Processos e Threads

65DSD

Fernando dos Santos
fernando.santos@udesc.br

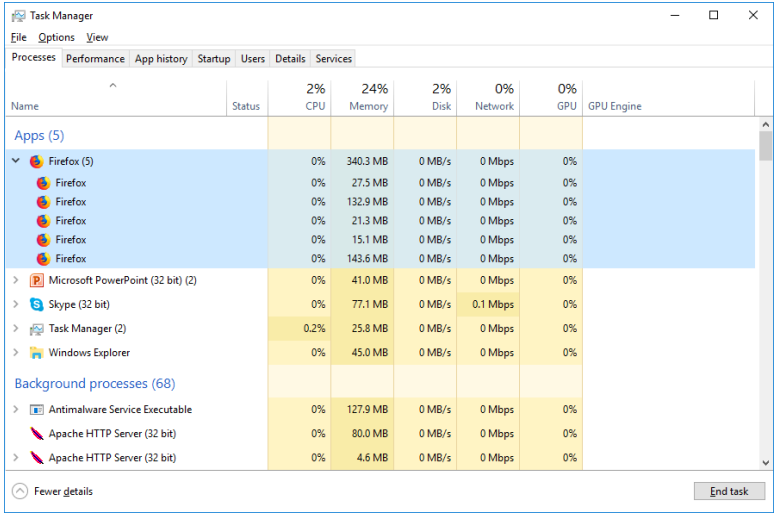
UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Leitura

- TANENBAUM, A. S. Sistemas operacionais modernos. 3. ed. São Paulo: Pearson, 2010. 653 p.
  - Capítulo 2
    - Seções 2.1 e 2.2

Processos

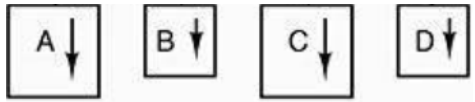
- Um processo é um **programa** em execução
  - Pode criar processos filhos



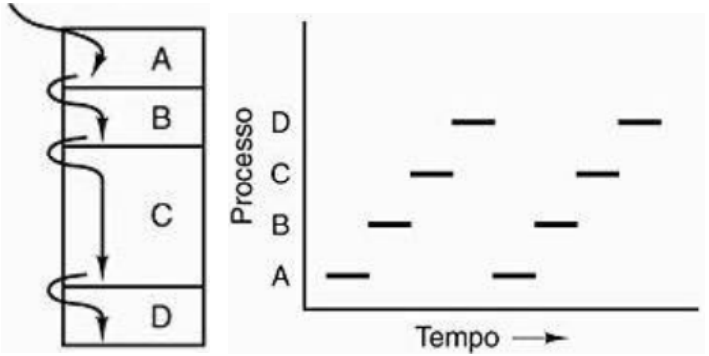
- Quantos processos podem ser executados **simultaneamente**?
  - em um computador com **apenas 1 CPU**

Processos

- Uma CPU executa **apenas um processo** por vez
- No decorrer de 1 segundo, a CPU pode executar vários processos
  - Dá ao usuário a ilusão de simultaneidade e paralelismo
    - **Pseudoparalelismo**
  - O mecanismo de trocas rápidas é chamado de **multiprogramação**



modelo conceitual de 4 processos independentes: cada qual c/ seu **contador de programa**

