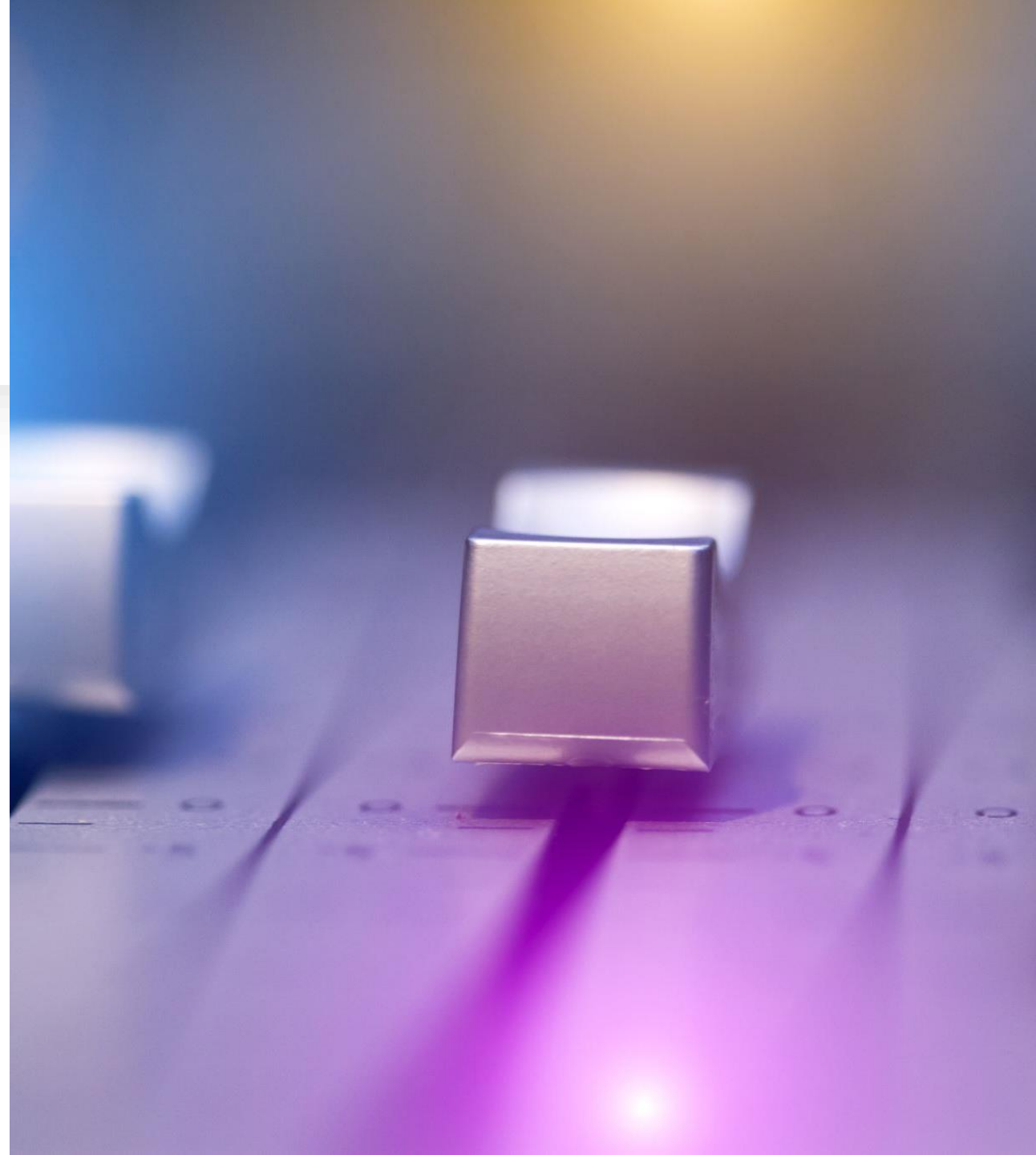


Semana 6 - Concorrência

Prof. Cassiano Moralles

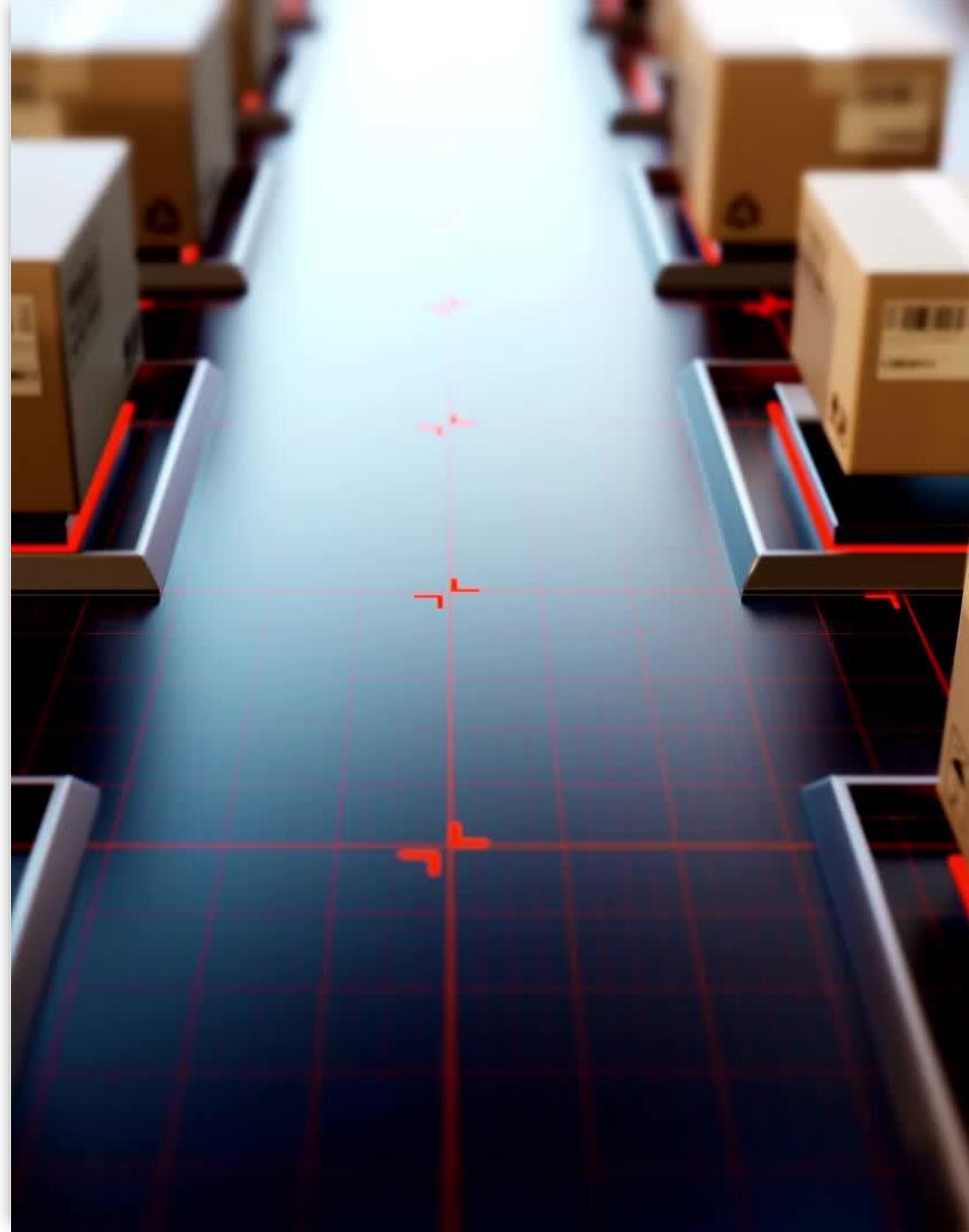
Concorrência

- Existem duas categorias de controle de unidades concorrentes.
- A mais natural é uma na qual, assumindo a disponibilidade de mais de um processador, diversas unidades do mesmo programa são executadas simultaneamente. Essa é a concorrência física. Um leve relaxamento desse conceito de concorrência permite ao programador e ao aplicativo de software assumirem a existência de múltiplos processadores fornecendo concorrência real, quando a execução real dos programas está ocorrendo de maneira intercalada em um processador.
- Essa é a concorrência lógica. É similar à ilusão de execução simultânea fornecida para diferentes usuários de um sistema de computação de multiprogramação de tempo compartilhado



