

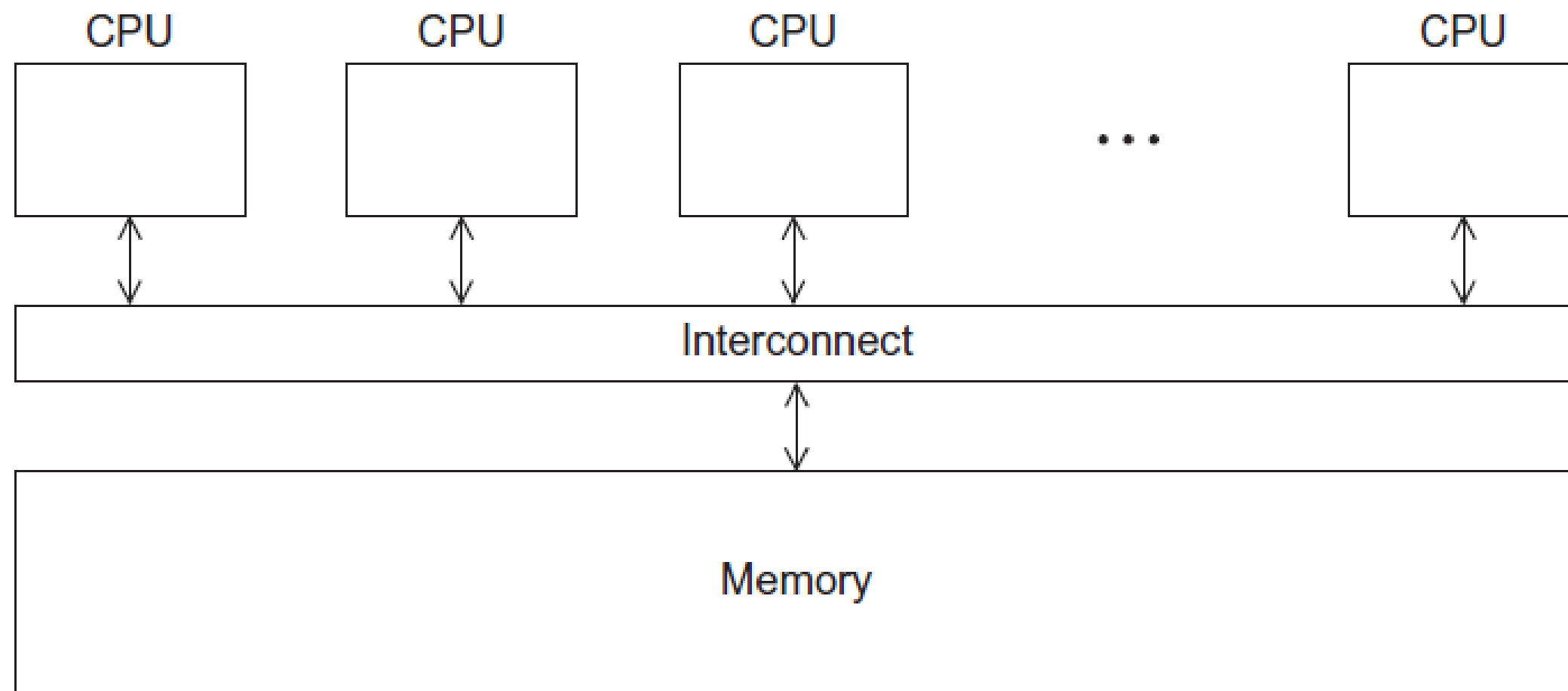
DISCIPLINA

Introdução à Computação Paralela - Pthreads -

Prof. Kayo Gonçalves

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

Relembrando...

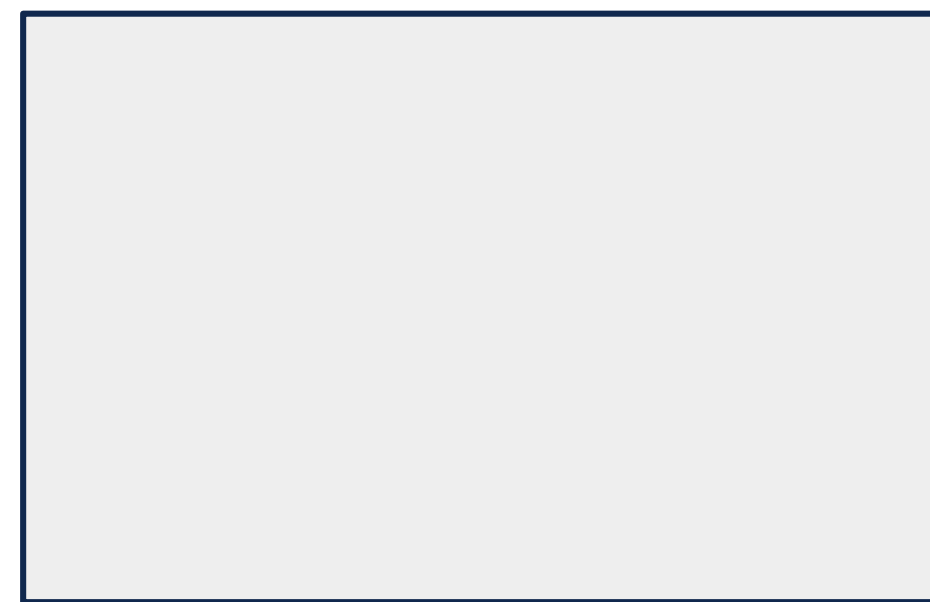


Não precisa mandar mensagem



Relembrando...

- Um processo é uma instância de programa em execução (ou suspensão)
- Threads são análogos a processos “leves”
- Em um programa que utiliza memória compartilhada, um único processo pode ter múltiplas threads



POSIX® Threads

- Conhecido como Pthreads
- Padrão para sistemas baseados em Linux
- Uma biblioteca que pode ser linkada com C/C++
- Especifica uma interface de programação de aplicações (API, application programming interface) para programação multi-thread