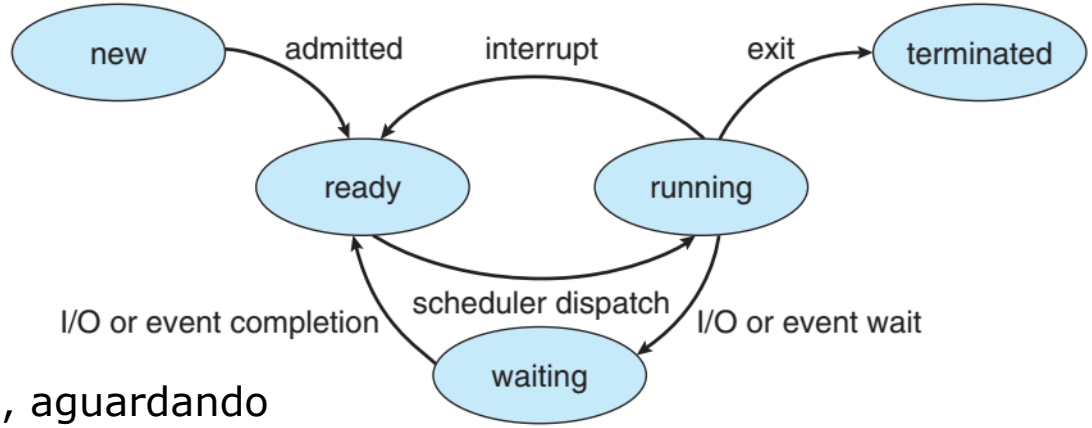


multiprogramação: rápido chaveamento entre processos

Processo vs. Programa: Diferença

- Programa → algoritmo/receita + entradas e saídas
- Processo → Atividade
  - Pode ser "pausada" e retomada
  - Inclui um **programa**, suas entradas e saídas, e um **estado**
- Exemplo: preparar pizza vs. primeiros socorros do filho(a)

Processos: estados

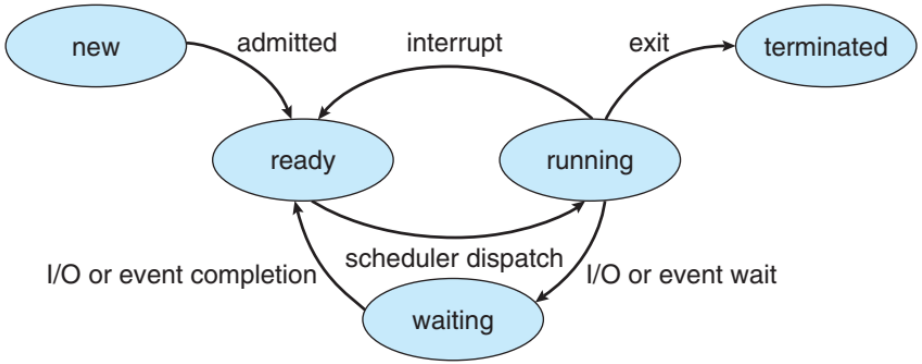


- **New**
  - está sendo criado
- **Ready**
  - pronto para ser executado, aguardando uma CPU
- **Running**
  - usando a CPU naquele instante
- **Waiting**
  - aguardando algum evento (ex: leitura do disco, ou evento)
- **Terminated**
  - finalizou a execução

Processos: transições

- Transições são gerenciadas pelo S.O.

1. **New → Ready**
   - S.O. finalizou a criação do processo
2. **Ready → Running**
   - S.O. selecionou o processo para ocupar a CPU
3. **Running → Ready**
   - **Escalonador** decide que o processo em execução já teve tempo suficiente e deve deixar outro processo ocupar a CPU
4. **Running → Waiting**
   - S.O. identifica que processo não pode prosseguir (ex: aguardando dados de arquivo)



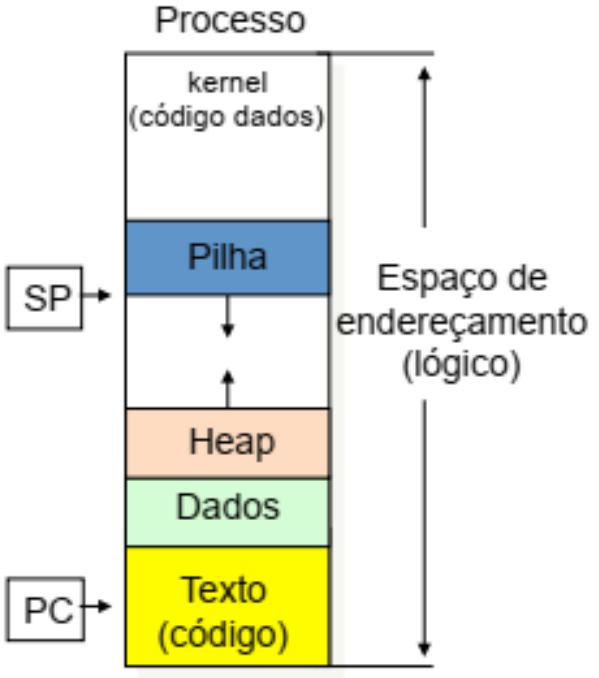
5. **Waiting → Ready**
   - S.O. identifica que o processo pode prosseguir (ex: dados foram recebidos)
6. **Running → Terminated**
   - O processo fez seu trabalho

