Unidade 2

# Programação Paralela

- Processos
- Threads
- Paralelismo em Java

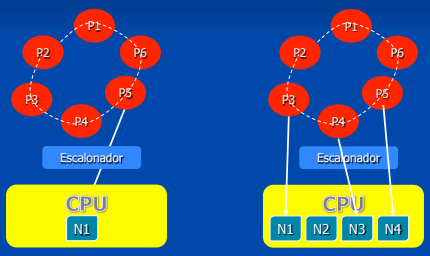
## Processos

- Definição
  - Um programa em execução em uma máquina
  - Identificado pelo seu PID (*Process Identifier*)
- Execução dos Processos
  - Um processador pode executar somente um processo a cada instante
  - Em um S.O. multi-tarefa, processos se alternam no uso do processador – cada processo é executado durante um *quantum* de tempo
  - Se houver N processadores, N processos podem ser executados simultaneamente

2

## Processos

- Escalonamento: 1 núcleo x n núcleos




3

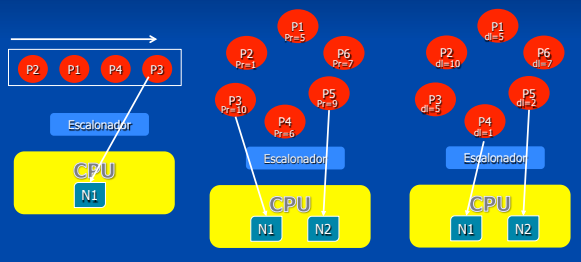
## Processos

- Escalonamento de Processos
  - O escalonador do S.O. seleciona o(s) processo(s) que deve(m) ser executado(s) pelo(s) processador(es)
- Algoritmo de Escalonamento
  - Define a ordem de execução de processos com base em uma fila, prioridade, deadline, ...
  - Em geral, os sistemas adotam uma política de melhor esforço para atender a todos os processos de maneira justa e igualitária
  - Processos do sistema e aplicações críticas (um alarme, por exemplo) exigem maior prioridade

4

## Processos

- Escalonamento: 1 núcleo x n núcleos




5

## Processos

- Estados de um Processo
  - Pronto: processo pronto para ser executado, mas sem o direito de usar o processador
  - Rodando: sendo executado pelo processador
  - Suspenso: aguarda operação de I/O, liberação de um recurso ou fim de tempo de espera
- Processos trocam de estado de acordo com:
  - Algoritmo de escalonamento
  - Troca de mensagens
  - Interrupções de hardware ou software

6

## Processos

- Ciclo de vida simplificado de um processo


7

## Processos

- Troca de Contexto / Preempção
  - O processo em execução é suspenso, e um outro processo passa a ser executado
  - Ocorre por determinação do escalonador ou quando o processo que estava sendo executado é suspenso
  - O contexto do processo suspenso deve ser salvo para retomar a execução posteriormente

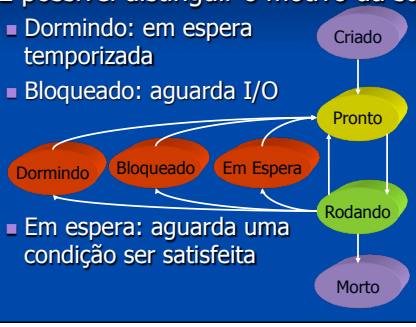
8

## Processos

- O contexto de um processo compreende:
  - O estado do processo
  - Informações para escalonamento
  - Dados para contabilização de uso
  - Um segmento de código
  - Um segmento de dados
  - Os valores dos registradores
  - O contador de programa
  - Uma pilha de execução
  - Arquivos, portas e outros recursos alocados

## Processos

- É possível distinguir o motivo da suspensão
  - Dormindo: em espera temporizada
  - Bloqueado: aguarda I/O
  - Em espera: aguarda uma condição ser satisfeita




## Processos

- Chamadas do Sistema Operacional Unix
  - Criar um Processo:
    - fork() cria uma cópia do processo atual
    - exec() carrega o código do processo

```
...
int pid = fork();            // cria cópia do processo
if  (pid < 0)                // erro na criação
   ...                       // trata o erro
else if (pid > 0)            // é o processo pai
   ...                       // executa código do pai
else if  (pid == 0)          // é o processo filho
   exec("/bin/filho", 0);    // executa código do filho
...
```

11

## Processos

- Chamadas do Sistema Operacional Unix
  - Suspender a Execução: sleep(<tempo>) ou wait() → reinicia com kill(<pid>,SIGWAIT)
  - Obter Identificador do Processo: getpid()
  - Aguarda o Fim dos Processos Criados: join()
  - Finalizar o Processo: exit(<retorno>)
  - Destruir um Processo: kill(<pid>,SIGKILL)

12

## Processos

- Interação com o Usuário no Unix
  - Processos são criados através da interface gráfica ou de comandos digitados na Shell
  - Comandos bloqueiam a Shell, a não ser que sejam seguidos de um &
  - Os processos em execução são listados com o comando ps –ef (ps –aux em versões antigas)
  - Processos são destruídos com kill -9 <pid>

13

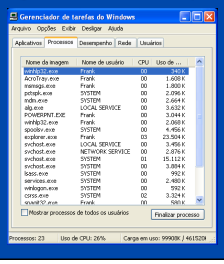
## Processos

- Chamadas da API do Windows
  - Criar um Processo: CreateProcess(<nome>, <comando>, ...) ou CreateProcessAsUser(<usuário>,<nome>, ...)
  - Obter o Identificador do Processo: GetCurrentProcessId()
  - Suspender a Execução: Sleep(<tempo>)
  - Finalizar o Processo: ExitProcess(<retorno>)
  - Destruir um Processo: TerminateProcess(<pid>, <retorno>)

14

## Processos

- Interação com o Usuário no Windows
  - Processos são criados através da interface gráfica ou de comandos digitados no Prompt
    - O Prompt não bloqueia aguardando aplicações Windows, que executam em outra janela
    - Processos são listados/destruídos pelo Gerenciador de Tarefas




15

## Threads

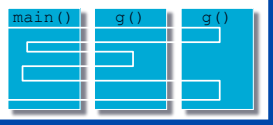
- Definição
  - *Threads* (linhas) de execução são atividades concorrentes executadas por um processo
  - Um processo pode ter uma ou mais *threads*
  - Threads pertencentes a um mesmo processo compartilham recursos e memória
- Suporte a Threads
  - Threads nativas do S.O.
  - Suporte de programação multi-thread
  - Linguagem de programação multi-threaded

16

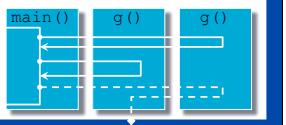
## Programação concorrente

- Todo programa, sendo *monothread* ou *multithread*, inicia com a execução da thread principal.