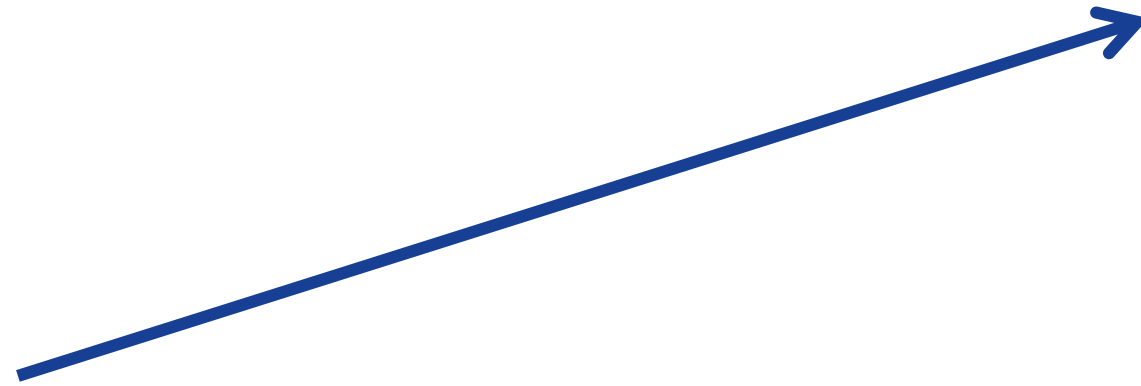


compilar

```
gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

```
g++ -g -Wall -o pth_hello pth_hello.c -lpthread
```

Link para a biblioteca Pthreads



Executar

./pth_hello <número de threads>

./pth_hello 1

Hello from the main thread
Hello from thread 0 of 1

Depende do código
Não obrigatório

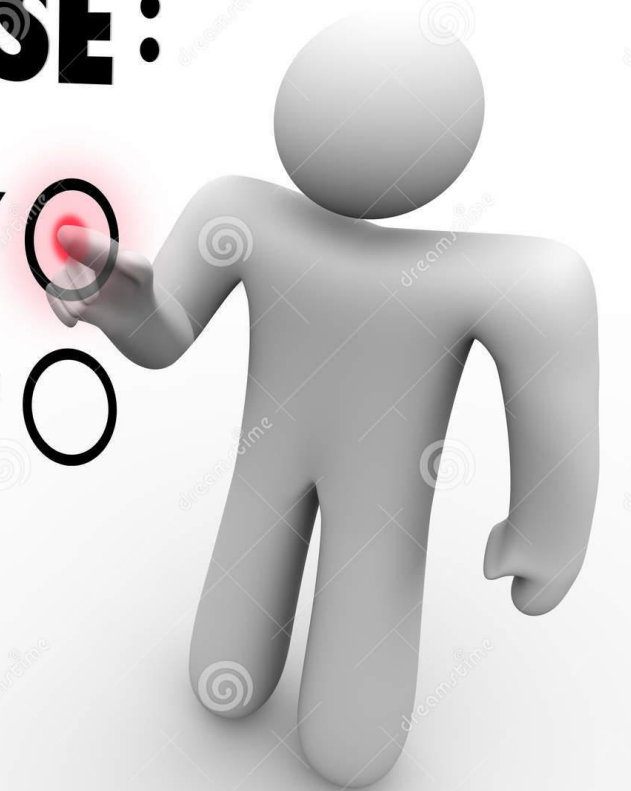
./pth_hello 4

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

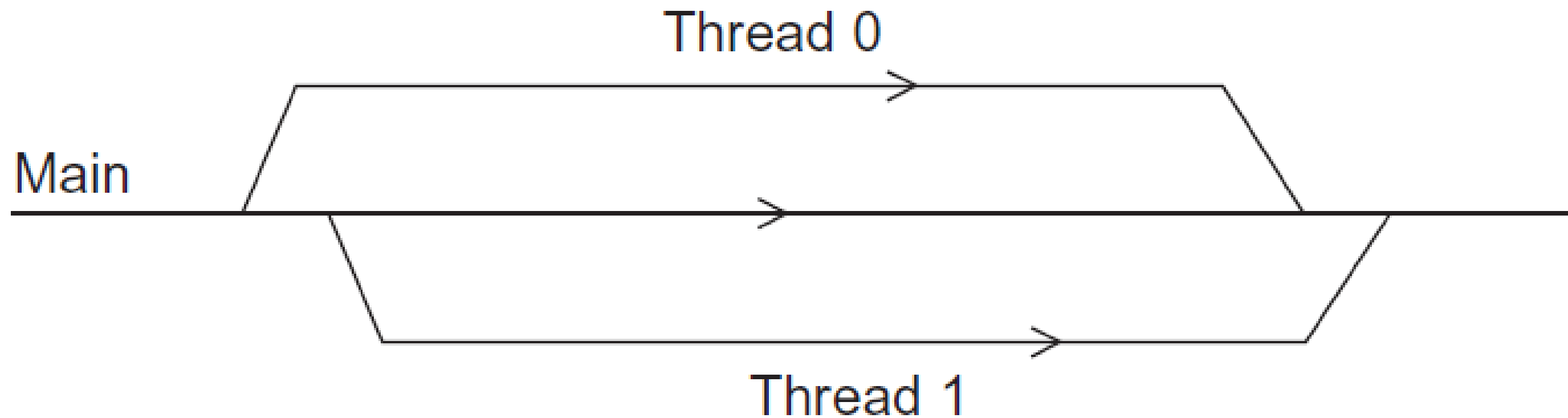
Sempre tem
main+threads
criadas

CHOOSE:

EASY ○
HARD ○



Como são as threads?



Main também é considerado uma thread



Hello World

Direto ao ponto...

- Você irá criar um código em C/C++
- Você especificará uma **única função** que será executada em uma **única thread**
 - Ao criar a thread, você especifica a função
- Teremos a thread **main** (`int main`) e as **criadas** (função que você especificou) ao mesmo tempo.
- As variáveis globais do código são compartilhadas entre **main** e **threads criadas**



Hello World (1)

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

Declara várias funções pthreads, constants, tipos, etc.

```
/* Global variable: accessible to all threads */
int thread_count;
```

```
void *Hello(void* rank); /* Thread function */
```

```
int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;
```

```
/* Get number of threads from command line */
thread_count = strtol(argv[1], NULL, 10);
```

```
thread_handles = malloc (thread_count*sizeof(pthread_t));
```



Hello World (2)

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

```
/* Global variable: accessible to all threads */
int thread_count;
```

```
void *Hello(void* rank); /* Thread function */
```

```
int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;
```

```
/* Get number of threads from command line */
thread_count = strtol(argv[1], NULL, 10);
```

```
thread_handles = malloc (thread_count*sizeof(pthread_t));
```

Variáveis globais (também para as threads)

Nº da thread



Hello World (3)

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

```
/* Global variable: accessible to all threads */
int thread_count;
```

```
void *Hello(void* rank); /* Thread function */
```

```
int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;
```

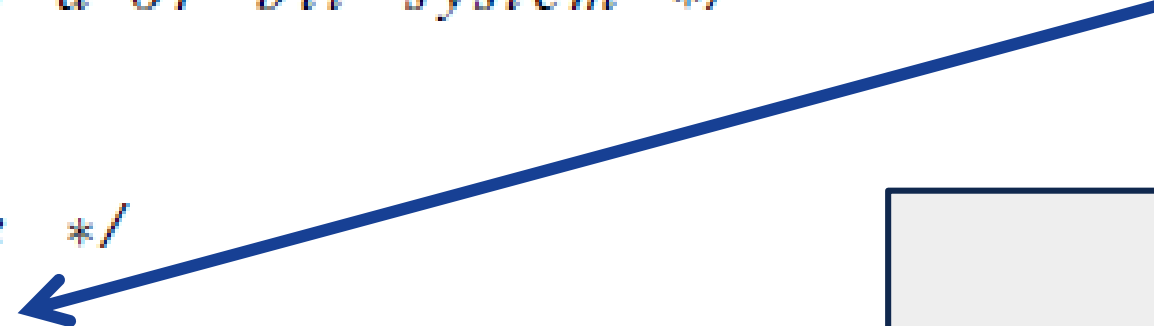
```
/* Get number of threads from command line */
thread_count = strtol(argv[1], NULL, 10);
```

```
thread_handles = malloc (thread_count*sizeof(pthread_t));
```

Ponteiro para as threads



Resgata número de threads
do argumento passado pelo
terminal (opcional)



Hello World (4)

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

/* Global variable: accessible to all threads */
int thread_count;

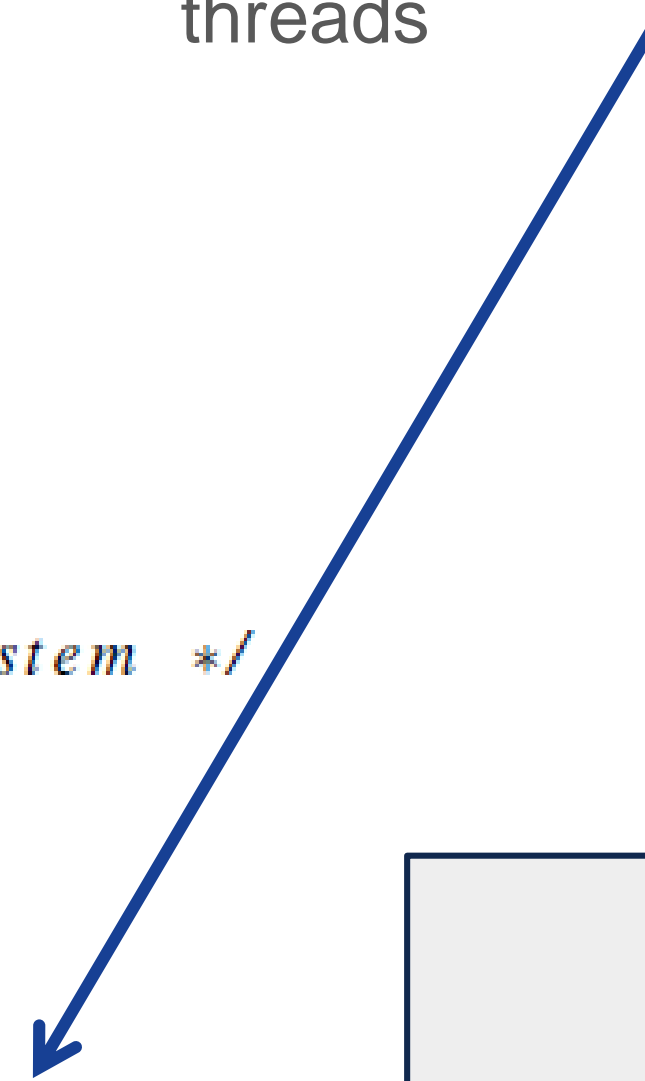
void *Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

    /* Get number of threads from command line */
    thread_count = strtol(argv[1], NULL, 10);

    thread_handles = malloc (thread_count*sizeof(pthread_t));
```

