



Programação II

Programação paralela - *Threads*

<http://dl.dropbox.com/u/3025380/prog2/aula14.pdf>

flavio.cec@unisul.br

Paralelismos

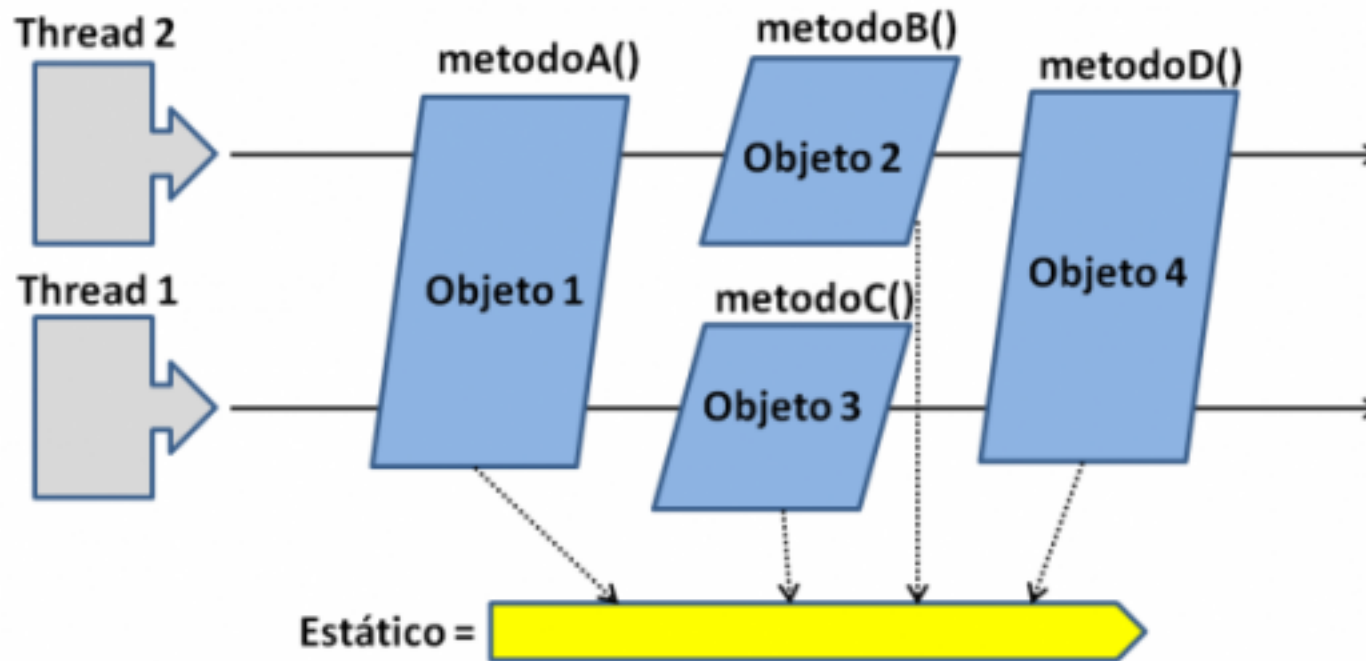
- O paralelismo acontece quando vários processos são executados como se fossem em paralelo.
- Um **processo** se caracteriza pelo ambiente onde o programa (sequência de instruções, composto por desvios, repetições (iterações) e chamadas de procedimentos e/ou funções) é executado.
- O sistema operacional tem a função de gerenciar esses processos.

Paralelismo

- Dentro de um processo podem existir vários sub-processos de execução paralela, que são chamados de processos leves ou ***threads***.
- O termo ***thread*** significa segmento ou linhas de execução.
- As threads também não são executadas em paralelo, ou seja, existe um **escalonador** de threads que decide quais segmento ou thread deve ser executado em uma unidade de tempo dentro de um determinado processo.

Paralelismo

- As **threads** de um mesmo **processo** compartilham o mesmo **espaço de endereçamento**.



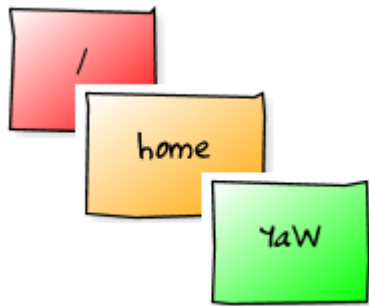
Por que utilizar threads?

Por que utilizar threads?

- Responsividade em Interfaces Gráficas: imagine se o seu navegador web parasse de carregar uma página só porque você clicou no menu “arquivo”;
- Sistemas multiprocessados: o uso de threads permite que o SO consiga dividir as tarefas entre todos os processadores disponíveis aumentando, assim, a eficiência do processo;

Por que utilizar threads?

- Processamento assíncrono ou em segundo plano: com threads um servidor de e-mail pode, por exemplo, atender a mais de um usuário simultaneamente;
- **Paralelizar aplicações, para melhorar o aproveitamento do tempo de processamento.**



	Thread 0	Thread 1
Core 0	CPU 0 General Use	CPU 4 General Use
Core 1	CPU 1 General Use	CPU 5 General Use
Core 2	CPU 2 writer thread	CPU 6 reader thread
Core 3	CPU 3 engine thread	CPU 7 engine thread*

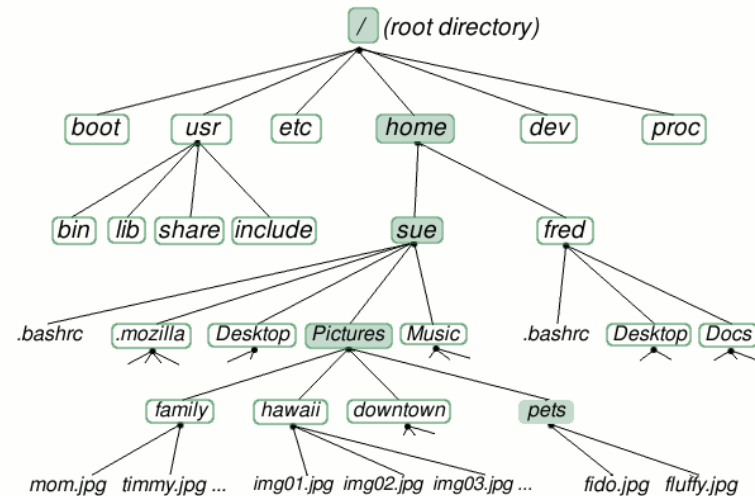
Por que utilizar threads?

Estudo de caso:

Salvar a estrutura de pastas do “Meus documentos”

Descrição

- Foi solicitado o desenvolvimento de uma aplicação em Java que tenha como função:
 - Ler todas as pastas que existam internas a pasta “meus documentos” do usuário da máquina;
 - Salve todas as pastas e documentos em um documento TXT.