Exemplo de um programa monothread




Exemplo de um programa multithread




- Mecanismos de sincronização e prioridade podem ser usados para controlar a ordem de execução de threads independentes e colaboradas.

17

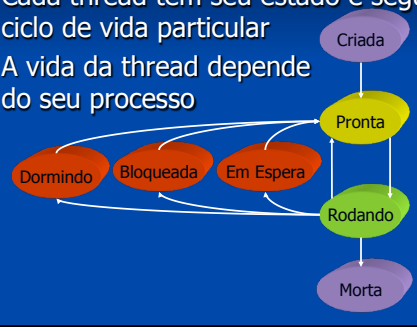
## Programação concorrente

- Por que usar threads ?
  - Permitir que um programa faça mais de uma coisa ao mesmo tempo. Cada thread pode fazer coisas diferentes ao mesmo tempo, por exemplo:
    - Uma thread baixando uma figura da rede e outra renderizando uma imagem;
    - Uma gerenciando a edição de um texto e outra cuidando da impressão de um outro documento em background;
    - Criação de várias threads para processamento de uma tarefa de forma paralela.
- Quando não usar threads ?
  - Quando um programa é puramente seqüencial.

18

## Threads

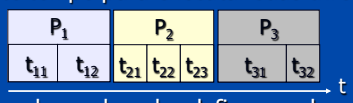
- Cada thread tem seu estado e segue um ciclo de vida particular
- A vida da thread depende do seu processo



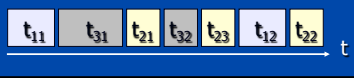

19

## Threads

- Escalonamento
  - Por Processo: escalonador aloca tempo para execução dos processos, que definem como usar este tempo para executar suas threads




  - Por Thread: escalonador define a ordem na qual as threads serão executadas




20

## Threads

- Troca de Contexto
  - Quando duas threads de um mesmo processo se alternam no uso do processador, ocorre uma troca de contexto parcial
  - Numa troca parcial, o contador de programa, os registradores e a pilha devem ser salvos
  - Uma troca de contexto parcial é mais rápida que uma troca de contexto entre processos
  - Uma troca de contexto completa é necessária quando uma thread de um processo que não estava em execução assume o processador

## Threads

- Threads X Processos

| | Processos | Threads |
|---|---|---|
| Troca de Contexto | Completa | Parcial |
| Área de Memória | Independente | Compartilhada |
| Comunicação | Inter-Processo | Intra-Processo |
| Código | Independente | Mesmo código |
| Suporte em S.O.'s | Quase todos | Os mais atuais |
| Suporte em Ling. Prog. | Quase todas | As mais recentes |

22

## Threads

- Threads POSIX
  - Padrão adotado pelos UNIX e outros S.O.'s
  - Criar uma Thread: pthread_create( <thread>, <atrib>, <rotina>, <args>);
  - Obter Ident. da Thread: pthread_self()
  - Suspender Execução: pthread_delay_np(<tempo>)
  - Finalizar a Thread: pthread_exit (<retorno>)
  - Linux usa as mesmas rotinas, mas cria um processo por thread → não é 100% POSIX

23

## Threads

- Threads no Windows
  - Criar uma Thread: CreateThread (<atribs>, <tam_stack>, <rotina>, <params>, <flags>,<thread_id>)
  - Obter Ident. da Thread: GetCurrentThreadId()
  - Suspender Execução: Sleep(<tempo>) ou SuspendThread (<thread>) → retomar com ResumeThread(<t>) ou SwitchToThread(<t>)
  - Finalizar a Thread: ExitThread(<retorno>)
  - Destruir uma Thread: TerminateThread (<thread>,<retorno>)

24

## Exclusão mútua

- Num sistema distribuídos existem recursos que não pode ser acessados simultaneamente por diferentes processos se desejarmos o correto funcionamento de um programa. Para evitar isso, é necessário mecanismos para garantir o acesso exclusivo dos recursos – chamamos isso de exclusão mútua.

## Exclusão mútua

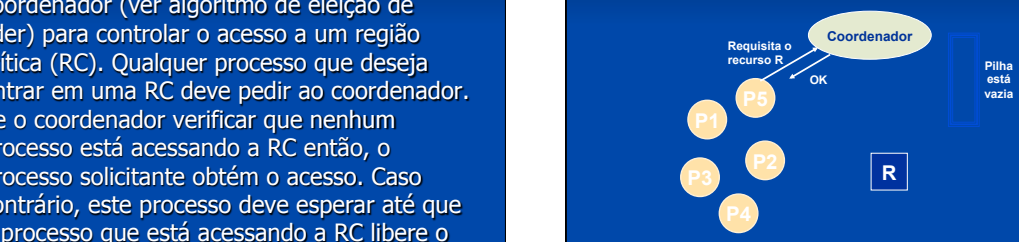
- Propriedade:
  - Exclusão mútua: dado um recurso que pode ser acessado por diferentes processos ao mesmo tempo, somente um processo por vez pode acessar esse recurso. Contudo, um processo que ganha acesso a um recurso deve liberá-lo para que outro processo ganhe acesso ao mesmo.
  - Starvation: qualquer processo que requisita um recurso deve recebê-lo em qualquer momento.

## Exclusão mútua

- Algoritmo centralizado:
  - Um processo do sistema é indicado como coordenador (ver algoritmo de eleição de líder) para controlar o acesso a um região crítica (RC). Qualquer processo que deseja entrar em uma RC deve pedir ao coordenador. Se o coordenador verificar que nenhum processo está acessando a RC então, o processo solicitante obtém o acesso. Caso contrário, este processo deve esperar até que o processo que está acessando a RC libere o recurso e avise o coordenador.

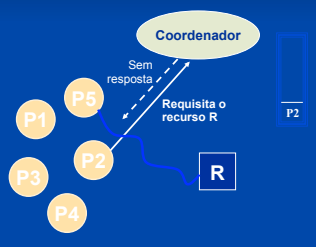
## Exclusão mútua

- Algoritmo centralizado – exemplo:



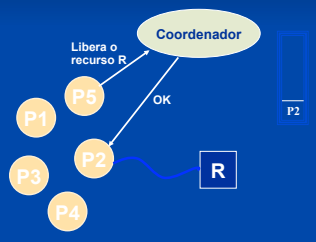

## Exclusão mútua

- Algoritmo centralizado – exemplo:




## Exclusão mútua

- Algoritmo centralizado – exemplo:



## Exclusão mútua

- Algoritmo baseado no token
- Neste algoritmo é definido um anel virtual em que um token circula entre os processos do sistema. Um processo de posse do token tem direito de acesso a uma determinada RC.
- Se ele não deseja acessar a RC o token deve ser passado adiante.

