SEMANA ACADÊMICA UNIFICADA DE

# ENGENHARIA DE SOFTWARE & SISTEMAS DE INFORMAÇÃO

# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

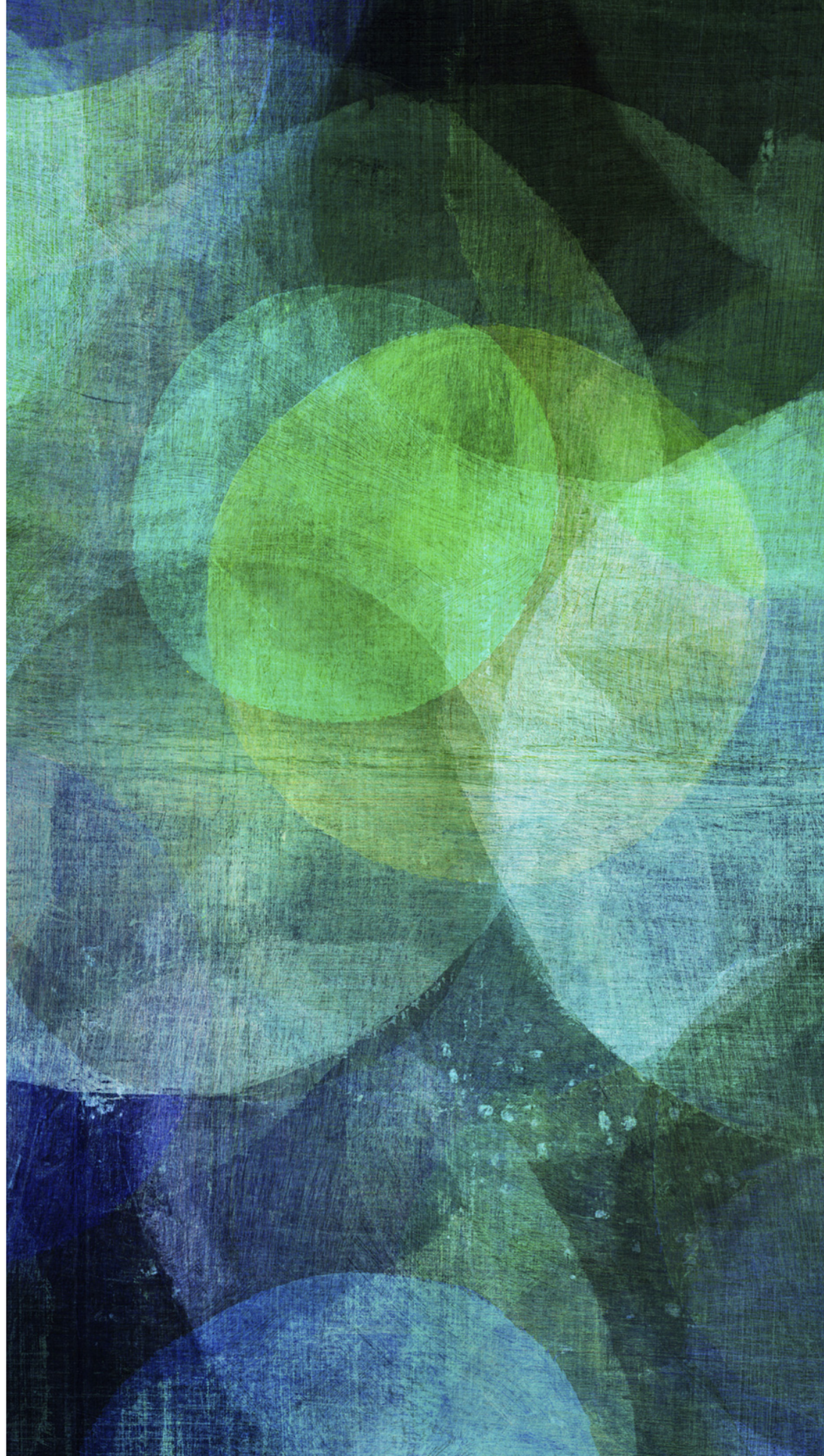
*Por Prof. DSc Bárbara Quintela*
*barbara@ice.ufjf.br*

