

Photo by [iggii](#) on [Unsplash](#)

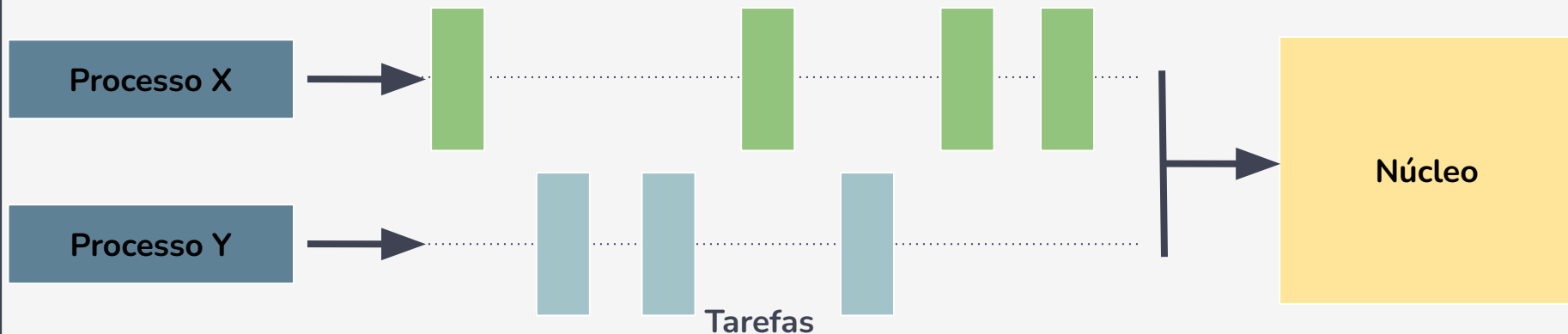
# Programação concorrente e distribuída

## Aula 4 - Memória compartilhada e controle de concorrência

Prof. João Robson

# Computação concorrente

- Permite que dois ou mais problemas possam ser executados ou obtenham progresso **de forma aparentemente simultânea** (“pseudoparalelismo”);
- Tarefas dentro de cada problema podem começar, serem executadas e concluídas em **períodos de tempo intercalados**;
- Tarefas não precisam ter uma ordem pré-definida, ou seja, **o resultado não é impactado pela ordem em que as instruções são executadas**.



# Corretude de programas concorrentes

---

- Apesar de permitir o aumento da eficiência na resolução de certos problemas, alguns desafios surgem com o modelo concorrente;
- Garantir a **corretude** de um programa, por exemplo, torna-se mais difícil:
  - Corretude é a propriedade de um algoritmo que garante que ele produza os resultados corretos em todas circunstâncias esperadas, de acordo com sua especificação;
- É possível distinguir a propriedade de corretude em dois tipos:
  - Propriedades de segurança: garantir que nada de ruim aconteça durante a execução.
  - Propriedades de “vivacidade” (*liveness*): garantir que coisas boas (que devem ocorrer) aconteçam durante a execução.
  - Exemplo: semáforos de trânsito:
    - Propriedade de segurança: sinais não devem ficar simultaneamente verdes em todas as direções;
    - Propriedade de *liveness*: um sinal que está vermelho deve ficar verde em algum momento.

