

# Programação concorrente em Sistemas de Memória compartilhada

Prof: Álvaro L. Fazenda  
([alvaro.fazenda@unifesp.br](mailto:alvaro.fazenda@unifesp.br))

# Concorrência em nível de Processos

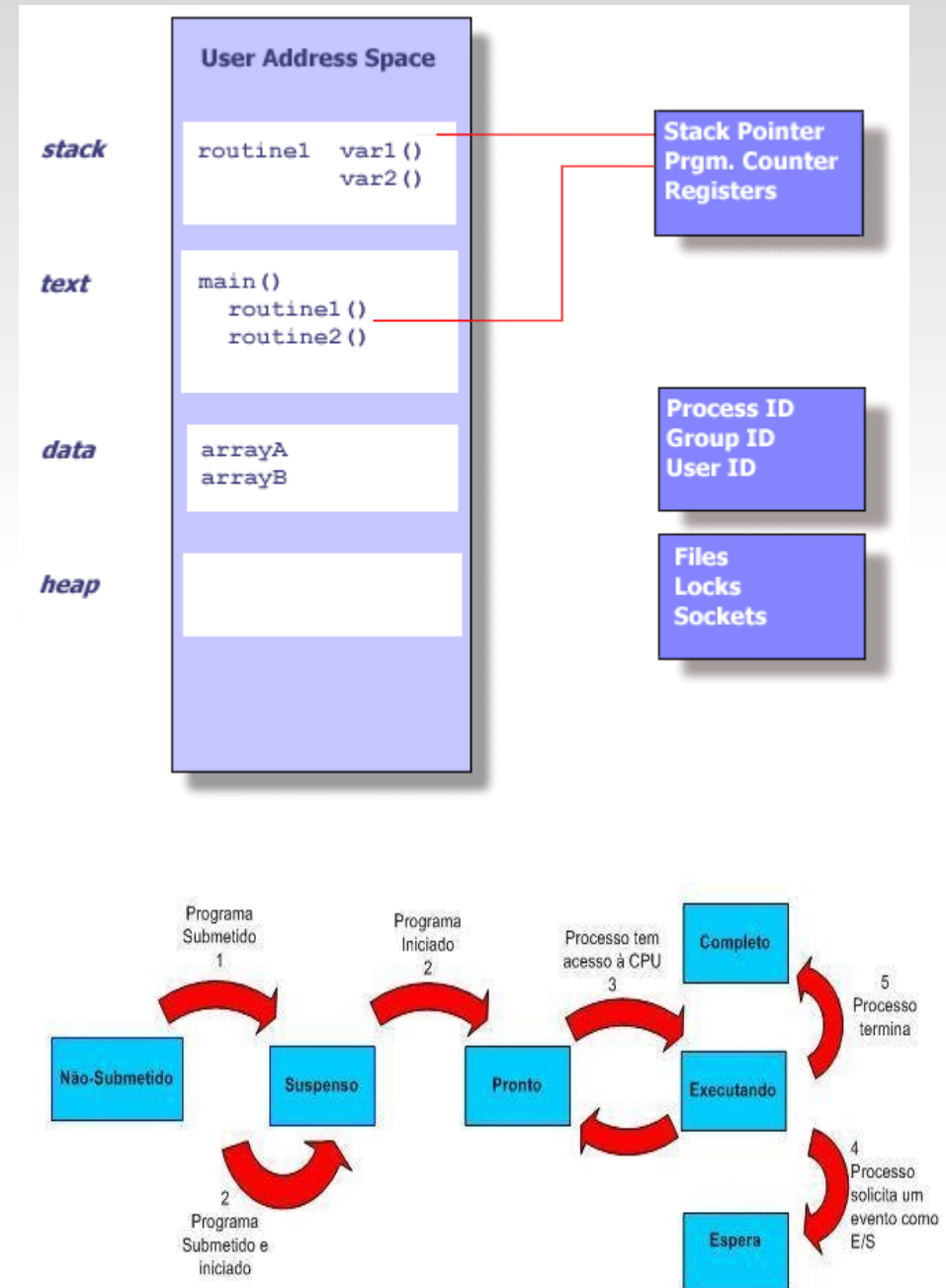
- Um processo é um programa em execução
- Processo é a unidade de computação que sistemas operacionais manipulam (criam, destroem, atribuem a processadores, retiram de processadores, etc)
- Processos tem espaço de endereçamento (memória) próprios e distintos
- Processos concorrentes podem ter execução simultânea ou não
- Os processos podem ser cooperantes ou não

# Conceitos importantes a revisar

- Processos
- SO multitarefas
- *Threads*

# Processo

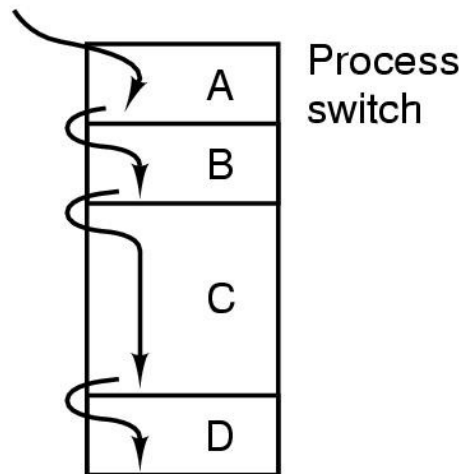
- Módulo executável carregável em um dado SO
- Assume diferentes estados
- Características: Código, Espaço de endereçamento, Registradores de uso geral, Apontador de programa indicando próxima instrução, Apontador de pilha, etc.



# SO Multitarefa

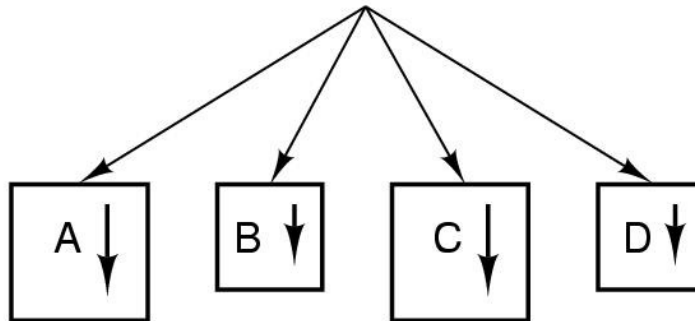
- Chaveia o processo em execução e escalona diversos processos
  - impressão de simultaneidade na execução dos processos.
- Troca de contexto:
  - Registradores de uso geral e apontadores devem ser preservados na memória
  - Os valores dos registradores de uso geral e dos apontadores do novo processo devem ser recuperados

One program counter

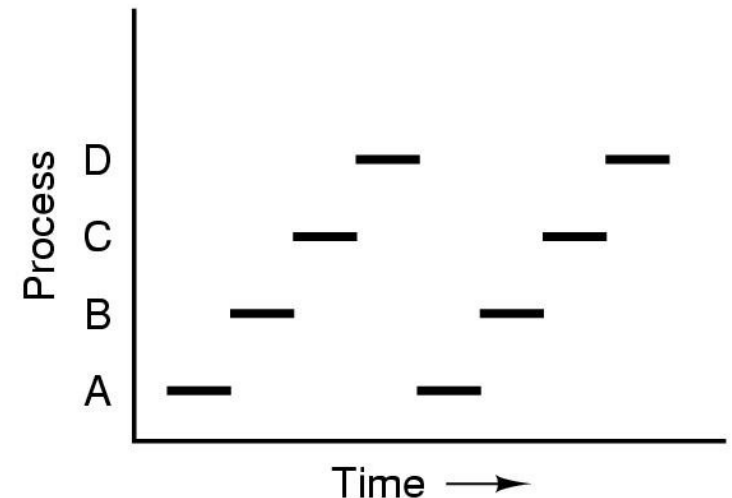


(a)