Motivações para o uso de concorrência

- Existem ao menos duas razões para projetar sistemas de software concorrentes. A primeira é a velocidade de execução dos programas. Hardware concorrente fornece uma maneira efetiva de aumentar a velocidade de execução de programas;
- Concorrência fornece um método diferente de conceituar soluções de programas para problemas. Muitos domínios de problema se prestam naturalmente à concorrência, da mesma forma que a recursão é uma maneira natural de projetar a solução de alguns problemas;



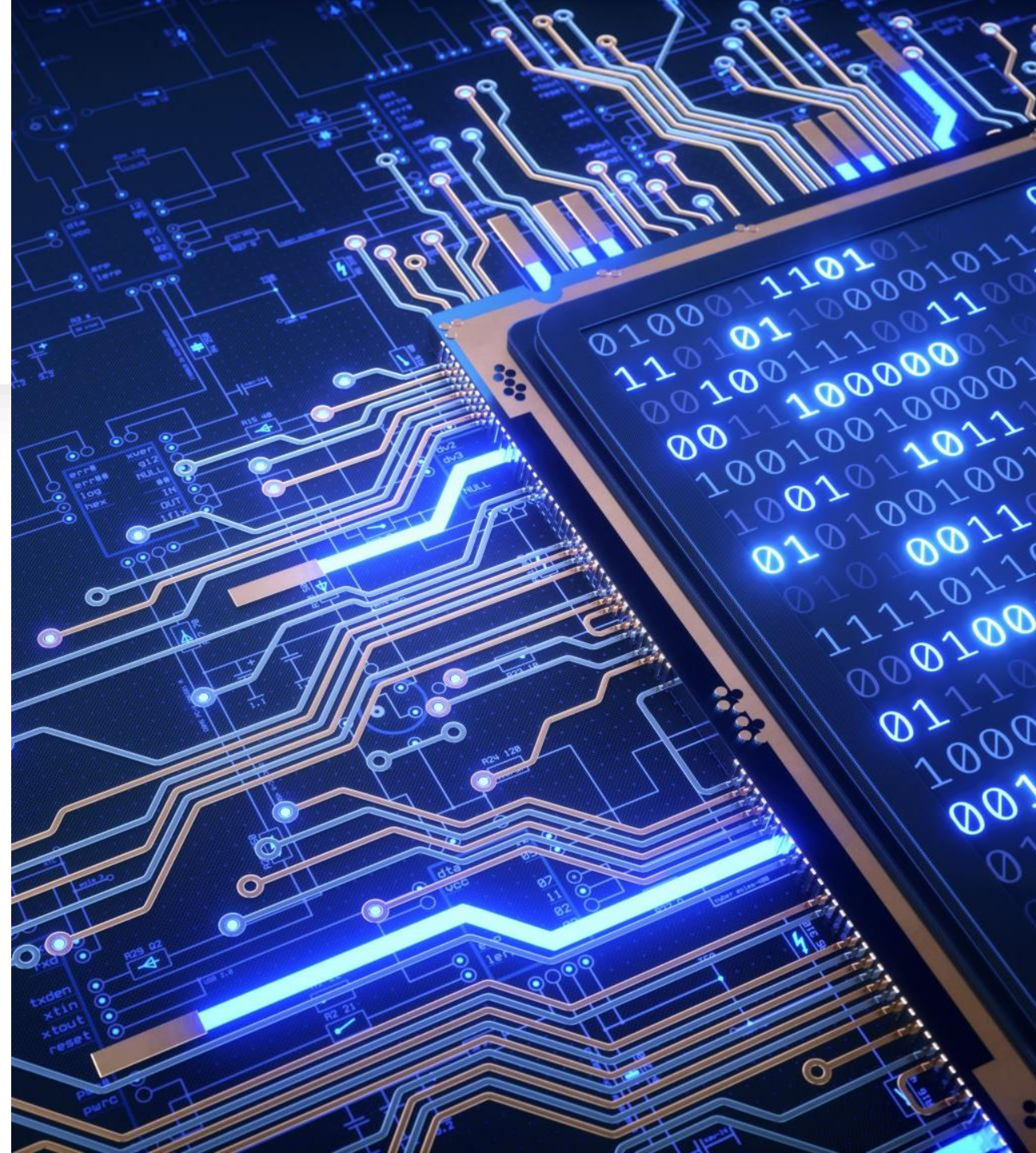
Concorrência

A concorrência na execução de software pode ocorrer em quatro níveis:

- no nível de instrução (executando duas ou mais instruções de máquina simultaneamente),
- no nível de sentença (executando duas ou mais sentenças na linguagem fonte simultaneamente),
- no nível de unidade (executando duas ou mais unidades de subprograma simultaneamente)
- no nível de programa (executando dois ou mais programas simultaneamente)

Concorrência no nível de instrução

- Este nível envolve a execução simultânea de múltiplas instruções de máquina através de técnicas como pipelining, onde diferentes estágios de várias instruções são processados ao mesmo tempo. Isso maximiza o uso do pipeline do processador, melhorando significativamente o desempenho de tarefas computacionalmente intensivas.



.....
.....
.....
.....

.....
.....
.....
.....

```
1 ; Suponha que temos um processador com pipeline de 5 estágios
2 ; As instruções podem ser algo como:
3 LOAD R1, A ; Carrega o valor de A no registrador R1
4 ADD R2, R1, B; Soma o valor do registrador R1 com B e armazena em R2
5 STORE R2, C ; Armazena o valor do registrador R2 em C
6 ; No pipeline, essas instruções podem estar sendo executadas simultaneamente em diferentes estágios
```

Concorrência no nível de instrução

- **Exemplo:** Um processador que utiliza pipelining para executar múltiplas instruções de máquina ao mesmo tempo.

Concorrência no nível de sentença

- Neste nível, sentenças de código em uma linguagem de alto nível são executadas simultaneamente utilizando threads. Cada thread pode executar uma tarefa independente, permitindo que programas realizem múltiplas operações ao mesmo tempo.
- Isso é particularmente útil em aplicações que necessitam de alta capacidade de resposta e interatividade, como interfaces gráficas de usuário (GUIs) e servidores web.

Concorrência no nível de sentença

- **Exemplo:** Utilizando threads em uma linguagem de alto nível como Python.

```
1  import threading
2
3  def func1():
4      print("Função 1")
5
6  def func2():
7      print("Função 2")
8
9  # Cria duas threads que executam func1 e func2 simultaneamente
10 t1 = threading.Thread(target=func1)
11 t2 = threading.Thread(target=func2)
12
13 t1.start()
14 t2.start()
15
16 t1.join()
17 t2.join()
18 |
```


Concorrência no nível de unidade

- concorrência é implementada através da execução paralela de unidades de subprograma, como funções ou métodos assíncronos. Ferramentas e frameworks de programação assíncrona permitem que desenvolvedores escrevam código que realiza operações de entrada/saída (I/O) de forma não bloqueante, melhorando a eficiência e a escalabilidade de aplicações que dependem de operações I/O intensivas, como consultas a bancos de dados ou chamadas de API.

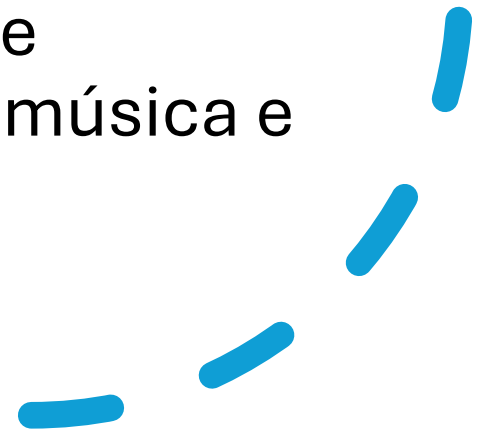
Concorrência no nível de unidade

- **Exemplo:** Executando múltiplas funções assíncronas em paralelo usando `async/await` em JavaScript.

```
1  async function fetchData1() {
2      let response = await fetch('https://api.example.com/data1');
3      let data = await response.json();
4      console.log(data);
5  }
6
7  async function fetchData2() {
8      let response = await fetch('https://api.example.com/data2');
9      let data = await response.json();
10     console.log(data);
11 }
12
13 async function runConcurrently() {
14     await Promise.all([fetchData1(), fetchData2()]);
15 }
16
17 runConcurrently();
18
```