## Exclusão mútua

- Algoritmo baseado no token - exemplo


O que tem que ser feito quando o processo que está com token falha?

## Deadlock

- Num sistema computacional existem recursos que não pode ser acessados simultaneamente por diferentes processos se desejarmos o correto funcionamento de um programa. Para evitar isso, é necessário mecanismos para garantir o acesso exclusivo dos recursos – chamamos isso de exclusão mútua.

## Deadlock

- Características do deadlock, condições necessárias:
  - Exclusão mútua: existência de pelo menos um recurso não compartilhável no sistema, isto é, somente um processo por vez pode utilizar esse recurso;
  - Segura espera: existência de um processo que está utilizando um recurso e está a espera por outros recursos que estão sendo utilizados por outros processos;

## Deadlock

- Recurso não preemptados: um recurso só pode ser liberado (desalocado) de forma voluntária pelo processo que está utilizando esse recurso;
- Espera circular: existência de um conjunto de processos (P0, P1, P2, ... Pn) bloqueados, tal que P0 espera por um recurso mantido por P1, P1 espera por um recurso mantido por P2, ..., Pn-1 espera por um recurso mantido por Pn e, por fim, Pn espera por um recurso mantido por P0.

## Detecção de deadlock centralizado

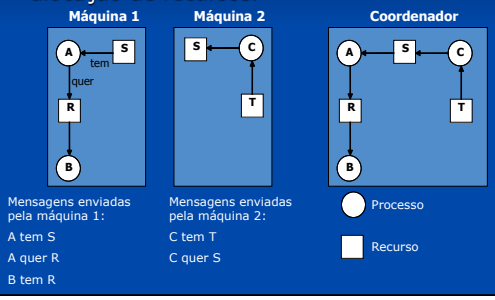
- Cada máquina mantém um grafo de alocação local de seus recursos pelos processos distribuídos.
- Um coordenador central recebe esses grafos locais (através de troca e mensagens) e assim constrói um grafo global (a união dos grafos locais).

## Detecção de deadlock centralizado

- Os grafos locais são enviados pelas máquinas locais ao coordenador toda vez que um recurso é alocado ou desalocado no sistema, ou seja, sempre que um arco é incluído ou excluído do grafo local.

## Detecção de deadlock centralizado

- Detecção de deadlock baseado em grafo de alocação de recursos.




Mensagens enviadas pela máquina 1:
A tem S
A quer R
B tem R

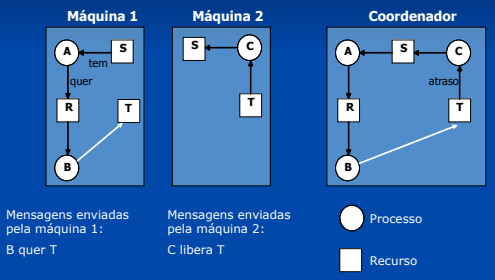
Mensagens enviadas pela máquina 2:
C tem T
C quer S

## Detecção de deadlock centralizado

- Quando o coordenador verifica um ciclo (espera circular) um deadlock é detectado e, para que o deadlock seja removido, o coordenador decide por matar um dos processos causadores do deadlock – um processo de maior ou menor índice.
- Como essas informações são enviadas através da rede pode ocorrer atrasos causando inversão na ordem de entrega das mensagens, causando dessa forma um falso deadlock.

## Detecção de deadlock centralizado

- Falso deadlock




Mensagens enviadas pela máquina 1:
B quer T

Mensagens enviadas pela máquina 2:
C libera T

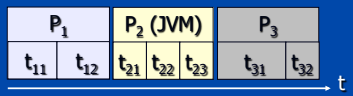
## Paralelismo em Java

- Máquina Virtual Java, Processos e Threads
  - Cada instância da JVM corresponde a um processo do sistema operacional hospedeiro
  - A JVM não possui o conceito de processos – apenas implementa threads internamente
  - Ao ser iniciada, a JVM executa o método main() do programa na thread principal, e novas threads podem ser criadas a partir desta
  - Threads de uma mesma JVM compartilham memória, portas de comunicação, arquivos e outros recursos

41

## Paralelismo em Java

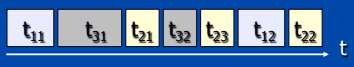
- Implementação de Threads no Java
  - Green Threads
    - Usadas em sistemas sem suporte a *threads*
    - Threads e escalonamento implementados pela máquina virtual Java




42

## Paralelismo em Java

- Implementação de Threads no Java
  - Threads Nativas
    - Usa threads nativas e escalonador do S.O.
    - Usada nos sistemas suportados oficialmente
      - Linux, Solaris e Windows
    - Ideal em máquinas com >1 processador




43

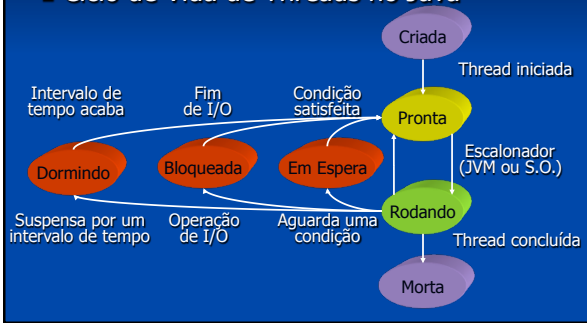
## Paralelismo em Java

- Estados das Threads em Java
  - Pronta: poderia ser executada, mas o processador está ocupado
  - Rodando: em execução
  - Suspensa:
    - Bloqueada: aguarda operação de I/O
    - Em Espera: aguarda alguma condição
    - Dormindo: suspensa por um tempo

44

## Paralelismo em Java

- Ciclo de Vida de Threads no Java




45

## Paralelismo em Java

- Threads na Linguagem Java
  - O conceito de thread está presente em Java através da classe java.lang.Thread, que é um tipo predefinido da linguagem
  - Java também oferece :
    - Mecanismos para sincronização e controle de concorrência entre threads
    - Classes para gerenciamento de grupos (pools) de threads
    - Classes da API que podem ser acessadas concorrentemente (*thread-safe*)

