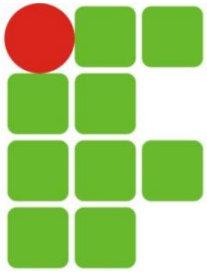


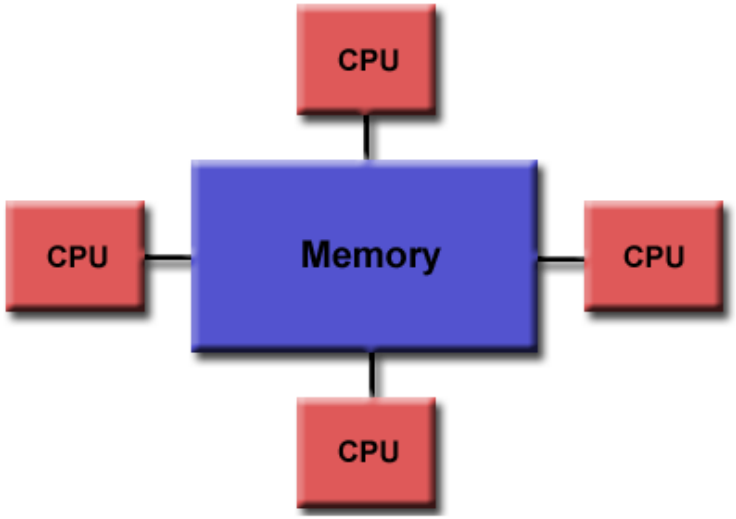
Instituto Federal de Minas Gerais
Campus Ouro Branco

Computação Paralela

Professor: Saulo Henrique Cabral Silva

Computação Paralela

- Introdução
- Motivação
- Desafios



- Programação paralela - Memória Compartilhada
- Processos - características
- Threads

Introdução

- Programação paralela é a **divisão de uma determinada aplicação em partes**, de maneira que essas partes possam ser executadas **simultaneamente**, por vários elementos de processamento.

- Os elementos de processamento devem **cooperar entre si** utilizando primitivas de **comunicação e sincronização**, realizando a quebra do paradigma de execução sequencial do fluxo de instruções.

- Objetivos
  - Alto **Desempenho** (Exploração Eficiente de Recursos)
  - Tolerância a falhas

Motivação

- Solução de aplicações **complexas** (científicas, industriais e militares)
  - Meteorologia
  - Prospecção de petróleo
    - Análise de local para perfuração de poções de petróleo
  - Simulação físicas
    - Aerodinâmica, energia nuclear
  - Matemática computacional
    - Análise de algoritmos para criptografia
  - Bioinformática
    - Simulação computacional da dinâmica molecular de proteínas
  - Games
  - Mineração de dados

Desafios

- SpeedUp
  - Fator de aceleração

$$SpeedUp = \frac{tempoSequencial}{tempoParalelo}$$

  - Existe um limite para o número de processadores
  - **Amdahl's Law**
    - Determina o **potencial** **de aumento de velocidade** a partir da porcentagem paralelizável. Considera um programa como uma **mistura de partes sequenciais e paralelas**.

