



# TÓPICOS ESPECIAIS – SISTEMAS EMBARCADOS

## *PROGRAMAÇÃO PARALELA E TEMPO REAL*

PROF. JOSENALDE OLIVEIRA

[josenalde.oliveira@ufrn.br](mailto:josenalde.oliveira@ufrn.br)

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - UFRN

# Questões de não determinismo e sincronismo

- Onde unidades de processamento (UP) podem executar assincronamente, é provável ocorrer não determinismo.
- Uma computação é não determinística se pode gerar saídas diferentes para uma mesma entrada. Se múltiplas threads executam de forma independente, a velocidade com que concluem tarefas varia de execução para execução e dependem de interações como sistema operacional

- Exemplo: thread id=0, com my\_x (privada) previsto = 7

thread id=1, com my\_x (privada) previsto = 19

- Supondo que cada thread execute o código `cout << id << "my_val: " << my_x;`
  - A saída poderá ser: `0 my_val: 7; 1 my_val: 19` OU `1 my_val: 19; 0 my_val: 7`
- Outro exemplo: cada thread calcula um inteiro e armazena em my\_val. É objetivo também adicionar este valor a uma variável compartilhada x. Portanto, cada thread executa algo assim:

```
my_val = computeVal(my_rank);  
x += my_val;
```

- Sabemos que uma adição exige carregar operandos da memória, colocar em registradores, e depois armazenar o resultado

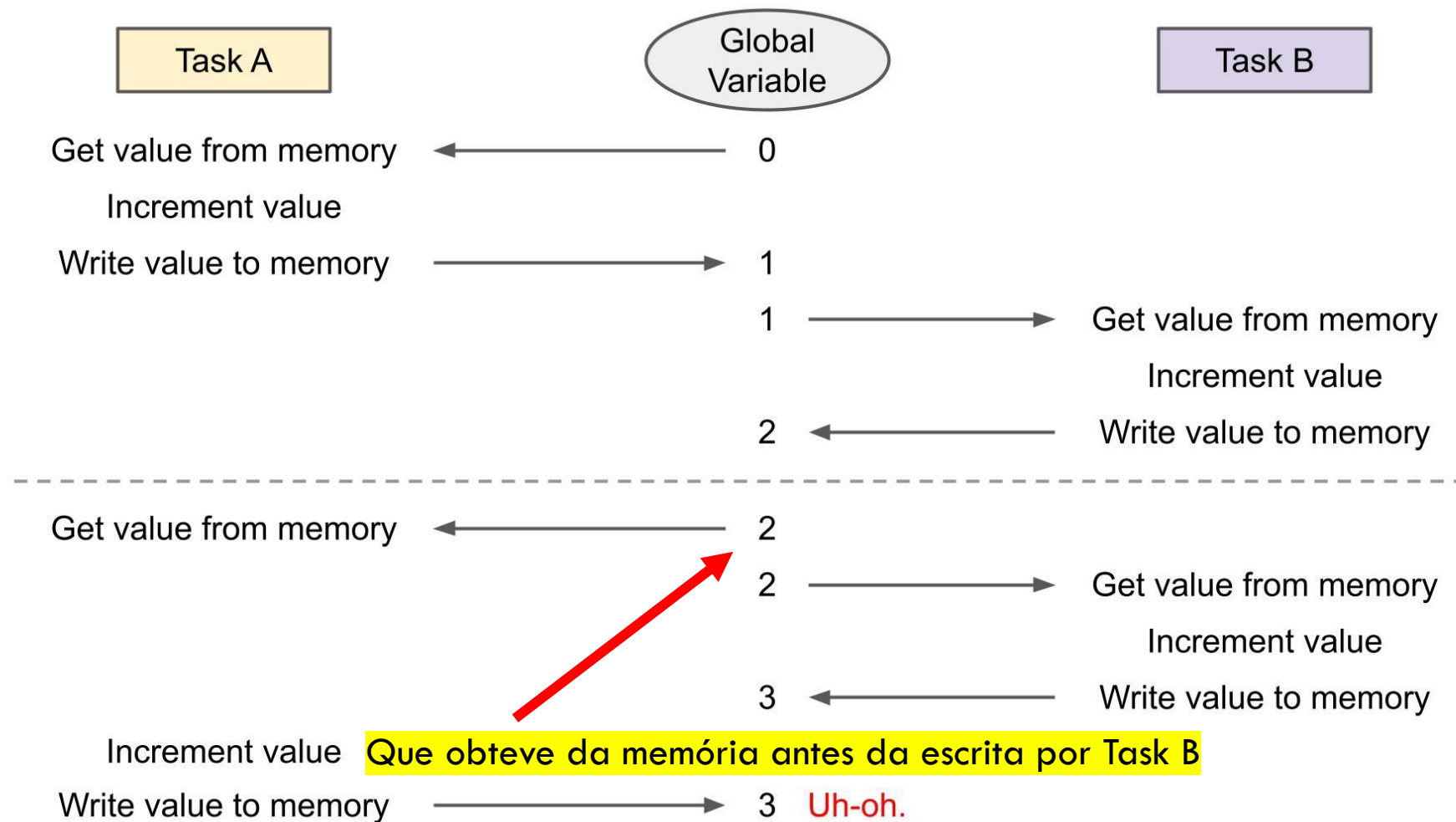
# Questões de não determinismo e sincronismo