Four program counters



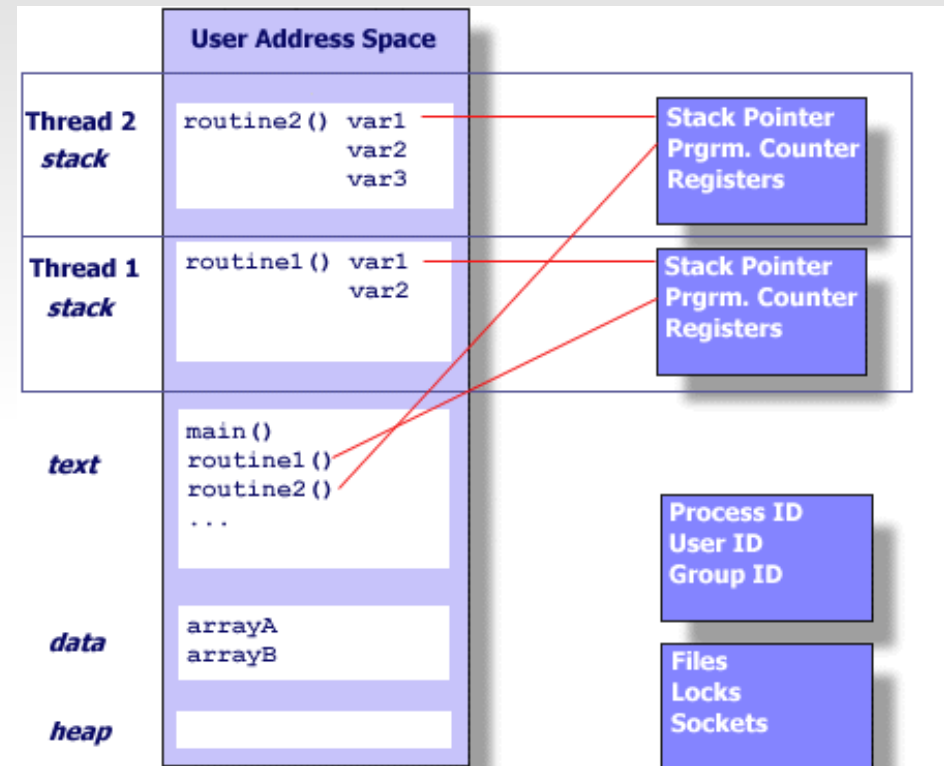
(b)



(c)

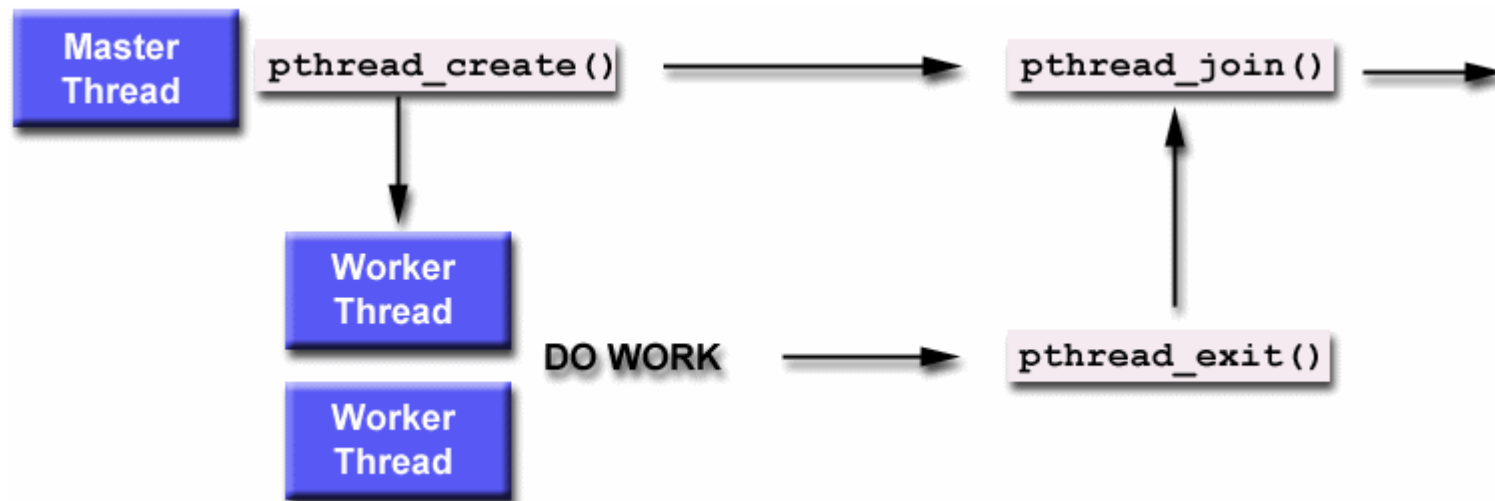
# Threads

- “Processo leve”
  - Fluxo de execução interno a um dado processo
- Características:
  - Apontadores de programa e de pilha próprios;
  - Herdam do processo pai:
    - Espaço de endereçamento e arquivos abertos.



# Processos e *Threads*

- Criação dinâmica de Processos requer grande tempo de execução
- Criação de *threads* é mais rápida (*thread* é um processo leve)
- Padronização para *Threads*: POSIX Standard 1003.1c, “*System Application Program Interface, Amendment 2: Threads Extension*”



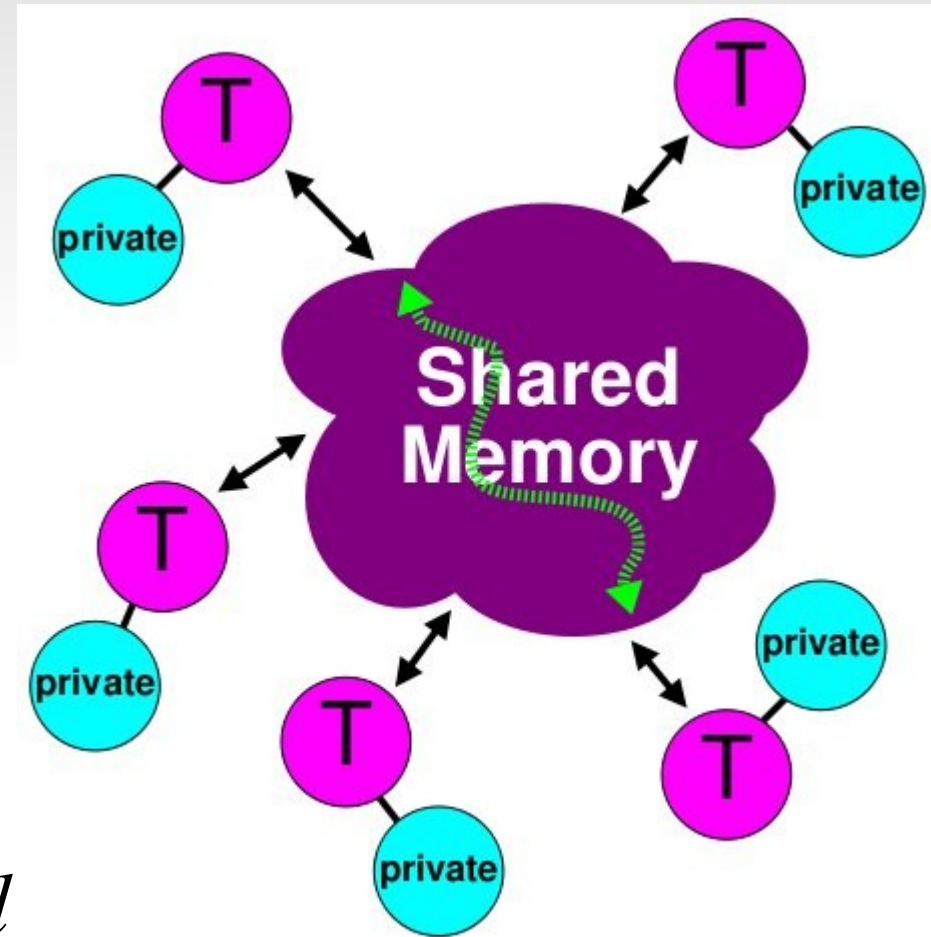
# Formas de expressar concorrência em vários níveis