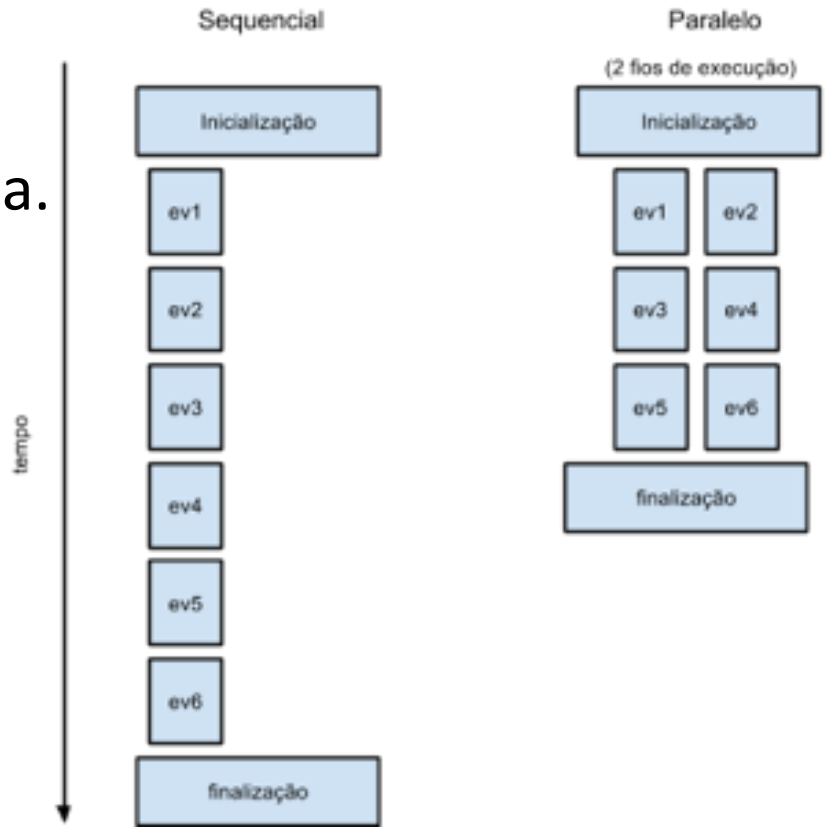
$$\mathbf{SpeedUp} = \frac{\mathbf{1}}{\frac{\mathbf{paralelo}}{\mathbf{nProcessadores}} + \mathbf{sequencial}}$$

Desafios

- **Custo de coordenação** (sincronização)
  - Necessidade de **troca de informação** entre threads
- **Divisão** adequada da computação entre os recursos
  - *Decomposição* do problema
- Complexidade de **implementação**
  - Particionamento de código e dados
  - Problema com **sincronismo**
  - **Dependência** de operações
  - **Balanceamento** de carga
  - *Deadlocks* (comunicação)

Desafios

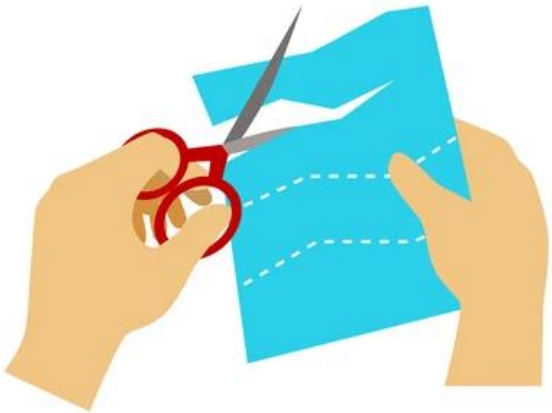
- Necessidade de conhecimento da máquina
  - Código dedicado a **máquina paralela**
  - Baixa **portabilidade**
  - Influencia
    - Paradigma utilizado para comunicação
    - Modelagem do problema
- Dificuldade na **conversão** da aplicação sequencial em paralela.
  - Algumas aplicações não são paralelizáveis!
- Dificuldade de **depuração**



Modelagem

- Podemos dividir basicamente em:
  - Modelos de **máquina**: descrevem as características das **máquinas**
  - Modelos de **programação**: permitem compreender aspectos ligados a **implementação e desempenho** de execução dos programas
  - Modelos de **aplicação**: representam o paralelismo de um **algoritmo**

- Vantagens
  - Permitem compreender o **impacto de diferentes aspectos** da aplicação na implementação de um programa paralelo, tais como:
    - Quantidade de **cálculo envolvido** total e por atividade concorrente
    - **Volume de dados** manipulado
    - **Dependência de informações** entre as atividades e execução