- Eventos possíveis:

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19

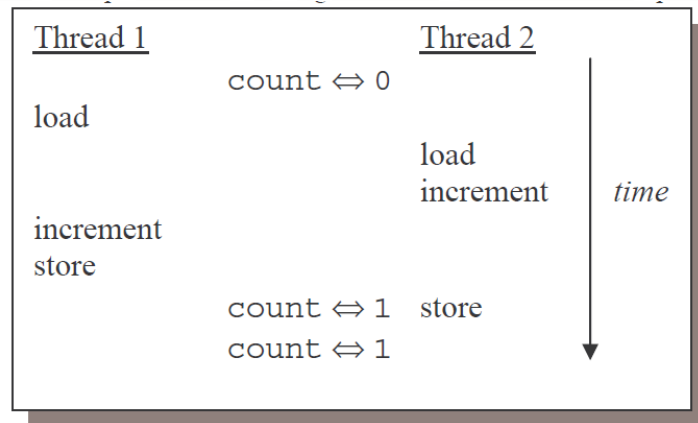
- Quando threads ou processos tentam simultaneamente acessar um recurso, e estes acessos podem resultar em resultados errados, diz-se que as threads ou processos estão em RACE CONDITION (condição de corrida). A saída depende de quem “ganhar a corrida”. No caso em questão a atualização da variável X. Só seria correto se uma das threads garantidamente terminasse antes da outra. Ou seja, a execução precisa ter **ATOMICIDADE**.
- Um bloco de comando que só pode ser executado por uma thread por vez é uma **SEÇÃO CRÍTICA** (mutualmente exclusivo) **garantido pelo(a) desenvolvedor(a) no paralelismo explícito.**

# Condição de corrida



# Questões de não determinismo e sincronismo

- **Exclusão mútua:** trecho de código executa com exclusão mútua se no máximo UMA THREAD pode executar o referido trecho a qualquer momento
- **Atomicidade:** termo utilizado e derivado da comunidade de bancos de dados, onde um conjunto de operações é atômico se OU TODOS EXECUTAM ATÉ O FIM (sem interrupções) ou NENHUM EXECUTA. Ou seja, não é possível verificar execuções parciais.
- No caso da contagem de objetos vista anteriormente:



O mecanismo mais comum é um **mutual exclusion lock** (mutex ou lock). Ideia é proteger a seção crítica. Antes da thread executar código na seção, precisa “obter” o mutex chamando função apropriada de **lock** e, ao fim, liberar, com **unlock**.

```
my_val = computeVal(my_rank);  
Lock(&add_my_val_lock);  
x += my_val;  
Unlock(&add_my_val_lock);
```

# Questões de não determinismo e sincronismo

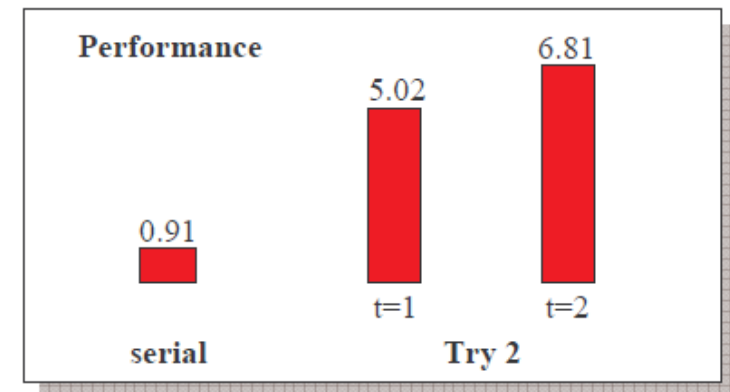
Task A	Mutex	Global Variable	Task B
Check for and take mutex	1	0	
Get value from memory	0	0	
	0	0	Check for and take mutex
	0	0	Wait/yield
Increment value	0	0	
Write value to memory	0	0	
Give mutex	0	1	
	1	1	Check for and take mutex
	0	1	Get value from memory
	0	1	Increment value
	0	1	Write value to memory
	0	2	Give mutex
	1	2	