<i><b>Granularity</b></i>	<i><b>Technology</b></i>	<i><b>Programming Model</b></i>
<i><b>Instruction Level</b></i>	<i><b>Superscalar</b></i>	<i><b>Compiler</b></i>
<i><b>Chip Level</b></i>	<i><b>Multicore</b></i>	<i><b>Compiler, OpenMP, MPI</b></i>
<i><b>System Level</b></i>	<i><b>SMP/cc-NUMA</b></i>	<i><b>Compiler, OpenMP, MPI</b></i>
<i><b>Grid Level</b></i>	<i><b>Cluster</b></i>	<i><b>MPI</b></i>



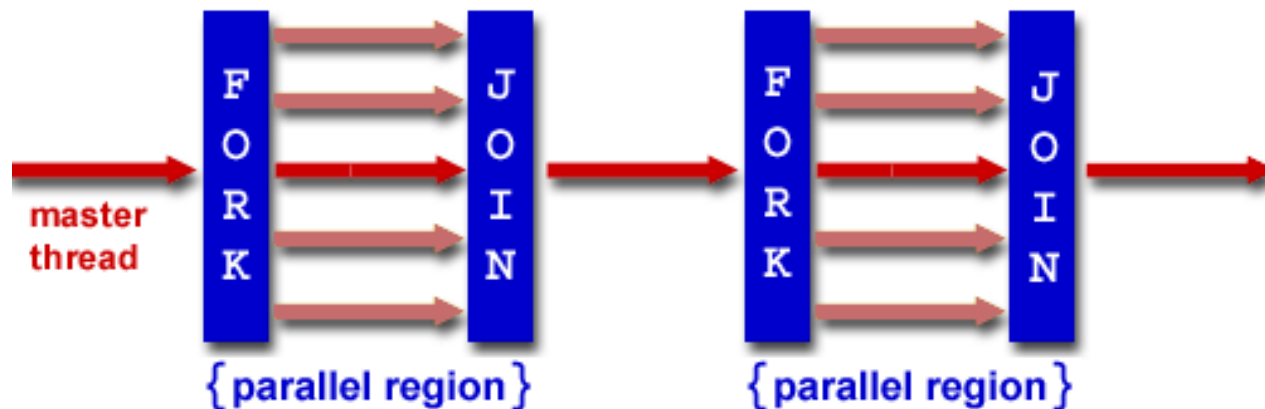
# Expressando Concorrência Sist. Memória Compartilhada

- Comunicações entre processos concorrentes através de acessos a dados em memória
- Modelo *Fork-Join*
  - Diretivas: *fork()*
  - *Threads* (*Posix Threads* e *Java Threads*)
  - *.NET System Threading*
  - *OpenMP*
- Outros modelos derivados: *Intel Threading Building Blocks* (TBB), etc.



# Modelo de programação *Fork-Join*

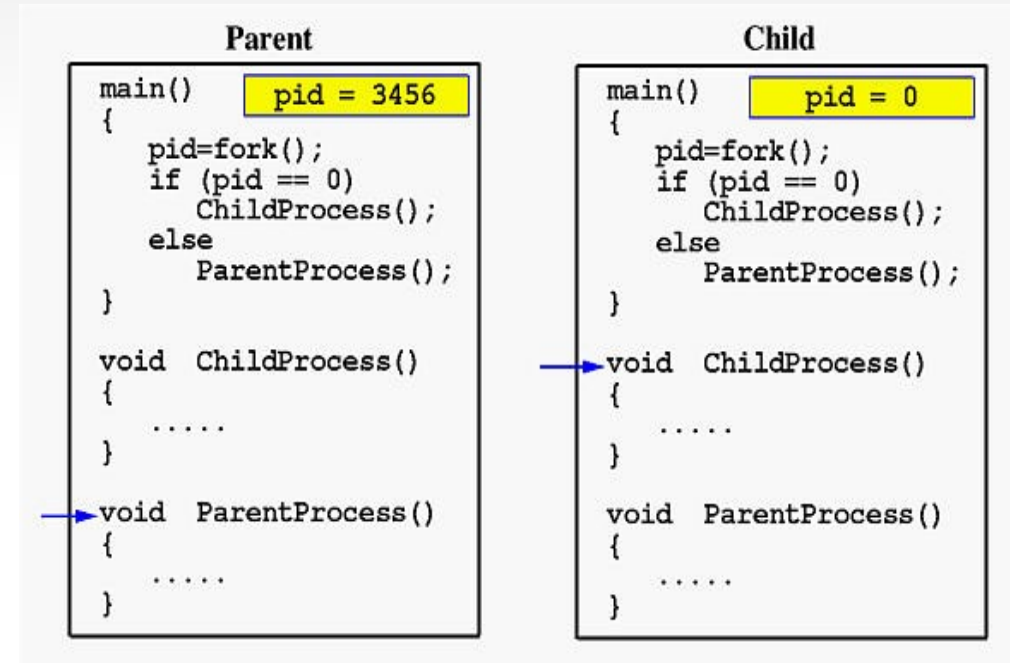
- *Connway, 1963 e Dennis & Van Horn, 1966*
- Um processo cria outros (*fork*)
- Processos executam concorrentemente
- Processos possuem o mesmo espaço de endereçamento
- Processo “pai” aguarda os “filhos” terminarem (*join*)



# Usando *fork()* para criar novo processo

- Primitiva *fork()* cria um novo processo
  - Filho do processo que executou a instrução
  - Sem argumentos de entrada
  - Retorna um *ID* de processo do SO

(ID=0 no contexto do processo filho; ID=identificador do processo recém criado no contexto do processo pai; -1 em caso de erro)



# Exemplo 1: Fork()

```
#include <stdio.h>
#include <sys/types.h>

#define MAX_COUNT 10

void ChildProcess(void);
/* child process prototype */
void ParentProcess(void);
/* parent process prototype */

void main(void)
{
    pid_t pid;

    pid = fork();
    if(pid==-1) /* erro */
        perror("impossivel de criar um
filho\n");
    else if (pid==0)
        ChildProcess();
    else
        ParentProcess();
}
```

```
void ChildProcess(void)
{
    int i, pid, parent;

    pid = getpid();
    parent = getppid();
    for (i=1; i<=MAX_COUNT; i++)
        printf(" Line from child, value=
%d\n", i);
    printf(" *** Child process (PID: %d,
parent: %d) is done ***\n", pid,
parent);
}

void ParentProcess(void)
{
    int i, pid, parent;

    pid = getpid();
    parent = getppid();
    for (i=1; i<=MAX_COUNT; i++)
        printf("Line from parent, value=
%d\n", i);
    printf("*** Parent (PID: %d, parent:
%d) is done ***\n",
pid, parent);
}
```

# Saída – Exemplo 1

```
./fork.x
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
  This line is from child, value = 1
  This line is from child, value = 2
  This line is from child, value = 3
This line is from parent, value = 6
  This line is from child, value = 4
This line is from parent, value = 7
This line is from parent, value = 8
This line is from parent, value = 9
This line is from parent, value = 10
  This line is from child, value = 5
  This line is from child, value = 6
*** Parent (PID: 1297, parent: 20532) is done ***
  This line is from child, value = 7
  This line is from child, value = 8
  This line is from child, value = 9
  This line is from child, value = 10
*** Child process (PID: 1298, parent: 1297) is done ***
```

# Posix Threads vs Fork

## Desempenho

**Criar novas *Threads* é muito mais rápido que criar novos processos com *fork()***

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Resultados para 50.000 criações de threads/processos

Fonte: *POSIX Threads Programming*.