Paralelismo de dados x paralelismo de tarefa

- Identifica a **concorrência** da aplicação:
  - Paralelismo de **dados**
    - Execução de uma **mesma atividade** sobre diferentes **partes de um conjunto de dados**
    - **Os dados determinam a concorrência** da aplicação e a forma como o cálculo deve ser distribuído na arquitetura
  - Paralelismo de **tarefa**
    - Execução paralela de **diferentes atividades sobre conjuntos distintos de dados**
    - Identificação das **atividades concorrentes** da aplicação e como essas atividades são distribuídas pelos recursos disponíveis

Memória compartilhada x troca de mensagens

- Identifica como é realizado o **compartilhamento de informações durante a execução**.

- Ligado diretamente ao tipo de arquitetura utilizado.

- Modelos:
  - **Memória Compartilhada**
    - As tarefas em execução compartilham um mesmo espaço de memória
    - Comunicação através do acesso a uma área compartilhada
  - **Troca de mensagens**
    - Não existe um espaço de endereçamento comum
    - Comunicação através de troca de mensagens usando a rede de interconexão.

Modelagem como um grafo

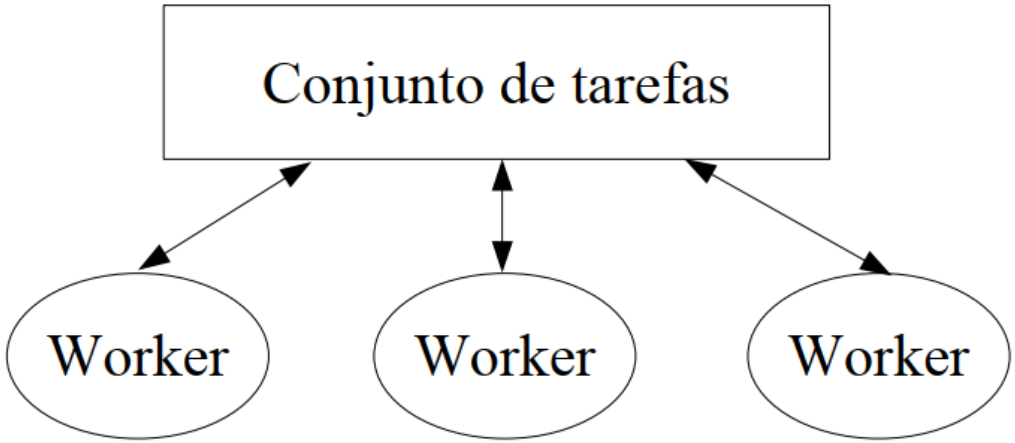
- As aplicações são modeladas usando um grafo que relaciona as tarefas e trocas de dados.
  - **Nós**: tarefas
  - **Arestas**: trocas de dados (Comunicações e/ou sincronizações)

- Modelos básicos
  - *Workpool*, mestre/escravo, *pipeline*, divisão e conquista e fases paralelas.