Processos: condições de término

- **Saída normal** (voluntária)
  - Finalizou seu trabalho normalmente
- **Saída por erro** (voluntária)
  - Processo se depara com erro fatal
- **Erro fatal** (involuntário)
  - Erro causado pelo processo
- **Cancelamento** por um outro processo (involuntário)
  - Um outro processo executou uma chamada do S.O. pedindo para "matar" o processo

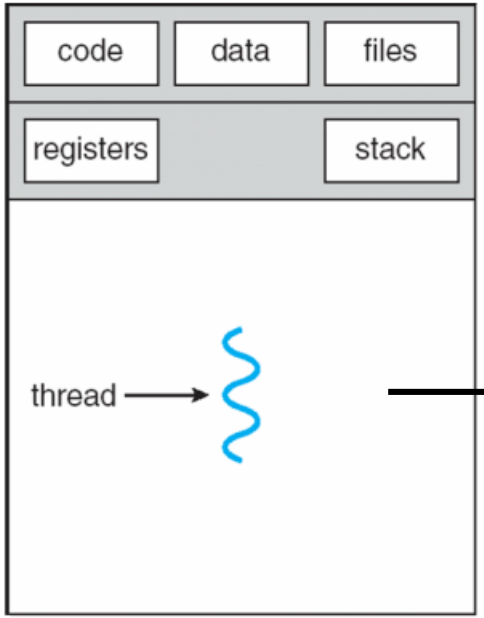
Processo: elementos

- Cada processo possui um **espaço de endereçamento** na memória
  - **Kernel**
    - Estrutura interna do S.O.
  - **Pilha**
    - Armazenar chamadas de função e passagem de parâmetros
    - Variáveis locais
  - **SP: Stack Pointer Register**
    - Endereço de retorno após executar função
  - **Heap**
    - Segmento para alocação dinâmica de memória
  - **Dados**
    - Segmento de variáveis globais
  - **PC: Program Counter Register**
    - Endereço da instrução a ser executada
  - **Texto**
    - Segmento com o código do programa



Thread

- É uma *linha*, ou *fluxo* de execução de instruções
- Todo processo possui ao menos um Thread



single-threaded process

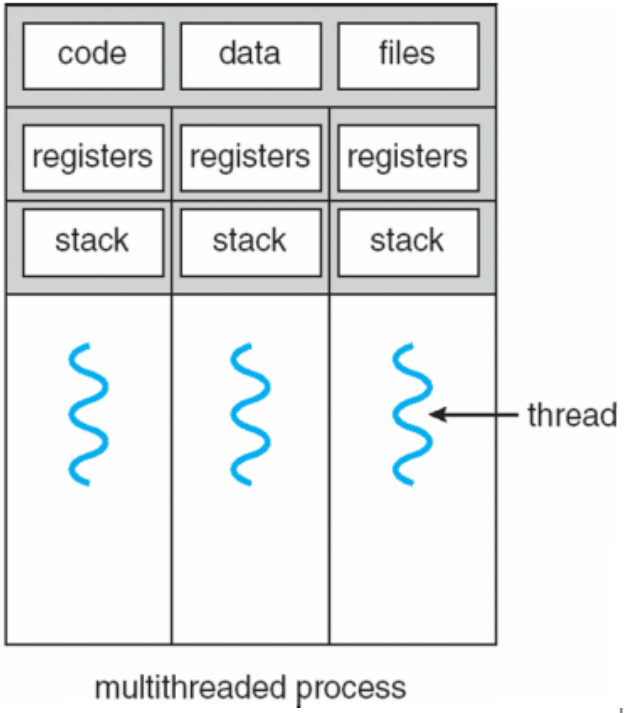
Exemplo de fluxo de execução:

```
int x = scanner.nextInt();
int y = Math.sqrt(x);
System.out.println(y);
```

Threads

- Um processo pode criar vários Threads
- Espaço de endereçamento é **compartilhado**
  - Segmentos de código, dados, heap, etc.