(<https://computing.llnl.gov/tutorials/pthreads/#CreatingThreads>). Agosto de 2017.

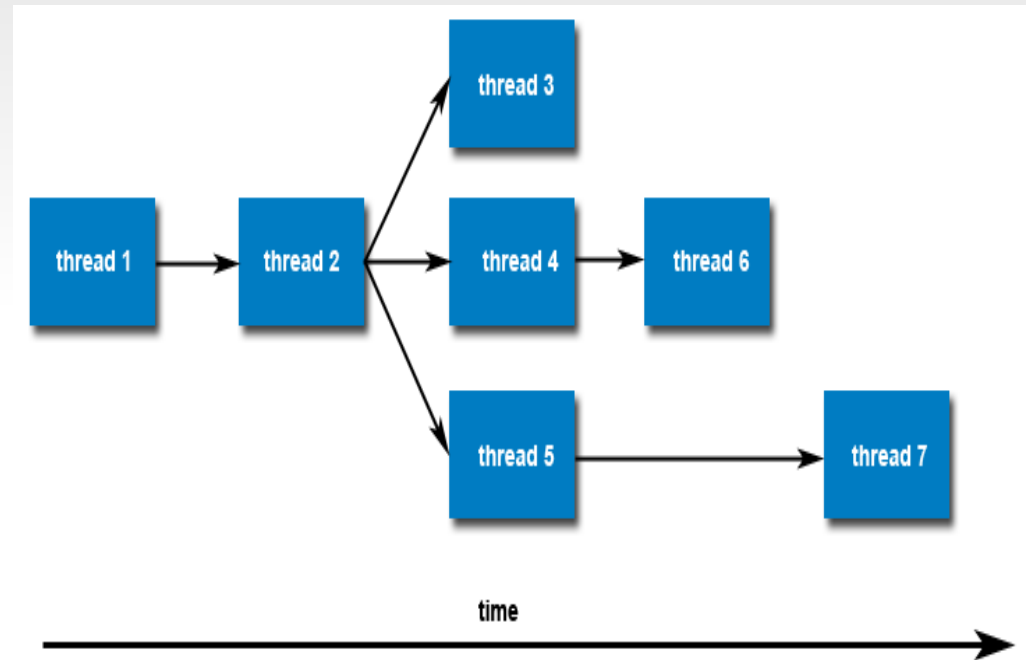
POSIX (*Portable Operating System Interface*, 1985) é uma família de normas definidas pelo IEEE (IEEE 1003), que tem como objetivo garantir a portabilidade do código-fonte em diferentes SOs aderentes a norma. A designação internacional da norma é ISO/IEC 9945. Sumariamente, define APIs para SOs tipo UNIX.

# Posix threads (PThreads) API: Criação de Threads

- `pthread_create(thread, attr, start_routine, arg)`
  - Cria uma nova thread e a executa imediatamente
    - Pode ser chamada qualquer número de vezes de qualquer local no código
  - Argumentos:
    - `thread`: Objeto do tipo `pthread_t` que permite a identificação da *thread* criada;
    - `attr`: atributo que deve ser utilizado para definir algumas características específicas da *thread*. (default=NULL);
    - `start_routine`: nome da função em C que será executada como um *thread*;
    - `arg`: único argumento que deve ser passado a função `start_routine` como dado de entrada/saída. Deve ser passado como referência a um ponteiro do tipo `void`. Pode-se usar `NULL` se nenhum argumento for passado e uma `struct` caso se deseje passar um conjunto de dados.

# Criação de *Threads* (cont.)

- O número máximo de *threads* é dependente da implementação
- Um *thread* existente pode criar outras *threads*. Não existe hierarquia explícita entre nenhuma *thread* criada
- O escalonamento das *threads* é dependente do SO ou pode ser imposto pela implementação.





# *Pthreads API:*

## *Destruição de Threads*

- Várias formas possíveis:
  - Retorno automático da *thread* após a execução da `start_routine`;
  - `pthread_exit(status)`: explicitamente termina uma thread.
    - Tipicamente utilizada quando o trabalho da thread finalizou-se e não mais será utilizada.
    - Deve ser chamada de dentro da própria thread;
  - Uma thread pode ser cancelada por outra através de `pthread_cancel()`;
  - O processo inteiro é cancelado por uma chamada a `exit` ou `exec`.

# Destruição de *threads* (cont.)

- Caso o programa principal (`main`) termine antes da *thread* que ele mesmo criou, através de uma chamada dele próprio a `pthread_exit`, as outras *threads* continuam executando normalmente;
- Caso o programa principal termine de forma usual todas as *threads* por ele criadas serão terminadas automaticamente quando `main` termina.

# Exemplo 2 - Pthread

```
#include <stdio.h>
#include <pthread.h>

#define MAX_COUNT 10
#define MAX_THREADS 2

void *ThreadProcess(void *th); /* thread process prototype */

int main(void)
{
    pthread_t t[MAX_THREADS]; // duas threads
    int i;

    for(i=0; i<MAX_THREADS; i++)
        pthread_create(&t[i], NULL, ThreadProcess, (void *) (i+1));

    pthread_exit(NULL);
}

void *ThreadProcess(void *th)
{
    int i, thid;

    thid = (int) th;
    for (i=1; i<=MAX_COUNT; i++)
        printf("Line from thread %d,value=%d\n", thid, i);

    pthread_exit(NULL);
}
```

# Compilação de programas em C com *PThreads*

- Usando GCC:

```
gcc -o <executavel> <fonte.c> -pthread
```

# Saída para o Exemplo 2

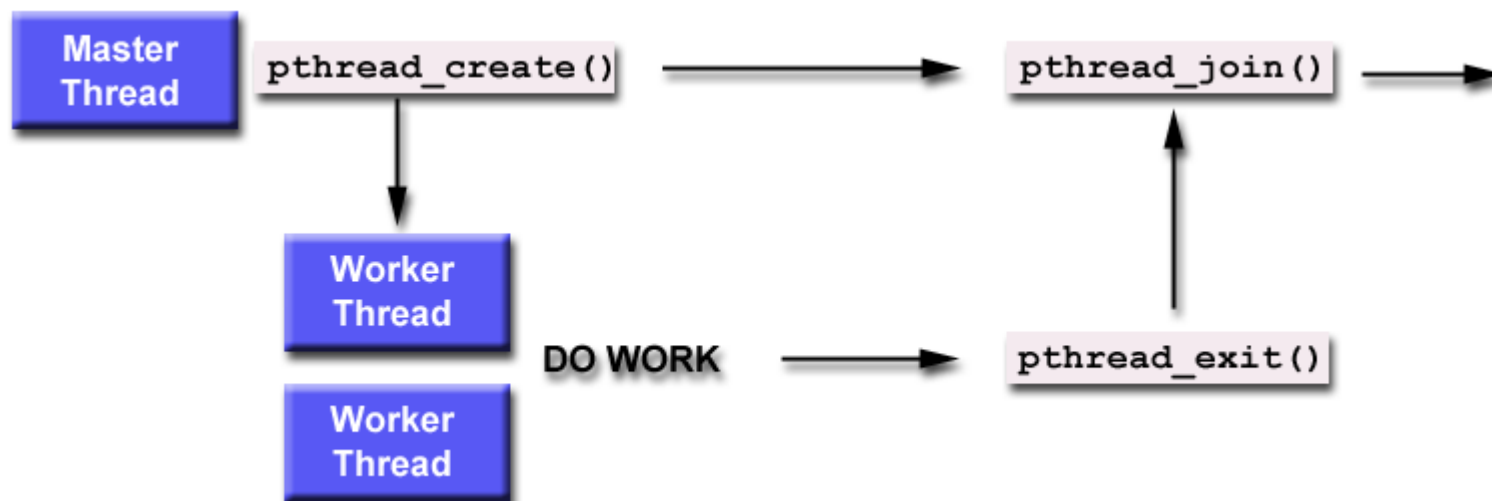
```
./pthreads-inicial-v2.x  
Line from thread 1,value=1  
Line from thread 1,value=2  
Line from thread 1,value=3  
Line from thread 1,value=4  
Line from thread 1,value=5  
Line from thread 1,value=6  
Line from thread 1,value=7  
Line from thread 1,value=8  
Line from thread 1,value=9  
Line from thread 1,value=10  
Line from thread 2,value=1  
Line from thread 2,value=2  
Line from thread 2,value=3  
Line from thread 2,value=4  
Line from thread 2,value=5  
Line from thread 2,value=6  
Line from thread 2,value=7  
Line from thread 2,value=8  
Line from thread 2,value=9  
Line from thread 2,value=10
```

