Programação concorrente em Sistemas de Memória compartilhada

Prof: Álvaro L. Fazenda
Profa. Denise Stringhini
(alvaro.fazenda@unifesp.br)

Concorrência em nível de Processos

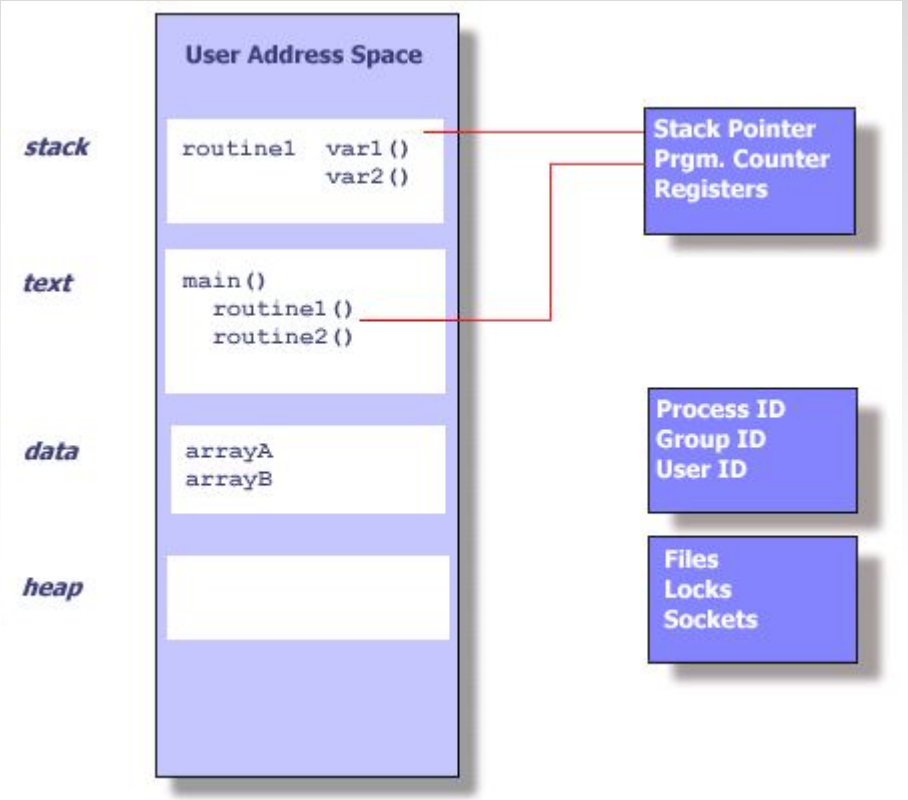
- Um processo é um programa em execução
- Processo é a unidade de computação que sistemas operacionais manipulam (criam, destroem, atribuem a processadores, retiram de processadores, etc)
- Processos tem espaço de endereçamento (memória) próprios e distintos
- Processos concorrentes podem ter execução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