*Parte 2 - 26/10/2018*

# AGENDA

- OpenMP
  - Escopo dos dados
  - Pragma for
  - Redução
    - Sincronização
- Java threads
  - Exemplo

# CONTINUA...



# COMO ESCREVER PROGRAMAS EM PARALELO

- Paralelismo de tarefas
  - Cada processador realiza uma tarefa distinta
- Paralelismo de dados
  - Divide os dados entre os processadores
    - Cada processador realiza operação similar sobre dados distintos

# FORMAS DE PARALELIZAR

- Restrições
  - Comunicação
  - Balanceamento de carga
  - Sincronização

# OPENMP – EXEMPLO SOMA DE N VALORES

- Solução sequencial de soma de n valores

```c
#include <stdio.h>
#define TAM 10
int main(){
    int vet[TAM] = {1,2,3,4,5,6,7,8,9,10};
    int i, soma = 0;
    for(i=0;i<TAM;i++){
        soma += vet[i];
    }
    return 0;
}
```

# OPENMP – EXEMPLO SOMA DE N VALORES

- Solução sequencial de soma de n valores

```c
#include <stdio.h>
#define TAM 10
int main(){
    int vet[TAM] = {1,2,3,4,5,6,7,8,9,10};
    int i, soma = 0;
    for(i=0;i<TAM;i++){
        soma += vet[i];
    }
    return 0;
}
```

- Se tem p processadores disponíveis e p<n
- Cada processador pode executar uma soma parcial de n/p valores

# OPENMP – EXEMPLO SOMA DE N VALORES

- Solução paralela de soma de n valores
  - O que é mais vantajoso?
    - Deixar a thread mestre concentrar a soma
    - Dividir trabalho entre threads pares e impares
    - etc.

# OPENMP – ESCOPO DOS DADOS

- Para cada região paralela
  - Ambiente de dados construído de acordo com cláusulas:
    - **Shared** - variável é comum entre as threads
    - **Private** - variável é nova
    - **First private** - variável é nova mas inicializada com valor inicial
    - **Default** - usada para definir valores padrão
    - **Last private** - ultimo valor da variável é copiado
    - **Reduction** - valor da variável é reduzido ao final

# OPENMP – ESCOPO DOS DADOS

- Exemplos

```c
#include <stdio.h>

int main(){
    int x= 1;
    #pragma omp parallel shared(x) num_threads(2)
    {
        x++;
        printf("%d\n", x);
    }
    printf("%d\n", x);
    return 0;
}
```

x = 2
x = 3
x = 3

*ou*

x = 3
x = 2
x = 3

# OPENMP – ESCOPO DOS DADOS

- Exemplos

```c
#include <stdio.h>

int main(){
    int x= 1;
    #pragma omp parallel shared(x) num_threads(2)
    {
        x++;
        printf("%d\n", x);
    }
    printf("%d\n", x);
    return 0;
}
```

```
x = 2
x = 3
x = 3
```

*ou*

```
x = 3
x = 2
x = 3
```

```c
#include <stdio.h>

int main(){
    int x= 1;
    #pragma omp parallel private(x) num_threads(2)
    {
        x++;
        printf("%d\n", x);
    }
    printf("%d\n", x);
    return 0;
}
```

Imprime qualquer coisa e depois
x = 1

# OPENMP – SINCRONIZAÇÃO

- Mecanismos
  - **Barrier** - sincroniza todas as threads no time
  - **Master** - somente thread mestre executa o bloco
  - **Critical** - somente uma thread de cada vez executa
  - **Atomic** - mesmo que critical mas para um local da memória (atualiza local da memória uma thread de cada vez)

# OPENMP – SINCRONIZAÇÃO

```c
#include <stdio.h>
#include <omp.h>
void foo(int x){
    printf("Thread: %d, x: %d\n", omp_get_thread_num(), x);
}
int main(){
    int x = 1;
    #pragma omp parallel num_threads(2)
    {
        #pragma omp master
        {
            x++;
        }
        foo(x);
    }
    return 0;
}
```