– Cada Thread contém:
  - Registradores SP e PC próprios
  - Stack



Threads: benefícios

- **Aumento de desempenho** das aplicações
  - Principalmente quando há grande volume de computação e E/S*

- Em termos de recursos necessários do Sistema Operacional:
  - É muito mais rápido de criar e destruir Threads do que Processos
    - Eventualmente, criar um Thread é 100 vezes mais rápido
  - Motivo: não é necessário "alocar recursos" (memória)
    - Threads compartilham o espaço de endereçamento.

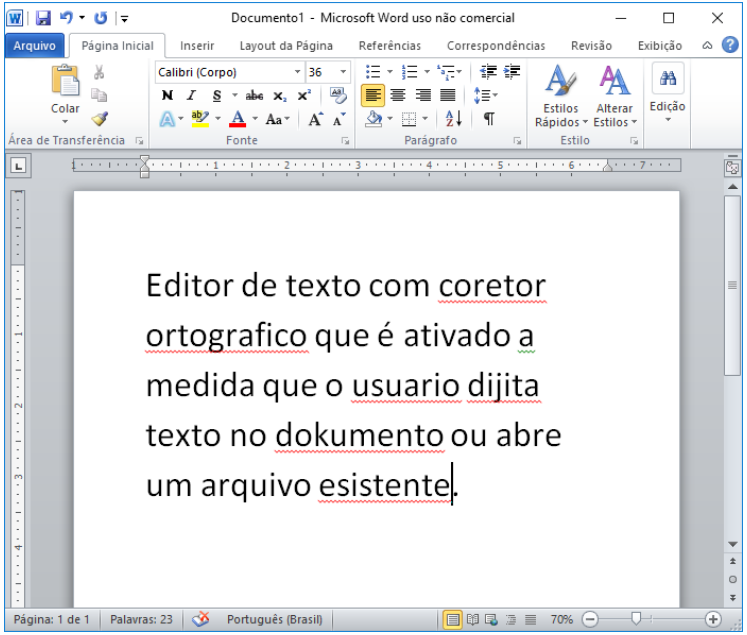
- Paralelismo **real** em equipamentos com múltiplas CPUs

*E/S = operações de entrada/saída (ex: leitura do disco rígido)

Threads: exemplo (1)

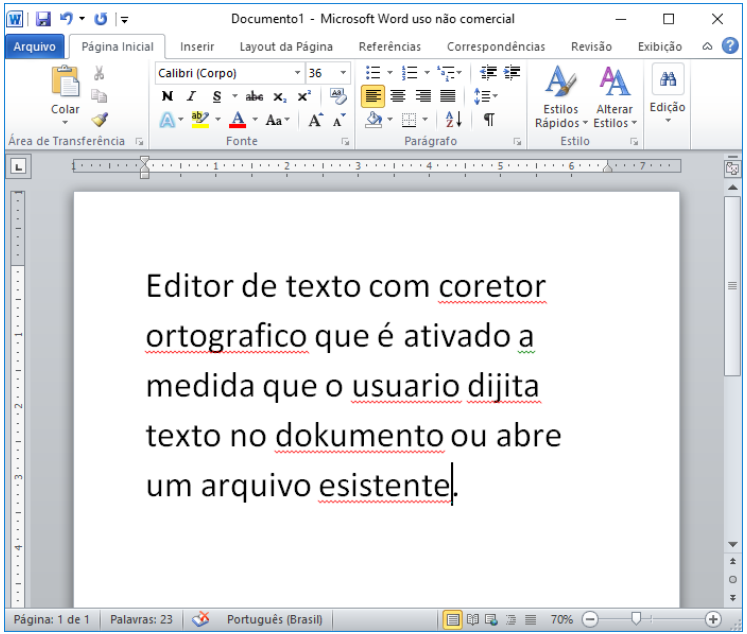
- Editor de texto com corretor ortográfico
  - Verifica se cada palavra digitada existe em um dicionário interno com milhares de palavras