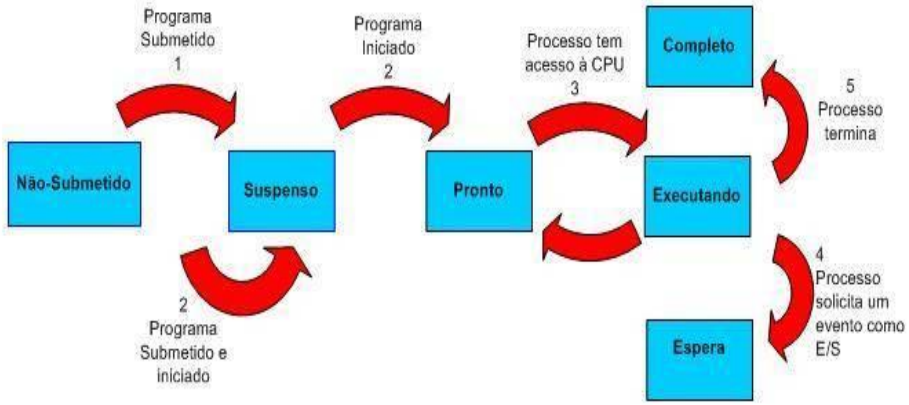
Conceitos importantes a revisar

- Processos
- SO multitarefas
- *Threads*

Processo

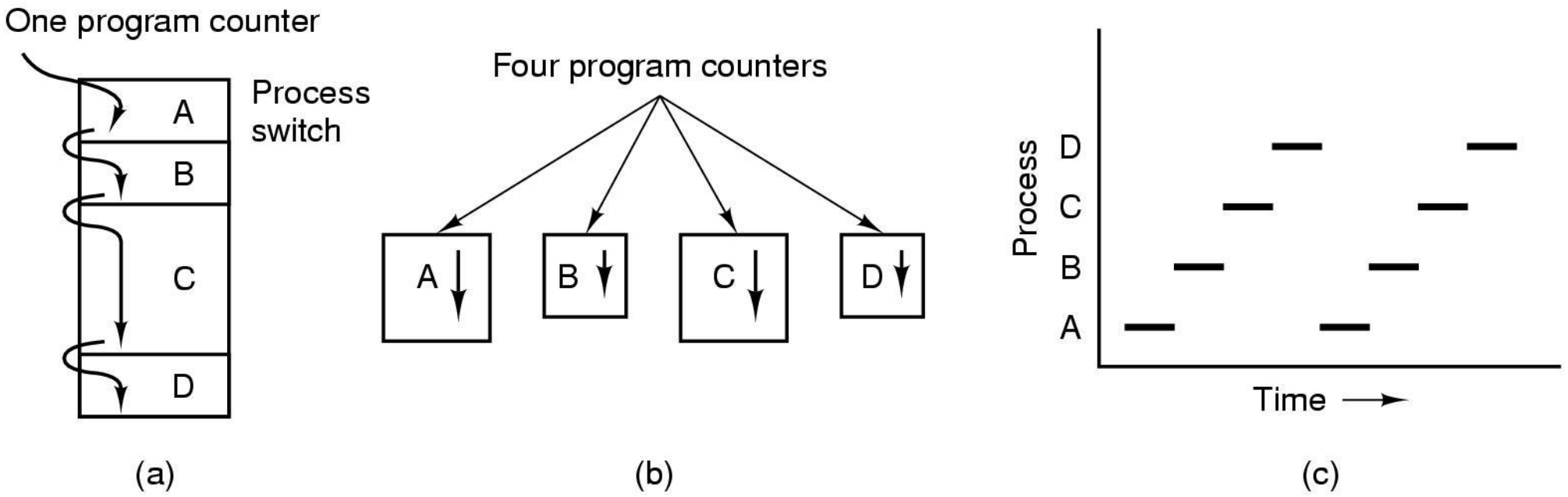
- Módulo executável carregável em um dado SO
- Assume diferentes estados
- Características: Código, Espaço de endereçamento, Registradores de uso geral, Apontador de programa indicando próxima instrução, Apontador de pilha, etc.