*foo(2),foo(2)*

*Ou*

*foo(2),foo(2)*

```c
int main(){
    int x = 1;
    #pragma omp parallel num_threads(2)
    {
        #pragma omp critical
        {
            x++;
            foo(x);
        }
    }
    return 0;
}
```

*foo(2),foo(3)*

```c
int main(){
    int x = 1;
    #pragma omp parallel num_threads(2)
    {
        x++;
        #pragma omp barrier
        foo(x);
    }
    return 0;
}
```

*foo(3),foo(3)*

# OPENMP – PRAGMA OMP FOR

```
#pragma omp for [clausulas]
  for(iexp ; test ; incr)
```

- Cláusulas podem ser `private, firstprivate, lastprivate, reduction, schedule, nowait`.
- Iterações do loop devem ser independentes
- O compartilhamento de dados padrão é shared

# OPENMP – PRAGMA OMP FOR

- Exemplo

```
 5    int main(){
 6        int i,j;
 7        int m[N][N];
 8        #pragma omp parallel num_threads(4)
 9        #pragma omp for private(j)
10        for(i=0 ; i < N ; i++){
11            for(j=0; j < N ; j++){
12                m[i][j] = i+2*j;
13            }
14        }
15        imprimeMat(m,N);
16    }
```

j deve ser declarado privado explicitamente

i automaticamente privado

Ponto de sincronização implícito no final do laço

# OPENMP – REDUCTION

- **Redução** - Combinação de variáveis locais de uma thread em uma variável única
  - É uma operação binária (como adição ou subtração)
  - Aplica mesmo operador a uma sequência de operandos de forma a obter um resultado único
  - Todos os resultados intermediários devem ser armazenados na mesma variável

# OPENMP – REDUCTION

- Restrições dos atributos:
  - Variáveis devem ser escalar
  - Cópia privada de cada variável é criada e iniciada dependendo da operação
  - devem ser declaradas como shared
  - Cópias são atualizadas pelas threads
  - Variável na operação de redução deve ser utilizada dentro de região paralela

# OPENMP – REDUCTION

- Exemplo:

```c
#include <stdio.h>
#include <omp.h>
#define N 10
int main () {
    int i, n, parte;
    float a[N], b[N], res; /* Some initializations */
    n = N;
    parte = 2;
    res = 0.0;
    for (i=0; i < n; i++)
    {
        a[i] = i * 1.0; b[i] = i * 2.0;
    }
    #pragma omp parallel for
    #pragma omp default(shared) private(i) schedule(static,parte) reduction(+:res)
    for (i=0; i < n; i++){
        res = res + (a[i] * b[i]);
        printf("Thread: %d \n",omp_get_thread_num());
    }

    printf("Resultado Final = %f\n",res);

    return 0;
}
```

# OPENMP – SCHEDULE

- Usada para determinar distribuição de trabalho entre as threads
- **Schedule (static, parte)**
  - Laço dividido igualmente em tamanhos definidos por "parte" que são distribuídos no estilo *round robin*
- **Schedule (dynamic, parte)**
  - Laço dividido igualmente em tamanhos definidos por "parte" que são distribuídos dinamicamente
- Se não especificar "parte" o padrão é parte ou *chunk* = 1

# OPENMP – SINCRONIZAÇÃO

- Podemos calcular chamadas recursivas do cálculo dos termos de Fibonacci em paralelo

```
3   int fib(int n) {
4       if (n < 2) return n;
5       else {
6           int x, y;
7       #pragma omp task shared(x)
8       {     x = fib(n-1); }
9       #pragma omp task shared(y)
10      {     y = fib(n-2); }
11      #pragma omp taskwait
12          return(x + y);
13      }
14  }
```

- Há necessidade de sincronizar para garantir que x e y foram calculados antes de retornar a soma

# OPENMP – SINCRONIZAÇÃO

- Diretiva **task**
  - especifica quais blocos podem ser executados de forma assíncrona pelas threads

- Diretiva **taskwait**
  - Sincroniza tarefa com tarefas filhas antes de continuar

