



# Programação concorrente

1º semestre de 2025

## Sincronização

Tema #05

Professor Marcelo Eustáquio

# Sincronização

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

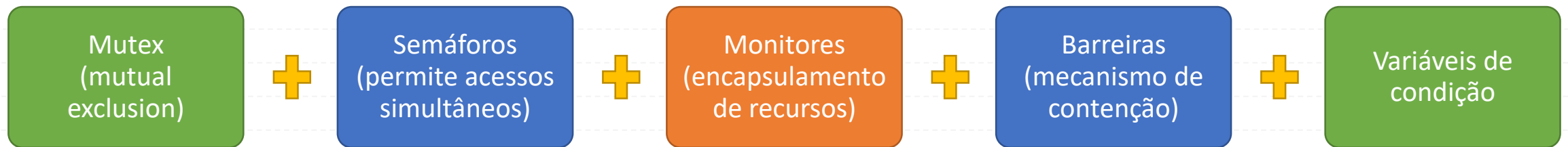


O objetivo principal da sincronização é evitar problemas como condições de corrida (race conditions) , inconsistências de dados e deadlocks , assegurando que o sistema funcione de maneira correta e previsível.

# Sincronização

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

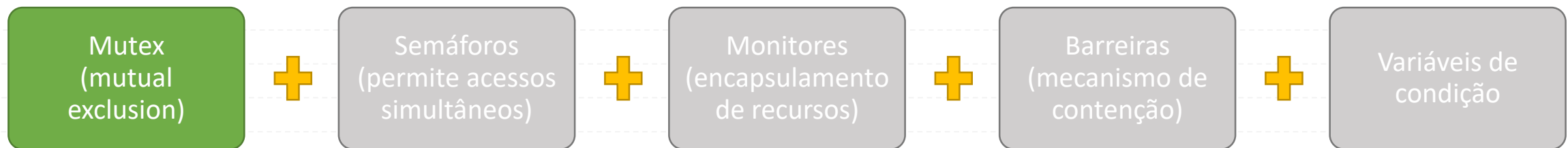
Para resolver esses problemas, várias estratégias de sincronização foram criadas:



# Mutex

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

Para resolver esses problemas, várias estratégias de sincronização foram criadas:



Um **mutex** é uma trava que garante que apenas uma thread possa acessar um recurso compartilhado por vez. Seu funcionamento está relacionado com thread que "bloqueia" o mutex antes de acessar o recurso, fazendo com que outras threads que tentarem acessar o mesmo recurso fiquem bloqueadas até que o mutex seja liberado.

# Mutex

**Mutex** (mutual exclusion) é uma trava que garante que apenas uma thread possa acessar um recurso compartilhado por vez. Em Python, no módulo `threading` é usado para implementar mutexes.

*No código a seguir, o método `with lock` garante que o mutex seja adquirido antes do acesso ao recurso e liberado automaticamente após o término do bloco. Sem o mutex, as threads poderiam sobrescrever mutuamente o valor de `counter`, resultando em um valor incorreto.*

```
import threading

# Recurso compartilhado
counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock: # Bloqueia o mutex
            counter += 1 # Acesso seguro ao recurso

# Criando threads
t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=increment)

t1.start()
t2.start()
t1.join()
t2.join()

print("Counter:", counter) # Saída esperada: 200000
```



# Mutex

Considere um sistema de reserva de assentos para um cinema que possui 100 assentos numerados de 1 a 100, e deve permitir que os clientes reservem assentos online. Vários clientes podem acessar o sistema simultaneamente, e cada cliente pode tentar reservar um ou mais assentos por vez.

Sem mecanismos de sincronização, podem ocorrer condições de corrida quando dois ou mais clientes tentam reservar o mesmo assento ao mesmo tempo. Isso pode resultar em problemas como:

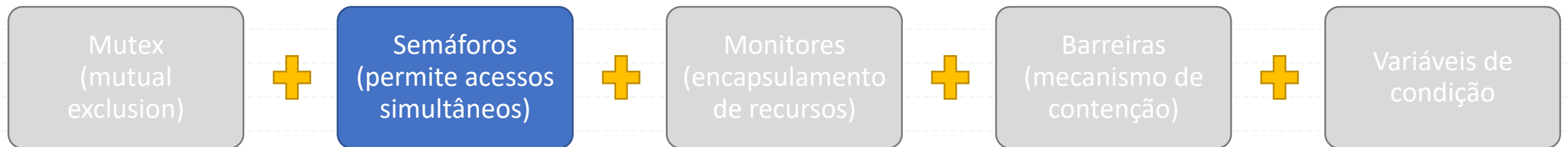
1. Reservas duplicadas: Dois clientes reservam o mesmo assento porque o sistema não verifica corretamente se o assento já está ocupado antes de confirmar a reserva.
2. Inconsistência no estado do sistema: O sistema pode exibir informações incorretas sobre a disponibilidade dos assentos, levando a uma experiência frustrante para os clientes.