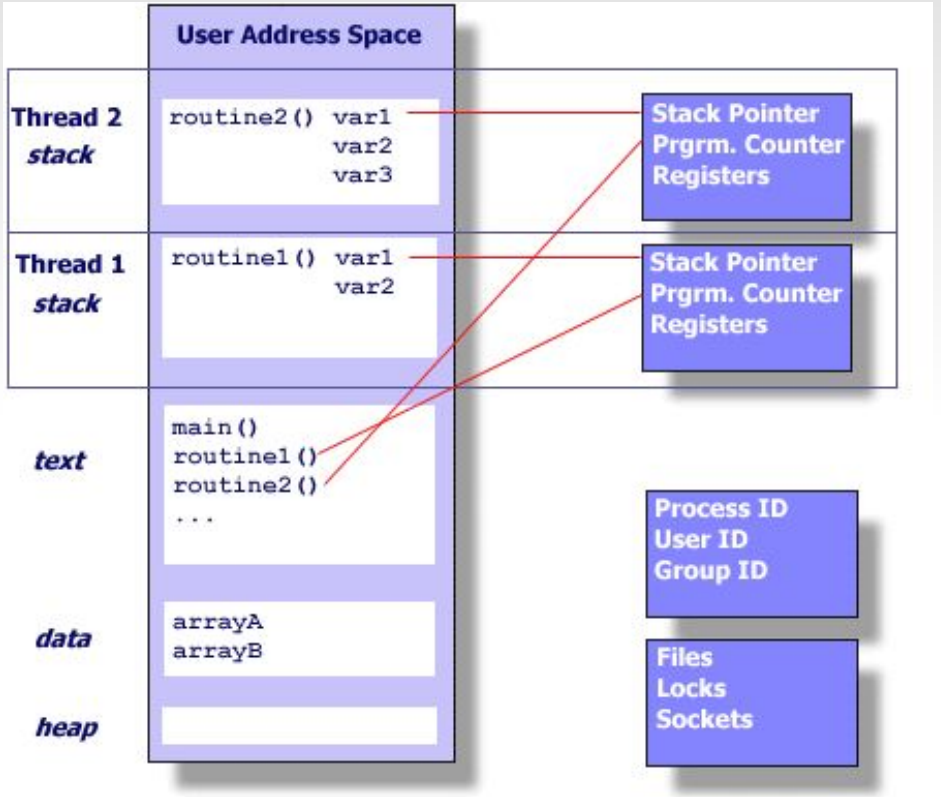
SO Multitarefa

- 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



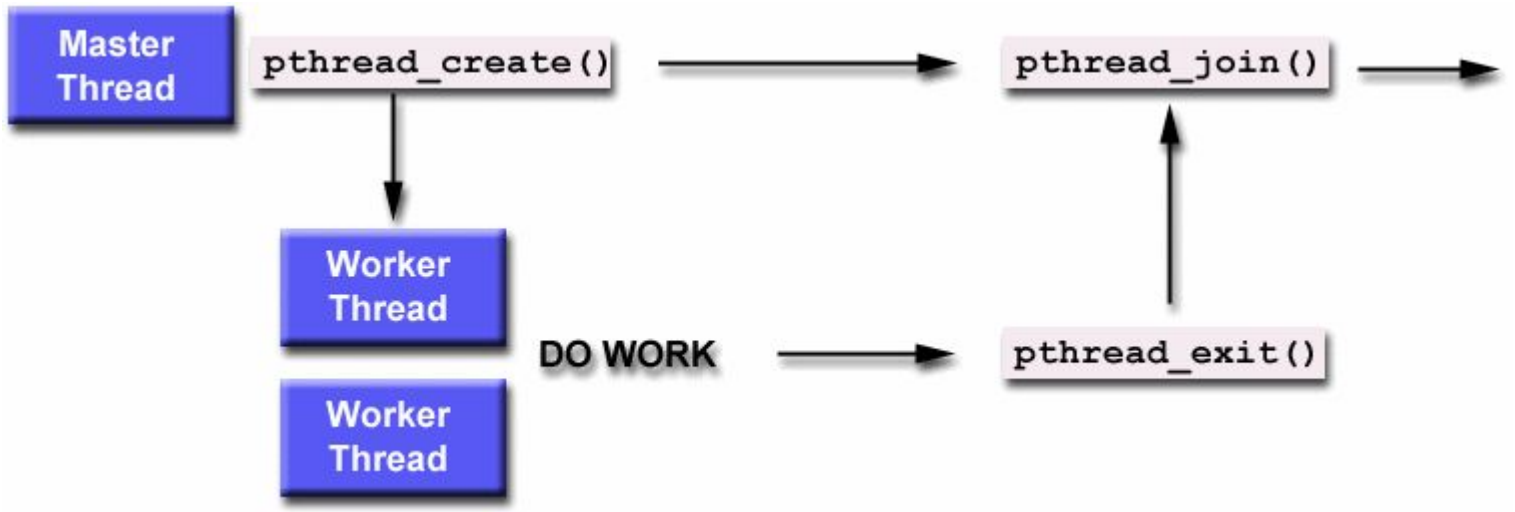
Threads

- "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*"