46

## Paralelismo em Java

- Classe Thread e interface Runnable

```
public class Thread extends Object implements Runnable {
   ...                    // Métodos da classe thread
}

public interface Runnable {
   public void run();  // Código executado por uma thread
}
```

47

## Paralelismo em Java

- Principais métodos de java.lang.Thread
  - Construtores: Thread(), Thread (Runnable), Thread(String), Thread(Runnable, String), ...
  - Método run(): contém o código executado pela Thread; herdado da interface Runnable; implementado pela Thread ou por um objeto passado como parâmetro para seu construtor
  - Método start(): inicia a Thread em paralelo com a Thread que a criou, executando o código contido no método run()

48

## Paralelismo em Java

- Criando Threads no Java usando herança
  - Definir uma classe que estenda Thread e implemente o método run()

```
public class MyThread extends Thread {
  public void run() {
    ...                                  // código da thread
  }
}
```

  - Criar thread e chamar start(), que executa run()

```
...
MyThread t = new MyThread();       // cria a thread
t.start();                         // inicia a thread
...
```

49

## Herdando a class `Thread`

```
class MinhaThread1 extends Thread {
  public void run() {
     System.out.println("Bom Dia !");
  }}
```

- O método `run()` contém o código que a thread executa.

```
class Teste1 {
  public static void main(String Args[]) {
     new MinhaThread1().start();
  }}
```

- Para executar a thread é necessário instanciar a classe `MinhaThread` e invocar o método `start()`.

50

## Paralelismo em Java

- Criando Threads no Java usando herança
  - Definir uma classe que implemente Runnable

```
public class MyRun implements Runnable {
  public void run() {
    ...                                  // código da thread
  }
}
```

  - Criar uma Thread passando uma instância do Runnable como parâmetro e chamar start()

```
...
Thread t = new Thread (new MyRun());  // cria a thread
t.start();                            // inicia thread
...
```

51

## Implementando a classe `Runnable`

```
class MinhaThread2 implements Runnable {
  public void run() {
     System.out.println("Bom Dia !");
  }
}
```

- Desta forma, a classe `MinhaThread` pode herdar uma outra classe.

```
class Teste2 {
  public static void main(String Args[]) {
     new Thread(new MinhaThread2()).start();
  }
}
```

52

## Paralelismo em Java

- Criando Threads no Java sem usar herança
  - Criar um Runnable, definindo o método run(), instanciar a thread passando o Runnable e chamar start()

```
...
Runnable myRun = new Runnable() {  // cria o runnable
   public void run() {
     ...                                  // código da thread
   }
};
...
new Thread(myRun).start();        // cria e inicia a thread
...
```

53

## Paralelismo em Java

- Criando Threads no Java sem usar herança
  - Criar uma Thread, definindo o método run(), e chamar start()

```
...
Thread myThread = new Thread() {  // cria a thread
   public void run() {
     ...                                  // código da thread
   }
};
...
myThread.start();                 // executa a thread
...
```

54

## Execução paralela de threads

```
class ImprimirThread_1 implements Runnable {
        String str;
        public ImprimirThread_1(String str) {
                this.str = str;
        }
        public void run() {
                for(;;)
                  System.out.println(str);
        }
}
class TesteConcorrente {
        public static void main(String Args[]) {
                new Thread(new ImprimirThread_1("A")).start();
                new Thread(new ImprimirThread_1("B")).start();
        }
}
```

## Execução paralela de threads

- Método `yield()`, cede o processamento para outra thread.

```
class ImprimirThread_2 implements Runnable {
        String str;
        public ImprimirThread_2(String str) {
                this.str = str;
        }
        public void run() {
           for(;;) {
              System.out.println(str);
              Thread.currentThread().yield();
           }}
}
class TesteConcorrente2 {
        public static void main(String Args[]) {
                new Thread(new ImprimirThread_2("A")).start();
                new Thread(new ImprimirThread_2("B")).start();
        }
}
```

## Paralelismo em Java

- Nome da Thread
  - Identificador não-único da Thread
  - Pode ser definido ao criar a Thread com Thread(String) ou Thread(Runnable, String)
  - Se não for definido na criação, o nome da Thread será "Thread-*n*" (com *n* incremental)
  - Pode ser redefinido com setName(String)
  - Pode ser obtido através do método getName()

57

## Paralelismo em Java

- Prioridade da Thread
  - Valor de 1 a 10; 5 é o *default*
  - Thread herda prioridade da thread que a criou
  - Modificada com setPriority(int)
  - Obtida com getPriority()
- Escalonamento
  - Threads com prioridades iguais são escalonadas em *round-robin*, com cada uma ativa durante um *quantum*

| | |
|---|---|
| 10 | A→B→G |
| 9 | D→E |
| 8 | F |
| 7 | G→H |
| 6 | I |
| 5 | J→K→L |
| 4 | M |
| 3 | N→O |
| 2 | P |
| 1 | Q |

58

## Prioridade de thread

```
class BaixaPrioridade extends Thread {
    public void run() {
      setPriority(Thread.MIN_PRIORITY);
      for(;;) {
      System.out.println("Thread de baixa prioridade
                              executando -> 1");
}}}
class AltaPrioridade extends Thread {
        public void run() {
          setPriority(Thread.MAX_PRIORITY);
       for(;;) {
          for(int i=0; i<5; i++)
             System.out.println("Thread de alta prioridade
                                  executando -> 10");
          try {
             sleep(100);
          } catch(InterruptedException e) {
             System.exit(0);
          }
}}}
```

InterruptedException lança exceção se a thread é interrompida pelo método interrupt();

## Prioridade de thread

```
class Lancador {
        public static void main(String args[]) {
                AltaPrioridade  a = new AltaPrioridade();
                BaixaPrioridade b = new BaixaPrioridade();
                System.out.println("Iniciando threads...");
                b.start();
              a.start();
                // deixa as outras threads iniciar a execução.
                Thread.currentThread().yield();
                System.out.println("Main feito");
        }
}
```

```
Iniciando threads...
Main feito
Thread de alta prioridade executando -> 10
Thread de alta prioridade executando -> 10
Thread de baixa prioridade executando -> 1
Thread de baixa prioridade executando -> 1
Thread de baixa prioridade executando -> 1
Thread de alta prioridade executando -> 10
Thread de alta prioridade executando -> 10
Thread de baixa prioridade executando -> 1
:
```

## Paralelismo em Java

- Outros métodos de java.lang.Thread
  - Obter Referência da Thread em Execução: currentThread()
  - Suspender Execução: sleep(long), interrupt()
  - Aguardar Fim da Thread: join(), join(long)
  - Verificar Estado: isAlive(), isInterrupted()
  - Liberar o Processador: yield()
  - Destruir a Thread: destroy()