Proposta de solução

- Desenvolver uma classe que leia a partir da pasta “meus documentos” todas as pastas e arquivos internas;
- Utilizar recursividade para varrer as pastas e subpastas;
- Adicionar o caminho das pastas em um ArrayList.
- Ao final salvar todos os registros do ArrayList em um arquivo TXT.

```

public List<String> recuperarListaArquivo(String caminhaRaiz) {
    final List<String> listaArquivos = new LinkedList<String>();

    listaArquivos.add(caminhaRaiz);

    this.lerRaizPasta(listaArquivos, caminhaRaiz);

    return listaArquivos;
}

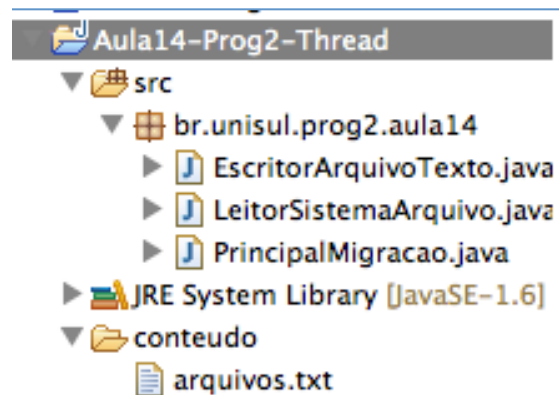
private void lerRaizPasta(List<String> listaArquivos, String caminhaRaiz) {
    File raiz = new File(caminhaRaiz);

    if(raiz != null && raiz.exists()) {
        File[] arquivos = raiz.listFiles();

        if(arquivos != null) {
            for(File arquivo : arquivos) {
                if(arquivo != null && arquivo.isHidden() == false) {
                    String caminho = arquivo.getAbsolutePath();
                    listaArquivos.add(caminho);

                    if(arquivo.isDirectory()) {
                        lerRaizPasta(listaArquivos, caminho);
                    }
                }
            }
        }
    }
}
}
}
}

```



```

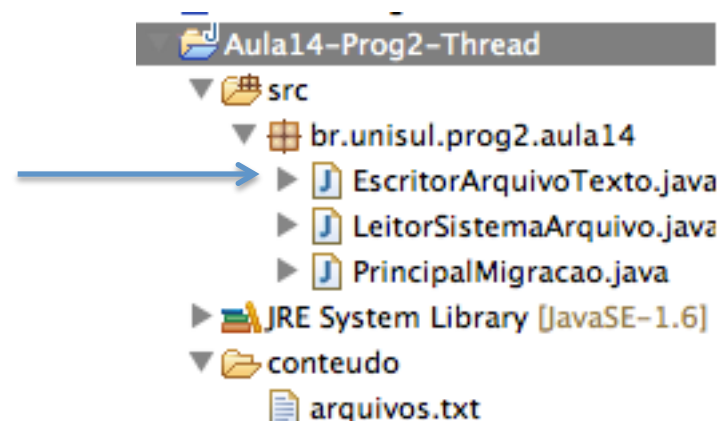
public void escreverConteudoArquivo(List<String> linhas) {
    OutputStream escritorByte = null;
    OutputStreamWriter escritorCaracter = null;
    BufferedWriter escritorPalavras = null;

    try {
        escritorByte = new FileOutputStream("conteudo/arquivos.txt", true);
        escritorCaracter = new OutputStreamWriter(escritorByte);
        escritorPalavras = new BufferedWriter(escritorCaracter);

        for(String linha : linhas) {
            escritorPalavras.write(linha);
            escritorPalavras.newLine();
        }
        escritorPalavras.flush();

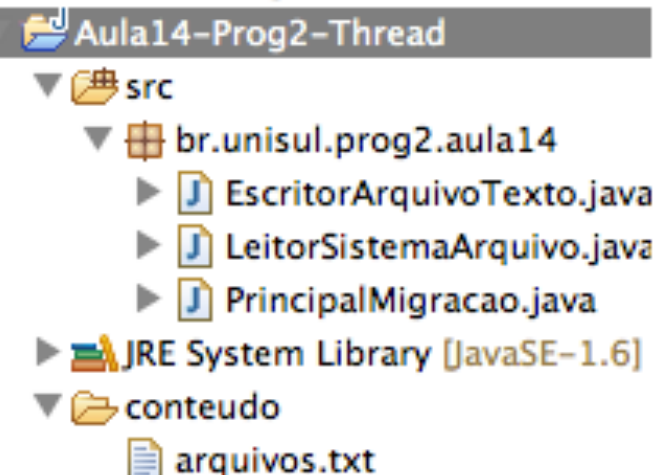
    } catch (FileNotFoundException e) {
        System.err.println(e);
    } catch (IOException e) {
        System.err.println(e);
    } finally {
        try {
            if(escritorPalavras != null) {
                escritorPalavras.close();
            }
            if(escritorCaracter != null) {
                escritorCaracter.close();
            }
            if(escritorByte != null) {
                escritorByte.close();
            }
        } catch (Exception e){}
    }
}

```



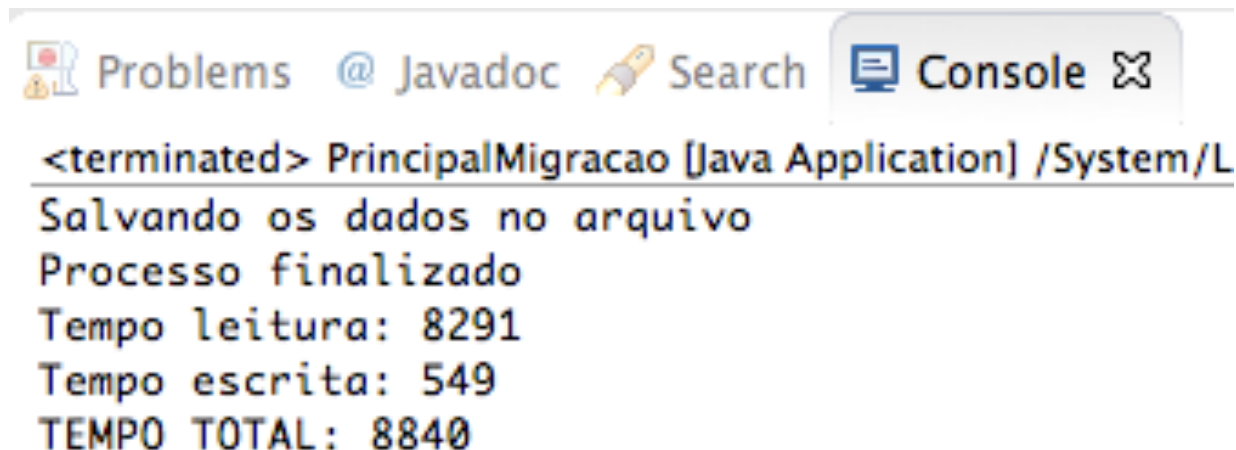
Classe principal

```
public class PrincipalMigracao {  
  
    public static void main(String[] args) {  
        LeitorSistemaArquivo leitor = new LeitorSistemaArquivo();  
        EscritorArquivoTexto escritor = new EscritorArquivoTexto();  
  
        List<String> arquivos = leitor.recuperarListaArquivo("/Users/flavioceci/Documents");  
  
        if(arquivos != null) {  
            System.out.println("Salvando os dados no arquivo");  
            escritor.escreverConteudoArquivo(arquivos);  
            System.out.println("Processo finalizado");  
        }  
    }  
}
```