Alocação espaço na
memória para o ponteiro das
threads



Hello World (5)

```
for (thread = 0; thread < thread_count; thread++)  
    pthread_create(&thread_handles[thread], NULL,  
                  Hello, (void*) thread);  
  
printf("Hello from the main thread\n");  
  
for (thread = 0; thread < thread_count; thread++)  
    pthread_join(thread_handles[thread], NULL);  
  
free(thread_handles);  
return 0;  
} /* main */
```

Main cria as threads. Cada ponteiro do vetor de threads irá referenciar uma thread.

NULL está no campo de atributo. Leia mais.

Hello é o nome da função

(void*) thread é o parâmetro da função **Hello** com casting para void

Hello World (6)

```
for (thread = 0; thread < thread_count; thread++)  
    pthread_create(&thread_handles[thread], NULL,  
                  Hello, (void*) thread);  
  
printf("Hello from the main thread\n");  
  
for (thread = 0; thread < thread_count; thread++)  
    pthread_join(thread_handles[thread], NULL);  
  
free(thread_handles);  
return 0;  
} /* main */
```

Estão rodando em
paralelo **main** e as
threads criadas

Hello World (7)

```
for (thread = 0; thread < thread_count; thread++)  
    pthread_create(&thread_handles[thread], NULL,  
        Hello, (void*) thread);
```

```
printf("Hello from the main thread\n");
```

← Main escreve.

```
for (thread = 0; thread < thread_count; thread++)  
    pthread_join(thread_handles[thread], NULL);
```

← Main espera a finalização das threads para desalocá-las uma por vez.

Pthread_join é bloqueante

```
free(thread_handles);  
return 0;  
} /* main */
```

← Main libera o vetor alocado dinamicamente

Hello World (8)

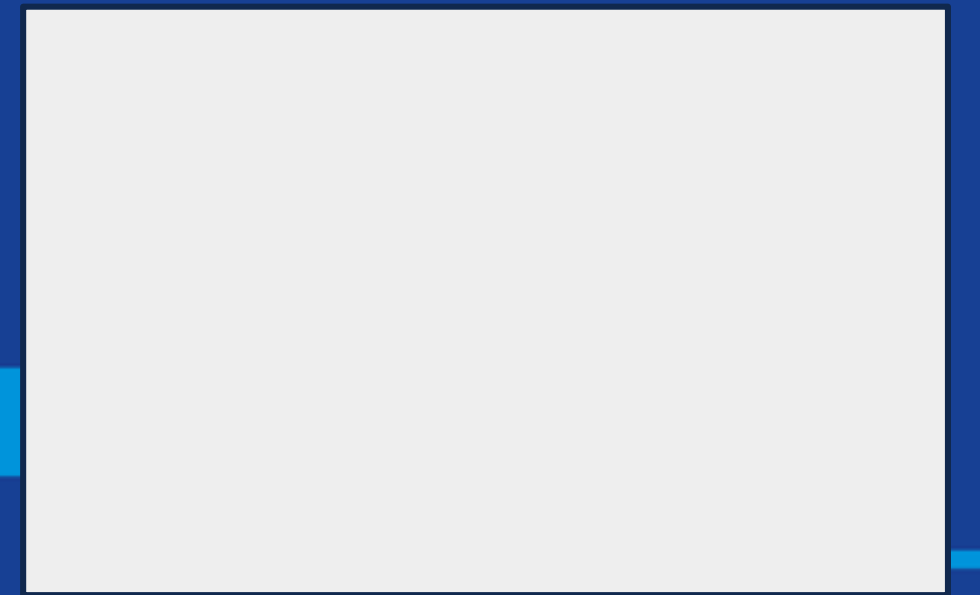
```
void *Hello(void* rank) {  
    long my_rank = (long) rank;  /* Use long in case of 64-bit system */  
  
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);  
  
    return NULL;  
} /* Hello */
```

← Thread escreve.

Thread_count é variável global

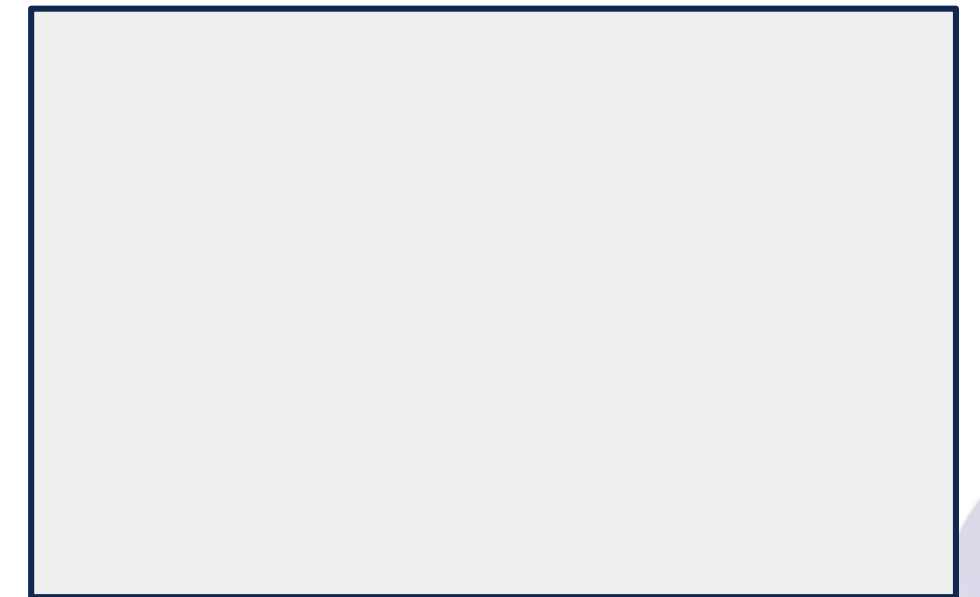
← Thread finaliza sua execução.
Retorno esperado pelo main em
pthread_join(.)