Concorrência no nível de programa

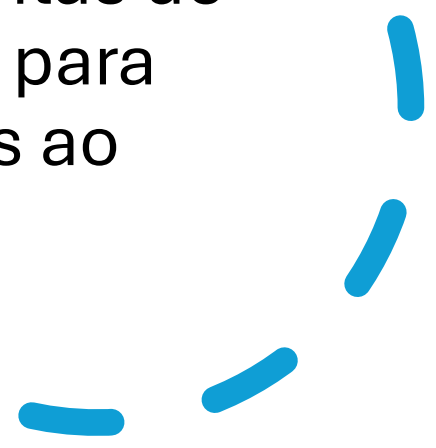
- No nível mais alto, múltiplos programas independentes são executados simultaneamente, muitas vezes gerenciados por um sistema operacional. Isso permite que diferentes aplicações utilizem os recursos do sistema de forma concorrente, melhorando a utilização geral do hardware e permitindo que usuários realizem múltiplas tarefas ao mesmo tempo, como edição de documentos enquanto ouvem música e navegam na internet.



```
# Executa dois scripts Python simultaneamente  
python script1.py &  
python script2.py &  
wait
```

Concorrência no nível de programa

- **Exemplo:** Utilizando ferramentas de gerenciamento de processos para executar múltiplos programas ao mesmo tempo.



Condições de Corrida

- As condições de corrida (*race conditions*) ocorrem em ambientes de programação concorrente quando dois ou mais processos ou threads tentam acessar e modificar dados compartilhados ao mesmo tempo.
- A sequência exata de operações pode ser imprevisível, levando a resultados incorretos ou inconsistentes.

Condições de Corrida

- Para entender melhor este conceito, vamos analisar um exemplo prático.
- Exemplo Prático de Condição de Corrida Considere um cenário onde duas threads estão incrementando uma variável compartilhada:

```
1  import threading
2
3  counter = 0 # Variável compartilhada
4
5  def increment():
6      global counter
7      for _ in range(100000):
8          counter += 1
9
10 # Criação das threads
11 thread1 = threading.Thread(target=increment)
12 thread2 = threading.Thread(target=increment)
13
14 # Início das threads
15 thread1.start()
16 thread2.start()
17
18 # Espera pela conclusão das threads
19 thread1.join()
20 thread2.join()
21
22 print("Valor final do counter:", counter)
23
```

PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS

ding.py"

Valor final do counter: 200000

PS D:\PYTHON> & d:/PYTHON/venv/Scripts/python.exe "d:/Or
ding.py"

Condições de Corrida

Análise do Exemplo

Neste exemplo, a expectativa é que a variável `counter` seja incrementada 200000 vezes (100000 vezes por cada thread), resultando no valor final de 200000. No entanto, devido à condição de corrida, o valor final de `counter` pode ser menor que 200000. Isso acontece porque as operações de leitura, incremento e escrita não são atômicas. Por exemplo:

1. Thread 1 lê o valor de `counter` (suponha que `counter` seja 0).
2. Thread 2 lê o valor de `counter` (também vê 0).
3. Thread 1 incrementa o valor lido (1) e escreve de volta em `counter`.
4. Thread 2 incrementa o valor lido (1) e escreve de volta em `counter`.

Apesar de ambas as threads terem incrementado `counter`, o valor final é apenas 1, pois a atualização de uma thread foi sobrescrita pela outra.

Solução para Evitar Condições de Corrida

Para evitar condições de corrida, é essencial garantir que a leitura, modificação e escrita da variável compartilhada sejam feitas de maneira atômica. Uma técnica comum é usar mecanismos de sincronização, como locks (travas).

```
1  import threading
2
3  counter = 0 # Variável compartilhada
4
5  def increment():
6      global counter
7      for _ in range(100000):
8          counter += 1
9
10 # Criação das threads
11 thread1 = threading.Thread(target=increment)
12 thread2 = threading.Thread(target=increment)
13
14 # Início das threads
15 thread1.start()
16 thread2.start()
17
18 # Espera pela conclusão das threads
19 thread1.join()
20 thread2.join()
21
22 print("Valor final do counter:", counter)
23
```

PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS

ding.py"

Valor final do counter: 200000

PS D:\PYTHON> & d:/PYTHON/venv/Scripts/python.exe "d:/Or
ding.py"

```
import threading

counter = 0
lock = threading.Lock() # Criação de uma trava

def increment():
    global counter
    for _ in range(100000):
        with lock: # Aquisição da trava
            counter += 1

# Criação das threads
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

# Início das threads
thread1.start()
thread2.start()

# Espera pela conclusão das threads
thread1.join()
thread2.join()

print("Valor final do counter:", counter)
```