- Versão **monothread**
  - Como fica o usuário enquanto a aplicação verifica a palavra digitada?
  - Como fica o usuário após abrir um arquivo existente?
    - Assumindo que o editor de texto sempre executa o corretor ortográfico em todo o documento ao abrir o arquivo



Threads: exemplo (1)

- Versão **c/ 2 threads**
  - Thread 1: é a aplicação (que interage com o usuário)
  - Thread 2: corretor ortográfico



  - Como fica o usuário enquanto a aplicação verifica a palavra digitada?
  - Como fica o usuário após abrir um arquivo existente?
    - Assumindo que o editor de texto sempre executa o corretor ortográfico em todo o documento ao abrir o arquivo

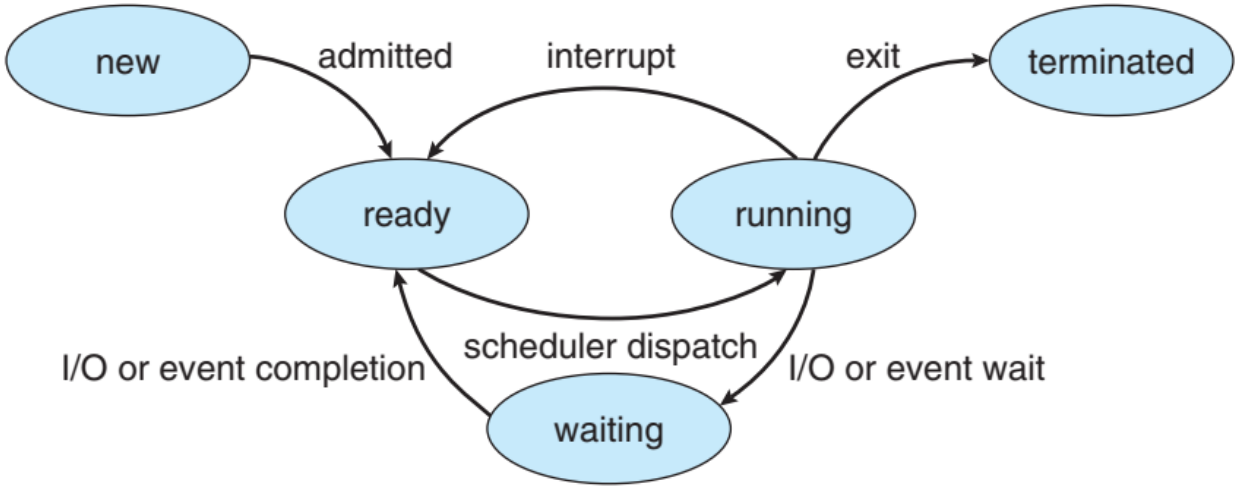
- Novo recurso: salvar o arquivo periodicamente
  - Como a aplicação se comportaria se **monothread** vs. **multithread**?

Threads: exemplo (2)

- Servidor web **monothread** vs. **multithread**

Threads: estados

- São os mesmos estados dos processos



Implementações em Java

Exemplo de Processo (1)