MODELOS

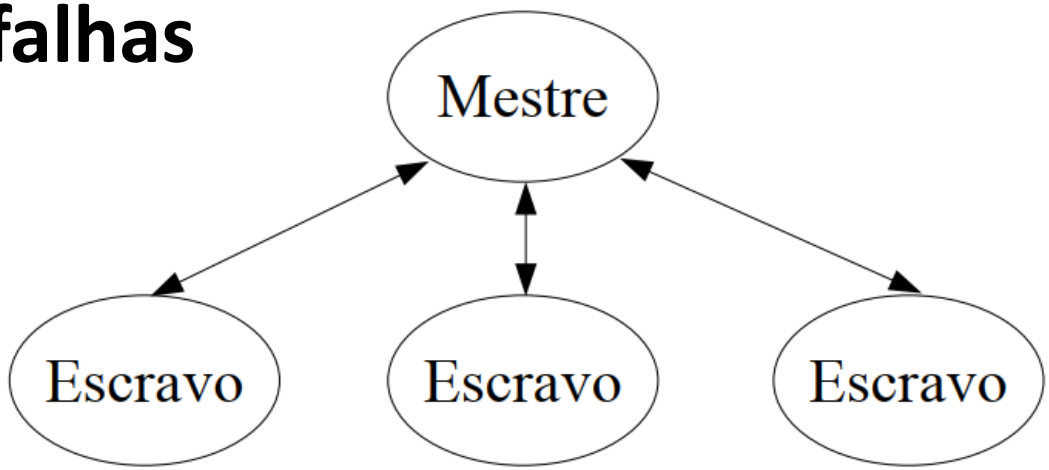
Workpool

- Tarefas disponibilizadas em uma estrutura de dados global (Memória compartilhada)
- Sincronização
- Balanceamento de carga.



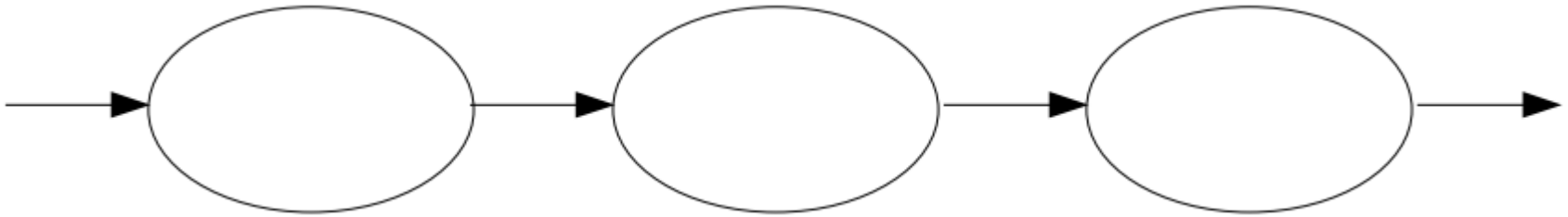
Mestre/Escravo (Task farming)

- Mestre escalona tarefas entre processos escravos
- Escalonamento **centralizado** – gargalo
- Maior **tolerância a falhas**



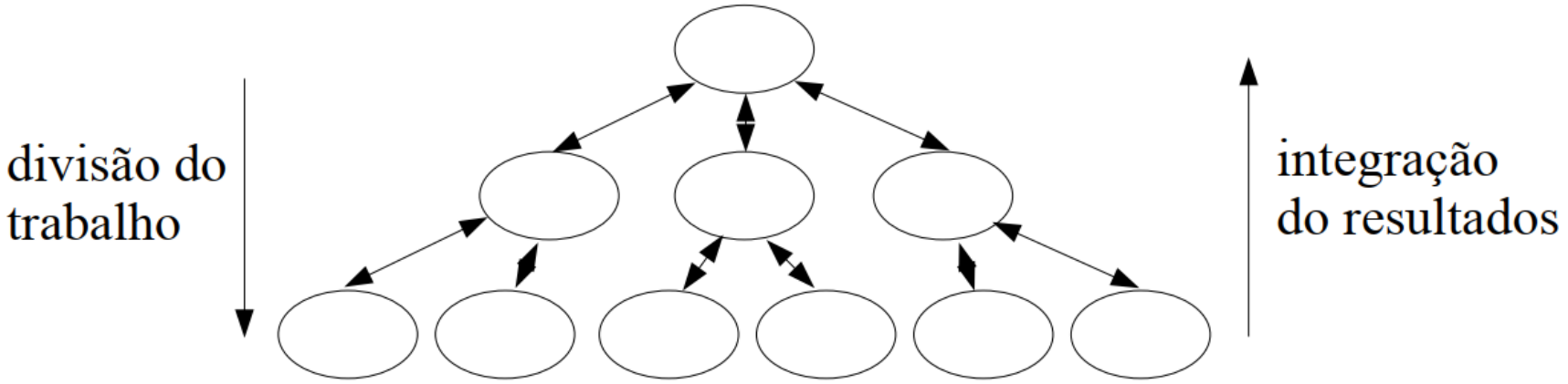
Pipeline

- *Pipeline* virtual
- Fluxo **contínuo de dados**
- Sobreposição de comunicação e computação