Características da proposta de solução

- A sua execução é feita de maneira serial;
 - O tempo total de execução é igual ao tempo total de leitura dos arquivos + o tempo total de escrita no novo arquivo TXT;

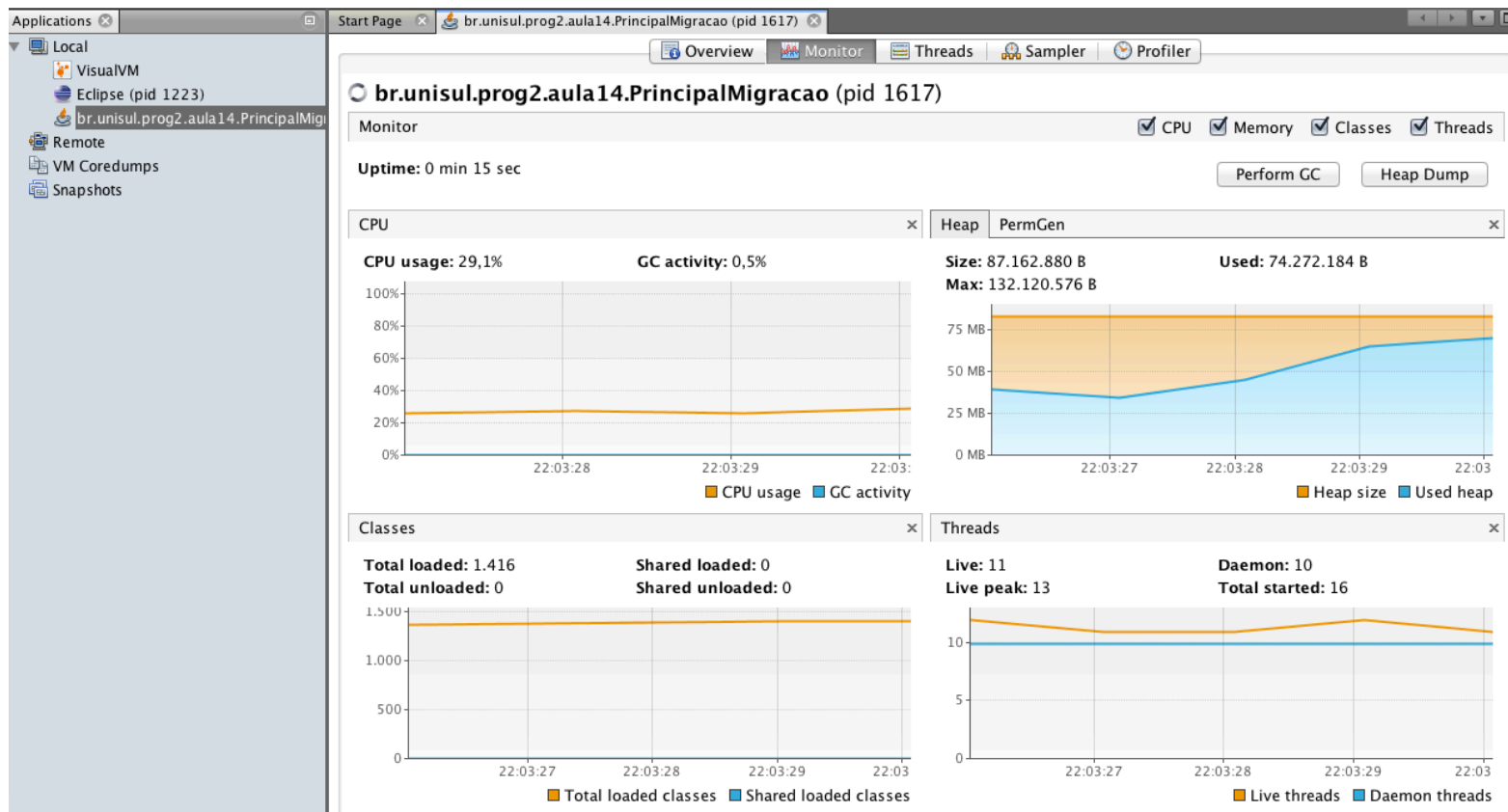


The screenshot shows an IDE interface with a toolbar at the top containing icons for Problems, Javadoc, Search, and Console. The Console window is active and displays the following output:

```
<terminated> PrincipalMigracao [Java Application] /System/L  
Salvando os dados no arquivo  
Processo finalizado  
Tempo leitura: 8291  
Tempo escrita: 549  
TEMPO TOTAL: 8840
```