61

## Paralelismo em Java

```
public class ThreadSleep extends Thread {
   private long tempo = 0;

   public ThreadSleep(long tempo) { this.tempo = tempo; }  // Construtor

   public void run() { // Código da Thread
      System.out.println(getName() +" vai dormir por "+ tempo +" ms.");
      try {
         sleep(tempo);
         System.out.println(getName() + " acordou.");
      } catch (InterruptedException e) { e.printStackTrace(); }
   }

   public static void main(String args[]) {
      for (int i=0; i<10; i++)
         // Cria e executa as Threads
         new ThreadSleep((long)(Math.random()*10000)).start();
   }
}
```

62

## Paralelismo em Java

- Método interrupt()
  - Um *interrupt* é uma indicação para a thread parar o que está fazendo e fazer outra coisa.
  - É decisão do programador o que a thread deve fazer diante de um *interrupt*, comumente a thread termina.

63

## Paralelismo em Java

- Interrompendo uma thread usando interrupt()
  - Uma thread pode interromper (chamando interrupt()) outra thread somente se este estiver em espera (wait()) ou dormindo (sleep()).
  - Em resposta, a thread em espera/dormindo continua a execução em exception através da criação de um objeto a partir InterruptedException.

64

## Paralelismo em Java

```
public class ThreadInterruptExemplo {
    public static void main(String[] args) {
        Thread1 t1 = new Thread1(); Thread2 t2 = new Thread2(t1);
        t1.start(); t2.start();
}}
class Thread2 extends Thread {
    private Thread t1;
    Thread2(Thread t1) {this.t1 = t1;}
    public void run() {
        int sleepTime = (int) (Math.random() * 3000);
        System.out.println(getName() + " dormindo por " + sleepTime
                           + " milisegundos.");
        try {
            Thread.sleep(sleepTime); // acorda após sleepTime
        } catch (InterruptedException e) {
        }
        t1.interrupt(); // interrompe a thread t1
}}
```

65

## Paralelismo em Java

```
class Thread1 extends Thread {
    int count = 0;
    public void run() {
        while (!isInterrupted()) {
            try {
                Thread.sleep((int) (Math.random() * 10000));
            } catch (InterruptedException e) {
                // Thread interrompida para fazer outra coisa...
                System.out.println(getName() + " quase terminando...");
                interrupt();
            }
            System.out.println(getName() + " " + count++);
        }
    }
}
```

66

## Paralelismo em Java

- Médodo join()
  - faz com que uma thread t2 espere por uma outra t1 que está em execução morra.
  - Após a morte de t1, a Thread t2 sai da suspensão e executa o que tem que executar antes de morrer.
  - A thread ao chamar join fica em suspensão;
  - A thread t2 deve ter a referência da outra t1 para chamar join: exemplo t1.join().

67

## Paralelismo em Java

```
public class JoinExample {
    public static void main(String[] args) {
        ThreadJ1 t1 = new ThreadJ1(); ThreadJ2 t2 = new ThreadJ2(t1);
        t1.start(); t2.start();
    }
}
class ThreadJ1 extends Thread {
    int count = 0;
    public void run() {
        try {
            Thread.sleep((int) (5000));
            System.out.println(getName() + " Acordou !");
        } catch (InterruptedException e) {
        }
    }
}
```

68

## Paralelismo em Java

```
class ThreadJ2 extends Thread {
    private Thread t1;
    ThreadJ2(Thread t1) {
        this.t1 = t1;
    }
    public void run() {
        int sleepTime = (int) (2000);
        System.out.println(getName() + " dormindo por " + sleepTime
                + " milisegundos.");
        try {
            Thread.sleep(sleepTime);
            t1.join(); // dorme junto com a thread t1
        } catch (InterruptedException e) {}
            System.out.println(getName() + " Acordei (do join) junto com a
                thread t1");
    }
}
```

69

## Paralelismo em Java

- Gerenciando a execução de threads
  - Executores gerenciam grupos de Threads
    - Interfaces Executor e ExecutorService (que estende a primeira; é mais completa)
  - São criados através da classe Executors:
    - SingleThreadExecutor usa uma thread para executar todas atividades submetidas sequencialmente

70

## Paralelismo em Java

- Gerenciando a execução de threads
  - São criados através da classe Executors:
    - FixedThreadPool usa um grupo de threads de tamanho fixo para executar tarefas. Se uma thread não estiver disponível para a tarefa, a tarefa é colocada numa fila até que outra termine;

71

## Paralelismo em Java

- Gerenciando a execução de threads
  - São criados através da classe Executors:
    - CachedThreadPool cria threads sob demanda para serem executadas em paralelo. Threads antigas disponíveis são reutilizadas para novas tarefas. Mas se uma thread não é usada por 60 segundos ela é terminada e removida do pool para economizar memória.

72

## Paralelismo em Java

- Exemplo de uso de Executor

```
import java.util.concurrent.*;

public class ExecutorTest {
   public static void main(String args[]) {
      ExecutorService exec = Executors.newSingleThreadExecutor();
 //   ExecutorService exec = Executors.newFixedThreadPool(5);
 //   ExecutorService exec = Executors.newCachedThreadPool();
      for (int i=0; i<10; i++) {
         // Cria e executa as threads
         exec.execute(new ThreadSleep((long)(Math.random()*10000)));
      }
      // Encerra o executor assim que as threads terminarem
      exec.shutdown();
   }
}
```



## Paralelismo em Java

- Uso de Executor com CompletionService
  - Muitas vezes é necessário esperar que todas as tarefas terminem para que a thread principal (main) realize uma computação;
  - O CompletionService pode ser usado para fazer com que a thread principal espere cada thread do pool termine sua tarefa para realizar uma computação;
  - A computação pode ser realizada em ordem diferente de quando a tarefa foi submetida.