# Questões de não determinismo e sincronismo

- Notamos no trecho de código anterior que não há ordem de acesso à seção crítica, apenas que uma thread por vez atualiza a variável x
- O mutex força serialização da seção crítica, portanto deve ser usado onde realmente for necessário e com execução o mais rápido possível, assim como as ISRs em interrupções

```
1 mutex m;
2
3 void count3s_thread (int id)
4 {
5     /* Compute portion of the array that this thread should work on */
6     int length_per_thread = length/t;
7     int start = id * length_per_thread;
8
9     for (i=start; i<start+length_per_thread; i+)
10    {
11        if (array[i] == 3)
12        {
13            mutex_lock(m);
14            count++;
15            mutex_unlock(m);
16        }
17    }
18 }
```

Esta solução garante atomicidade, mas eleva o tempo de processamento, devido ao tempo lock/unlock

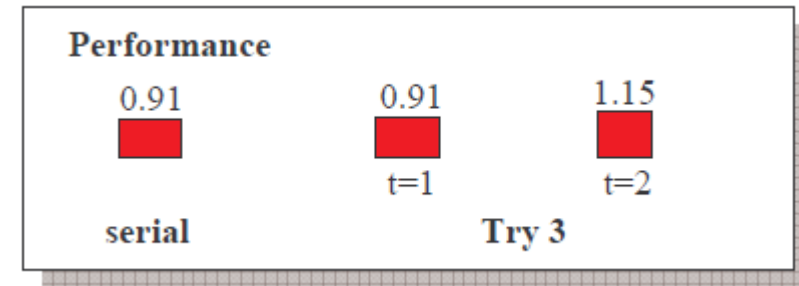


# Questões de não determinismo e sincronismo

- Comparando os códigos 1 e 2, no SO Windows 11, com Processador Intel® Core™ i7-10510U quad-core com 2 threads por core (8 threads), 16GBRam, com o g++ (MinGW.org GCC Build-2) 9.2.0, usando a função clock() de <ctime> em C++: 0.001 / 0.02 (media 10 execuções)
- Vamos então aumentar a granularidade (menos comunicação, mais computação). Ao invés de acessar a seção crítica sempre que COUNT for incrementada, podemos acumular a contribuição local numa variável PRIVADA à thread, private\_count, e acessar a seção crítica para atualizar a COUNT compartilhada apenas uma vez por thread.

```
1 private_count [MaxThreads];
2 mutex m;
3
4 void count3s_thread (int id)
5 {
6     /* Compute portion of the array that this thread
7     int length_per_thread = length/t;
8     int start = id * length_per_thread;
9
10    for (i=start; i<start+length_per_thread; i++)
11    {
12        if (array[i] == 3)
13        {
14            private_count [t]++;
15        }
16    }
17    mutex_lock(m);
18    count += private_count [t];
19    mutex_unlock(m);
20 }
```

- No código 3, 0.002s





## Outro exemplo com somatório

- Cálculo do número PI com N termos

```
1 void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11        factor = 1.0;
12    else
13        factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16        my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 } /* Thread_sum */
```

# Tipos de atributos MUTEX

o atributo padrão é PTHREAD\_MUTEX\_NORMAL ou PTHREAD\_MUTEX\_DEFAULT: causa deadlock se thread tenta acessar lock sem antes ter havido unlock

o atributo PTHREAD\_MUTEX\_RECURSIVE permite usar à thread usar o mesmo mutex "lockado". Ao ser criado o atributo é NORMAL. Depois muda com settype. Exemplo:

```
pthread_mutexattr_t attr;  
pthread_mutex_t mutex;  
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);  
pthread_mutex_init(&mutex, &attr);
```

```
pthread_mutexattr_t attr; //criado como PTHREAD_MUTEX_NORMAL  
pthread_mutex_t mutex;  
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);  
pthread_mutex_init(&mutex, &attr);
```

```
struct {  
    int a;  
    int b;  
    int c;  
} A;  
  
f() {  
    pthread_mutex_lock(&mutex);  
    A.a++;  
    g();  
    A.c = 0;  
    pthread_mutex_unlock(&mutex);  
}
```

```
g() {  
    pthread_mutex_lock(&mutex);  
    A.b += A.a;  
    pthread_mutex_unlock(&mutex);  
}
```

<https://www.ibm.com/docs/pt-br/aix/7.3?topic=programming-using-mutexes>