Características da proposta de solução

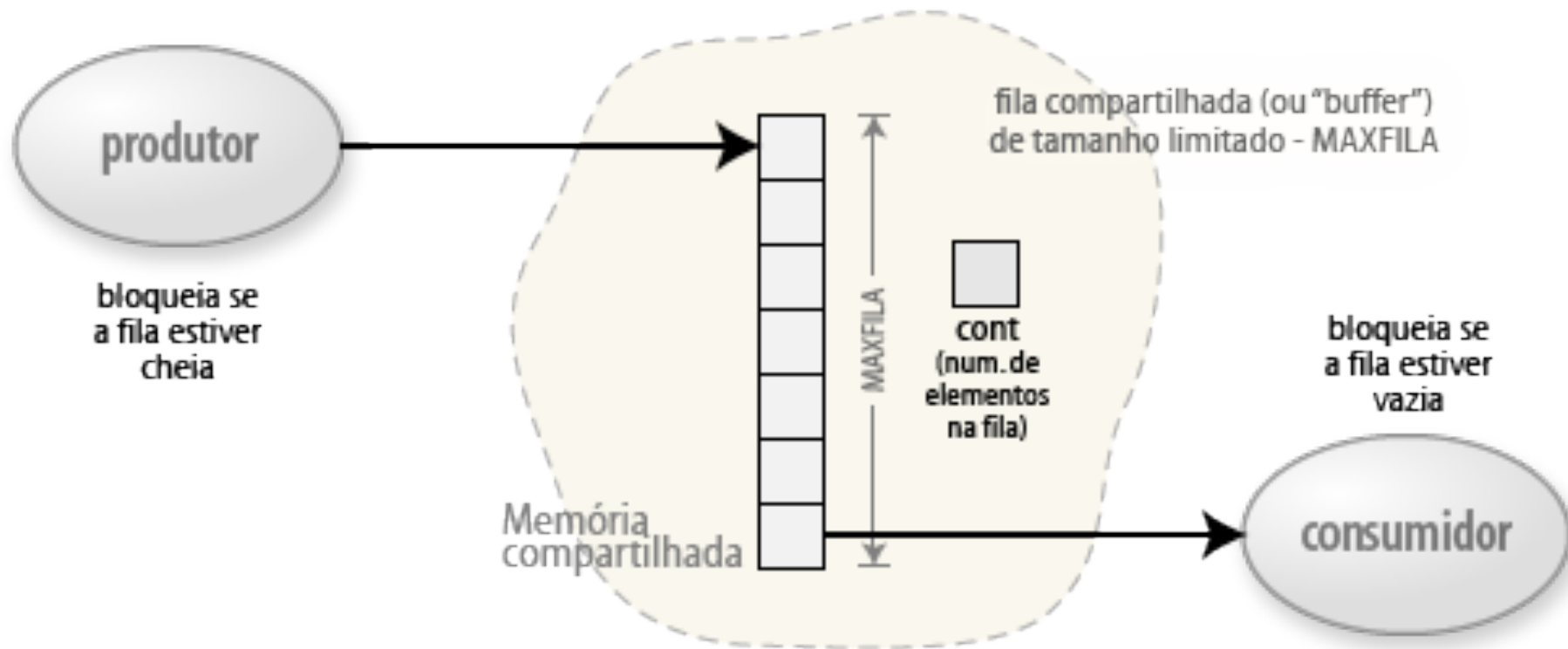
- Pode não estar utilizando o recurso físico da máquina, como deveria;



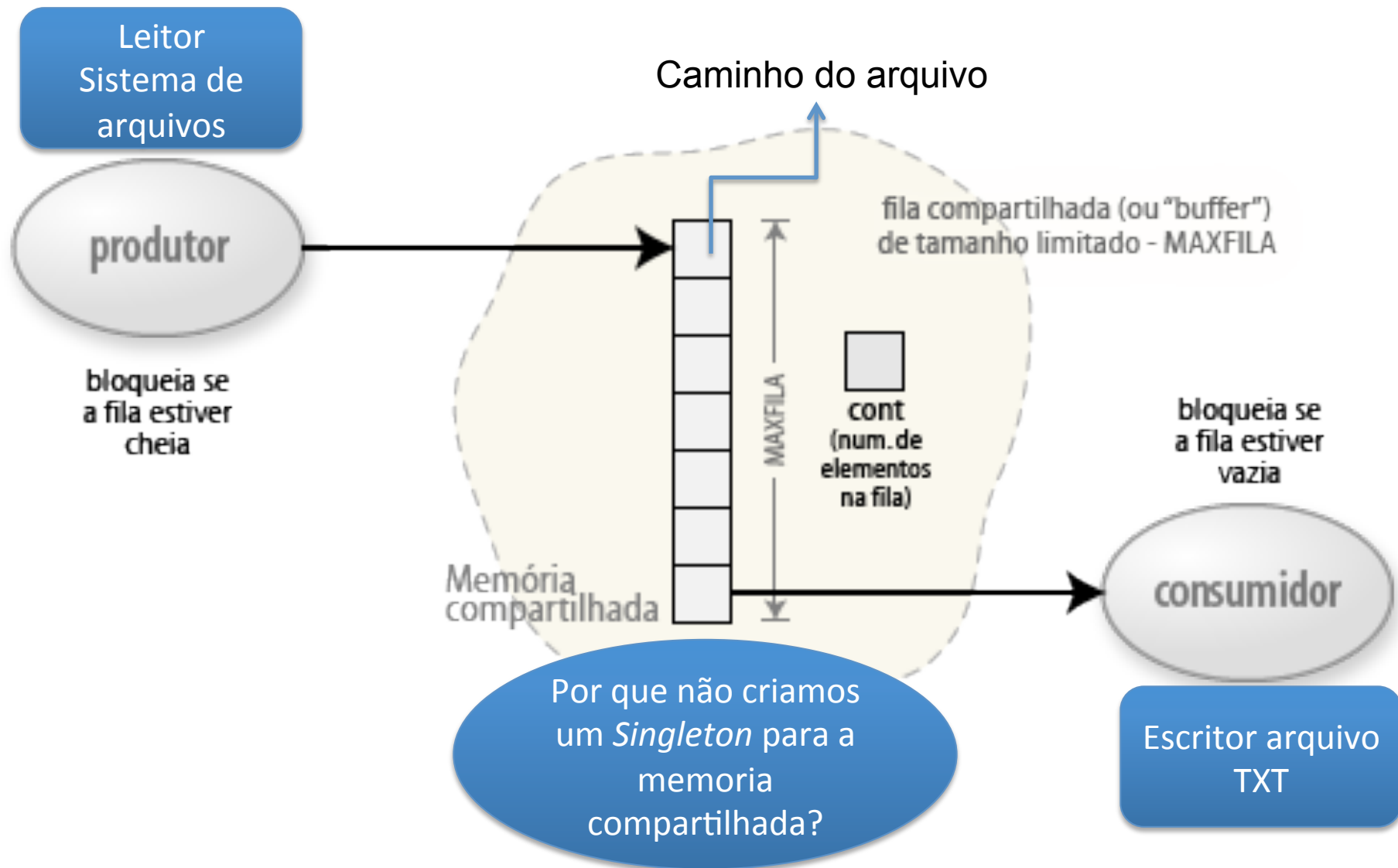


Como podemos melhorar essa
execução?