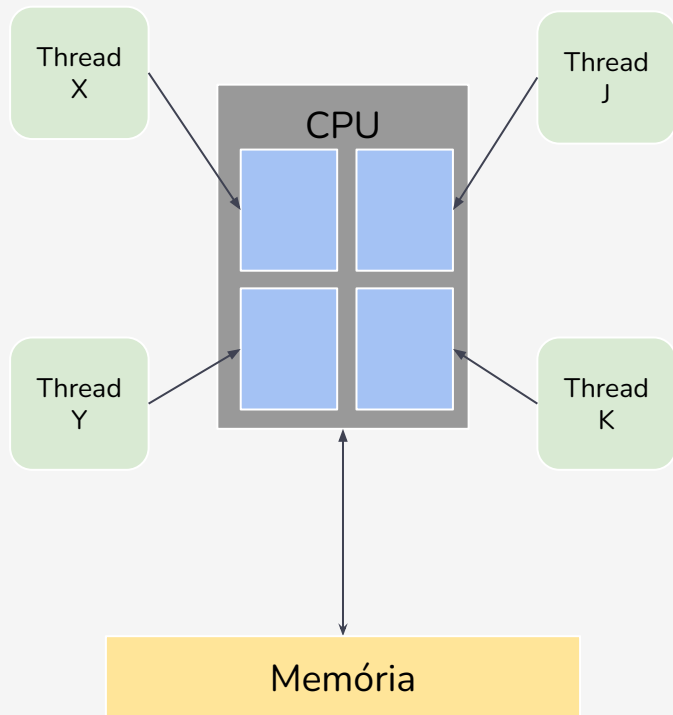
# Corretude de programas concorrentes

---

- Em geral, no caso específico de programas concorrentes, tem-se as seguintes propriedades:
  - Propriedades de segurança:
    - Exclusão mútua;
  - Propriedades de “vivacidade”:
    - Ausência de *deadlocks* (“impasses”);
    - Ausência de inanição (*starvation*) ou ausência de *lockout*.
- Nesse sentido, um programa está **correto** se nenhuma das possibilidades de intercalação entre as instruções viola as propriedades de corretude.

# Memória compartilhada

- Em aulas anteriores, foi visto que não basta aumentar o número de *threads* para resolver determinado problema. Em geral, é necessário utilizar algum mecanismo que garanta:
  - **Sincronia**, que são restrições na intercalação ou escalonamento dos processos/*threads* e/ou
  - **Comunicação**, ou seja, troca de dados entre processos/*threads*;
- Uma das formas de realizar isso é por meio da **memória compartilhada**;
- Essa estratégia funciona para computadores com processos sendo executados no mesmo processador e compartilhando a mesma memória.



# Memória compartilhada

---

- O uso da memória compartilhada por processos ou *threads* diferentes depende de algum algoritmo que garanta consistência nos valores acessados:
  - Exemplo: três *threads* sobrescrevendo uma variável X, com valor inicial 0, da seguinte forma:

```
// thread 1
```

```
...
```

```
X++
```

```
...
```

```
// thread 2
```

```
...
```

```
X++
```

```
...
```

```
// thread 3
```

```
...
```

```
X++
```

```
...
```

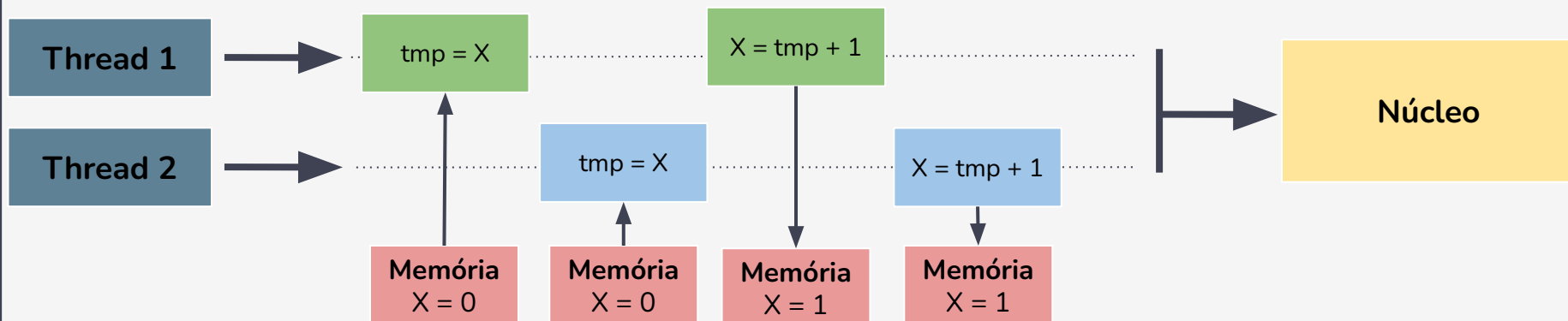
- Qual valor final de X?
  - Esperado: 3
  - Real: 3, 4, 5...
  - Por quê?