# OPENMP – SINCRONIZAÇÃO

- As diretivas são usadas de forma integrada com as outras do OpenMP
- Espera-se uma diretiva **`parallel`** para disparar as threads

```
16  int main(){
17      int num, result;
18      scanf("%d", &num);
19      #pragma omp parallel
20      {
21          #pragma omp single nowait
22          {
23              result = fib(num);
24          }
25      }
26      printf("Resultado: %d \n", result);
27      return 0;
28  }
```

# OPENMP – SECTIONS

- Cada **section** é executada por uma thread do grupo

- Há um ponto de **sincronização** implícito no final
  - A não ser que use **`nowait`**

- Se tiver mais threads do que seções o OpenMP decide quais executam os blocos

# OPENMP – SECTIONS

- Exemplo

```c
#include <stdio.h>

#define N 10000

int main () {
    int i, n;
    float a[N], b[N], c[N];

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    n = N;
```

```c
    #pragma omp parallel shared(a,b,c,n) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < n/2; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=n/2; i < n; i++)
                c[i] = a[i] + b[i];
        }  /* fim das sections */
    }  /* fim das sections paralelas */
```

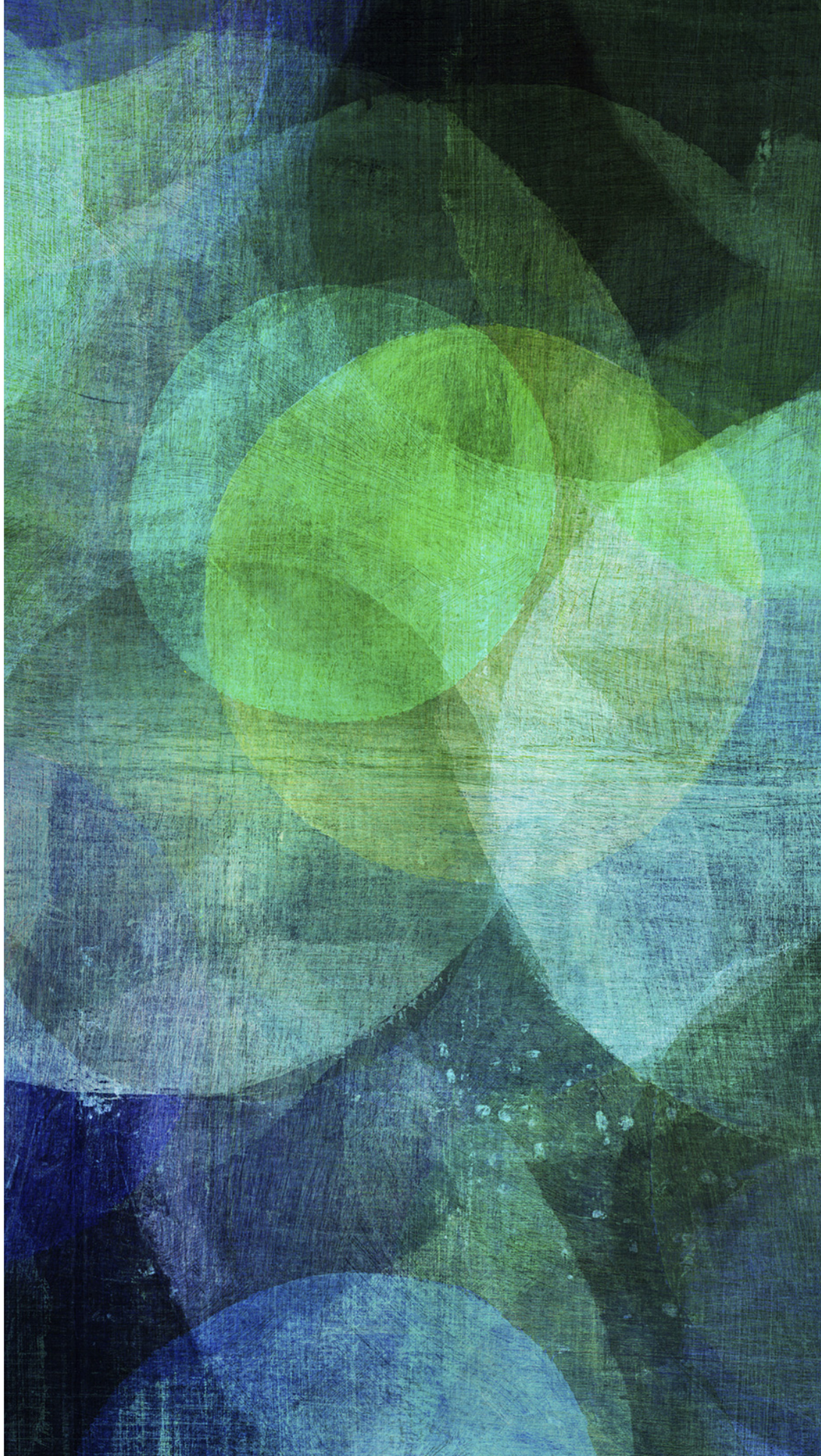
# OPENMP – REFERÊNCIAS

- https://computing.llnl.gov/tutorials/openMP/

- https://www.ibm.com/developerworks/br/aix/library/au-aix-openmp-framework/index.html

- https://www.revista-programar.info/artigos/paralelizacao-de-aplicacoes-com-openmp/

# JAVA THREADS



# JAVA THREAD INTRODUÇÃO

- Quando uma aplicação Java é executada
  - JVM cria um objeto do tipo thread
  - Tarefa da thread é executa o que está no método main()