Variáveis Compartilhadas



Variáveis Compartilhadas

- Podem introduzir erros sutis e confusos que não existiam em memória distribuída
- Minimize o uso de variáveis compartilhadas.
- Minimize escrita de 2 ou mais threads (incluindo **main**) em variáveis compartilhadas
 - Se for necessário, **garanta** 1 escrita por vez



Exemplo de problema/erro

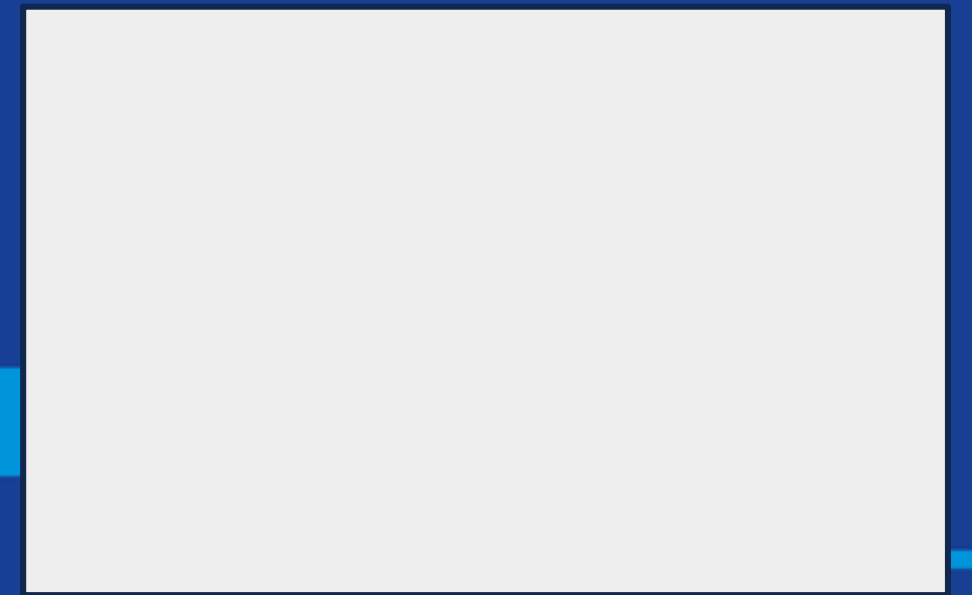
```
1 void *incrementa_contador(void* desnecessario) {  
2     // "contador" e "n" são variáveis compartilhadas  
3     for (int i=0; i<n; i++) {  
4         contador += 1;  
5     }  
6     return NULL;  
7 }
```

A **thread A** pode resgatar um valor (ex 30) antes mesmo da **thread B** ter atualizado o seu valor (ex 31).

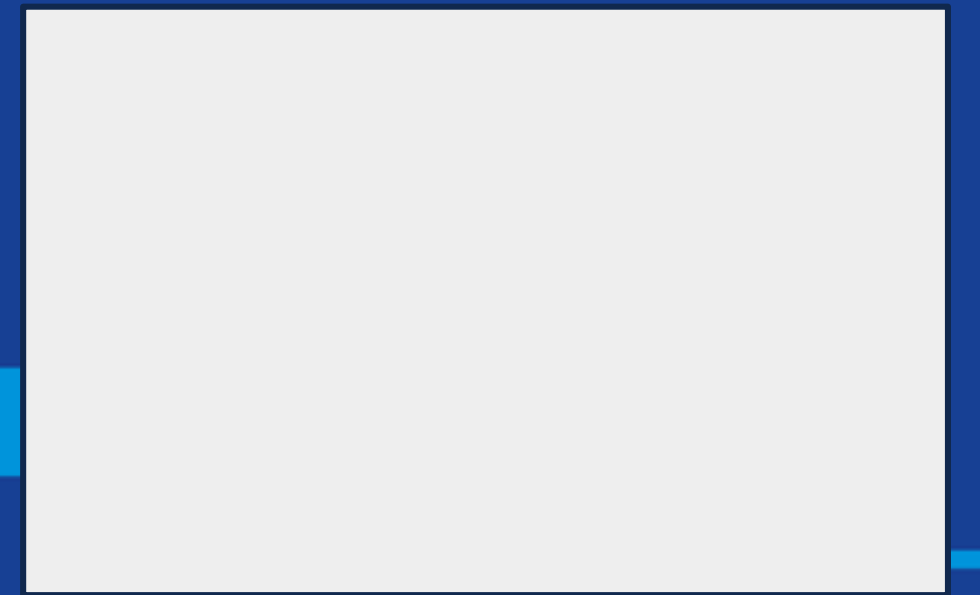
- **Thread B** guardará na memória o valor 31
- **Thread A** também
- O valor correto seria 32

***Repetindo: Minimize
leitura+escrita de 2 ou mais
threads (incluindo main) em
variáveis compartilhadas***

Dica de ouro, platina, diamante, mestre, grão-mestre, desafiante



Multiplicação de Matrizes



Multiplicação de Matrizes

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
 x_1
 \vdots
 x_{n-1}

=

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

Multiplicação de Matrizes

Código Serial

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0	y_0
x_1	y_1
\vdots	\vdots
\vdots	$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots	\vdots
x_{n-1}	y_{m-1}



Multiplicação de Matrizes (1)

Código Paralelo

- Suponha matriz $\mathbf{a}_{6 \times 6}$ e $\mathbf{x}_{6 \times 1}$, resultado em $\mathbf{y}_{6 \times 1}$
- Utilizaremos 3 threads (de 0 a 2)
- Não utilizaremos **main** na multiplicação (facilitar implementação)
- Cada **thread** computa o resultado de 2 linhas de \mathbf{y}

Thread	Components of \mathbf{y}
0	$y[0], y[1]$
1	$y[2], y[3]$
2	$y[4], y[5]$



Multiplicação de Matrizes (2)

Código Paralelo

Thread	Components of y
0	$y[0], y[1]$
1	$y[2], y[3]$
2	$y[4], y[5]$

Thread 0: y_0

```
y[0] = 0.0;  
for (j = 0; j < n; j++)  
    y[0] += A[0][j]* x[j];
```

Caso geral da linha y_i

```
y[i] = 0.0;  
for (j = 0; j < n; j++)  
    y[i] += A[i][j]* x[j];
```

Multiplicação de Matrizes (3)

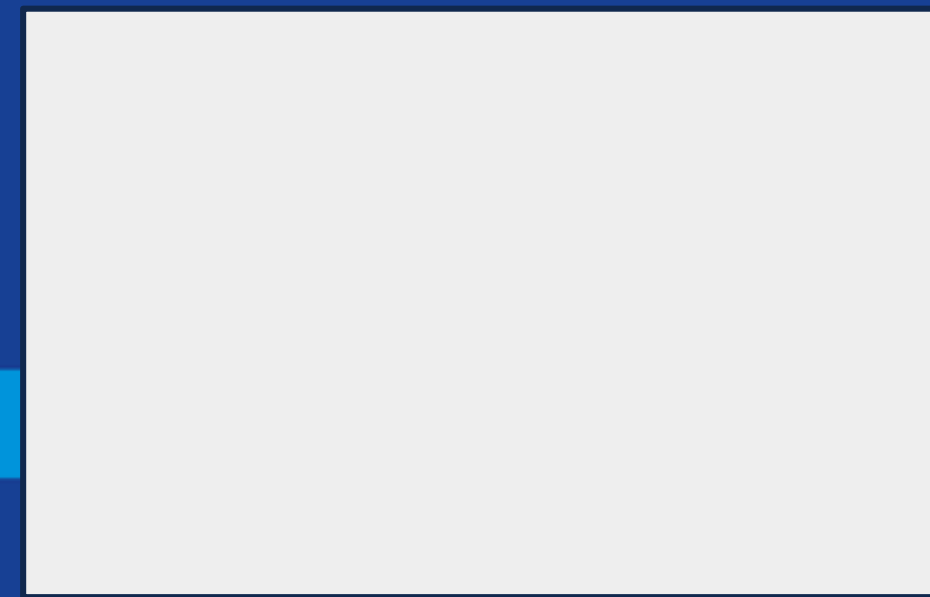
Código Paralelo

```
void *Pth_mat_vect(void* rank) {  
    long my_rank = (long) rank;  
    int i, j;  
    int local_m = m/thread_count;  
    int my_first_row = my_rank*local_m;  
    int my_last_row = (my_rank+1)*local_m - 1;  
  
    for (i = my_first_row; i <= my_last_row; i++) {  
        y[i] = 0.0;  
        for (j = 0; j < n; j++)  
            y[i] += A[i][j]*x[j];  
    }  
  
    return NULL;  
} /* Pth_mat_vect */
```