Para resolver esses problemas, é necessário usar um mutex (trava) para garantir que apenas uma thread (representando uma solicitação de reserva) possa modificar o estado dos assentos por vez.

# Semáforos

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

Para resolver esses problemas, várias estratégias de sincronização foram criadas:



Um **semáforo** é uma variável inteira que controla o acesso a um conjunto de recursos. Possui as operações ***wait()***, que decrementa o semáforo; e ***signal()***, que incrementa o semáforo, permitindo que outra thread prossiga. Além dessas operações, pode ser classificado em **binário**, que controla o acesso a um único recurso (similar a um mutex) ou **contador**, que controla o acesso a múltiplos recursos.



# Semáforos

**Semáforo** é uma variável inteira que controla o acesso a um conjunto de recursos, permitindo que múltiplas threads acessem um recurso até um limite máximo. Em Python, o módulo `threading` fornece a classe `Semaphore`.

```
import threading

semaphore = threading.Semaphore(3) # Semáforo com limite de 3 threads
.....

def access_resource(thread_id):
    print(f"Thread {thread_id} tentando acessar o recurso...")
    with semaphore: # Adquire o semáforo
        print(f"Thread {thread_id} acessou o recurso.")
        threading.Event().wait(1) # Simula uso do recurso
    print(f"Thread {thread_id} liberou o recurso.")

# Criando threads

threads = [threading.Thread(target=access_resource, args=(i,)) for i in range(5)]
for t in threads: t.start()
for t in threads: t.join()
```

*O semáforo permite que no máximo 3 threads acessem o recurso simultaneamente (as outras threads aguardam até que uma das threads libere o semáforo.*

# Semáforos

Considere um sistema que controle o acesso a uma sala de reuniões em um escritório: a sala tem capacidade máxima para 5 pessoas e funcionários podem entrar e sair da sala ao longo do dia, mas é necessário garantir que:

1. Nenhum funcionário entre na sala se ela já estiver cheia (capacidade máxima atingida).
2. Quando um funcionário sai da sala, ele libera espaço para outros entrarem.

O sistema deve permitir que múltiplos funcionários tentem acessar a sala simultaneamente, mas respeitando a capacidade máxima. Para resolver esse problema, usaremos semáforos, que são mecanismos de sincronização adequados para controlar o acesso a recursos limitados, como a sala de reuniões neste caso.

# Sincronização

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

Para resolver esses problemas, várias estratégias de sincronização foram criadas:



Um **monitor** é uma construção de alto nível que encapsula um recurso compartilhado e suas operações de sincronização. Nesse caso, as threads só podem acessar o recurso a partir de métodos definidos pelo monitor, garantindo exclusão mútua automaticamente. Observação: Linguagens como Java fornecem suporte nativo para monitores usando a palavra-chave `synchronized`.

# Monitores

**Monitor** é uma construção de alto nível que encapsula um recurso compartilhado e suas operações de sincronização. Em Python, os monitores podem ser implementados usando o decorador `@synchronized` ou diretamente com `Lock` (Python não possui uma implementação explícita de monitores, mas o comportamento pode ser simulado usando classes e locks)

*No código a seguir, a classe `Monitor` encapsula o recurso (`resource`) e as operações seguras (`incremente`). Nesse caso, o lock garante que apenas uma thread execute o método por vez.*

```
import threading

class Monitor:
    def __init__(self):
        self.lock = threading.Lock()
        self.resource = 0

    def increment(self):
        with self.lock:
            self.resource += 1
            print(f"Resource incremented to {self.resource}")

# Criando instância do monitor
monitor = Monitor()

def task():
    for _ in range(5):
        monitor.increment()

threads = [threading.Thread(target=task) for _ in range(3)] # Criando threads

for t in threads: t.start()
for t in threads: t.join()
```