# Exemplo 3: Passagem de argumentos

```
struct thread_data{
    int  thread_id;
    int  sum;
    char *message;
};
struct thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}
int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
    ...
}
```

# Pthreads API: Juntando Threads

- `pthread_join(threadid, status)`
- A rotina `pthread_join` bloqueia o chamador até que a *thread* especificada por `threadid` termine;
  - Permite sincronização entre *threads*
  - Cada thread pode atender a uma única ação “join”
  - Threadid: ID da thread
  - Status: status da operação “join”



# Exemplo 4: *Join*

```
#include <pthread.h>
#include <stdio.h>
#include <math.h>
#define NUM_THREADS    4

void *BusyWork(void *t)
{
    int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld
starting\n",tid);
    for (i=0; i<1000000; i++)
        result = result + sin(i)*tan(i);
    printf("Thread %ld done.", tid);
    printf("Result = %e\n", result);
    pthread_exit((void*) t);
}
```

```
int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    int rc;
    long t;
    void *status;

    for(t=0; t<NUM_THREADS; t++) {
        printf("Main:create thread %ld\n",t);
        rc = pthread_create(&thread[t], NULL,
BusyWork, (void *)t);
        if (rc) {
            printf("ERROR:create=%d\n", rc);
            exit(-1); }
    }

    for(t=0; t<NUM_THREADS; t++) {
        rc = pthread_join(thread[t],
&status);
        if (rc) {
            printf("ERROR:join code=%d\n", rc);
            exit(-1); }
        printf("Main:join thread %ld", t);
        printf(" status=%ld\n", (long)status);
    }

    printf("Main:Exiting.\n");
    pthread_exit(NULL);}
```



# Saída para exemplo 4

```
./join-simples.x
Main:create thread 0
Main:create thread 1
Thread 0 starting...
Thread 1 starting...
Main:create thread 2
Thread 2 starting...
Main:create thread 3
Thread 3 starting...
Thread 0 done. Result = -3.153838e+06
Main:join thread 0 status=0
Thread 2 done. Result = -3.153838e+06
Thread 1 done. Result = -3.153838e+06
Main:join thread 1 status=1
Main:join thread 2 status=2
Thread 3 done. Result = -3.153838e+06
Main:join thread 3 status=3
Main:Exiting.
```

# *Thread* “juntável” ou não

- *Threads* podem ser *joinable* (juntáveis) ou *detached* (destacadas)
- Esta característica é definida pelo atributo `attr` (de `pthread_create`)
- O padrão POSIX especifica que *threads* por padrão (*default*) devem ser juntáveis
- A função `pthread_detach` é usada para tornar uma *thread* destacada (independente do atributo)

# Definindo os atributos attr

- Passos necessários:
  - Declaração de uma variável de atributo do tipo `pthread_attr_t`
  - Inicialização do atributo com `pthread_attr_init()`
  - Definição do *status* com `pthread_attr_setdetachstate()`:
    - `PTHREAD_CREATE_JOINABLE`
    - `PTHREAD_CREATE_DETACHED`
  - Liberação do atributo, após o termino, com: `pthread_attr_destroy()`



# Java Threads: Criação

- Funcionamento similar a *Pthreads* mas com orientação a objetos
- O corpo de uma *thread* é o seu método `run()`.
  - Métodos básicos para se lidar com threads: `run()`, `start()`, `stop()`, `sleep()`.
- Toda *thread* em Java tem um nome (tipo `String`)
  - Se não for dado um nome será atribuído um pelo sistema
- Duas formas possíveis:
  - 1) Objeto do tipo `Runnable`;
  - 2) Subclasse de `Thread`.

# Objeto Runnable

- Deve-se definir um método `run` que contenha o código a ser executado na *thread*
- O objeto `Runnable` deve ser passado para o construtor da classe `Thread`
- Exemplo:

```
import java.lang.*;
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread named:" +
            Thread.currentThread().getName());
    }
    public static void main(String args[]) {
        new Thread(new HelloRunnable(), "Thead A").start();
    }
}
```

# Subclasse de Thread

- Exemplo:

```
import java.lang.*;
public class HelloThread extends Thread {
    public HelloThread (String nome) {    // construtor
        super(nome);    // chama construtor da superclasse
    }
    public void run() {
        System.out.println("Hello from a thread named:" +
            Thread.currentThread().getName());
    }
    public static void main(String args[]) {
        new HelloThread("Thread A").start();
    }
}
```

# Runnable VS Thread

- Ambos os casos invocam `Thread.start` para iniciar uma nova *thread*
- Com `Runnable` pode-se ter uma subclasse de uma outra classe qualquer com `Threads`
- Subclasse de `Thread` é levemente mais simples de usar

# Exemplo 5 – *JavaThread*

```
public class printthreadv2 implements Runnable {
    private long count;

    public printthreadv2(long account) {    // construtor
        count = account;
    }

    public void run() {
        long i;
        for(i=1; i<=count; i++)
            System.out.println("Line from " +
                Thread.currentThread().getName() + " value=" + i);
    }

    public static void main(String args[]) {
        long MAX_THREADS = 2;
        long MAX_COUNT    = 10;
        long i;
        String s;
        for(i=0; i<MAX_THREADS; i++) {
            s = "Thread " + ((char) (65+i));
            new Thread(new printthreadv2(MAX_COUNT), s).start();
        }
    }
}
```



# Saída do exemplo 5

```
java printthreadv2
Line from Thread A value=1
Line from Thread A value=2
Line from Thread A value=3
Line from Thread A value=4
Line from Thread A value=5
Line from Thread A value=6
Line from Thread A value=7
Line from Thread A value=8
Line from Thread A value=9
Line from Thread A value=10
Line from Thread B value=1
Line from Thread B value=2
Line from Thread B value=3
Line from Thread B value=4
Line from Thread B value=5
Line from Thread B value=6
Line from Thread B value=7
Line from Thread B value=8
Line from Thread B value=9
Line from Thread B value=10
```

# Multithreads para melhorar desempenho - exemplo

- Lembrando que  $\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = e = 2.71828182846\dots$
- Pode-se calcular  $\left(1 + \frac{1}{n}\right)^n$  para  $n$  grande
- Cálculo sequencial: cálculo direto
- Cálculo *multithread*:
- Calcular:

$$x_i = \left(1 + \frac{1}{n}\right)^{n/M}$$

$$x = \prod_{i=1}^M \left(1 + \frac{1}{n}\right)^{n/M}$$

# Exemplo 6 – Pthreads - e

```
#include <stdio.h>
#include <pthread.h>

#define N 1000000000
#define MAX_THREADS 8

void *calcula(void *res){
    int k;
    double p = 1 , x , y;
    x = 1 + 1.0/N;
    for(k=0; k<(N/MAX_THREADS); k++)
        p = p*x;
    //copia valor p/variável de retorno
    *( (double*)res ) = p;
    pthread_exit(NULL);
}
```

```
int main(void){
    pthread_t t[MAX_THREADS];
    double v[MAX_THREADS];
    double final;
    int i;

    for(i=0; i<MAX_THREADS; i++) {
        pthread_create(&t[i], NULL,
                      calcula,
                      (void *) &v[i]);
    }

    final = 1.;
    for(i=0; i<MAX_THREADS; i++) {
        pthread_join(t[i], NULL);
        final = final*v[i];
    }

    printf("Resultado=%lf\n", final);
    return 0;
}
```

# Desempenho (*PThreads*):

- Configuração:
  - CPU: Intel(R) Core(TM)2 Quad
    - Q8200 @ 2.33GHz
  - Memória principal: 2Gb
- 1 Thread: 4.759s
- 2 Threads: 2.401s
- 4 Threads: 1.219s
- 8 Threads: 1.315s !!