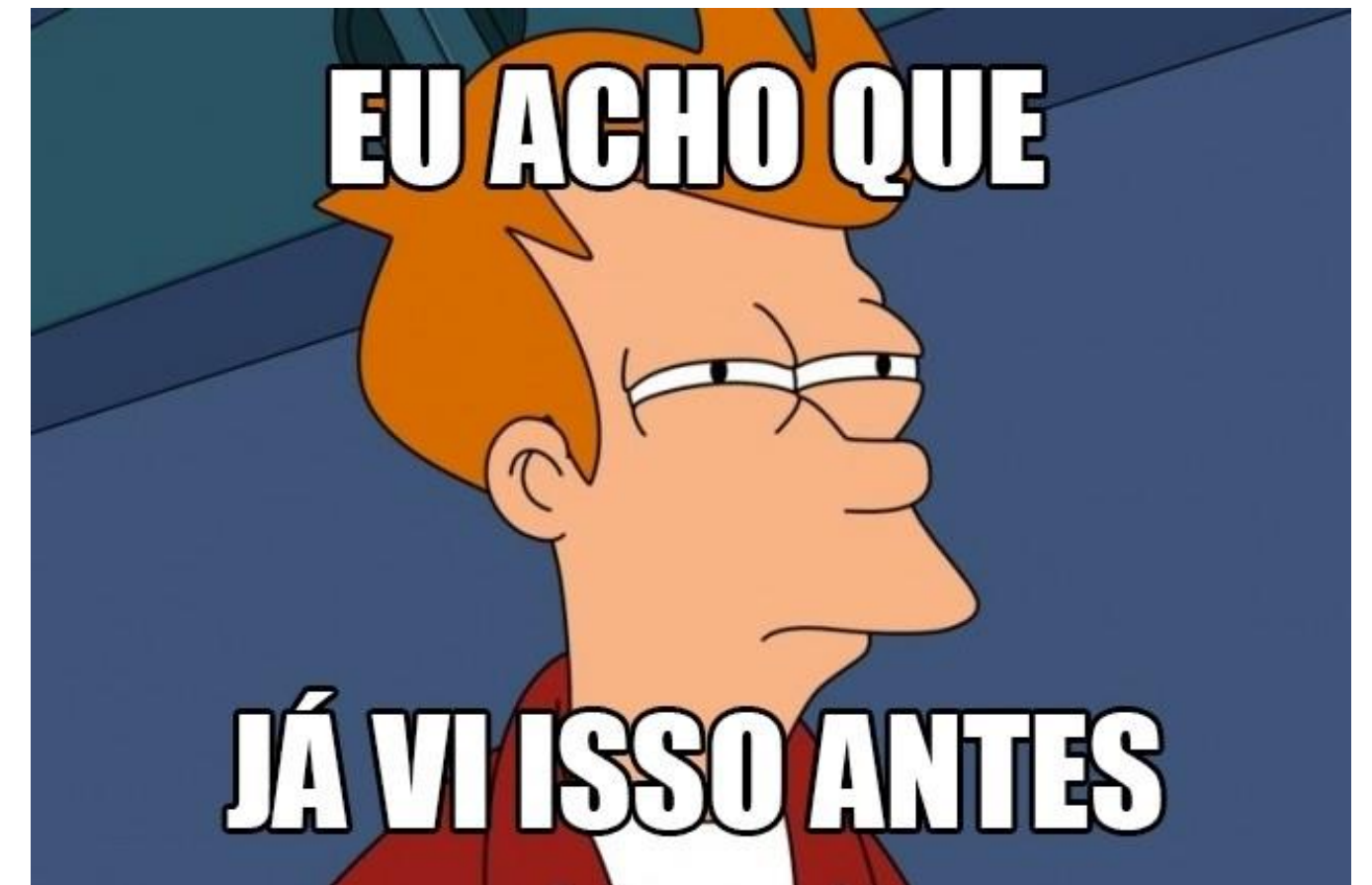
ô loco
meu!

Região Crítica



Condição de corrida (1)

- É o comportamento do software em que o resultado do processo é inesperadamente dependente da sequência ou do tempo de **outros eventos**.
- Exemplo: Execute **soma+=1** em um loop com soma como uma variável compartilhada



Condição de corrida (2)

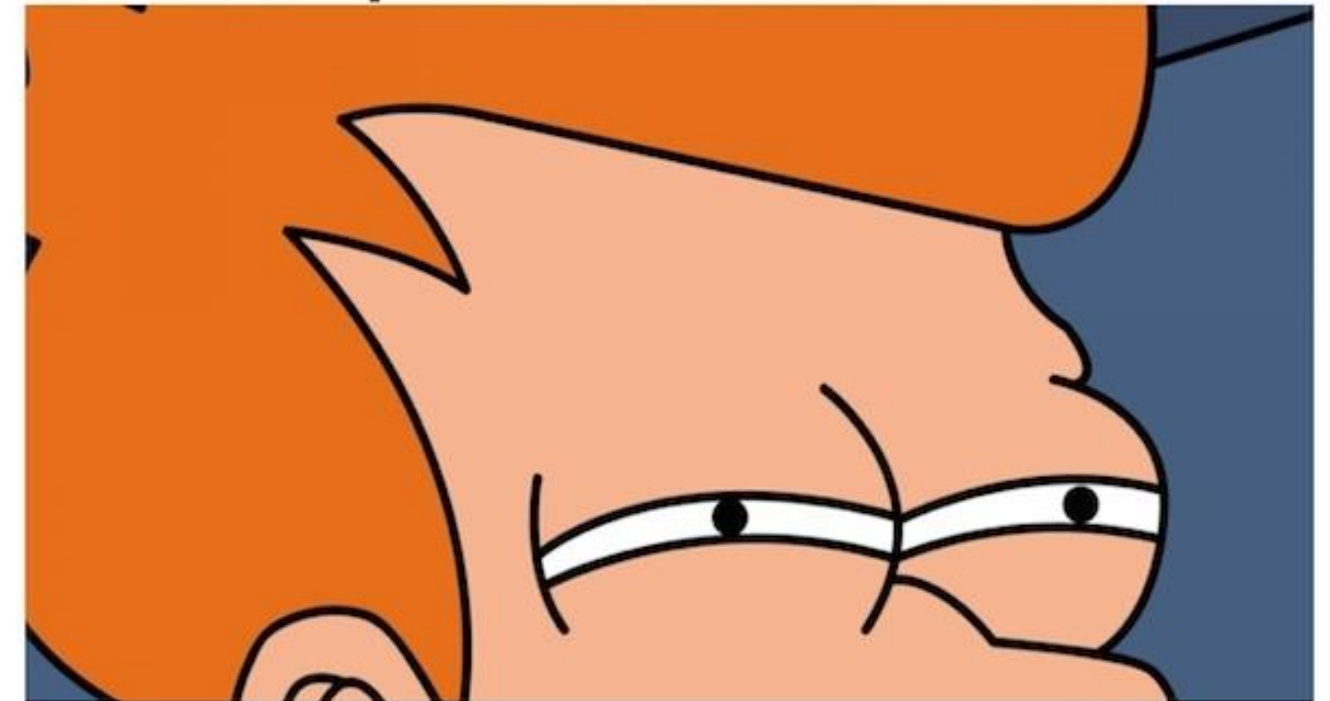
- Utilizamos multitarefa, pipeline e multicore
- Realizar **soma+=1** requer
 - Busca de instrução
 - Busca de operandos da memória
 - Execução da soma
 - Escreve resultado na memória



Região Crítica (1)

- Uma **região crítica** é um segmento do código que acessa variáveis compartilhadas e tem que ser executada **atômicamente**.
- Exemplo: O próprio **soma+=1**

Eu acho que



já vi isso antes...