## Paralelismo em Java

- Uso de Executor com CompletionService

```
import java.util.concurrent.*;
final class GenLongTask implements Callable<Long> {
    private long tempo = 0;
    GenLongTask() {
        this.tempo = (long) (Math.random() * 10000);
    }
    public Long call() {
          System.out.println("** "+Thread.currentThread().getName()
                                  +"tempo = "+tempo);
        try {    // Long operations
            Thread.sleep(tempo);
        } catch (InterruptedException e) {e.printStackTrace();}
        return new Long(tempo);
    }
}
```



## Paralelismo em Java

- Uso de Executor com CompletionService

```
public class ExecutorTest3 {
   public static void main(String args[]) {
      ExecutorService threadPool = Executors.newFixedThreadPool(4);
      CompletionService<Long> pool =
                  new ExecutorCompletionService<Long>(threadPool);
      for (int i = 0; i < 10; i++) {
         pool.submit(new GenLongTask());
      } // Tarefas submetidas !
      for (int i = 0; i < 10; i++) {
    // Thread main faz a computação à medida que cada thread termina
         try {
            Long result = pool.take().get();
         System.out.println(Thread.currentThread().getName()
                                                + " result= " + result);
```



## Paralelismo em Java

- Uso de Executor com CompletionService

```
} catch (InterruptedException e) {e.printStackTrace();
          } catch (ExecutionException e) {e.printStackTrace();
          }
      }
      System.out.println("Antes do shutdown");
      threadPool.shutdown();
      System.out.println("Depois do shutdown");
   }
}
```



## Paralelismo em Java

- Escalonamento de atividades
  - Um ScheduledExecutorService permite escalonar a execução de atividades, definindo um atraso para início da atividade e/ou um período de repetição entre execuções
    - SingleThreadScheduledExecutor usa uma thread para todas as atividades escalonadas
    - ScheduledThreadPool cria um grupo de threads para executar as atividades

## Paralelismo em Java

- Escalonamento com ScheduledExecutor

```
import java.util.concurrent.*;
public class ScheduleThread extends Thread {
   public void run () { System.out.println(getName() + " executada.");
   }
   public static void main(String args[]) {
      ScheduledExecutorService exec =
                  Executors.newSingleThreadScheduledExecutor();
      for (int i=0; i<10; i++) {
         long tempo = (long) (Math.random()*10000);
         System.out.println("Thread-" + i
                              + " será executada em "+ tempo + "ms.");
         // Escalona a execução da thread
         exec.schedule(new ScheduleThread(),
                        tempo, TimeUnit.MILLISECONDS);
      }
      exec.shutdown();
   }}
```



## Paralelismo em Java

- Execução periódica com ScheduledExecutor

```
import java.util.concurrent.*;
public class CountdownThread extends Thread {
   private static long tempo = 0, cont = 10, espera = 5, intervalo = 1;
   public CountdownThread(long tempo) { this.tempo = tempo; }
   public void run () {
      System.out.println(tempo +"segundos para o encerramento.");
      tempo--; }
   public static void main(String args[]) {
      ScheduledExecutorService exec =
                  Executors.newSingleThreadScheduledExecutor();
     exec.scheduleAtFixedRate(new CountdownThread(cont),
                                    espera, intervalo, TimeUnit.SECONDS);
     try { exec.awaitTermination(cont+espera, TimeUnit.SECONDS);
     } catch (InterruptedException ie) { ie.printStackTrace(); }
     exec.shutdown();
   }
}
```



## Paralelismo em Java

- Escalonamento usando Timer

```
import java.util.*;
public class Relogio1 {
    public static void main(String[] args) {
        Timer t = new Timer();
        t.schedule(new TimerTask() {
            public void run() {
                System.out.println(new Date().toString());
            }
        }, 3000, 1000); //Espera 3s e print a cada 1s
    }
}
```



## Paralelismo em Java

- Variáveis locais em Threads
  - A classe ThreadLocal permite criar variáveis locais para Threads (ou seja, que têm valores distintos para cada Thread)

```
// Cria variável local
static ThreadLocal valorLocal = new ThreadLocal();
...
// Define o valor da variável para esta Thread
valorLocal.set(new Integer(10));
// Obtém o valor de var correspondente a esta Thread
int valor = ((Integer) valorLocal.get()).intValue();  // valor = 10
// Outra thread pode acessar a mesma variável e obter outro valor
int valor = ((Integer) valorLocal.get()).intValue();  // valor = 0
```



## Paralelismo em Java

- Exemplo1: ThreadLocal

```
public class ExemploThreadLocal {
    public static void main(String[] args) {
        Obj1 o1 = new Obj1();
        ProcThread t1 = new ProcThread(o1, "Texto da ThreadLocal t1");
        ProcThread t2 = new ProcThread(o1, "Texto da ThreadLocal t2");
        t1.start(); t2.start();
        System.out.println("Sou a "+Thread.currentThread().getName()
 +": "+  t1.tL1.get());
    }
}
class Obj1 {
    public void print() {
        System.out.println("Sou a "+Thread.currentThread().getName()
                            +"no Obj1: "+Obj2.tL2.get());
}}
class Obj2 {
    public static ThreadLocal<String> tL2 = new ThreadLocal<String>();
}
```



## Paralelismo em Java

- Exemplo1: ThreadLocal

```
class ProcThread extends Thread {
    ThreadLocal<String> tL1;
    public Obj1 o1;
    public String processo;
    ProcThread(Obj1 o1,String processo) {
        tL1 = new ThreadLocal<String>();
        this.o1 = o1;
        this.processo = processo;
        tL1.set(processo); // Thread Main faz esse set
        System.out.println("**"+Thread.currentThread().getName()+
                            ": "+tL1.get());
    }
    public void run() {
        Obj2.tL2.set(processo); // A thread corrente vai em Obj2 para setar a tL2
        o1.print();
        System.out.println(this.tL1.get()); //Dá null, pois só a thread Main fez set
}}
```

## Paralelismo em Java

- Exemplo2: ThreadLocal

```
class My_Thread extends Thread {
    outroObj1 o1; outroObj2 o2;
    ThreadLocal<Double> t1; int n;
    My_Thread(outroObj1 o1, outroObj2 o2, ThreadLocal<Double> tl, int n) {
        this.o1 = o1; this.o2 = o2;
        this.n = n; this.t1 = tl;
        System.out.println(Thread.currentThread().getName() + "* no construtor
                    de My_Thread Integer= " + tl.get());
    }
    public void run() {
        System.out.println(Thread.currentThread().getName() + "* tem
                    Integer= "+ t1.get());
        o1.set(new Double(n));
        for (int i = 1; i < 2; i++)
            System.out.println(Thread.currentThread().getName() + "*** get:
                ThreadLocal para: " + o2.get());
    }}
```

85

## Paralelismo em Java

- Exemplo2: ThreadLocal

```
class outroObj1 {
    Double i = 0.0;
    ThreadLocal<Double> tLocal;
    outroObj1(ThreadLocal<Double> tLocal) {
        this.tLocal = tLocal;
        System.out.println(Thread.currentThread().getName()
            + " no construtor de outroObj1: ThreadLocal = " + tLocal.get());
    }
    public void set(Double i) {
        tLocal.set(i);
        System.out.println(Thread.currentThread().getName()
                    + "** no outroObj1 set: ThreadLocal para: " + tLocal.get());
    }
}
```