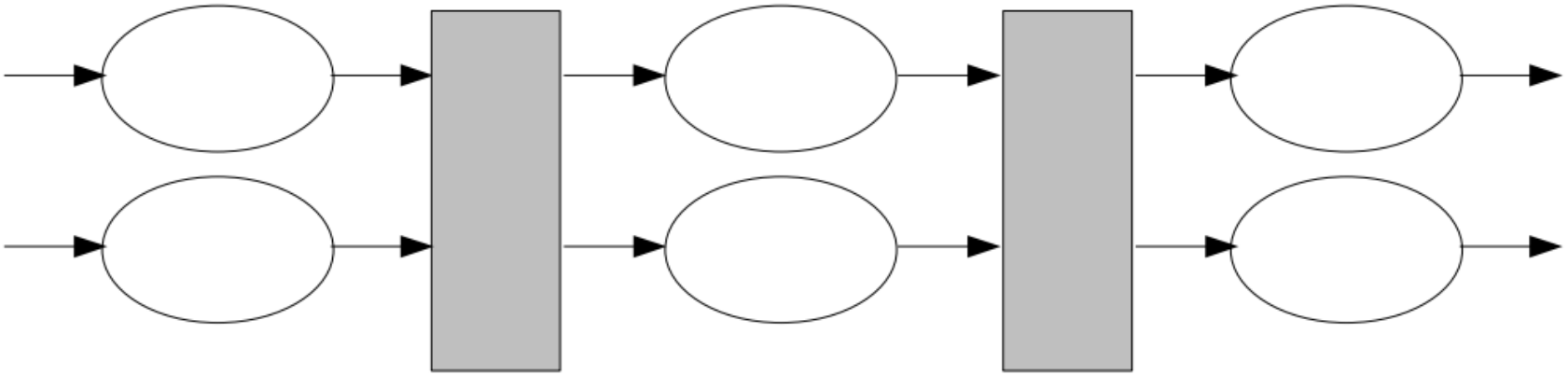
Divisão e conquista (Divide and Conquer)

- Processos organizados em uma hierarquia (pai e filhos)
- Processo **pai divide trabalho** e repassa uma **fração deste aos seus filhos**
- Integração dos resultados de forma recursiva
- Dificuldade de **balanceamento de carga** na divisão das tarefas



Fases paralelas

- Etapas de **computação** e **sincronização**
- Problema de balanceamento de carga
  - Processos que acabam antes???
- Overhead de comunicação
  - Comunicação é realizada **ao mesmo tempo**



PROGRAMAÇÃO PARALELA – MEMÓRIA COMPARTILHADA

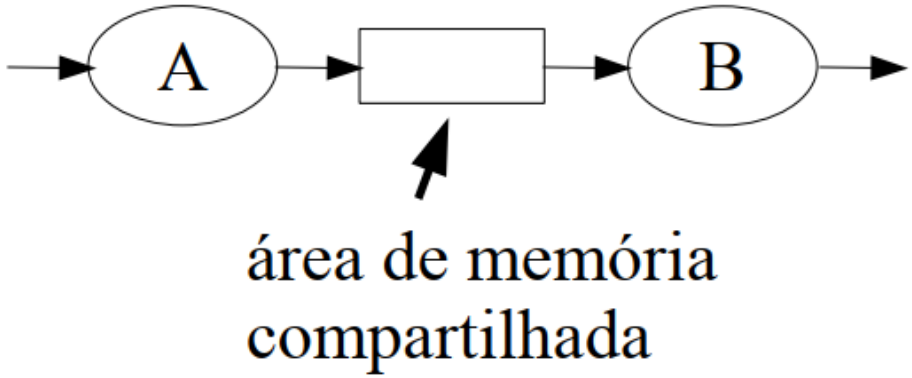
Programação Paralela Memória Compartilhada

- A comunicação entre as threads é realizada através de acessos do tipo *load* e *store* a uma área de **endereçamento comum**.