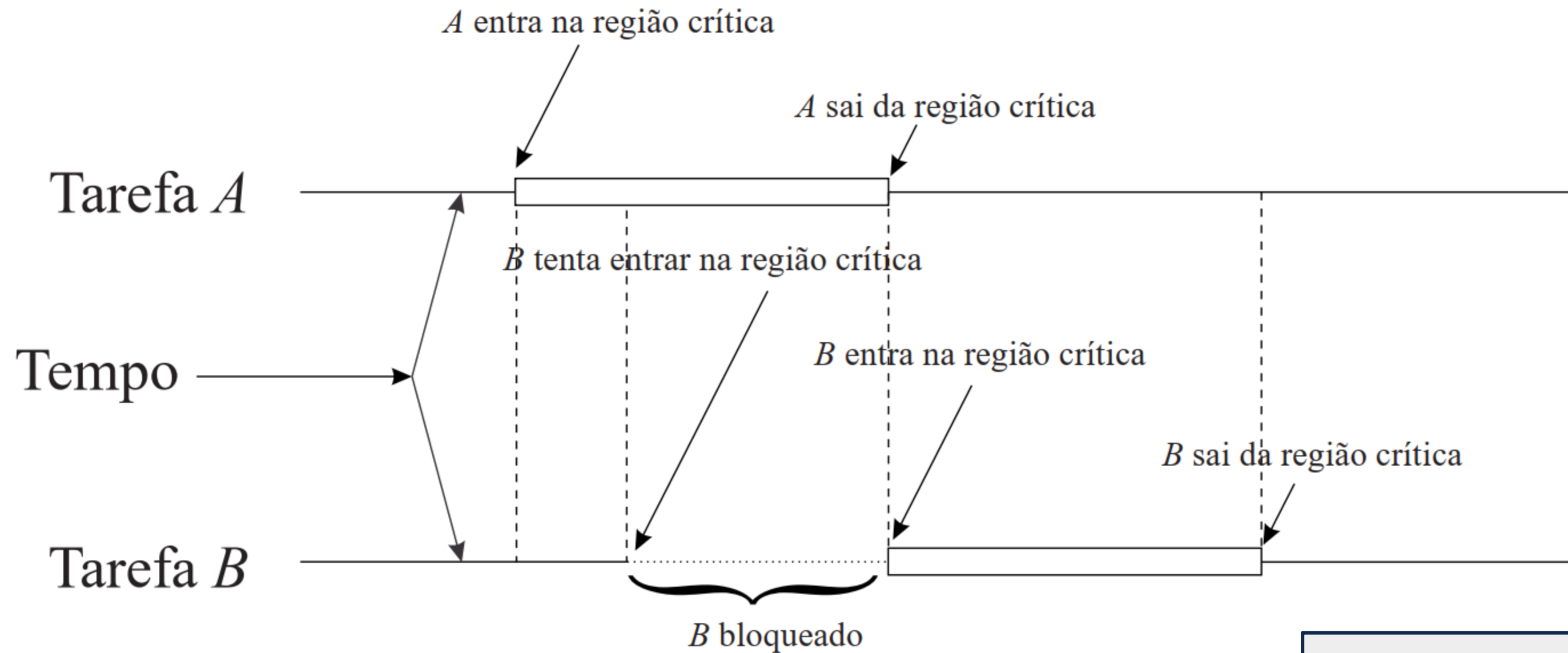
Região Crítica (2)

- Execução **soma+=1** tem que ser atômica
 - Busca de instrução
 - Busca de operandos da memória
 - Execução da soma
 - Escreve resultado na memória
- Como implementar? Exclusão mútua

Começou?
Termine.

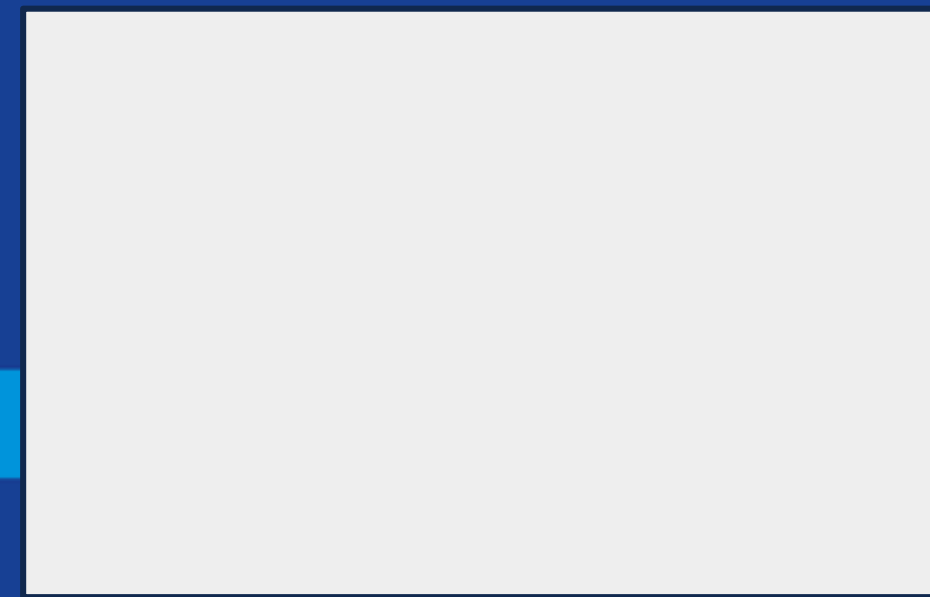


Exclusão Mútua



- Busy-Waiting (espera ocupada) – não comumente utilizada
- Semáforo
- Mutex (caso especial do semáforo)

Exclusão Mútua



Exclusão Mútua por Exemplos (1)

Vamos Estimar PI utilizando a fórmula de Leibniz

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots + (-1)^n \frac{1}{2n+1} + \dots \right)$$

Exclusão Mútua por Exemplos (2)

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots + (-1)^n \frac{1}{2n+1} + \dots \right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

Solução – Sem Exclusão Mútua (1)

Código paralelo

Cada **thread** irá executar “**n** dividido por **nº de threads**” operações (somas)

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots + (-1)^n \frac{1}{2n+1} + \dots \right)$$

Solução – Sem Exclusão Mútua (2)

Código paralelo

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)  /* my_first_i is even */
        factor = 1.0;
    else /* my_first_i is odd */
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }

    return NULL;
} /* Thread_sum */
```

Solução – Sem Exclusão Mútua (2)

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Note que quanto maior n , a estimativa com uma thread se torna melhor (e não com 2 threads).

O que aconteceu?

Solução – Sem Exclusão Mútua (3)

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0)  /* my_first_i is even */  
        factor = 1.0;  
    else  /* my_first_i is odd */  
        factor = -1.0;  
  
    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {  
        sum += factor/(2*i+1);  
    }  
  
    return NULL;  
}  /* Thread_sum */
```

Condição de corrida
sem exclusão mútua



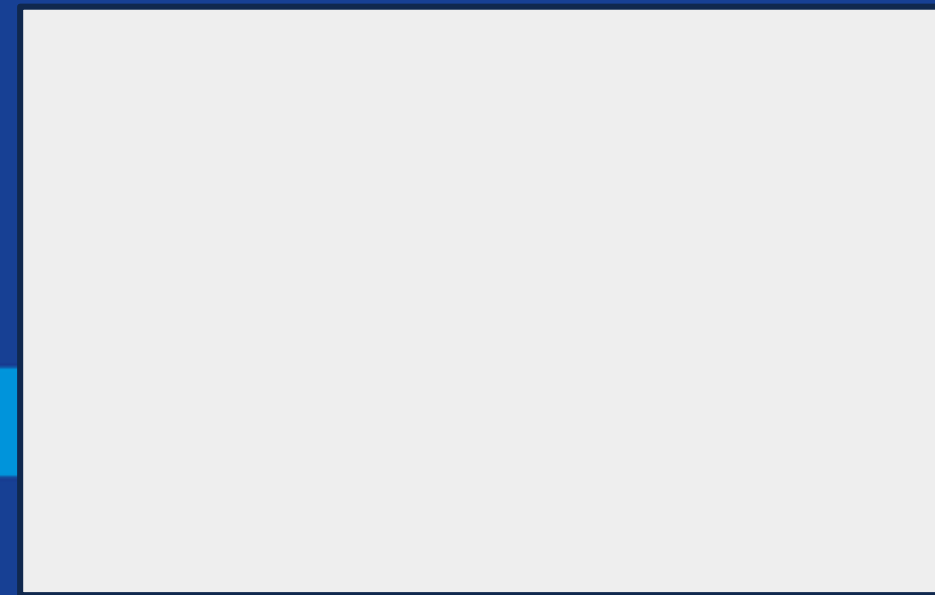
Solução – Sem Exclusão Mútua (3)

Na verdade, o código anterior não é solução do problema. O Cálculo do PI está errado e não segue a fórmula

Temos que corrigir o código.