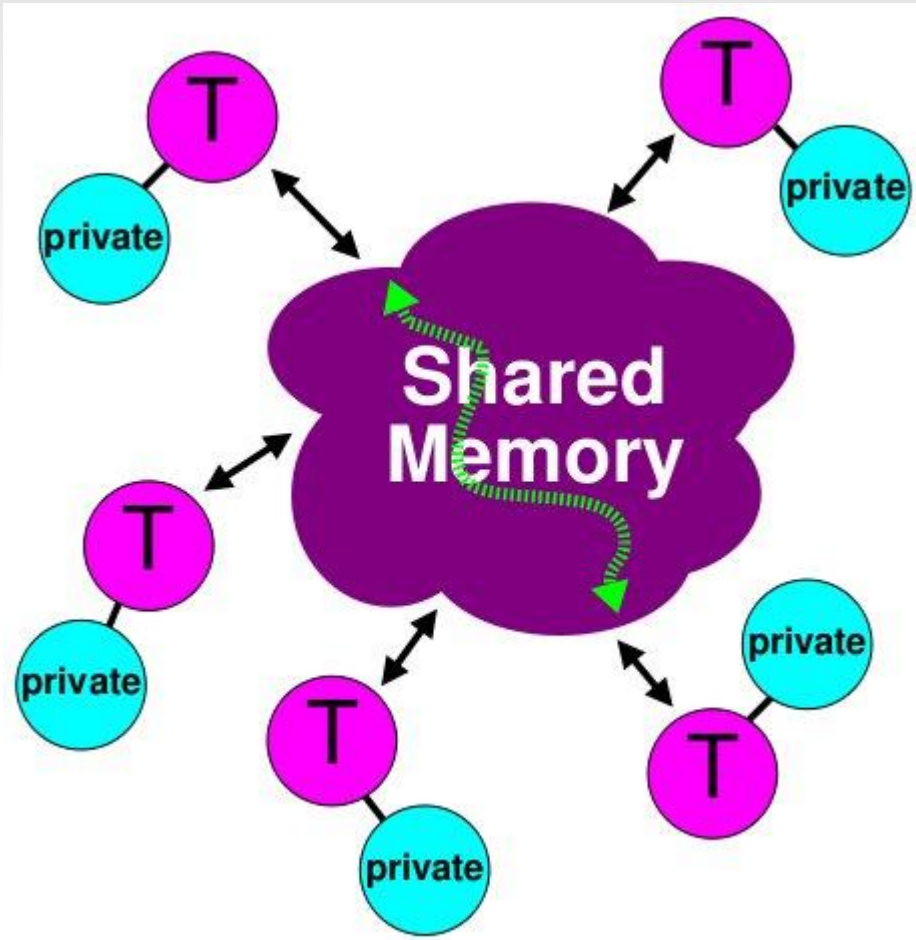


Formas de expressar concorrência em vários níveis

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

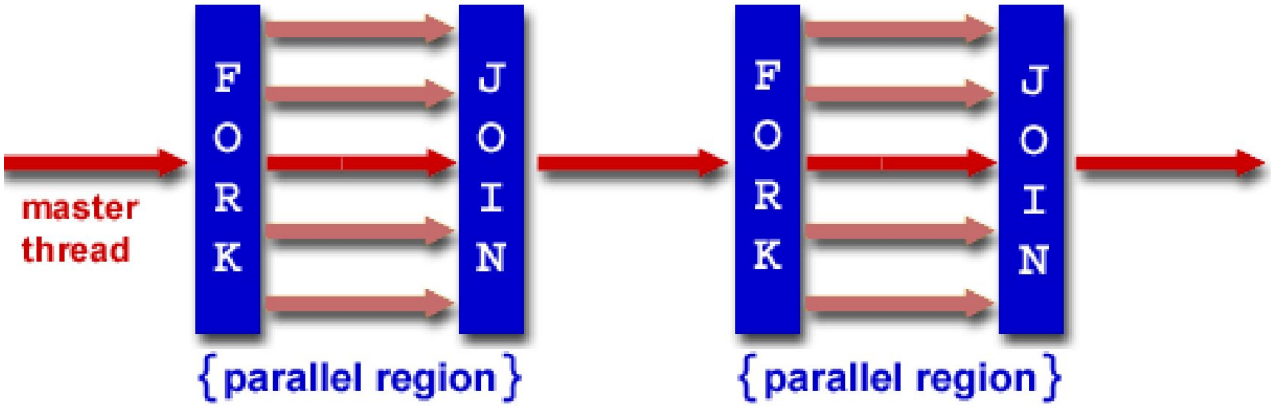
Expressando Concorrência
Sist.Memória Compartilhada

- Comunicações entre processos concorrentes através de acessos a dados em memória
- Modelo *Fork-Join*
  - Diretivas*: fork()*
  - *Threads (Posix Threads* e *Java Threads)*
  - OpenMP
- Outros modelos derivados: *Data Parallel C++ (DPC++), SYCL, OpenAcc, OpenCL*, etc.