# Exemplo 7 – Java threads - e

```
class RunThE implements Runnable {
    private long    N;
    private int     MaxThreads;
    // resultado da operação
    public double valor;

    // construtor
    public RunThE(long aN,
                   int aMaxThreads) {
        N = aN;
        MaxThreads = aMaxThreads;
    }

    public void run() {
        int k;
        double p = 1, x, y;
        x = 1 + 1.0/N;
        for(k=0 ;k<N/MaxThreads; k++)
            p = p*x;
        valor = p;
    }
}
```

```
public class CalcThE {
    public static void
    main(String args[]) {
        int MaxThreads = 8, i;
        long N = 1000000000;
        Thread[] th;
        RunThE[] rh;
        double resfinal;

        rh = new RunThE[MaxThreads];
        th = new Thread[MaxThreads];
        for(i=0; i<MaxThreads; i++) {
            rh[i] = new RunThE(N,
                               MaxThreads);
            th[i] = new Thread(rh[i]);
            th[i].start();
        }

        resfinal = 1;
        try {
            for(i=0; i<MaxThreads; i++) {
                th[i].join();
                Resfinal *= rh[i].valor;
            }
        } catch (InterruptedException e)
        { System.out.println("Excecao"); }
        System.out.println("Resultado=" +
                           resfinal);
    }
}
```

# OpenMP (<http://openmp.org>)

- Padrão *de fato* para o modelo de programação de memória compartilhada *fork-join*
  - *OpenMP API specification for parallel programming*
  - Portabilidade garantida devido ao padrão
- Paralelismo explícito
  - Diretivas de programação com interface para linguagem C e Fortran
- Compiladores traduzem as diretivas para as primitivas do sistema operacional baseadas em *threads*
  - diretivas expressas como comentários
    - Permite execução sequencial do programa paralelizado se o compilador não for munido com OpenMP.
  - Combina código sequencial com código paralelo
  - Permite aplicação incremental

# Modo de operação do



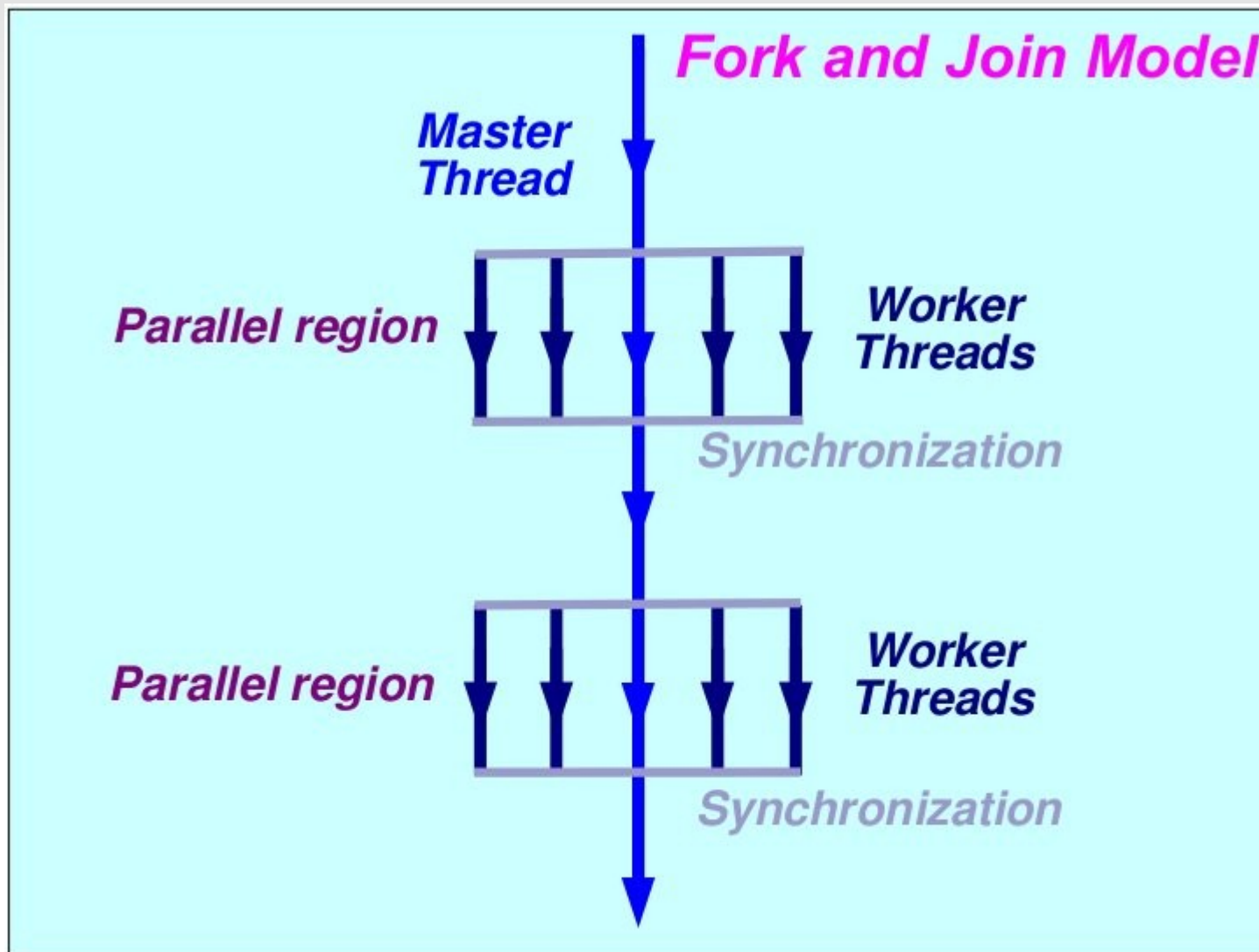
- Uma região paralela é a estrutura básica do paralelismo *OpenMP*
  - Define uma seção paralela do programa
- Inicia a execução com uma única *thread*
  - Denominada *thread master*
- Criação automática de *threads* em regiões paralelas
  - *Overhead* para criação e destruição
    - Importante minimizar o número de regiões paralelas abertas
  - Existência de variáveis privadas ou compartilhadas

# Uso do OpenMP

- Fácil utilização
- Trata, frequentemente, de paralelismo de dados em laços
- Construção *work-sharing*
  - Processamento concorrente para cada conjunto de dados
- Sincronização de *threads* implícita
  - Barreira implícita no término de cada laço.
  - Processos aguardam o final da execução do último conjunto de dados para continuar
- O sincronismo pode se tornar explícito se desejado
- Seções críticas sobre dados compartilhados podem ser definidas pelo programador