- Para utilização correta da área de memória compartilhada é necessário que as threads **coordenem seus acessos** utilizando primitivas de sincronização.

Programação Paralela Memória Compartilhada

- Execução sequencial de tarefas
  - O resultado de uma tarefa é comunicado a outra tarefa através da **escrita** em uma posição de memória compartilhada.
  - **Sincronização implícita**, isto é, uma tarefa só é executada **após o término** da tarefa que a precede.



Programação Paralela Memória Compartilhada

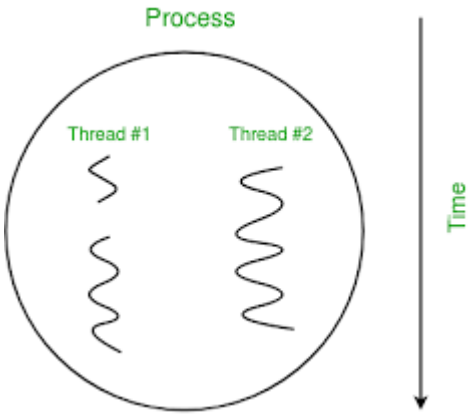
- Execução concorrente de tarefas
  - Não existe sincronismo implícito
  - **Seção crítica**: conjunto de instruções de acesso a uma área de memória compartilhada acessada por diversos fluxos de execução
  - É **responsabilidade do programador** de fazer uso dos mecanismos de **sincronização** para garantir a correta utilização da área compartilhada para comunicação entre os fluxos de execução distintos.
  - Mecanismos mais utilizados para exclusão mútua no acesso a memória: *mutex*, operações de criação e bloqueia dos fluxos de execução (*create* e *join*)

Multiprogramação leve

- **Multithreading**: permite a criação de vários fluxos de execução (threads) no interior de um processo.

- **Thread**: também chamado de processo leve, em referência ao fato de que os recurso de processamento alocados a um processo são compartilhados por todas suas threads ativas

- As threads **compartilham dados** e se "comunicam" através **da memória alocada** ao processo.



IMPLEMENTAÇÃO DE THREADS

Java Threads

- Toda aplicação Java **tem pelo menos** uma thread (sem considerar o system thread)

- Do ponto de vista do desenvolvedor, o programa começa com uma thread, chamada **main thread**.

- A partir da **main thread** *(“mãe”/principal)* é possível criar **novas** threads.

Java Threads

- Cada thread é **associada com uma instância** da classe Thread.

- Duas estratégias possíveis para criar uma thread:
  - Instanciando a classe Thread;
  - Delegando criação/gerência da thread para "executor" (*high-level concurrency objects*)

Principais métodos da Classe Thread

- **run( )**: é o método que executa as atividades de uma thread. Quando este método finaliza, a thread também termina.

- **start( )**: método que dispara a execução de uma thread. Este método chama o método run() antes de terminar.

- **sleep(int x)**: método que coloca a thread passa dormir por x milissegundos.