# Memória compartilhada

- A operação de incremento não é atômica, ou seja, ela é composta de várias instruções:

```
// X++ decomposto  
tmp = X  
X = tmp + 1
```

- Dessa forma, entre a primeira e a segunda instrução, pode haver uma troca de contexto entre as *threads*:



# Exclusão mútua

---

- Com isso, o valor de X, que no caso mostrado, deveria ser 2 ao fim, fica como 1;
- Para resolver esse problema, é necessário utilizar o conceito de **exclusão mútua** em regiões críticas do código:
  - Região crítica é uma **seção de código** em que um processo/thread **acessa recursos compartilhados** que podem ser modificados por outros processos ou threads. Nesse contexto, **é crucial garantir** que **apenas um processo/thread** tenha **acesso exclusivo aos recursos compartilhados**;
  - O acesso exclusivo é garantido com algum mecanismo de **exclusão mútua**:
    - Monitores;
    - Semáforos;
    - Locks (“travas”);
    - Barreiras.



# Exclusão mútua - Exemplo

---

- Dois vizinhos (João e Maria) compartilham um **quintal**;
- João tem um cachorro;
- Maria tem um gato;
- O cachorro e o gato não podem ficar juntos no mesmo instante no quintal;
- Como coordenar a entrada dos animais no quintal?
  - Com algum protocolo;
  - Opções:
    - **Maria olha pela janela e vê se quintal tá vazio**
      - Quintal é muito grande.
    - **Maria pode ir até a casa de João e bater na porta**
      - Demorado e João pode não estar em casa
  - E agora?

# Exclusão mútua - Exemplo

---

- João e Maria decidem usar bandeiras bem altas, visíveis de longe:
  - Se Maria quiser liberar gato no quintal:
    - Ela ergue sua bandeira;
    - Quando a bandeira de João é abaixada, ela solta seu gato;
    - Quando seu gato volta, ela abaixa sua bandeira.
  - E João segue um protocolo um pouco mais complexo:
    - Ele ergue sua bandeira;
    - Enquanto a bandeira de Maria está erguida:
      - Ele abaixa sua bandeira;
      - Espera até que a bandeira de Maria seja abaixada;
      - E enfim, ergue sua bandeira.
    - Assim que sua bandeira estiver erguida e a dela estiver abaixada, ele solta seu cachorro;
    - Quando seu cachorro volta, ele abaixa sua bandeira.

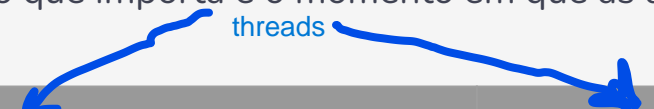
# Exclusão mútua - Exemplo

---

- O protocolo funciona, em um nível intuitivo, pelos seguinte motivo:
  - Se cada um ergue sua bandeira e, em seguida, olha para a bandeira do outro, então pelo menos um verá a bandeira do outro erguida (claramente, o último a olhar verá a bandeira do outro erguida) e não deixará seu animal de estimação entrar no quintal
- No entanto, essa observação não prova que os animais de estimação nunca estarão juntos no quintal;
- Pode-se supor, por contradição, que há uma maneira dos animais acabarem juntos no quintal e que na última vez que Maria e João ergueram suas bandeiras, eles olharam para a bandeira do outro antes de enviar o animal de estimação para o quintal.
  - Quando Maria olhou pela última vez, sua bandeira já estava totalmente erguida. Ela não deve ter visto a bandeira de João, ou não teria soltado o gato, então João deve não ter completado a elevação de sua bandeira **antes de Maria começar a olhar**.
  - Segue-se que **quando João olhou pela última vez**, depois de erguer sua bandeira, **deve ter sido depois de Maria começar a olhar**, então ele deve ter visto a bandeira de Maria erguida e não teria soltado seu cachorro, **uma contradição**.

# Exclusão mútua - Exemplo

- Dessa forma, mesmo que o ato de erguer a bandeira ocorra simultaneamente, o protocolo funciona, pois o que importa é o momento em que as ações começam e terminam:



The diagram shows the word "threads" in blue text. Two blue arrows originate from it: one points to the "Ações de Maria" column header, and the other points to the "Ações de João" column header.