Espera Ocupada



Espera Ocupada

- Uma **thread** testa repetidamente uma condição, mas, efetivamente, não realiza trabalho útil até a condição ter um valor apropriado
- Tome cuidado com compiladores otimizadores!

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

flag é inicializada com 0 pela
thread **main**



PI - Solução 1 – EO dentro do loop FOR

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank); ← Espera Ocupada
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```

PI - Solução 2 – EO fora do loop FOR (1)

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor, my_sum = 0.0; ← my_sum não é compartilhada  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;
```

PI - Solução 2 – EO fora do loop FOR (2)

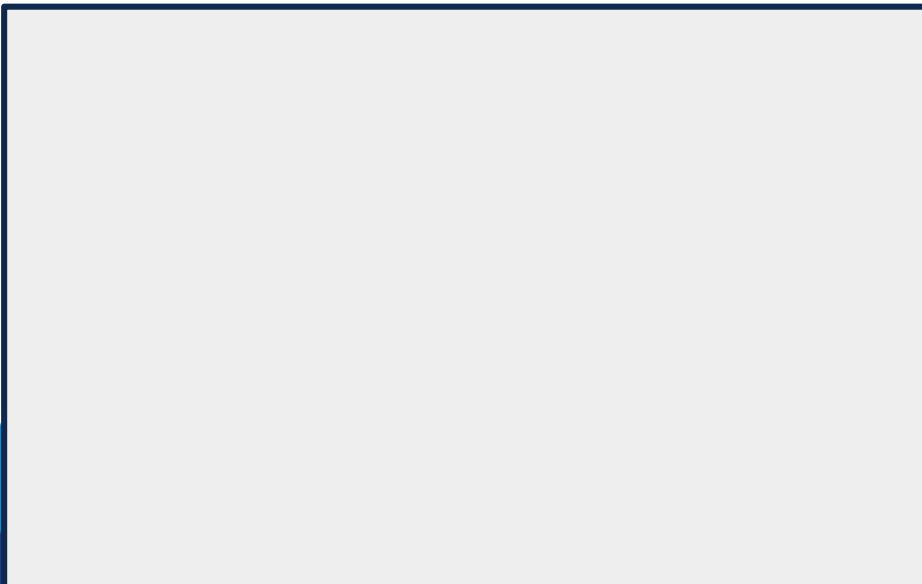
```
for (i = my_first_i; i < my_last_i; i++, factor = -factor)
    my_sum += factor/(2*i+1);

while (flag != my_rank);
sum += my_sum;
flag = (flag+1) % thread_count;

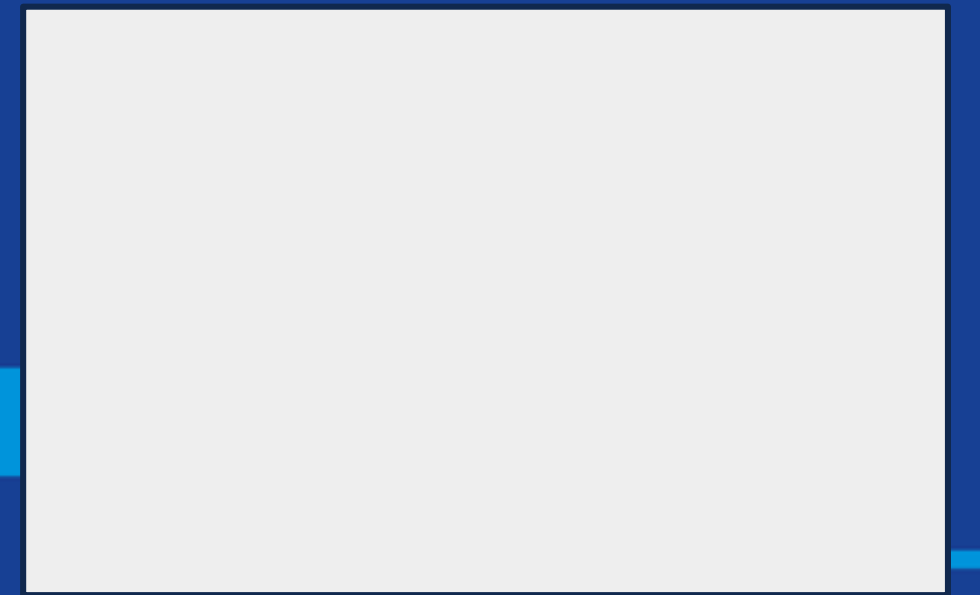
return NULL;
} /* Thread_sum */
```

Não há condição de corrida porque **my_sum** não é compartilhada

Espera Ocupada



Mutex



Mutex (1)

- Uma **thread** executando espera-ocupada pode continuamente utilizar a CPU e não realizar computação útil alguma.
- Mutex (**M**utual **E**xclusion – Exclusão Mútua) é um tipo especial de variável que pode ser utilizada para restringir acesso a uma região crítica a uma **única thread** por vez.



Mutex (2)

- Você pode LOCK (travar) ou UNLOCK (destravar)
 - **LOCK:** Ninguém pode executar este trecho de código, exceto eu
 - **UNLOCK:** Trecho de código liberado. Todos podem tentar LOCK
- Linhas entre LOCK e UNLOCK são uma região crítica com exclusão mútua



Mutex (3)

- O padrão pthreads inclui tipos de mutex: **pthread_mutex_t**

```
int pthread_mutex_init(  
    pthread_mutex_t*      mutex_p    /* out */  
    const pthread_mutexattr_t* attr_p /* in */);
```

- Para finalizar mutex, utilize

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```

Mutex (4)

- Para LOCK:

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

- Para UNLOCK:

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```



PI - Solução 1 – Mutex (1)

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
    double my_sum = 0.0;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;
```

My_sum não é compartilhada

PI - Solução 1 – Mutex (2)

```
for (i = my_first_i; i < my_last_i; i++, factor = -factor) {  
    my_sum += factor/(2*i+1);  
}  
pthread_mutex_lock(&mutex);  
sum += my_sum;  
pthread_mutex_unlock(&mutex);  
  
return NULL;  
} /* Thread_sum */
```

Não há condição de corrida porque **my_sum** não é compartilhada

Região crítica com mutex

Resultados Comparativos

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

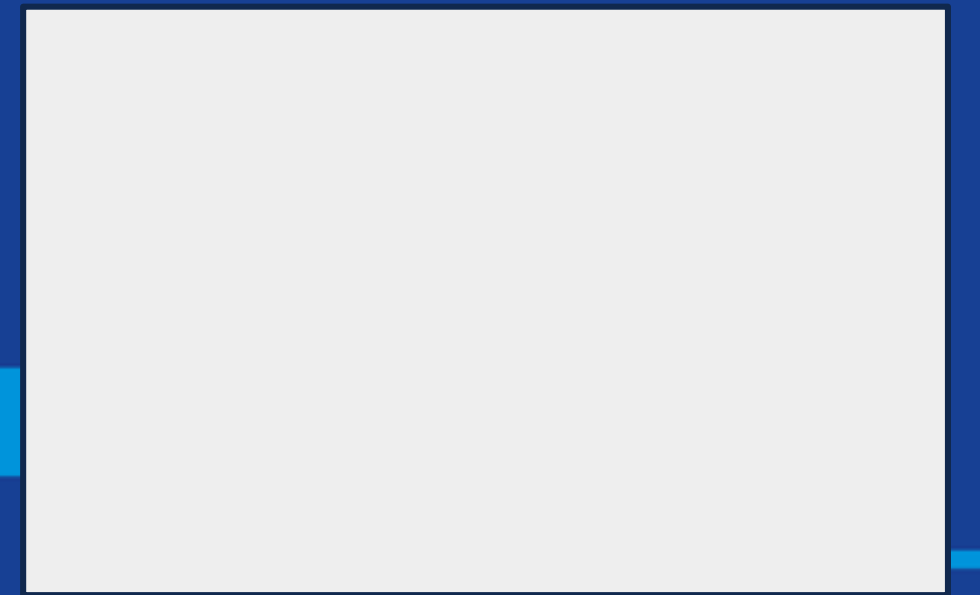
Melhoria no tempo de execução

Não há melhoria no tempo de execução

O tempo de execução (em segundos) para PI utilizando $n=108$ termos em um sistema 2 quad-cores (total de 8 cores)