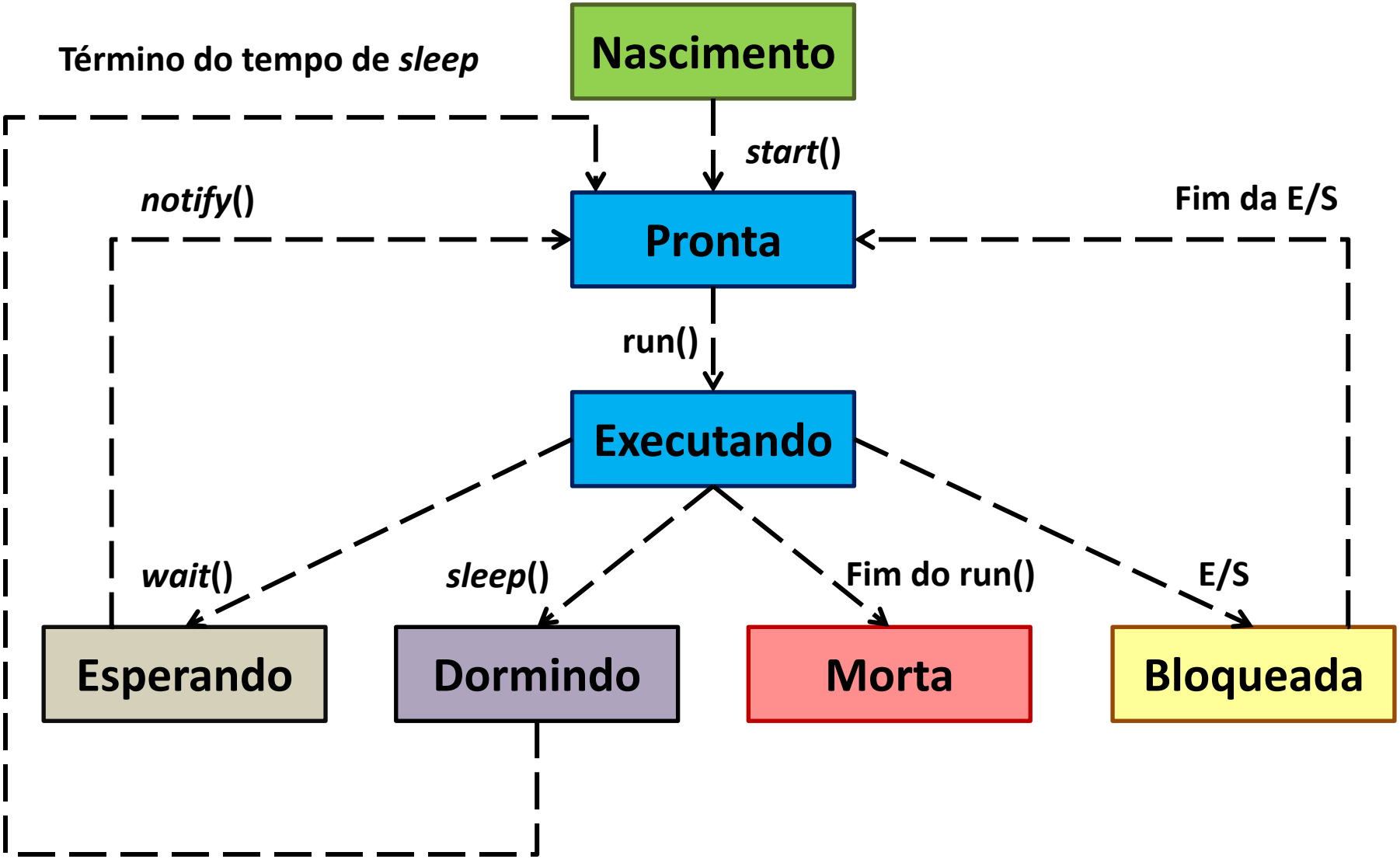
Principais métodos da Classe Thread

- **join( )**: método que espera o término da thread para qual foi enviada a mensagem para ser liberada.

- **interrupt( )**: método que interrompe a execução de uma thread.

- **interrupted( )**: método que testa se uma thread está ou não interrompida.