Condições de Corrida

Solução para Evitar Condições de Corrida

- Para evitar condições de corrida, é essencial garantir que a leitura, modificação e escrita da variável compartilhada sejam feitas de maneira atômica. Uma técnica comum é usar mecanismos de sincronização, como locks (travas).
- Neste código, a utilização da trava (lock) garante que apenas uma thread por vez possa acessar e modificar a variável counter. Assim, evitamos a condição de corrida e asseguramos que o valor final de counter seja o esperado.



Condições de Corrida - Conclusão

- Condições de corrida são problemas críticos em programação concorrente que podem levar a resultados inesperados e bugs difíceis de rastrear.
- Utilizar mecanismos de sincronização, como locks, é essencial para garantir a corretude dos programas concorrentes.
- Ao entender e aplicar esses conceitos, podemos desenvolver sistemas mais robustos e confiáveis.



Deadlocks

- Deadlocks são situações em sistemas concorrentes onde dois ou mais processos ou threads ficam permanentemente bloqueados, cada um esperando por um recurso que o outro possui.
- Isso cria um ciclo de dependências que impede todos os envolvidos de continuar a execução.
- Para entender melhor, vamos explorar um exemplo prático e como prevenir deadlocks.

Deadlocks

Exemplo Prático de Deadlock

Imagine que temos dois recursos, `R1` e `R2`, e duas threads, `T1` e `T2`. Ambas precisam de acesso exclusivo a esses recursos para realizar suas tarefas. A situação de deadlock pode ocorrer da seguinte forma:

1. Thread T1 adquire o recurso `R1`.
2. Thread T2 adquire o recurso `R2`.
3. Thread T1 tenta adquirir o recurso `R2` e é bloqueada, pois `R2` está sendo usado por `T2`.
4. Thread T2 tenta adquirir o recurso `R1` e é bloqueada, pois `R1` está sendo usado por `T1`.

Agora, `T1` e `T2` estão esperando indefinidamente uma pela outra, resultando em um deadlock.

```
import threading

# Recursos compartilhados
lock1 = threading.Lock()
lock2 = threading.Lock()

def thread1():
    with lock1:
        print("Thread 1: adquiriu lock1")
        threading.Event().wait(0.1) # Simula trabalho
        with lock2:
            print("Thread 1: adquiriu lock2")

def thread2():
    with lock2:
        print("Thread 2: adquiriu lock2")
        threading.Event().wait(0.1) # Simula trabalho
        with lock1:
            print("Thread 2: adquiriu lock1")

# Criação das threads
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)

# Início das threads
t1.start()
t2.start()

# Espera pela conclusão das threads
t1.join()
t2.join()
```

PS D:\PYTHON> & d:/PYTHON/ve
Thread 1: adquiriu lock1
Thread 2: adquiriu lock2


Prevenção de Deadlocks

- **Ordem Total para Aquisição de Recursos:** Estabelecer uma ordem hierárquica para adquirir recursos. Se todas as threads adquirirem os recursos na mesma ordem, os deadlocks podem ser evitados.

```
1  import threading
2
3  # Recursos compartilhados
4  lock1 = threading.Lock()
5  lock2 = threading.Lock()
6
7  def thread1():
8      with lock1:
9          print("Thread 1: adquiriu lock1")
10         with lock2:
11             print("Thread 1: adquiriu lock2")
12
13  def thread2():
14      with lock1:
15          print("Thread 2: adquiriu lock1")
16          with lock2:
17              print("Thread 2: adquiriu lock2")
18
19  # Criação das threads
20  t1 = threading.Thread(target=thread1)
21  t2 = threading.Thread(target=thread2)
22
23  # Início das threads
24  t1.start()
```

PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS JUP

```
PS D:\PYTHON> & d:/PYTHON/venv/Scripts/python.exe d:/OneDri
Thread 1: adquiriu lock1
Thread 1: adquiriu lock2
Thread 2: adquiriu lock1
Thread 2: adquiriu lock2
PS D:\PYTHON>
```


A glowing green padlock is positioned on the left side of the slide, set against a dark background with a complex, glowing circuit board pattern. The padlock itself is translucent and emits a bright green light, with some internal details visible. The circuit lines are thin and white, creating a web-like pattern across the dark space.