  - Thread iniciada automaticamente
  - Comandos executados sequencialmente

# JAVA THREAD INTRODUÇÃO

- Pode-se criar mais threads
  - Extendendo a classe Thread
    - Instanciando objeto dessa classe
  - Implementando interface Runnable
    - Passando objeto da nova classe como argumento do construtor da classe Thread

- Nos dois casos a tarefa da thread deve ser descrita pelo

# CÓDIGOS DE EXEMPLO DE THREADS EM JAVA NO GITHUB

- SimpleThread.java
- ThreadExemplo.java

Compile : **`javac ThreadExemplo.java`**

Execute : **`java ThreadExemplo`**

# JAVA THREAD

- Como Java não permite herança múltipla estender a classe Thread restringe criação de subclasses
- Através da interface Runnable é possível criar classes que representem uma thread sem precisar extender
  - Cria thread através de instanciação de objeto thread usando objeto que implementa interface Runnable

http://www.inf.puc-rio.br/~inf1621/java2.pdf

# CÓDIGOS DE EXEMPLO DE THREADS EM JAVA NO GITHUB

- SimpleThread2.java
- Teste.java

Compile : **`javac Teste.java`**

Execute : **`java Teste`**

# JAVA THREAD – METHODS

- **`start()`**
  - Inicia execução da thread - só pode ser chamado uma vez
- **`yield()`**
  - Suspende execução da thread corrente e escalona outro
- **`sleep(t)`**
  - Faz com que a thread fique suspensa por t segundos
- **`wait()`**
  - Faz com que a thread fique suspensa até que seja explicitamente reativada por outra thread

# JAVA THREADS – PRIORIDADES

- Java permite atribuição de prioridades
- Threads com menor prioridade são escalonadas com menor frequência
  - Round robin

```
setPriority(int p)

getPriority()
```