86

## Paralelismo em Java

- Exemplo2: ThreadLocal

```
class outroObj2 {
    Double i = 0.0;
    ThreadLocal<Double> tLocal;
    outroObj2(ThreadLocal<Double> tLocal) {
        this.tLocal = tLocal;
    }
    public Double get() {
        return (Double) tLocal.get();
    }
}
```

87

## Paralelismo em Java

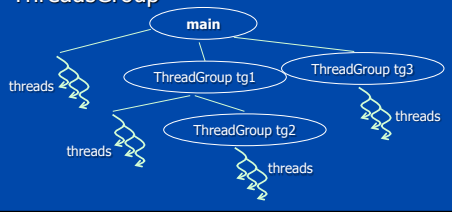
- Exemplo2: ThreadLocal

```
public class ThreadLocalTest {
    public static void main(String args[]) {
    ThreadLocal<Double> tLocal = new ThreadLocal<Double>();
    tLocal.set(11.0);
    outroObj1 o1 = new outroObj1(tLocal);
    outroObj2 o2 = new outroObj2(tLocal);
    System.out.println(Thread.currentThread().getName() + "* tem Integer= "
                                        + tLocal.get());
    My_Thread m1 = new My_Thread(o1, o2, tLocal, 6);
    tLocal.set(22.0);
    My_Thread m2 = new My_Thread(o1, o2, tLocal, 7);
    tLocal.set(33.0);
    My_Thread m3 = new My_Thread(o1, o2, tLocal, 8);
    System.out.println(Thread.currentThread().getName() + "* tem Integer= "
                                        + tLocal.get());
    m1.start(); m2.start(); m3.start();
    }}
```

88

## Paralelismo em Java

- Grupos de Threads
  - A classe ThreadGroup define um grupo de Threads que são controladas conjuntamente
  - Um ThreadGroup pode conter outras sub ThreadsGroup




89

## Paralelismo em Java

- Grupos de Threads
  - O grupo da Thread é definido usando o construtor Thread(ThreadGroup, ...)

```
// Cria grupo de Threads
ThreadGroup myGroup = new ThreadGroup("MyThreads");
...
// Cria Threads e as insere no grupo
Thread myThread1 = new MyThread(myGroup,"MyThread"+i);
Thread myThread2 = new MyThread(myGroup,"MyThread"+i);
...
// Interrompe todas as Threads do grupo
myGroup.interrupt();
...
```

```
public static void main (String [] args) {
    ThreadGroup tg1 = new ThreadGroup ("A");
    ThreadGroup tg2 = new ThreadGroup (tg1, "B");
}
```

90

## Paralelismo em Java

- A ThreadGroup tem três métodos que permitem modificar o estado corrente de todas threads dentro de um grupo:
  - resume
  - stop
  - Suspend
- Estes métodos aplicam a mudança de estado apropriada para cada thread do grupo e seus subgrupos.

91

## Paralelismo em Java

- Grupos de Threads
  - É possível, por exemplo, iniciar ou suspender todas as threads de grupo de uma só vez;
  - Toda thread criada pela thread main pertence implicitamente ao grupo main:
    - Thread.*currentThread().getThreadGroup().getName()*
  - Com a getThreadGroup é possível obter outras informações do grupo de thread:
    - Número de threads
    - Nome dessas threads

92

## Paralelismo em Java

- Informações do Grupos de Threads

```
class EnumerateTest {
   void listCurrentThreads() {
      ThreadGroup currentGroup = Thread.currentThread().getThreadGroup();
      int numThreads;
      Thread[] listOfThreads;
      numThreads = currentGroup.activeCount();
      listOfThreads = new Thread[numThreads];
      currentGroup.enumerate(listOfThreads);
      for (int i = 0; i < numThreads; i++) {
         // Imprime o nome das threads pertencente ao grupo currentGroup
         System.out.println("Thread #" + i + " = " + listOfThreads[i].getName());
      }
   }
}
```

93

## Paralelismo em Java

- Interrompe threads de um grupo

```
public class InterruptThreadGroup {
   public static void main(String[] args) {
      ThreadGroup tg= new ThreadGroup("MyThreads");
      // Cria Threads e as insere no grupo
      Thread myThread1 = new MinhaThread(tg,"MyThread1");
      Thread myThread2 = new MinhaThread(tg,"MyThread2");
      myThread1.start(); myThread2.start();
      try {
         Thread.sleep(2000); // espera 2s
      } catch (InterruptedException e) {}
      tg.interrupt(); // Interrompe todas as Threads do grupo
      //Thread.currentThread().getThreadGroup().interrupt();
         System.out.println(Thread.currentThread().getName() + "
                              interrompendo Thread-1 e Thread-2.");
}}
```

94

## Paralelismo em Java

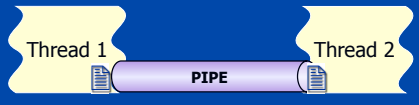
- Interrompe threads de um grupo

```
class MinhaThread extends Thread {
   public MinhaThread(ThreadGroup myGroup, String string) {
      super(myGroup, string);
   }
   public void run() {
      synchronized ("A") {
         System.out.println(getName() + " vai esperar.");
         try {
            "A".wait();
         } catch (InterruptedException e) {
            System.out.println(getName() + " interrompida.");
         }
         System.out.println(getName() + " terminando.");
      }
}}
```

95

## Pipes

- Pipes = canais de comunicação
  - Duas threads podem trocar dados por um pipe → comunicação de 1 para 1
  - Threads devem rodar na mesma máquina virtual → comunicação local



## Pipes

- Disponíveis em Java através das classes PipedInputStream e PipedOutputStream
- Principais métodos:
  - Criação dos Pipes:
    is = new PipedInputStream();
    os = new PipedOutputStream(is);
  - Envio e recepção de bytes / arrays de bytes:
    is.read() // bloqueante
    os.write(dado); os.flush();
  - Exceção: java.io.IOException

## Pipes

- Uso de Pipes com fluxos (*streams*) de dados
  - Criação do fluxo:
    in = new java.io.DataInputStream(is);
    out = new java.io.DataOutputStream(os);
  - Envio e recepção de dados:
    <*tipo*> var = in.read<*tipo*>();
    out.write<*tipo*>(var);
    onde <*tipo*> = Int, Long, Float, Double, etc.