# Monitores

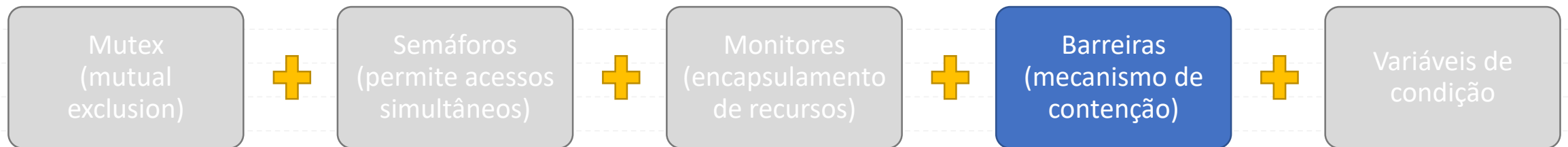
Considere um sistema que gerencia o acesso a uma impressora compartilhada em uma rede de computadores. Sabe-se que a impressora pode processar apenas um documento por vez, e várias máquinas podem enviar solicitações de impressão simultaneamente. Para garantir que os documentos sejam impressos na ordem em que as solicitações foram feitas, sem conflitos ou sobreposições, será necessário implementar um mecanismo de sincronização usando monitores .

Um monitor é ideal para este cenário porque ele encapsula a lógica de sincronização e controle de acesso ao recurso compartilhado (a impressora). O monitor gerencia a fila de solicitações e garante que as operações ocorram de forma segura e ordenada.

# Sincronização

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

Para resolver esses problemas, várias estratégias de sincronização foram criadas:



Uma **barreira** faz com que um grupo de threads espere até que todas tenham alcançado um certo ponto no programa e pode ser usado em cenários onde as threads precisam sincronizar seus progressos antes de continuar.

# Barreiras

**Barreira** é uma construção que faz com que um grupo de threads espere até que todas tenham alcançado um certo ponto dentro do código-fonte. Em Python, o módulo `threading` fornece a classe `Barrier`.

*No código a seguir, todas as threads chegam à barreira e só prosseguem quando o número de threads (3, no caso) estiver presente.*

```
import threading

# Barreira para 3 threads
barrier = threading.Barrier(3)

def worker(thread_id):
    print(f"Thread {thread_id} iniciada.")
    threading.Event().wait(1)      # Simula algum trabalho
    print(f"Thread {thread_id} chegou à barreira.")
    barrier.wait()                # Aguarda as outras threads
    print(f"Thread {thread_id} passou pela barreira.")

# Criando threads

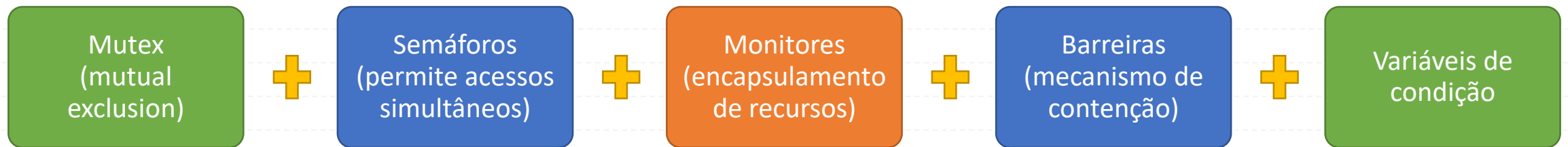
threads = [threading.Thread(target=worker, args=(i,)) for i in range(3)]

for t in threads: t.start()
for t in threads: t.join()
```

# Sincronização

Sincronização, em programação concorrente e paralela, diz respeito à coordenação de múltiplas threads (ou processos) com o objetivo de garantir que eles cooperem adequadamente ao acessar recursos compartilhados ou executar tarefas dependentes.

Para resolver esses problemas, várias estratégias de sincronização foram criadas:



**Variáveis de condição** permitem que threads esperem até que uma condição específica seja satisfeita, sendo geralmente usadas em combinação com mutexes para implementar padrões mais complexos de sincronização.





# Programação concorrente



1º semestre de 2025

## Sincronização

Tema #05

Professor Marcelo Eustáquio





Dado um grande arquivo de texto, divida-o em partes e conte as palavras em paralelo usando múltiplos processos. Combine os resultados no final e use a biblioteca multiprocessing para implementar a contagem paralela e sincronize os resultados finais.

```
def count_words_sequential(file_path):
```

```
    with open(file_path, 'r') as file:  
        text = file.read()
```

```
    word_count = len(text.split())
```

```
    return word_count
```

```
# Exemplo de uso
```

```
if __name__ == "__main__":
```

```
    file_path = "large_text_file.txt" # Substitua pelo caminho do seu arquivo
```

```
    total_words = count_words_sequential(file_path)
```

```
    print(f"Total de palavras no arquivo: {total_words}")
```