Estratégia: Produtor X Consumidor



Vamos rever o estudo de caso



Relembrando o que é um *singleton*

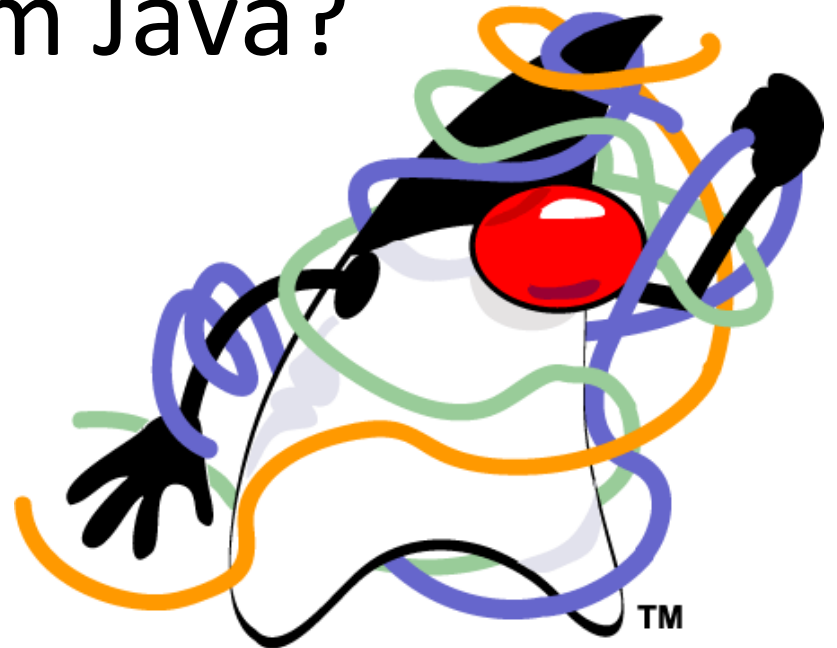
- Padrão de projeto criado para disponibilizar apenas uma instância de um objeto.
 - No caso de termos uma estratégia produtor e consumido com acesso a uma “memória compartilhada” o uso do *singleton* é muito bem vindo!!!

Singleton
<u>– singleton : Singleton</u>
<u>– Singleton()</u>
<u>+ getInstance() : Singleton</u>

Vamos construir o *singleton* da fila

```
public class ServicoMemoriaCompartilhada {  
  
    private List<String> memoriaCompartilhada = new ArrayList<String>();  
  
    private static ServicoMemoriaCompartilhada instancia = new ServicoMemoriaCompartilhada();  
  
    private ServicoMemoriaCompartilhada(){  
    };  
  
    public static ServicoMemoriaCompartilhada getInstancia() {  
        return instancia;  
    }  
  
    public void adiciona(String elemento) {  
        this.memoriaCompartilhada.add(elemento);  
    }  
  
    public String recupera() {  
        return this.memoriaCompartilhada.remove(0);  
    }  
  
    public boolean temElemento() {  
        return !this.memoriaCompartilhada.isEmpty();  
    }  
}
```

Como podemos paralelizar
operações em Java?



Paralelismo com Java

- ***Thread***: A classe deve estender (herdar) a classe Thread do Java.
 - Dessa forma deve-se ler que o novo objeto “é uma” thread.
 - Deve implementar o método run();
 - É iniciando pelo método start();
- ***Runnable***: É uma interface que solicita a implementação do método run();
- Deve ser encapsulado dentro de uma Thread.

Paralelismo com Java

- Mas quando utilizar Thread ou Runnable?
 - Uma **Thread** é uma extensão de uma classe, ou seja, só se deve trabalhar com essa estratégia quando a classe como um todo represente a operação.
 - No caso da **Runnable**, pode pensar como uma tarefa menor que será executada.
 - Vale lembrar que ela também precisa de uma Thread para se executada.

Exemplo de Thread em Java

Criando threads

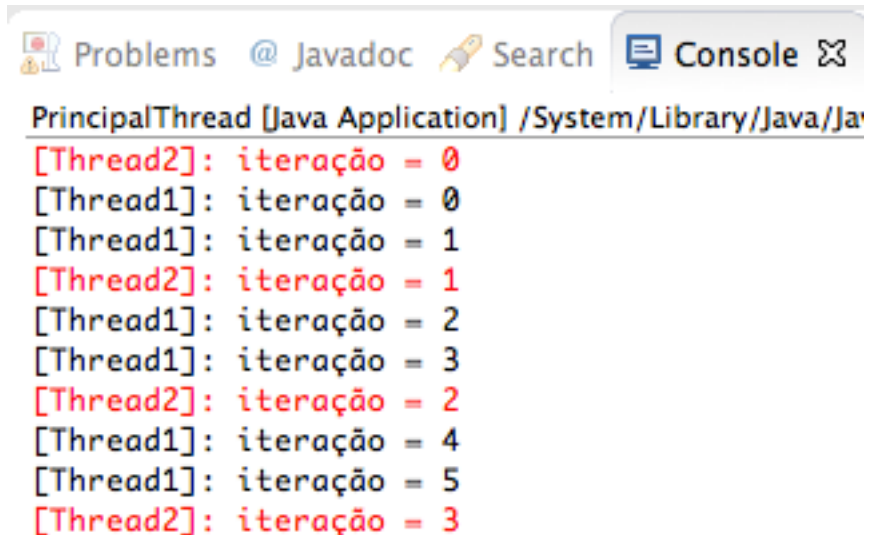
```
public class Thread1 extends Thread {  
  
    @Override  
    public void run() {  
        this.inicio();  
    }  
  
    private void inicio() {  
        try {  
            for(int i = 0; i < 1000; i++) {  
                System.out.println("[Thread1]: iteração = "+i);  
                Thread.sleep(250);  
            }  
        } catch (InterruptedException e) {  
            System.err.println(e);  
        }  
    }  
}
```

Criando threads

```
public class Thread2 extends Thread {  
  
    @Override  
    public void run() {  
        this.inicio();  
    }  
  
    private void inicio() {  
        try {  
            for(int i = 0; i < 500; i++) {  
                System.err.println("[Thread2]: iteração = "+i);  
                Thread.sleep(500);  
            }  
        } catch (InterruptedException e) {  
            System.err.println(e);  
        }  
    }  
}
```

Executando as threads

```
public class PrincipalThread {  
  
    public static void main(String[] args) {  
        Thread1 thread1 = new Thread1();  
        thread1.start();  
  
        Thread2 thread2 = new Thread2();  
        thread2.start();  
    }  
}
```



The screenshot shows an IDE interface with a toolbar at the top containing icons for Problems, Javadoc, Search, and Console. The Console tab is active, displaying the output of a Java application. The output shows two threads, Thread1 and Thread2, each performing a series of iterations. Thread1's iterations are shown in black text, while Thread2's iterations are shown in red text. The iterations are interleaved, indicating that both threads are running concurrently. The output ends with Thread2 completing its 3rd iteration.

```
PrincipalThread [Java Application] /System/Library/Java/Ja  
[Thread2]: iteração = 0  
[Thread1]: iteração = 0  
[Thread1]: iteração = 1  
[Thread2]: iteração = 1  
[Thread1]: iteração = 2  
[Thread1]: iteração = 3  
[Thread2]: iteração = 2  
[Thread1]: iteração = 4  
[Thread1]: iteração = 5  
[Thread2]: iteração = 3
```