Prevenção de Deadlocks

- **Tentar Bloquear (Try-Lock):** Em vez de bloquear indefinidamente, as threads tentam adquirir os locks e, se não conseguirem, liberam os recursos que já possuem e tentam novamente.
- **Detecção e Recuperação de Deadlock:** Monitorar o sistema para detectar deadlocks e, uma vez detectados, tomar ações para recuperar, como matar um dos processos envolvidos.

Prevenção de Deadlocks

Explicação do Código

- Definição dos Locks:** `lock1` e `lock2` são criados usando `threading.Lock()`, representando os recursos que as threads tentarão adquirir.
- Função `try_lock_example`:** Esta função é executada por cada thread. Ela tenta adquirir `lock1` e `lock2` com um timeout de 0.1 segundos. Se não conseguir adquirir ambos os locks, ela libera qualquer lock que tenha adquirido e tenta novamente após um breve intervalo (`time.sleep(0.1)`).
- Criação e Início das Threads:** Duas threads (`t1` e `t2`) são criadas e iniciadas, cada uma executando a função `try_lock_example` com um nome de thread passado como argumento.
- Sincronização das Threads:** `t1.join()` e `t2.join()` garantem que o programa principal espere até que ambas as threads tenham terminado sua execução antes de continuar.
- Prevenção de Deadlocks:** Ao utilizar `try_lock` com timeouts e liberar os locks adquiridos se não for possível obter todos os necessários, o código evita situações de deadlock.

```
1 import threading
2 import time
3
4 # Recursos compartilhados
5 lock1 = threading.Lock()
6 lock2 = threading.Lock()
7
8 def try_lock_example(thread_name):
9     while True:
10         print(f"{thread_name} tentando adquirir lock1")
11         acquired_lock1 = lock1.acquire(timeout=0.1)
12         if acquired_lock1:
13             print(f"{thread_name} adquiriu lock1")
14             try:
15                 print(f"{thread_name} tentando adquirir lock2")
16                 acquired_lock2 = lock2.acquire(timeout=0.1)
17                 if acquired_lock2:
18                     print(f"{thread_name} adquiriu lock2")
19                     # Realiza o trabalho com ambos os recursos adquiridos
20                     print(f"{thread_name} está realizando trabalho com ambos os locks")
21                     time.sleep(0.2) # Simula trabalho
22                     print(f"{thread_name} liberando lock2")
23                     lock2.release()
24                     print(f"{thread_name} liberando lock1")
25                     lock1.release()
26                     break
27                 else:
28                     print(f"{thread_name} não conseguiu adquirir lock2, liberando lock1")
29                     lock1.release()
30             except Exception as e:
31                 print(f"{thread_name} encontrou um erro: {e}")
32                 lock1.release()
33             time.sleep(0.1) # Espera um pouco antes de tentar novamente
34
35 # Criação das threads
36 t1 = threading.Thread(target=try_lock_example, args=("Thread 1",))
37 t2 = threading.Thread(target=try_lock_example, args=("Thread 2",))
38
39 # Início das threads
40 t1.start()
41 t2.start()
42
43 # Espera pela conclusão das threads
44 t1.join()
45 t2.join()
46
47 print("Execução concluída")
48
```

PROBLEMS 13 OUTPUT DEBUG CONSOLE TERMINAL PORTS JUPYTER

Thread 1 está realizando trabalho com ambos os locks
Thread 1 liberando lock2
Thread 1 liberando lock1
Thread 2 tentando adquirir lock1
Thread 2 adquiriu lock1
Thread 2 tentando adquirir lock2
Thread 2 adquiriu lock2
Thread 2 está realizando trabalho com ambos os locks
Thread 2 liberando lock2
Thread 2 liberando lock1
Execução concluída
PS D:\PYTHON>



Concorrência

- Cada nível de concorrência não só apresenta desafios específicos, como a necessidade de sincronização para evitar condições de corrida e deadlocks, mas também oferece oportunidades significativas para otimizar a eficiência e o desempenho dos sistemas computacionais, aproveitando ao máximo os recursos disponíveis.