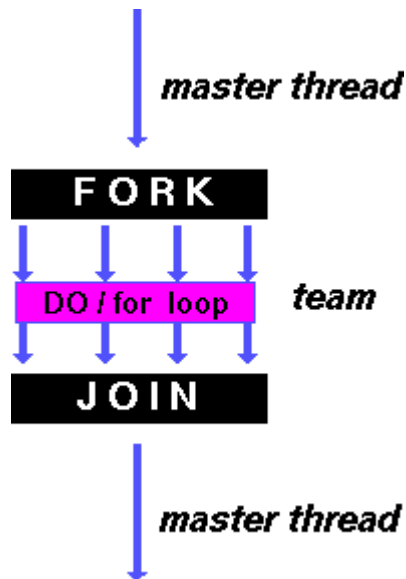
# Modo de execução do OpenMP



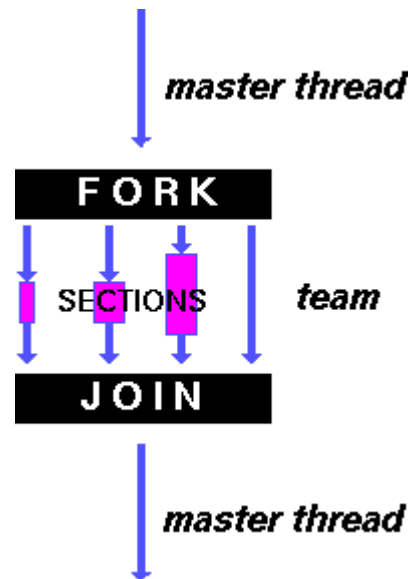
# Tipos de paralelismo

## Work-Share

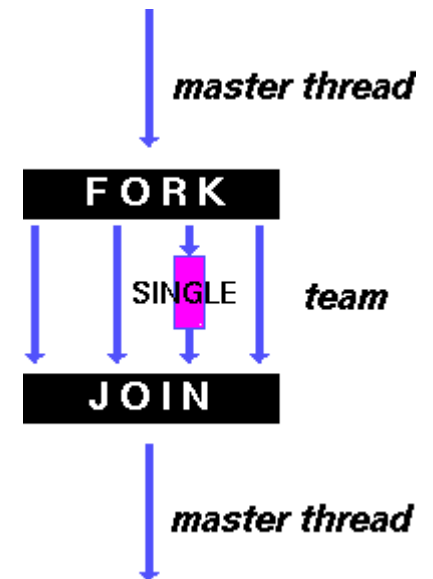
Laços paralelos.  
**Paralelismo de dados**



Seções paralelas:  
cada *thread* executa uma seção  
**Paralelismo funcional**



Execução serial dentro de uma seção paralela



# Diretivas e Sentinelas

- Uma diretiva é uma linha especial de código fonte com significado especial apenas para determinados compiladores
- Uma diretiva se distingue pela existência de uma sentinela no começo da linha.
- As sentinelas do OpenMP são:
  - Fortran: `! $OMP` (ou `C $OMP` ou `* $OMP`)
    - O início e fim das seções devem ser indicadas por diretivas específicas
  - C/C++: `#pragma omp`
    - Seguem o padrão de diretivas de compilação para C/C++
    - Cada diretiva se aplica no máximo a próxima instrução, que deve ser um bloco estruturado

# Regiões Paralelas

- Um código dentro da região paralela é executado por todas as *threads*.
- Sintaxe em C/C++:
  - Exige incluir a biblioteca: <omp.h>

```
#pragma omp parallel
{
    // bloco executado em paralelo
}
```

# Cláusulas

- Especificam informação adicional na diretiva de região paralela:
- C/C++:
  - `#pragma omp parallel [clausulas]`
- Cláusulas são separadas por vírgula ou espaço no Fortran, e por espaço no C/C++

# Atributos de dados

- Dentro de uma região paralela as variáveis podem ser compartilhadas (todas as *threads* vêem a mesma cópia) ou privada (cada *thread* tem a sua própria cópia).
  - Definidas como cláusulas na diretiva OpenMP
- Cláusulas *SHARED*, *PRIVATE* e *DEFAULT* em C/C++:
  - **shared(list)**
    - Todas as *threads* acessam o mesmo endereço de memória
  - **private(list)**
    - Nenhuma associação com o dado global
      - Valores são indefinidos na entrada e saída
      - Referencias são sempre para o dado local
  - **default(shared|none)**
    - Define valores *default*:
      - *Shared* implica em todos os dados compartilhados
      - *None* implica que nada será definido por *default*
        - » Todos os dados necessitam de atributos

# Definindo Paralelismo no OpenMP

- *OpenMP Team := Master + Workers*
- Uma região paralela é um bloco de código executado por todas as *threads* simultaneamente
  - A *thread* Mestre tem ID = 0
  - Ao iniciar uma região paralela todos os *threads* estão sincronizados
  - Regiões paralelas podem ser aninhadas, mas esta característica é dependente da implementação
  - Regiões paralelas podem ser condicionadas por "*if*"

# Funções úteis em OpenMP

Função	Utilidade
<code>omp_get_num_threads()</code>	Retorna o número de threads em execução
<code>omp_set_num_threads(n)</code>	Define o número de threads (n)
<code>omp_get_thread_num()</code>	Retorna o ID da Thread
<code>omp_get_num_procs()</code>	Retorna o número de processadores do sistema
<code>omp_in_parallel()</code>	Retorna verificação se no dado ponto do programa está-se ou não em uma região paralela
<code>omp_get_wtime()</code>	Retorna o <i>Wall Clock Time</i> do sistema
<code>omp_get_wtick()</code>	Retorna o número de <i>ticks</i> ou intervalos regularmente espaçados ditados pela frequência do <i>clock</i> entre um segundo, no sistema

Muitas outras mais...



# Hello World com OpenMP

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int th_id, nthreads;
#pragma omp parallel private(th_id)
    {
        th_id = omp_get_thread_num();
        printf("Hello World from thread %d\n", th_id);
        if (th_id==0) {
            nthreads = omp_get_num_threads();
            printf("There are %d threads\n", nthreads);
        }
    }
    return 0;
}
```

# Compilação de programas em C/C++ com OpenMP

- Compiladores disponíveis: Intel C/C++ e Fortran-95, GCC e GFORTRAN, G95, PGI, PathScale, Absoft
- Usando GCC:

```
gcc -fopenmp -o <executavel> <fonte.c>
```

# Laços Paralelos

- Principal fonte de paralelismo em muitas aplicações
- Se as iterações de um laço são independentes (podem ser executadas em qualquer ordem), então pode-se executar iterações diferentes em *threads* diferentes
- Exemplo: considerando 2 *threads* e o laço:

```
for (i = 0; i < 100; i++) {  
    a[i] = a[i] + b[i];  
}
```

- Pode-se fazer as iterações 0-49 em uma *thread* e as iterações 50-99 na outra

# Laços Paralelos (cont.)

- Usa-se uma diretiva que aparece dentro de uma região paralela e indica como o trabalho deve ser compartilhado entre as *threads*
- Sintaxe C/C++:
  - `#pragma omp for [clausulas]`
- Sem cláusulas adicionais, a diretiva DO/FOR particionará as iterações o mais igualmente possível entre as *threads*
  - Contudo, isto é dependente de implementação e ainda há alguma ambiguidade:
    - Ex.: 7 iterações, 3 *threads*. Podem ser particionados como 3+3+1 ou 3+2+2

# Exemplo ilustrativo

## For-loop with independent iterations

```
for (int i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

## For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for  
for (int i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