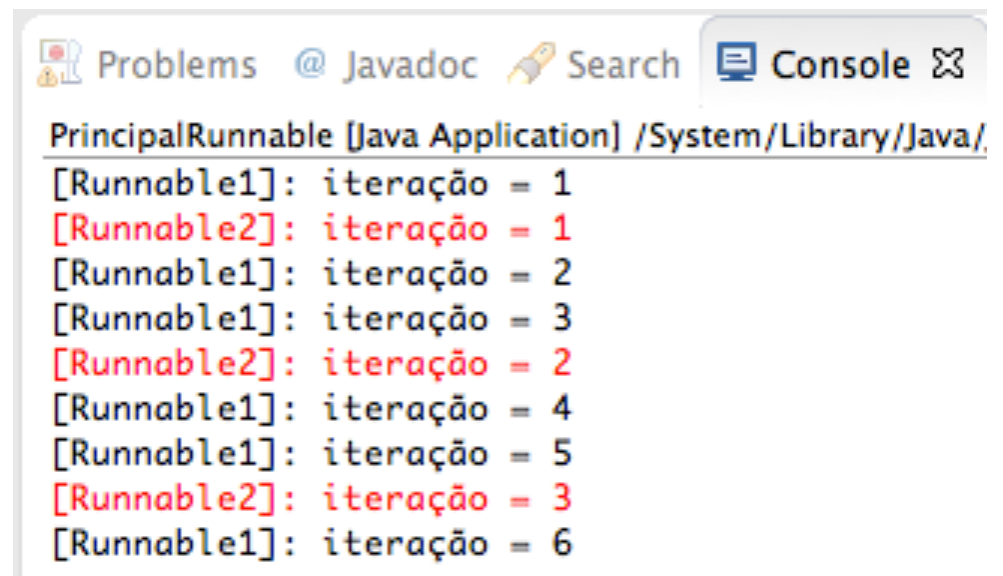
Convertendo Thread para Runnable

Criando Runnable

```
public class Thread1 implements Runnable {  
  
    @Override  
    public void run() {  
        this.inicio();  
    }  
  
    private void inicio() {  
        try {  
            for(int i = 0; i < 1000; i++) {  
                System.out.println("[Runnable1]: iteração = "+i);  
                Thread.sleep(250);  
            }  
        } catch (InterruptedException e) {  
            System.err.println(e);  
        }  
    }  
}
```

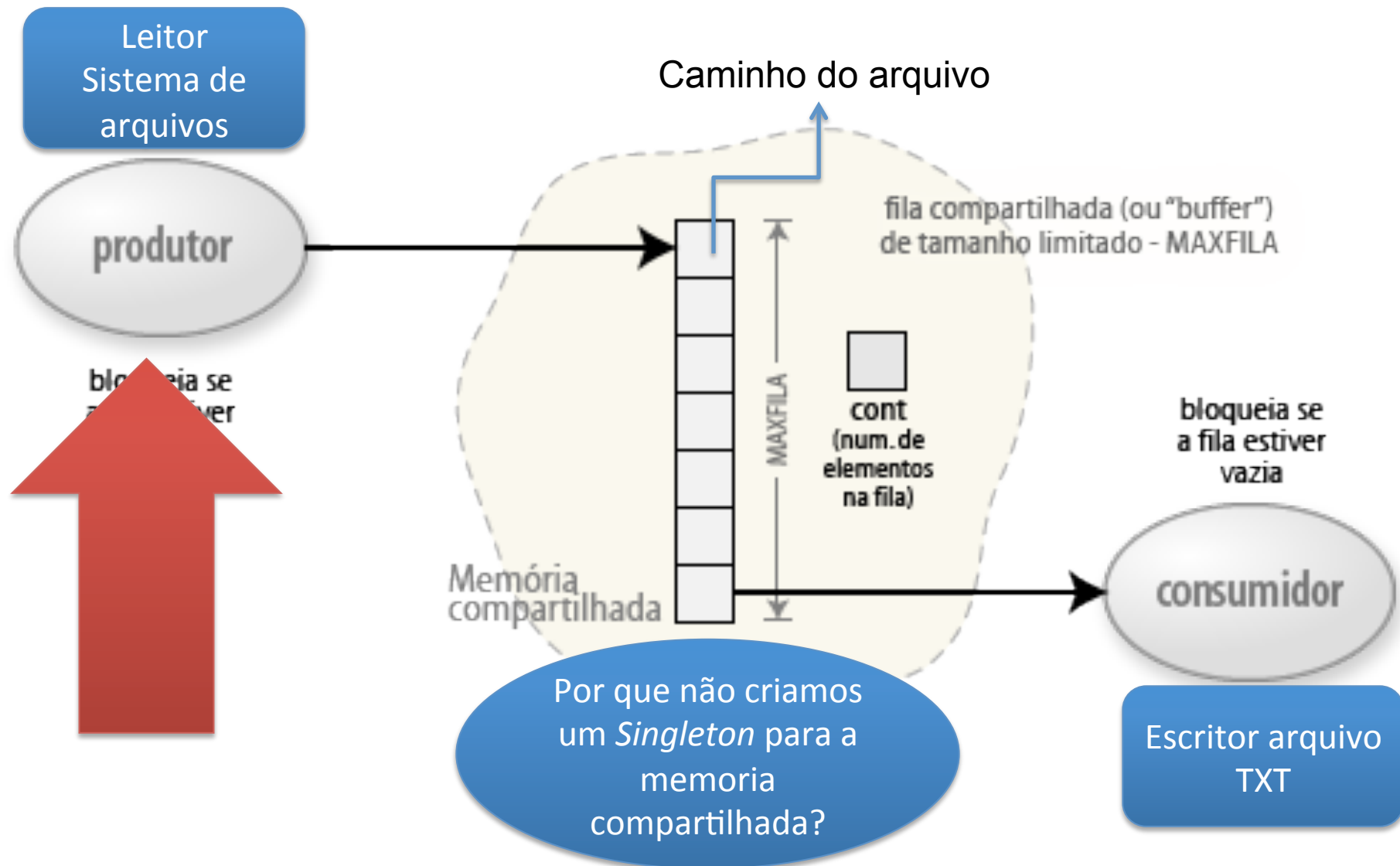
Executando Runnable

```
public class PrincipalRunnable {  
  
    public static void main(String[] args) {  
        Thread thread = null;  
  
        thread = new Thread(new Thread1());  
        thread.start();  
  
        thread = new Thread(new Thread2());  
        thread.start();  
    }  
}
```



```
Problems @ Javadoc Search Console ✕  
PrincipalRunnable [Java Application] /System/Library/Java/  
[Runnable1]: iteração = 1  
[Runnable2]: iteração = 1  
[Runnable1]: iteração = 2  
[Runnable1]: iteração = 3  
[Runnable2]: iteração = 2  
[Runnable1]: iteração = 4  
[Runnable1]: iteração = 5  
[Runnable2]: iteração = 3  
[Runnable1]: iteração = 6
```

Voltando ao nosso estudo de caso...