Modelo de programação *Fork-Join*

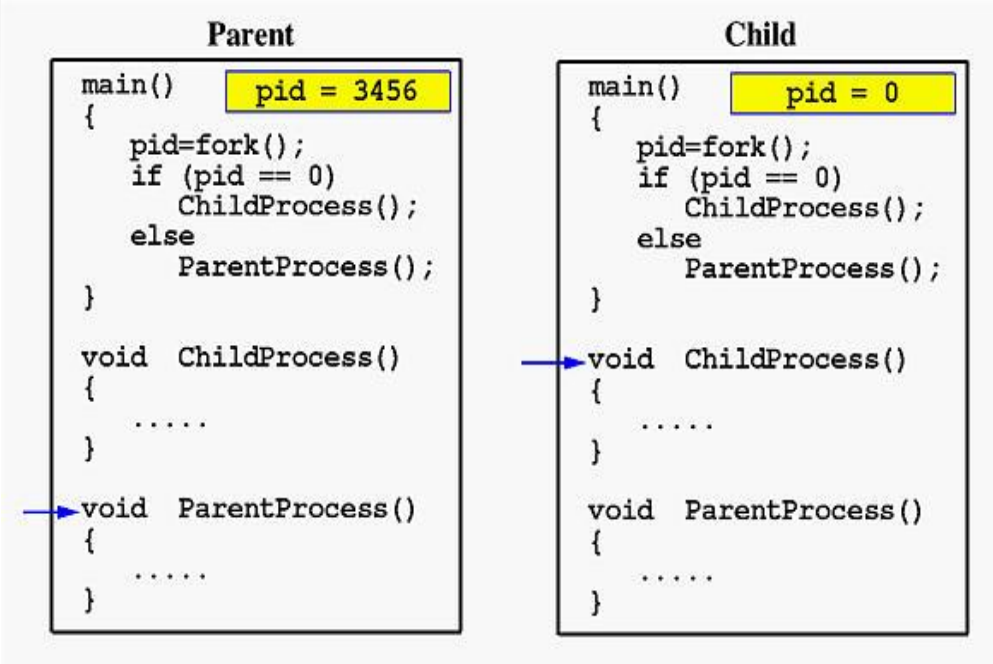
- *Connway, 1963 e Dennis & Van Horn, 1966*
- Um processo cria outros (*fork*)
- Processos executam concorrentemente
- Processos possuem o mesmo espaço de endereçamento
- Processo "pai" aguarda os "filhos" terminarem (*join*)



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()

```c
#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();
}
```

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

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

Saída – Exemplo 1

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

Fonte: POSIX *Threads Programming.* (https://computing.llnl.gov/tutorials/pthreads/#CreatingThreads).

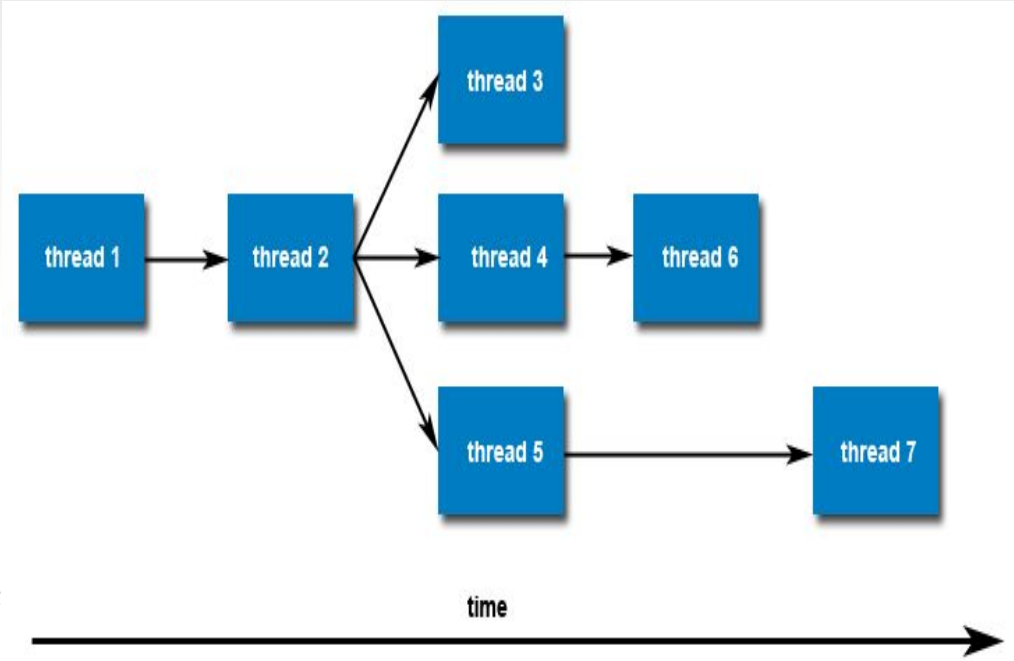
POSIX (*Portable Operating System Interface,* 1985) é uma família de normas definidas pelo IEEE (IEEE 1003), que tem como objetivo garantir a portabilidade do código-fonte em diferentes SOs aderentes a norma. A designação internacional da norma é ISO/IEC 9945. Sumariamente, define APIs para SOs tipo 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 a ser passado para a função `start_routine` como dado de E/S. Referência à um ponteiro do tipo `void`. Pode-se usar `NULL` ou uma `struct` para 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*.
    - Utilizada quando o trabalho da *thread* finalizou-se.
    - Deve ser chamada de dentro da própria *thread*;
  - Uma *thread* pode ser cancelada por outra através de `pthread_cancel()`;
  - O processo 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*

```c
#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
```