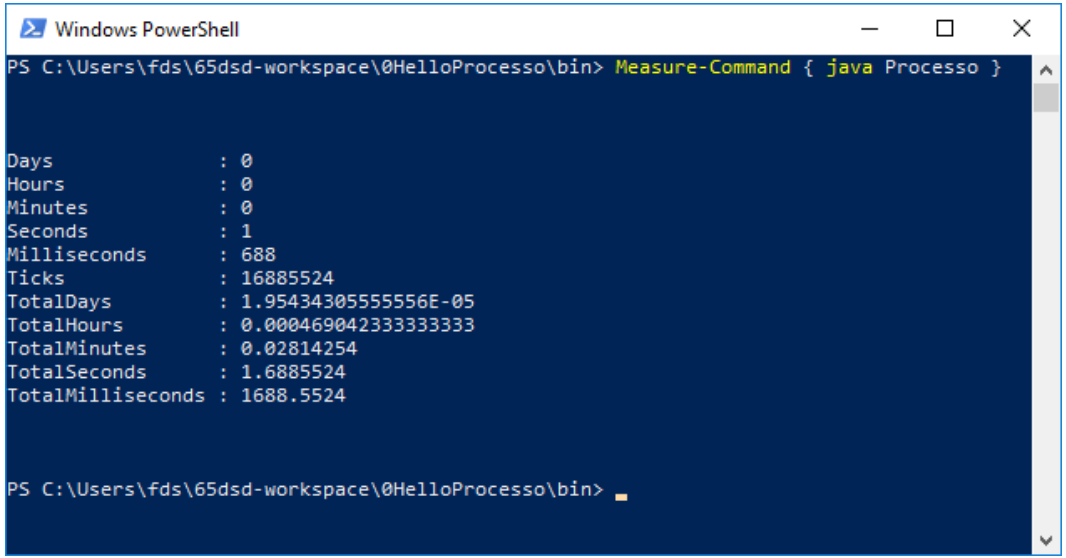
- Qualquer aplicação Java (execução do *main*) é um processo

```
import java.io.IOException;

public class Processo {
   public static void main(String[] args) throws IOException {
      System.out.println("Primeiro Processo.");
      System.out.println("Pressione Enter para sair. ");
      // aguarda usuario pressionar Enter
      System.in.read();
      System.out.println("Terminei");
   }
}
```

Exemplo de Processo (1)

- Como medir o **tempo de execução** de um processo/aplicação?
- Windows PowerShell
  - **`Measure-Command { aplicação }`**
  - **`Measure-Command { aplicação | Out-Default }`**
    - para mostrar saídas



Exemplo de Thread (1)

- Estender a classe *Thread*
- Implementar o método *run()*
  - Operações que o Thread faz

```
public class MeuThread extends Thread {
   @Override
   public void run() {
      System.out.println("Sou o Thread "+this.getId()+". Tchau");
   }
}
```

Exemplo de Thread (1)

- Criação e ativação de Threads

```
public class Processo {
   public static void main(String[] args) {
      System.out.println("Processo que cria Threads");
      MeuThread t1 = new MeuThread();
      t1.start();
      MeuThread t2 = new MeuThread();
      t2.start();
      System.out.println("Processo Finalizado");
   }
}
```

- Atividade
  - Coloque um *breakpoint* no método *run()* de MeuThread
  - Execute em modo depuração e observe quantos Threads existem

Exercício 1

- Desenvolva um programa que, dado um intervalo numérico, imprime todos os números primos existentes. Exemplo:
  - Intervalo: 5 a 25
  - Primos existentes: 5, 7, 11, 13, 17, 19, 23
    - A ordem de impressão é irrelevante

- Versão 1: Apenas o processo (sem Threads)
- Versão 2: Criar um Thread que imprime os primos no intevalo
- Versão 3: Criar 2 Threads: *[inicio ... metade] [metade+1, fim]*
- Versão 4: Criar vários Threads, um para cada sub-intervalo

- Ao executar estas versões, qual é mais rápida?
  - Testar com intervalos grandes. Ex: 2 a 1 milhão
  - Usar Windows PowerShell para medir o tempo de execução

Exercício 2

- Modificar o Exercício 1 (versão com várias Threads) para que o programa informe quantos primos existem no intervalo informado

- Como esperar que um *Thread* finalize?
  - Criação assíncrona vs. criação síncrona de threads (próximo slide)

Criação de Threads

- **Criação Assíncrona**
  - Assim que o thread pai cria um thread filho, o pai retorna sua execução para que ele e seus(s) filho(s) sejam executados concorrentemente
    - É o comportamento do Java