Ações de <u>Maria</u>	Ações de <u>João</u>
Maria abre janela	João abre janela
Maria ergue bandeira	João ergue bandeira
Maria termina de erguer bandeira	João termina de erguer bandeira
Maria olha para bandeira de João	João olha para bandeira de Maria
Maria solta gato	João abaixa bandeira
...	...

# Exclusão mútua - Threads e memória

---

- Como já foi mostrado, *threads* de um mesmo processo compartilham alguns dados entre si (código executável, variáveis globais, recursos (arquivos, conexões de rede, etc.), **heap**):
  - **Heap**: região de **memória dinâmica** utilizada pelos programas de computador para **alocar** e **gerenciar** objetos ou estruturas de dados **durante sua execução**.
- Mas as **threads** também possuem **estruturas individuais**:
  - **Pilha**: estrutura que armazena informações sobre a chamada de funções e métodos, incluindo parâmetros passados, variáveis locais e endereço de retorno.
  - **Registradores**: memória localizada na CPU, usada para armazenar dados temporários relativos a execução de uma thread.

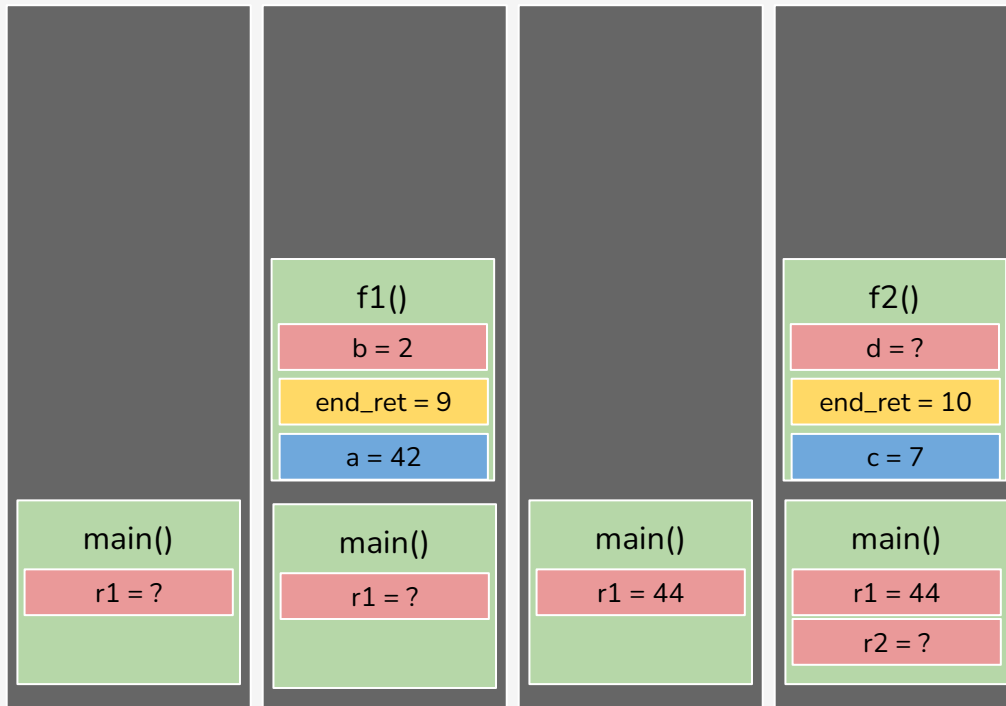
RAM

# Exclusão mútua - Threads e memória

- Funcionamento da pilha:

ender\_retorno(linha) ? ponto p/ voltar

```
0 func f1(a):  
1     b = 2  
2     return a + b  
3  
4 func f2(c):  
5     d = f1(c)  
6     return c + d  
7  
8 main  
9     r1 = f1(42)  
10    r2 = f2(7)
```