Estados da thread



Sincronização de Threads em Java

- O controle de execução de sessões críticas em Java é garantido por um mecanismo de monitores

- **Sessão de código (RC)** é executada por apenas uma thread em um determinado instante de tempo

- Thread que deseja executar a RC, pede permissão para execução;
  - **Se tiver permissão**, executa região bloqueando entrada para outras threads
  - **Senão tiver permissão** (outra thread está executando), bloqueia até liberar

Sincronização de Threads em Java

- Todos os objetos Java são potencialmente monitores
- Objetos e classes possuem um *lock* intrínseco
- Para fazer chamada a algum método de um objeto compartilhado (monitor), deve-se **obter o *lock***
- O mesmo para métodos de classe (static)
- A palavra *synchronized* é **usada para se obter o *lock*** de um objeto monitor (ou classe) em java

Sincronização de classe

- Garante que **a penas uma thread** vai executar um (*synchronized*) método no escopo de uma classe (*static method*)

```java
package testlanguage;

import java.util.ArrayDeque;
import java.util.Deque;

public class MyBuffer {

    private static Deque<Object> pilha;

    public MyBuffer(){
        pilha = new ArrayDeque<Object>();
    }

    synchronized static public void insere( Object _o ) {
        pilha.push(_o );
    }

    synchronized static public Object retira() {
        return pilha.pop();
    }
}
```

Sincronização de Bloco

- Garante que apenas uma thread vai executar o conjunto de instruções definidas dentro do bloco.

- É usado um objeto auxiliar (ex. this) para garantir **exclusão mútua**

```java
public class MyBuffer2 {
    private static Deque<Object> pilha;

    public MyBuffer2(){
        pilha = new ArrayDeque<Object>();
    }

    public void insere( Object _o ) {
        synchronized (this) {
            pilha.push(_o );
        }
    }

    synchronized public Object retira() {
        Object aux;
        synchronized (this) {
            aux = pilha.pop();
        }
        return aux;
    }
}
```

Sincronização de Bloco (métodos independentes)

- São usados **objetos auxiliares distintos** para evitar exclusão mútua (*interleave*) entre métodos.

```java
public class Exemplo3 {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized (lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized (lock2) {
            c2++;
        }
    }
}
```

Sincronização Reentrante

- Uma thread **não pode obter um *lock* que está com outra thread**
- Mas uma thread pode obter um *lock* que ela já possui
- Permite um **método (*synchronized*) chamar** outro método também *synchronized*
- Evita que uma thread **cause o bloqueio de si mesma**

Guarded Blocks

- Usados para coordenar as ações das threads
- Um "*guarded block*" começa perguntando (*polling*) se uma condição é verdadeira

```java
public class Exemplo4 {
    // controle é uma variável compartilhada
    private static boolean controle = true;

    public synchronized void metodoControlado() {
        while (!controle) {
            try {
                wait();
                /* bloqueia a thread até algum evento ser
                notificado. Note que pode ser qualquer outro evento*/
            } catch (InterruptedException e) {
            }
        }
        controle = false;
        System.out.println("Foi ativado com sucesso!");
    }
}
```

Guarded Blocks

- Sempre use o "*wait*" dentro de um loop que testa se a condição esperada foi satisfeita (Abordagem de Hansen).
  - Não assuma que a interrupção foi para a condição específica que estávamos esperando

- Mesmo que o evento tenha sido o que estávamos esperando, **pode ser que a condição não seja mais a mesma** quando a thread ganhar permissão de executar

Wait

- Quando uma thread executa o *wait*, ela **precisa ter o *lock* intrínseco do objeto**
- Executar o *wait* dentro de um método *synchronized* é uma maneira simples de obter o *lock*
- Quando uma thread chama o *wait*, **a thread libera o *lock* e suspende a execução**

NotifyAll

- Uma outra thread que obtenha o *lock* vai chamar *notifyAll*, que **informa** a todas as threads que estejam **esperando por aquele *lock* que algo importante aconteceu**

```
public synchronized void notifyMetodoControlado(){
    controle = true;
    notifyAll();
}
```