# JAVA THREADS SINCRONIZAÇÃO

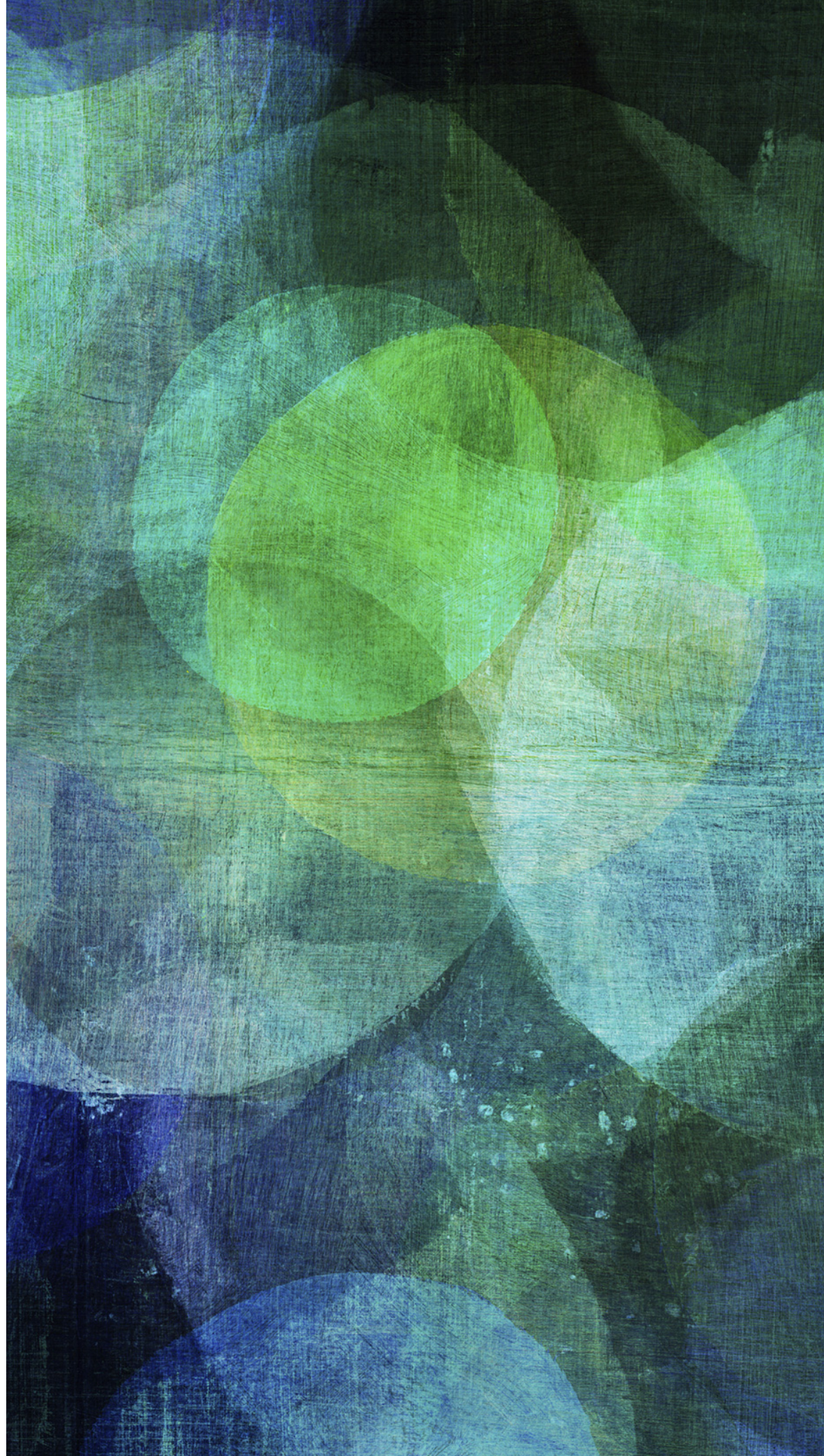
- Cada thread possui uma pilha distinta
  - Versões diferentes de variáveis locais

- Memória dinâmica (heap) de um programa é compartilhada
  - Duas threads podem acessar os mesmos atributos de um objeto de forma concorrente
    - Restringe com método **`synchronized`**

# JAVA THREADS SINCRONIZAÇÃO

- Mecanismo de sincronização de Java
  - Baseado em monitores
  - Operações **`wait()`** e **`nofifyAll()`**
    - So podem ser usados dentro de métodos **`synchronized`**

# EXTRA



# ANDROID THREADS

- Tutorial de app com thread
  - https://www.101apps.co.za/articles/using-threads-tutorial.html
- https://developer.android.com/topic/performance/threads

# REFERÊNCIAS OPENMP – NO ANDROID

- https://www.softwarecoven.com/parallel-computing-with-openmp-in-android/

- https://www.hindawi.com/journals/misy/2016/4513486/

SEMANA ACADÊMICA UNIFICADA DE

# ENGENHARIA DE SOFTWARE & SISTEMAS DE INFORMAÇÃO

# INTRODUÇÃO A PROGRAMAÇÃO PARALELA

*Por Prof. DSc Bárbara Quintela*
*barbara@ice.ufjf.br*

*Parte 2 - 26/10/2018*