Transformando a classe em Thread

```
public class LeitorSistemaArquivo extends Thread {
```

```
    @Override  
    public void run() {  
        // TODO Auto-generated method stub  
        super.run();  
    }
```

Método que será
invocado quando
a thread for iniciada

Devemos invocar esse método dentro do método run()

```
    public List<String> recuperarListaArquivo(String caminhaRaiz) {  
        final List<String> listaArquivos = new LinkedList<String>();  
  
        listaArquivos.add(caminhaRaiz);  
  
        this.lerRaizPasta(listaArquivos, caminhaRaiz);  
  
        return listaArquivos;  
    }
```

Como vamos passar esse
parâmetro na execução da
Thread?

Continua...

Transformando a classe em Thread

```
public class LeitorSistemaArquivo extends Thread {  
    private String caminhoRaiz;  
  
    private ServicoMemoriaCompartilhada memoria = ServicoMemoriaCompartilhada.getInstancia();  
  
    @Override  
    public void run() {  
        this.recuperarListaArquivo();  
    }  
  
    public LeitorSistemaArquivo(String caminhoRaiz) {  
        this.caminhoRaiz = caminhoRaiz;  
    }  
  
    private void recuperarListaArquivo() {  
        memoria.adiciona(caminhoRaiz);  
  
        this.lerRaizPasta(caminhoRaiz);  
    }  
}
```

↑
Acesso direto a memoria
compartilhada

Continuando...

Transformando a classe em Thread

```
private void recuperarListaArquivo() {
    memoria.adiciona(caminhoRaiz);

    this.lerRaizPasta(caminhoRaiz);
}

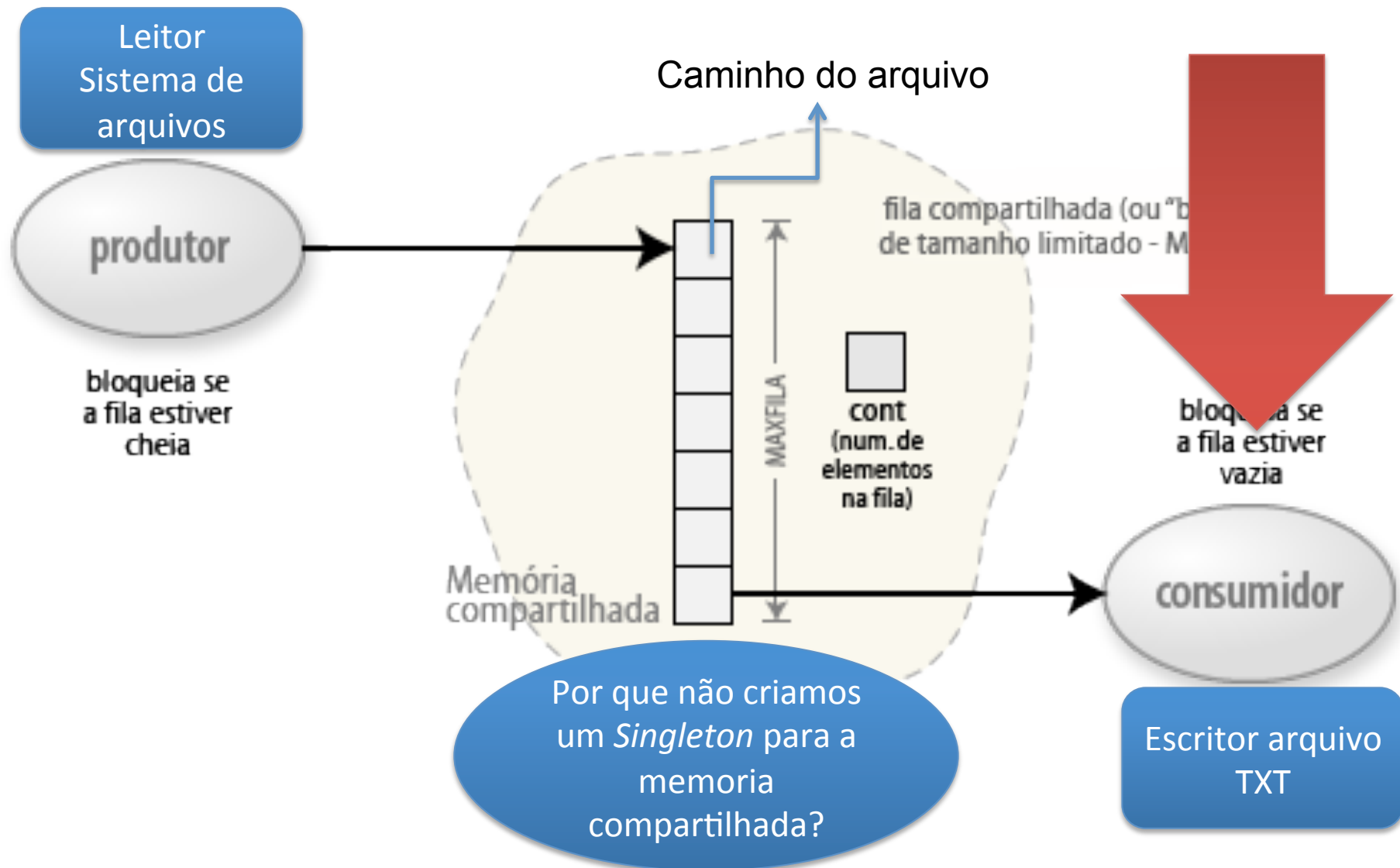
private void lerRaizPasta(String caminhoRaiz) {
    File raiz = new File(caminhoRaiz);

    if(raiz != null && raiz.exists()) {
        File[] arquivos = raiz.listFiles();

        if(arquivos != null) {
            for(File arquivo : arquivos) {
                if(arquivo != null && arquivo.isHidden() == false) {
                    String caminho = arquivo.getAbsolutePath();
                    memoria.adiciona(caminho);

                    if(arquivo.isDirectory()) {
                        lerRaizPasta(caminho);
                    }
                }
            }
        }
    }
}
```