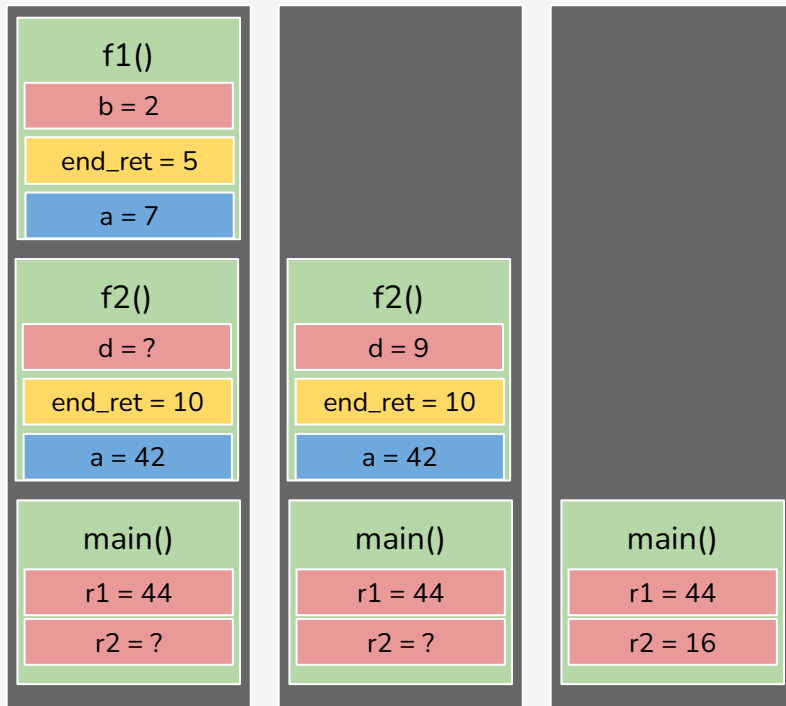


# Exclusão mútua - Threads e memória

- Funcionamento da pilha:

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10

```
func f1(a):  
    b = 2  
    return a + b  
  
func f2(c):  
    d = f1(c)  
    return c + d  
  
main  
    r1 = f1(42)  
    r2 = f2(7)
```



# Exemplo de memória compartilhada em Java

---

- Um programa que utilize *threads* para atualizar um contador, em que cada chamada do método *run()* deve atualizar o contador.



# Ausência de *deadlock*

~~Um é o inverso do outro~~

- Quando dois ou mais processos ou *threads* estão livres de impasses (*deadlocks*), isso significa que eles **não ficam bloqueados** aguardando recursos que estão sendo utilizados por outros processos bloqueados;
- Exemplo de João e Maria (livre de *deadlocks*):
  - Se um dos animais quiser entrar no quintal, cedo ou tarde consegue;
  - Se ambos quiserem entrar, pelo menos um irá eventualmente conseguir:
    - João e Maria erguem suas bandeiras. João, ao perceber que a bandeira de Maria está erguida, abaixa sua bandeira, permitindo que o gato dela entre no quintal.
- Exemplo de situação com *deadlock*:
  - João e Maria casam e vão viver na mesma casa.
  - João quer fazer um bife na frigideira; Maria, um ovo frito;
  - João pega a frigideira e Maria o óleo;
  - João fica aguardando Maria liberar o óleo. Maria espera João liberar a frigideira;
  - Enquanto João não fizer o bife, ele não libera a frigideira. Enquanto Maria não fritar o ovo, não libera o óleo. **fica bloqueado para sempre**

# Ausência de inanição (starvation-free)

---

- Há uma garantia de que todos os processos/threads têm oportunidade de realizar progresso (ou seja, ter tempo na CPU), sem serem adiados ou impedidos de executar de forma permanente
  - Exemplo de Maria e João:
    - Não é livre de inanição;
    - Toda vez que há conflito entre João e Maria, João cede;
    - Se Maria usar o quintal repetidamente, o cachorro nunca poderá usá-lo;
- Nas próximas aulas, será mostrado formas de garantir a propriedade de ausência de inanição.

# Referências

---

- Herlihy, M., & Shavit, N. (2008). The art of multiprocessor programming. Morgan Kaufmann.
- Ben-Ari, M. (1982). Principles of Concurrent Programming. Prentice Hall.