- **Criação Síncrona**
  - Quando o thread pai cria um ou mais filhos, ele deve esperar que todos os filhos terminem para voltar a executar
  - Estratégia de criação conhecida como **fork-join**
    - **fork**: criar de thread
    - **join**: aguardar até que thread seja finalizado (reencontra o pai)
  - Java disponibiliza operação *Thread.join()*

Threads síncronas e join()

- Aguardar finalização de thread em Java

```
public class Processo {

   public static void main(String[] args) throws InterruptedException {
      System.out.println("Processo que cria Threads");
      MinhaThread t1 = new MinhaThread();
      t1.start();
      MinhaThread t2 = new MinhaThread();
      t2.start();

      // aguardar t1 e t2 finalizarem
      t1.join();
      t2.join();

      System.out.println("Processo Finalizado");
   }
}
```

Exercício 2

- Modificar o Exercício 1 (versão com várias Threads) para que o programa informe quantos primos existem no intervalo informado.

Interface Thread

- Alternativa para especificação de Thread
- Maior flexibilidade e polimorfismo

```
public class MeuRunnable implements Runnable {
   private String nome;

   public MeuRunnable(String nome) {
      this.nome = nome;
   }
   public void run() {
      System.out.println("Sou o runnable"
                                 +nome);
   }
}
```

```
public class Processo {

   public static void main(String[] args) {
      System.out.println("Processo que cria Threads");

      MeuRunnable run1 = new MeuRunnable("R1");
      MeuRunnable run2 = new MeuRunnable("R2");

      Thread t1 = new Thread(run1);
      t1.start();

      Thread t2 = new Thread(run2);
      t2.start();

      System.out.println("Processo Finalizado");
   }
}
```

UDESC

Executor e ExecutorService

- Um objeto *Executor* executa *Runnables*
  - Cria e gerencia um grupo de Threads: *pool de theads*

- *ExecutorService* estende *Executor* e disponibiliza métodos de para gerenciamento do ciclo de vida do *Executor*

ExecutorService: Criação

- `ExecutorService executor = Executors.newCachedThreadPool()`
  - Cria um grupo de Threads que adiciona Threads conforme necessário, mas reutiliza Threads existentes sempre que possível

- `ExecutorService executor = Executors.newFixedThreadPool(quantidade);`
  - Cria um grupo com uma quantidade fixa de Threads

- `ExecutorService executor = Executors.newScheduledThreadPool (quantidade);`
  - Cria um grupo de Threads que suporta agendamento

ExecutorService: Uso

- Opera com *Runnables*

```
public class MeuRunnable implements Runnable {
   private String nome;

   public MeuRunnable(String nome) {
      this.nome = nome;
   }
   public void run() {
      System.out.println("Sou o runnable"
                         +nome);
   }
}
```

```
public class Processo {

   public static void main(String[] args) {
      System.out.println("Processo que cria Threads");

      MeuRunnable run1 = new MeuRunnable("R1");
      MeuRunnable run2 = new MeuRunnable("R2");

      ExecutorService executor =
                     Executors.newCachedThreadPool();

      executor.execute(runner1);
      executor.execute(runner2);

      executor.shutdown();

      System.out.println("Processo Finalizado");
   }
}
```

Exercício 3

- Usar *ExecutorService* no Exercício 1 (versão com várias Threads)

ExecutorService: aguardar término

- `Executor.awaitTermination(long time, TimeUnit unit)`
  - Retorna o controle para o chamador quando:
    - Todos os Runnables em execução estão concluídos (retorna true)
    - Expirou o tempo de espera (retorna false)

ExecutorService: aguardar término

- Exemplo de uso:

```
// ... Criação do ExecutorService e adição dos Runnables...

executor.shutdown();

boolean terminou;
try {
   terminou = executor.awaitTermination(10, TimeUnit.SECONDS);
   if (terminou) {
      System.out.println("Todos os threads terminaram.");
   }else {
      System.out.println("Ainda ha threads trabalhando. Aguardando...");
   }
} catch (InterruptedException e) {
   System.out.println("A thread principal foi interrompida");
   e.printStackTrace();
}
```

Exercício 4

- Usar ExecutorService no Exercício 2 (quantidade de primos)