Agora vamos alterar o consumidor



Alterando a classe Escritor...

```
public class EscritorArquivoTexto extends Thread {

    private ServicoMemoriaCompartilhada memoria = ServicoMemoriaCompartilhada.getInstancia();

    @Override
    public void run() {
        this.escreverConteudoArquivo();
    }

    public void escreverConteudoArquivo() {
        OutputStream escritorByte = null;
        OutputStreamWriter escritorCaracter = null;
        BufferedWriter escritorPalavras = null;

        try {
            escritorByte = new FileOutputStream("conteudo/arquivos.txt", true);
            escritorCaracter = new OutputStreamWriter(escritorByte);
            escritorPalavras = new BufferedWriter(escritorCaracter);

            while(memoria.temElemento()) {
                String linha = memoria.recupera();

                escritorPalavras.write(linha);
                escritorPalavras.newLine();
            }
            escritorPalavras.flush();

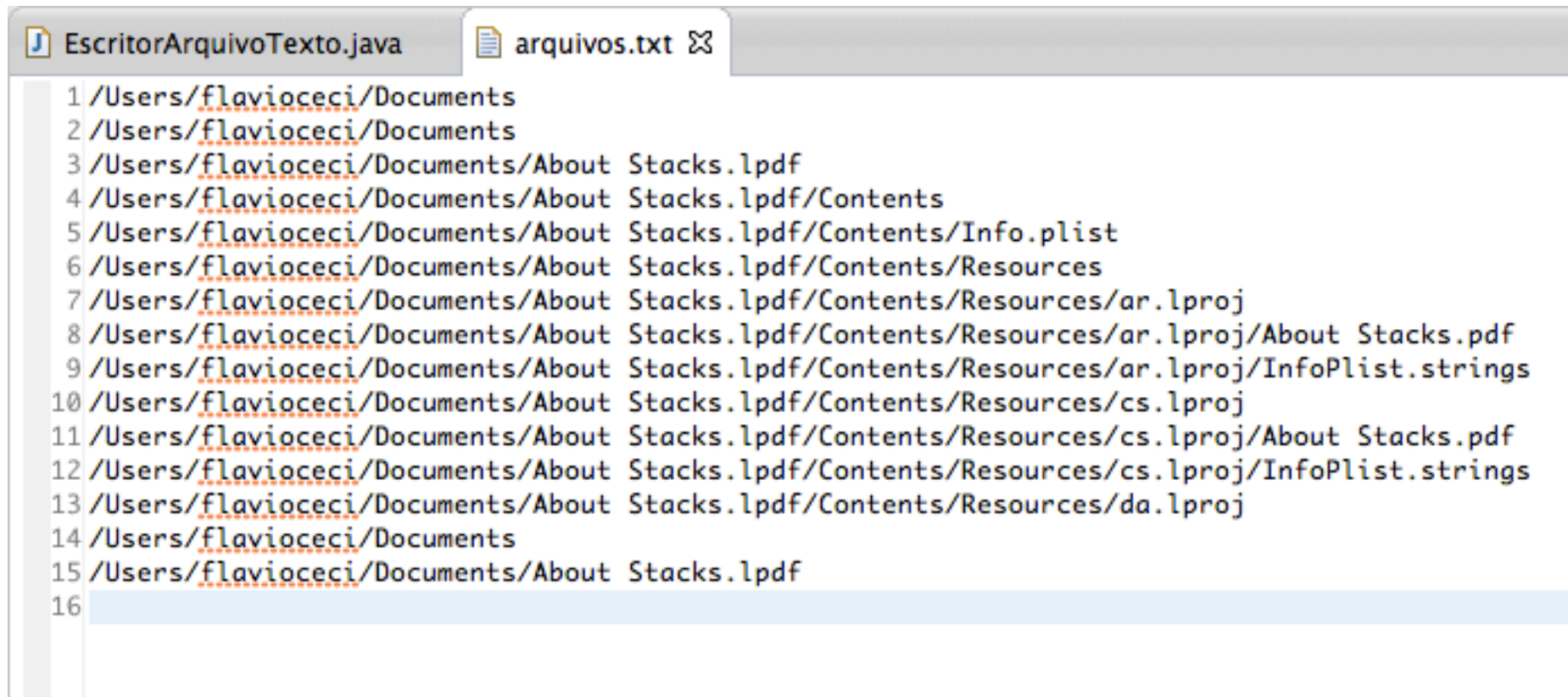
        } catch (FileNotFoundException e) {
            System.err.println(e);
        }
    }
}
```

Alterando a classe Principal

```
public class PrincipalMigracaoThread {  
    public static void main(String[] args) throws InterruptedException {  
        ServicoMemoriaCompartilhada memoria = ServicoMemoriaCompartilhada.getInstancia();  
        LeitorSistemaArquivo leitor = new LeitorSistemaArquivo("/Users/flavioceci/Documents");  
        EscritorArquivoTexto escritor = new EscritorArquivoTexto();  
        long inicio = System.currentTimeMillis();  
  
        leitor.start();  
  
        Por que  
        Isso? { while(memoria.temElemento() == false) {  
                System.out.println("Esperando preencher adicionar elementos na memoria");  
            }  
        escritor.start();  
  
        while(leitor.isAlive() || escritor.isAlive()) {  
            Thread.sleep(500);  
        }  
  
        System.out.println("TEMPO TOTAL: "+(System.currentTimeMillis()-inicio));  
    }  
}
```

Vamos ao resultado...

Onde estão os outros registros?



```
EscritorArquivoTexto.java arquivos.txt ✕
1 /Users/flavioceci/Documents
2 /Users/flavioceci/Documents
3 /Users/flavioceci/Documents/About Stacks.lpdf
4 /Users/flavioceci/Documents/About Stacks.lpdf/Contents
5 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Info.plist
6 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources
7 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/ar.lproj
8 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/ar.lproj/About Stacks.pdf
9 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/ar.lproj/InfoPlist.strings
10 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/cs.lproj
11 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/cs.lproj/About Stacks.pdf
12 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/cs.lproj/InfoPlist.strings
13 /Users/flavioceci/Documents/About Stacks.lpdf/Contents/Resources/da.lproj
14 /Users/flavioceci/Documents
15 /Users/flavioceci/Documents/About Stacks.lpdf
16
```

Por que aconteceu essa situação???

Vamos entender a situação













- Estamos trabalhando com uma estratégia produtor x consumidor;
 - Utilizando uma classe como memória compartilhada;
- Aparentemente a classe consumidora está consumindo mais rápido que a classe produtora pode produzir;
 - Por conta disso a thread ***EscritorArquivoTexto*** “morre” antes da ***LeitorSistemaArquivo*** finalizar.

Exercício

- Desenvolva uma estratégia para corrigir o problema entre a thread produtora e consumidora.
- Baixe o projeto do eclipse disponível no link abaixo:

<http://dl.dropbox.com/u/3025380/prog2/aula14.zip>

Exercício

- ▼  Aula14-Prog2-Thread
 - ▼  src
 - ▼  br.unisul.prog2.aula14
 - ▶  exemplos
 - ▶  serial
 - ▼  thread
 - ▶  EscritorArquivoTexto.java
 - ▶  LeitorSistemaArquivo.java
 - ▶  PrincipalMigracaoThread.java
 - ▶  ServicoMemoriaCompartilhada.java
 - ▶  JRE System Library [JavaSE-1.6]
 -  conteudo