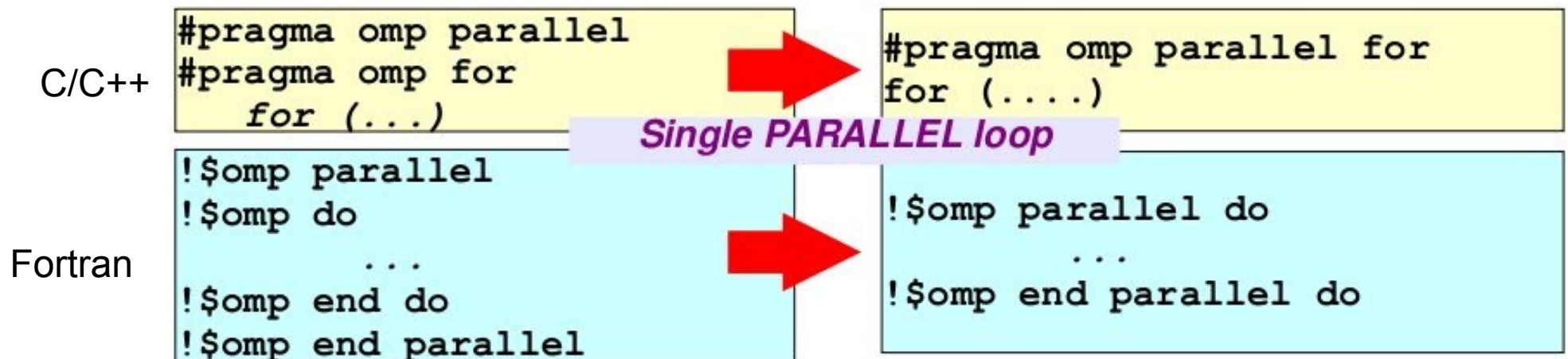
Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
← i=0-199 →	← i=200-399 →	← i=400-599 →	← i=600-799 →	← i=800-999 →
a[i]	a[i]	a[i]	a[i]	a[i]
+	+	+	+	+
b[i]	b[i]	b[i]	b[i]	b[i]
=	=	=	=	=
c[i]	c[i]	c[i]	c[i]	c[i]

# Paralelizando laço

```
/* Exemplo 3 */  
/* Laco perfeitamente paralelizavel */  
#pragma omp parallel  
#pragma omp for  
for (i=1; i<n; i++) {  
    b[i]= (a[i] - a[i-1])*0.5; }  
/* end parallel for */
```

# A diretiva DO/FOR paralela

- Esta construção é tão comum que existe uma forma que combina a região paralela e a diretiva DO/FOR:
- Sintaxe C/C++:
  - `#pragma omp parallel for [clausulas]`



# Aplicando a diretiva DO/FOR paralela

```
/* Exemplo 3 - Laco paralelizavel */
```

```
#pragma omp parallel for
```

```
for (i=1; i<n; i++) {
```

```
    b[i]= (a[i] - a[i-1])*0.5; }
```

```
/* end parallel for */
```

```
/* Exemplo 4 */
```

```
#pragma omp parallel for
```

```
for (i=0; i < n; i++) {
```

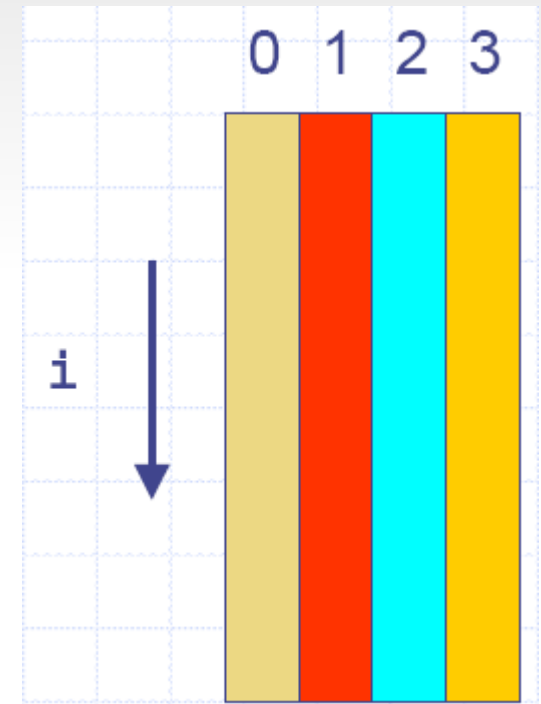
```
    z[i] = a * x[i] + y; }
```



# Variáveis Privadas e Compartilhadas (exemplo)

- Exemplo: cada *thread* inicia a sua própria coluna de uma matriz compartilhada:

```
#pragma omp parallel default(none) \  
    private (i, myid) shared(a,n)  
{  
    myid = omp_get_thread_num();  
  
    for(i = 0; i < n; i++){  
        a[i][myid] = 1.0; }  
} /* end parallel */
```



# Quais variáveis devem ser compartilhadas e privadas?

- A maioria das variáveis são compartilhadas
  - *Defaults:*
    - O índices dos laços são privados
    - Variáveis temporárias dos laços são compartilhadas
    - Variáveis apenas de leitura são compartilhadas
    - *Arrays* principais são compartilhadas
    - Escalares do tipo *write-before-read* são usualmente privados
  - A decisão pode ser baseada em fatores de desempenho

# Valor inicial de variáveis privadas

- Variáveis privadas não tem valor inicial no início da região paralela.
- Para dar um valor inicial deve-se utilizar a cláusula *FIRSTPRIVATE*:

– Exemplo:

```
b = 23.0;
```

```
. . . . .
```

```
#pragma omp parallel firstprivate(b) private(i,myid)
```

```
{
```

```
myid = omp_get_thread_num();
```

```
#pragma omp for
```

```
for (i=0; i<n; i++) {
```

```
    b += c[i]*2; }
```

```
d[myid] = b;
```

```
}
```

# Reduções

- Uma redução produz um único valor a partir de operações associativas como soma, multiplicação, máximo, mínimo, e , ou.

– Exemplo:

```
b = 0;  
for (i=0; i<n; i++) {  
    b += a[i]; }
```

- Permitir que apenas uma *thread* por vez atualize a variável *b* impede todo o paralelismo
- Ao invés disto, cada *thread* pode acumular sua própria cópia privada, então essas cópias são reduzidas para dar o resultado final

# Reduções (cont.)

- Uso da cláusula REDUCTION:
  - C/C++: `reduction(op:list)`
    - Onde *op* pode ser:

Operador (op)	Operação	Valor inicial
+	Adição	0
-	Subtração	0
*	Multiplicação	1
&	E lógico	Todos os bits em 1
	OU lógico	Todos os bits em 0
^	Equivalente (lógica)	Todos os bits em 0
&&	Não equivalência (lógica)	Todos os bits em 1
	Máximo	0

# Redução – Exemplo:

```
b = 0;

#pragma parallel private (i) reduction (+:b)
{
    #pragma omp for
    for (i = 0; i < n; i++) {
        b = b + c[i]; }
}

/* omp end parallel */

/* b retorna soma de todos os elementos da matriz
*/
```

# Variáveis de ambiente do SO para OpenMP

- OMP\_NUM\_THREADS – especifica o número de *threads* para serem usadas durante a execução de regiões paralelas
  - O valor padrão para esta variável é o número de processadores
  - Pode-se definir um número de *threads* independente do número de processadores físicos disponíveis

# Exemplo 7 – OpenMP - e

```
#include <stdio.h>
#include <omp.h>
#define N 10000000000

int main(int argc, char
    *argv[]){
    double x, final;
    int i;

    final = 1;
    x = 1 + 1.0/N;
```

```
#pragma omp parallel
    default(none) private(i)
    shared(x) reduction(*:final)
    {
        #pragma omp for
            for(i=0; i<N; i++) {
                final = final*x;
            }
    }

    printf("Resultado=
    %lf\n",final);

    return 0;
}
```



# Seções paralelas (paralelismo funcional)

Cada seção é  
executada  
por uma  
*thread*

```
#pragma omp parallel default(none) \  
    shared(n,a,b,c,d) private(i)  
{  
    #pragma omp sections nowait  
    {  
        #pragma omp section  
        for (i=0; i<n-1; i++)  
            b[i] = (a[i] + a[i+1])/2;  
  
        #pragma omp section  
        for (i=0; i<n; i++)  
            d[i] = 1.0/c[i];  
  
    } /*-- End of sections --*/  
  
} /*-- End of parallel region --*/
```