreads, int nIntervals)
    {
        this.myId = myId;
        this.nThreads = nThreads;
        this.nIntervals = nIntervals;
        dX = 1.0/(double)nIntervals;
        myResult = 0;
        myIntervals = 0;
    }

    public double getMyResult() { return myResult; }

    public int getMyIntervals() { return myIntervals; }
```

44

Programação Distribuída / José Marinho

44

Computação Paralela Baseada em Múltiplas Threads

```
@Override
public void run()
{
    double i, xi;

    myResult = 0;

    if(nIntervals < 1 || nThreads < 1 || myId <1 || myId > nThreads){
        return;
    }

    for (i = myId-1 ; i < nIntervals; i += nThreads) {
        xi = dX*(i + 0.5);
        myResult += (4.0/(1.0 + xi*xi));
        myIntervals++;
    }

    myResult *= dX;
}
```

45

Programação Distribuída / José Marinho

45

Computação Paralela Baseada em Múltiplas Threads

```
public static void main(String[] args) throws InterruptedException
{
    ParallelPi []t; //Working threads
    int i, nThreads;
    long nIntervals;
    double pi = 0.0;

    if(args.length != 2){
        System.out.println("Sintaxe: java ParallelPi <número de\" +
            \" intervalos> <número de threads>");
        return;
    }

    nIntervals = Integer.parseInt(args[0]);
    nThreads = Integer.parseInt(args[1]);

    t = new ParallelPi[nThreads];

    for(i=0; i<nThreads; i++){
        t[i] = new ParallelPi(i+1, nThreads, nIntervals);
        t[i].start();
    }
}
```

46

Programação Distribuída / José Marinho

46

Computação Paralela Baseada em Múltiplas Threads

```
pi = 0;
for(i=0; i<nThreads; i++){
    t[i].join();
    pi += t[i].getMyResult();
}

System.out.println("Valor aproximado de pi: " + pi);

} //main
} //class ParallelPi
```

47

Programação Distribuída / José Marinho

47

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

- O protocolo HTTP/1.0 (RFC 1945) está na origem da Web
- Criado com o objectivo de permitir a partilha de documentos através da Internet
- Inclui comandos (formato *ascii*) destinados a obter recursos da rede e a submeter dados aos servidores
- Foi melhorado através de versões sucessivas
- Foi identificada a necessidade/vantagem de existência de hiperligações
- O exemplo apresentado apenas suporta o método GET
- Cada pedido novo é atendido por uma *thread* dedicada

48

Programação Distribuída / José Marinho

48

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
import java.io.*;
import java.net.*;
import java.util.*;

public class WebServerDemo
{
    protected String docRoot; // DIRECTORY OF HTML PAGES AND OTHER FILES
    protected int port; // PORT NUMBER OF WEB SERVER
    protected ServerSocket ss; // SOCKET FOR THE WEB SERVER

    // HANDLER FOR A HTTP REQUEST (INNER CLASS)
    class Handler extends Thread
    {
        protected Socket socket;
        protected PrintWriter pw;
        protected BufferedOutputStream bos;
        protected BufferedReader br;
        protected File docRoot;
    }
}
```

49

Programação Distribuída / José Marinho

49

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
public Handler(Socket _socket, String _docRoot) throws Exception
{
    socket=_socket;
    // GET THE ABSOLUTE DIRECTORY OF THE FILEPATH
    docRoot=new File(_docRoot).getCanonicalFile();
}

public void run()
{
    try {
        br = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        bos = new BufferedOutputStream(socket.getOutputStream());
        pw = new PrintWriter(socket.getOutputStream(), true);

        // READ HTTP REQUEST FROM USER (HOPEFULLY GET /FILE..... )
        String line = br.readLine();

        // SHUTDOWN ANY FURTHER INPUT - ONLY ONE REQUEST PER TCP CONNECTION
        socket.shutdownInput();
    }
}
```

50

Programação Distribuída / José Marinho

50

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
if(line == null){ //EOF
    socket.close();
    return;
}

// If NOT A GET REQUEST, THE SERVER WILL NOT SUPPORT IT
if(!line.toUpperCase().startsWith("GET")){
    pw.println("HTTP/1.0 501 Not Implemented");
    pw.println();
    socket.close();
    return;
}

// ELIMINATE ANY TRAILING ? DATA, SUCH AS FOR A CGI GET REQUEST
StringTokenizer tokens = new StringTokenizer(line, " ?");
tokens.nextToken(); //SKIPS FIRST TOKEN (GET)
String req = tokens.nextToken();
```

51

Programação Distribuída / José Marinho

51

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
// CONSTRUCTS THE PATH TO THE REQUESTED RESOURCE
String name;
if(req.startsWith("/") || req.startsWith("\\\\")){
    name = this.docRoot+req;
}else{
    name = this.docRoot+File.separator+req;
}

// GET ABSOLUTE FILE PATH
File file = new File(name).getCanonicalFile();