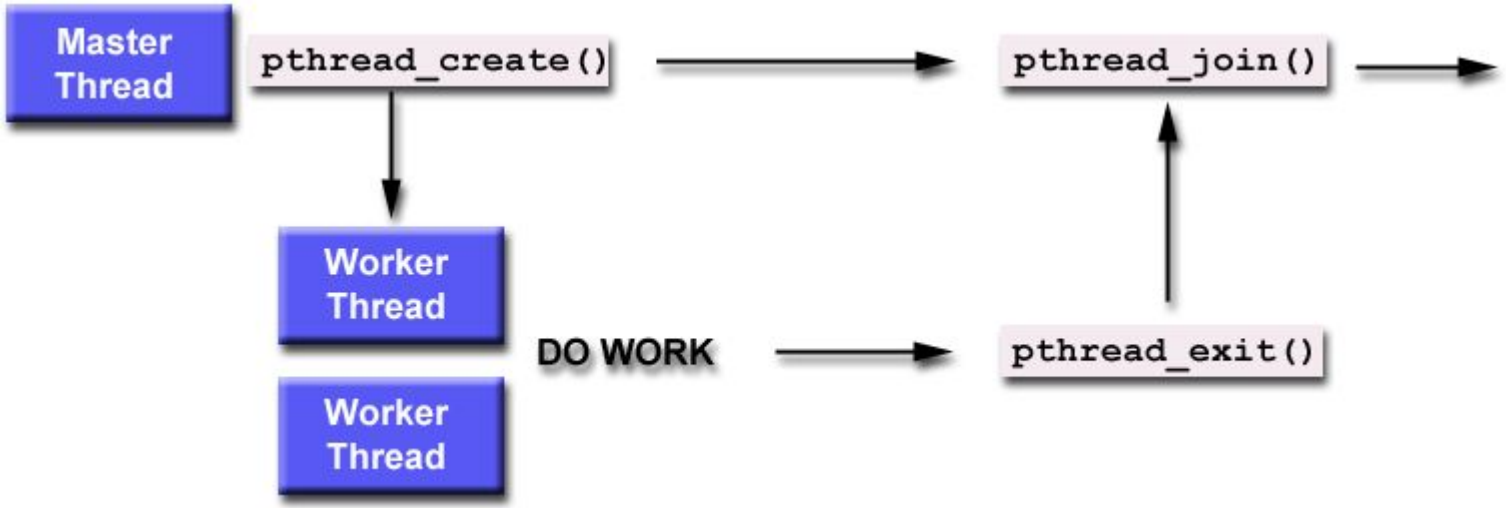
Exemplo 3: Passagem de argumentos

```
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*"



Saída para exemplo 4

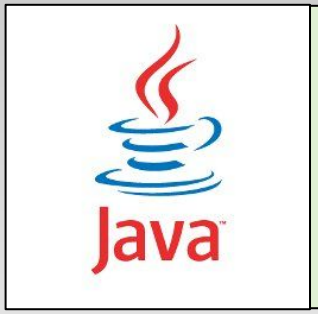
```
./join-simples.x
Main:create thread 0
Main:create thread 1
Thread 0 starting...
Thread 1 starting...
Main:create thread 2
Thread 2 starting...
Main:create thread 3
Thread 3 starting...
Thread 0 done. Result = -3.153838e+06
Main:join thread 0 status=0
Thread 2 done. Result = -3.153838e+06
Thread 1 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) interface `Runnable`;
2) Subclasse de `Thread`.

Interface *Runnable*

- Deve-se definir um método `run` que contenha o código a ser executado na *thread*
- O objeto com interface `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();  }
}
```

https://www.online-java.com/jpPlND6G14

`Runnable` vs `Thread`

- Ambos os casos invocam `Thread.start` para iniciar uma nova *thread*
- Com interface `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 acount) {  // construtor
        count = acount; }

    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(); }
    }
}
```

https://www.online-java.com/E8Z0YL6zTf

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
```