Semáforos



Semáforo (1)

- Mecanismo que permite que **n thread** acessem região crítica.
- Mutex é um caso especial de semáforo com contador 1
- É um inteiro não negativo que pode realizar duas operações (além de inicializar e destruir)
 - P ou DOWN: reduz o número (como **LOCK**)
 - V ou UP: incrementa o número (como **UNLOCK**)
- Estas instruções são atômicas



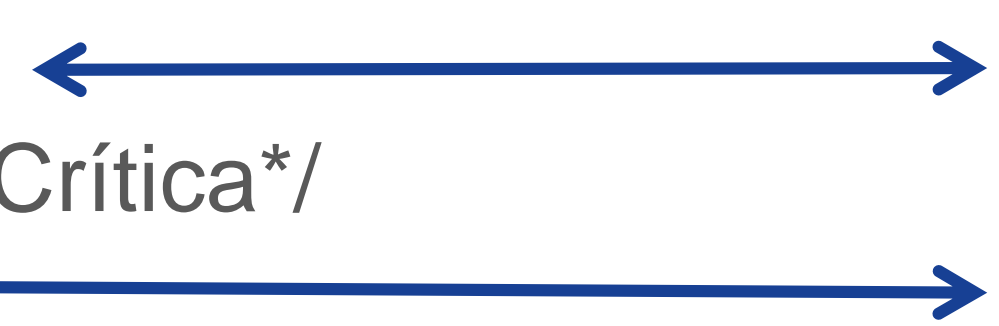
Semáforo (2)

- Seja s um semáforo
 - Down(s): se $s > 0$, reduza s . Caso contrário, bloqueie a thread
 - Up(s): Se existe thread(s) bloqueada(s) em s , libere uma delas. Caso contrário, incremente s .
 - Não é especificado qual thread é liberada



Semáforo (3)

<pre>void p1(void) { for (;;) { Down(s1) /*Região Crítica*/ Up(s1) ... } ... }</pre>	<pre>void p2(void) { for (;;) { Down(s1) /*Região Crítica*/ Up(s1) ... } ... }</pre>
--	--



Funções de threads diferentes, mas bloqueiam a mesma região crítica

Semáforo (4)

Função	Descrição
sem_init	Inicialização
sem_wait	Down
sem_trywait	Down não bloqueante. Tenta DOWN e retorna BOOL dizendo se conseguiu (TRUE) ou não (FALSE)
sem_post	Up
sem_get_value	Retorna o valor do contador do semáforo
sem_destroy	Destrói o semáforo

Semáforo (5)

Semáforo não faz parte de Pthreads. Precisa adicionar esta biblioteca

```
#include <semaphore.h>
int sem_init(
    sem_t*      semaphore_p    /* out */,
    int         shared         /* in  */,
    unsigned    initial_val    /* in  */);
```



Semáforo (5)

```
#include <semaphore.h>

int  sem_init      (sem_t *sem, int pshared, unsigned int value)
int  sem_wait      (sem_t *sem)
int  sem_trywait   (sem_t *sem)
int  sem_post      (sem_t *sem)
int  sem_getvalue  (sem_t *sem, int *sval)
int  sem_destroy   (sem_t *sem)
```

Se **pshared == 0**, semáforo compartilhado entre threads de um mesmo processo

Se **pshared != 0**, semáforo compartilhado entre processos.

Value: Contador. Para mutex, **value=1**.

Exemplo 1

```
#include <semaphore.h>

...
sem_t slots;
...
status = sem_init(&slots, 0, 10);
...
status = sem_wait(&slots);
...
status = sem_post(&slots);
...
```

Exemplo 2

```
// C program to demonstrate working of Semaphores
```

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

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

```
#include <semaphore.h>
```

```
#include <unistd.h>
```

```
sem_t mutex;
```

```
void* thread(void* arg)
```

```
{
```

```
    //wait
```

```
    sem_wait(&mutex);
```

```
    printf("\nEntered..\n");
```

```
    //critical section
```

```
    sleep(4);
```

```
    //signal
```

```
    printf("\nJust Exiting...\n");
```

```
    sem_post(&mutex);
```

```
}
```

```
int main()
```

```
{
```

```
    sem_init(&mutex, 0, 1);
```

```
    pthread_t t1,t2;
```

```
    pthread_create(&t1,NULL,thread,NULL);
```

```
    sleep(2);
```

```
    pthread_create(&t2,NULL,thread,NULL);
```

```
    pthread_join(t1,NULL);
```

```
    pthread_join(t2,NULL);
```

```
    sem_destroy(&mutex);
```

```
    return 0;
```

```
}
```

The background is a solid dark blue. It features several horizontal bars of a lighter blue color. There are two bars at the top left, one at the top right, and two at the bottom left. A single bar is at the bottom center, partially overlapping a white rectangular box in the bottom right corner. The word "Barreira" is centered in the middle of the image in a white, bold, sans-serif font.

Barreira

Barreira

- Espera um certo número de threads executar a barreira. Enquanto este número não é alcançado, as threads que executaram a barreira permanecem bloqueadas. Quando alcançado o número, todas as threads são liberadas.
- Precisa criar a barreira e inicializá-la.



Exemplo (1)

```
42 main () // ignora argumentos
43 {
44     time_t    now;
45     char      buf [27];
46
47     // Cria uma barreira com contador de 3
48     pthread_barrier_init (&barrier, NULL, 3);
49
50     // Inicializa 2 threads: thread 1 e thread 2
51     pthread_create (NULL, NULL, thread1, NULL);
52     pthread_create (NULL, NULL, thread2, NULL);
53
54     // Neste momento, thread1 e thread2 estão executando
55
56     // Agora espera pela finalização
57     time (&now);
58     printf ("main () esperando na barreira em %s", ctime_r (&now, buf));
59     pthread_barrier_wait (&barrier);
60
61     // Após este ponto, todas as 3 threads foram concluídas
62     time (&now);
63     printf ("Barreira em main() finalizada em %s", ctime_r (&now, buf));
64 }
```

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <pthread.h>
4
5 pthread_barrier_t    barrier; // Barreira
```

Exemplo (2)

```
7 void * thread1 (void *not_used)
8 {
9     time_t now;
10    char buf [27];
11
12    time (&now);
13    printf ("thread1 começando em %s", ctime_r (&now, buf));
14
15    // Faça computação
16    // Vamos apenas dormir aqui...
17    sleep (20);
18    pthread_barrier_wait (&barrier);
19    // Após este ponto, todas as 3 threads foram concluídas
20    time (&now);
21    printf ("Barreira em thread1() finalizada em %s", ctime_r (&now, buf));
22 }
```

Exemplo (3)

```
24 void * thread2 (void *not_used)
25 {
26     time_t  now;
27     char    buf [27];
28
29     time (&now);
30     printf ("thread2 começando em %s", ctime_r (&now, buf));
31
32     // Faça computação
33     // Vamos apenas dormir aqui...
34     sleep (40);
35     pthread_barrier_wait (&barrier);
36     // Após este ponto, todas as 3 threads foram concluídas
37     time (&now);
38     printf ("Barreira em thread2() finalizada e %s", ctime_r (&now, buf));
39 }
```