// CHECK TO SEE IF REQUEST DOESN'T START WITH OUR DOCUMENT ROOT
if(!file.getAbsolutePath().startsWith(
    this.docRoot.getAbsolutePath())){
    pw.println("HTTP/1.0 403 Forbidden");
    pw.println();
}
// ... OR CHECK TO SEE IF IT'S MISSING
} else if(!file.exists()){
    pw.println("HTTP/1.0 404 File Not Found");
    pw.println();
}
// ... OR CHECK TO SEE IF IT CAN'T BE READ FOR SECURITY REASONS
```

52

Programação Distribuída / José Marinho

52

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
// ... OR CHECK TO SEE IF IT CAN'T BE READ FOR SECURITY REASONS
} else if(!file.canRead()){
    pw.println("HTTP/1.0 403 Forbidden");
    pw.println();
// ... OR CHECK TO SEE IF ITS ACTUALLY A DIRECTORY, AND NOT A FILE
} else if(file.isDirectory()){
    sendDir(pw,file,req);
// ... OR CHECK TO SEE IF IT'S REALLY A FILE
}else{
    sendFile(bos, pw, file.getAbsolutePath());
}

} catch(Exception e) { e.printStackTrace(); }

try {
    socket.close();
} catch(Exception e) { e.printStackTrace(); }

} //RUN()
```

53

Programação Distribuída / José Marinho

53

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
protected void sendFile(BufferedOutputStream bos,
                        PrintWriter pw, String filename)
{
    try {
        BufferedInputStream bis = new java.io.BufferedInputStream(
            new FileInputStream(filename));
        byte[] data = new byte[10*1024];
        int read;

        pw.println("HTTP/1.0 200 Okay");
        pw.println();

        read = bis.read(data);
        while(read != -1){
            bos.write(data,0,read); read = bis.read(data);
        }
        bos.flush(); bis.close();
    } catch(Exception e){
        pw.flush(); bos.flush();
    }
}
```

54

Programação Distribuída / José Marinho

54

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
protected void sendDir(PrintWriter pw, File dir, String req)
{
    try {
        pw.println("HTTP/1.0 200 Okay");
        pw.println(); pw.flush();

        pw.print("<html><head><title>Dir
```

Directory of XPTO/

[file1](#)
[Dir -> subdir1](#)
[file2](#)

HTTP/1.0 200 Okay

```
<html><head><title>Directory of XPTO </title></head><body><h1>Directory of
XPTO</h1><table border="0">
<tr><td><a href="XPTO/file1">file1</a></td></tr>
<tr><td><a href="XPTO/subdir1/">Dir ->subdir1 </a></td></tr>
<tr><td><a href="XPTO/file2">file2 </a></td></tr>
</table></body></html>
```

```
if(contents[i].isDirectory()){
    pw.print("/");
}
```

The `<tr>` tag defines a row in an HTML table
The `<td>` tag defines a standard cell in an HTML table.

55

Programação Distribuída / José Marinho

55

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
protected void sendDir(PrintWriter pw, File dir, String req)
{
    try {
        pw.println("HTTP/1.0 200 Okay");
        pw.println(); pw.flush();

        pw.print("<html><head><title>Directory of " + req);
        pw.print("</title></head><body><h1>Directory of " + req);
        pw.println("</h1><table border=\"0\">");

        File[] contents=dir.listFiles();

        for(int i=0;i<contents.length;i++){
            pw.print("<tr><td><a href=\"");
            pw.print(req);
            pw.print(contents[i].getName());
            if(contents[i].isDirectory()){
                pw.print("/");
            }
        }
    }
}
```

56

Programação Distribuída / José Marinho

56

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
        pw.print("\n>");

        if(contents[i].isDirectory()){
            pw.print("Dir -> ");
        }

        pw.print(contents[i].getName());
        pw.print("</a></td></tr>");
    } // FOR

    pw.println("</table></body></html>");

    }catch(Exception e){
        pw.flush();
    }
}
} // INNER CLASS HANDLES (THREAD)
```

57

Programação Distribuída / José Marinho

57

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
// CHECK THAT A FILEPATH HAS BEEN SPECIFIED AND A PORT NUMBER
protected void parseParams(String[] args) throws Exception
{
    if(args.length != 2){
        System.err.println ("Syntax: java "+this.getClass().getName()+
                               " docRoot port");
        System.exit(0);
    }

    this.docRoot = args[0];
    this.port = Integer.parseInt(args[1]);
}

public void ProcessRequests(String[] args) throws Exception {
    System.out.println ("Checking for parameters");
    parseParams(args);

    System.out.print ("Starting web server..... ");
}
```

58

Programação Distribuída / José Marinho

58

Exemplo de um Servidor HTTP/1.0 Concorrente Elementar

```
this.ss = new ServerSocket(this.port);

System.out.println ("OK");

for (;;) {
    Socket accept = ss.accept();

    // START A NEW HANDLER INSTANCE TO PROCESS THE REQUEST
    new Handler(accept, docRoot).start();
} // WebServerDemo

public static void main(String[] args) throws Exception
{
    WebServerDemo webServerDemo = new WebServerDemo();
    webServerDemo.ProcessRequests(args);
}
```

59

Programação Distribuída / José Marinho

59

Bibliografia

- REILLY, David; REILLY, Michael - *Java Network Programming & Distributed Computing* - Addison-Wesley
- <http://download.oracle.com/javase/tutorial/essential/>

60

Programação Distribuída / José Marinho

60