## Pipes

```
public class Producer extends Thread {
   private DataOutputStream out;
   private Random rand = new Random();
   public Producer(PipedOutputStream os) {
      out = new DataOutputStream(os);   }
   public void run() {
     while (true)
        try {
           int num = rand.nextInt(1000);
           out.writeInt(num);
           out.flush();
           sleep(rand.nextInt(1000));
         } catch(Exception e) { e.printStackTrace(); }
   }
}
```

## Pipes

```
public class Consumer extends Thread {
   private DataInputStream in;
   public Consumer(PipedInputStream is) {
      in = new DataInputStream(is); }
   public void run() {
      while(true)
         try {
            int num = in.readInt();
            System.out.println("Número recebido: " + num);
         } catch(IOException e) {  e.printStackTrace(); }
   }
}
```

## Pipes

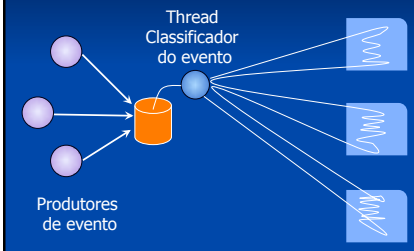
```
public class PipeTest {
   public static void main(String args[]) {
     try {
       PipedOutputStream out = new PipedOutputStream();
       PipedInputStream in = new PipedInputStream(out);

       Producer prod = new Producer(out);
       Consumer cons = new Consumer(in);

       prod.start();
       cons.start();
     } catch (IOException e) { e.printStackTrace(); }
   }
}
```

## Pipes

- Uso de Pipes com fluxos de objetos
  - Criação do fluxo:
    in = new java.io.ObjectInputStream(is);
    out = new java.io.ObjectOutputStream(os);
  - Envio e recepção de objetos:
    <*classe*> obj = (<*classe*>) in.readObject();
    out.writeObject(obj);
    onde <*classe*> = classe do objeto lido/enviado
  - Envio e recepção de Strings:
    - String str = in.readUTF();
    - out.writeUTF(str);

## Pipes

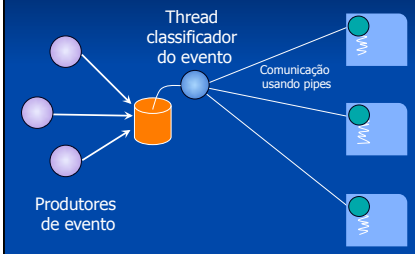
- Exemplo




Classe/objeto que consome o evento através de invocação de método pela thread Classificador

## Pipes

- Exemplo 2




Melhor desempenho: cada thread faz a execução em paralelo

## Pipes – Exemplo 2

```
import java.util.*;
import java.io.*;
public class PipeTeste2 {
  public static void main(String args[]) {
    try {
      PipedOutputStream[] out = new PipedOutputStream[3];
      for (int i = 0; i < 3; i++) {
        out[i] = new PipedOutputStream();
        PipedInputStream in = new PipedInputStream(out[i]);
        new Consumer2(in).start();
      }
      new Producer2(out).start();
    } catch (IOException e) {
      e.printStackTrace();
    }}
}
```

## Pipes – Exemplo 2

```
class Producer2 extends Thread {
  private DataOutputStream[] out =
                        new DataOutputStream[3];
  private Random rand = new Random();
  public Producer2(PipedOutputStream[] os) {
      for(int i=0;i<3;i++){
        out[i]=new DataOutputStream(os[i]);
  }}
  public void run() {
    while (true)
      try {
        int num = rand.nextInt(1000);
        System.out.println("Produtor enviou " + num);
        for (int i = 0; i < 3; i++) {
          out[i].writeInt(num);
```

## Pipes – Exemplo 2

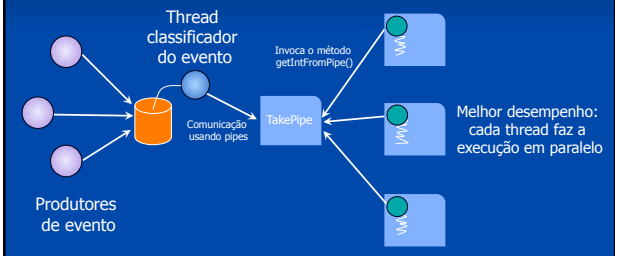
```
          out[i].flush();
          sleep(rand.nextInt(3));
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

## Pipes – Exemplo 2

```
class Consumer2 extends Thread {
  private DataInputStream in;
  public Consumer2(PipedInputStream is) {
      in = new DataInputStream(is);
  }
  public void run() {
    while (true)
      try {
        int num = in.readInt(); // Bloqueante
        System.out.println(Thread.currentThread().
            getName() + " Consumidor recebeu: " + num);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}
```

## Pipes

- Exemplo 3: 3 consumidores usando o mesmo pipe, funciona ??




## Pipes – Exemplo 3

```
import java.util.*;
import java.io.*;
public class PipeTeste3 {
   public static void main(String args[]) {
      try {
         PipedOutputStream out = new PipedOutputStream();
         PipedInputStream in = new PipedInputStream(out);
         Producer3 prod = new Producer3(out);
         TakePipe tp = new TakePipe(in);
         new Consumer3(tp).start();
         new Consumer3(tp).start();
         prod.start();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }}
```

## Pipes – Exemplo 3

```
class Producer3 extends Thread {
   private DataOutputStream out;
   private Random rand = new Random();
   public Producer3(PipedOutputStream os) {
         out = new DataOutputStream(os); }
   public void run() {
      while (true)
         try {
            sleep(10);
            int num = rand.nextInt(1000);
            System.out.println("Produtor enviou " + num);
            out.writeInt(num);
            out.flush();
         } catch (Exception e) {e.printStackTrace();}
   }
}
```

## Pipes – Exemplo 3

```
class Consumer3 extends Thread {
   private TakePipe tp;
   public Consumer3(TakePipe tp) {
         this.tp = tp;
   }
   public void run() {
      while (true) {
         try {
            int num = tp.getIntFromPipe();
            System.out.println(Thread.currentThread().
                getName() + " Consumidor recebeu: " + num);
            Thread.yield();
         } catch (Exception e) {
            e.printStackTrace();
         }
}}}
```

## Pipes – Exemplo 3

```
class TakePipe {
   private DataInputStream in;
   int num;
   public TakePipe(PipedInputStream is) {
      in = new DataInputStream(is);
   }
   public int getIntFromPipe() {
      try {
         num = in.readInt();
         System.out.println("TakePipe recebeu: " + num);
      } catch (IOException e) {
         e.printStackTrace();
      }
      return num;
   }
}
```