3. SINCRONIZAÇÃO

1.

	Task A	Task B	Task C
1	P(SA)	P(SB)	P(SC)
2	P(SA)	.	P(SC)
3	P(SA)	.	P(SC)
4	.	.	.
5	.	.	.
6	.	V(SC)	V(SB)
7	V(SB)	V(SA)	V(SB)
8	END	.	V(SA)
9		END	END

Semáforos	a)	b)	c)	d)	e)	f)	g)
SA	2	3	2	0	3	1	1
SB	0	0	1	0	1	0	1
SC	2	2	1	3	3	3	1

a)
SA: 2 , 1 , 0
SB: 0 , 0
SC: 2 , 1 , 0
TA:

P(SA)	P(SA)	P(SA)*							
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)*	*	*							
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)	P(SC)*							
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Todos processos travaram.

b)
SA: 3 , 2 , 1 , 0 , 1 , 2
SB: 0 , 1 , 2 , 3
SC: 2 , 1 , 0 , 1
TA:

P(SA)	P(SA)	P(SA)	.	.	.	V(SB)	END		
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)*	*	*	*	*	*	*	.	.	.
0	1	2	3	4	5	6	7	8	9	10

.	V(SC)	V(SA)	.	END					
11	12	13	14	15	16	17	18	19	20

TC:

P(SC)	P(SC)	P(SC)*	*	*	*	*	*	*	*
0	1	2	3	4	5	6	7	8	9	10

*	*	.	.	V(SB)	V(SB)	V(SA)	END		
11	12	13	14	15	16	17	18	19	20

Não deu DEADLOCK.

c)
SA: 2 , 1 , 0 , 1
SB: 1 , 0 , 1
SC: 1 , 0 , 1
TA:

P(SA)	P(SA)	P(SA)*	*	*	*	*	.	.	.
0	1	2	3	4	5	6	7	8	9	10

V(SB)	END								
11	12	13	14	15	16	17	18	19	20

TB:

P(SB)	.	.	.	.	V(SC)	V(SA)	.	END	
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)*	*	*	*	*	P(SC)*	*	*	*
0	1	2	3	4	5	6	7	8	9	10

*	*								
11	12	13	14	15	16	17	18	19	20

DEADLOCK na tarefa C.

d)
SA: 0 , 1 , 0 , 1 , 0
SB: 0 , 1 , 0 , 1
SC: 3 , 2 , 1 , 2
TA:

P(SA)*	*	*	*	*	*	*	*	P(SA)*	*
0	1	2	3	4	5	6	7	8	9	10

*	*	P(SA)*	*						
11	12	13	14	15	16	17	18	19	20

TB:

P(SB)*	*	*	*	*	*	.	.	.	.
0	1	2	3	4	5	6	7	8	9	10

V(SC)	V(SA)	.	END						
11	12	13	14	15	16	17	18	19	20

TC:

P(SC)	P(SC)	P(SC)	.	.	V(SB)	V(SB)	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK na tarefa A.

e)
SA: 3 , 2 , 1 , 0 , 1 , 2
SB: 1 , 0 , 1 , 2 , 3
SC: 3 , 2 , 1 , 0 , 1
TA:

P(SA)	P(SA)	P(SA)	.	.	.	V(SB)	END		
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)	.	.	.	.	V(SC)	V(SA)	.	END	
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)	P(SC)	.	.	V(SB)	V(SB)	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

Não deu DEADLOCK.

f)
SA: 1 , 0 , 1 , 0 , 1 , 0
SB: 0 , 1 , 0 , 1 , 2
SC: 3 , 2 , 1 , 0 , 1
TA:

P(SA)	P(SA)*	*	*	*	*	*	*	P(SA)*	*
0	1	2	3	4	5	6	7	8	9	10

*	*	.	.	.	V(SB)	END			
11	12	13	14	15	16	17	18	19	20

TB:

P(SB)*	*	*	*	*	*	.	.	.	.
0	1	2	3	4	5	6	7	8	9	10

V(SC)	V(SA)	.	END						
11	12	13	14	15	16	17	18	19	20

TC:

P(SC)	P(SC)	P(SC)	.	.	V(SB)	V(SB)	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

Não deu DEADLOCK.

g)
SA: 1 , 0 , 1 , 0
SB: 1 , 0
SC: 1 , 0 , 1 , 0
TA:

P(SA)	P(SA)*	*	*	*	*	*	P(SA)*	*	
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)	.	.	.	.	V(SC)	V(SA)	.	END	
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)*	*	*	*	*	P(SC)*	*	*	
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Tarefa A e C travaram.

1.2.

	Task A	Task B	Task C
1	P(SA)	P(SB)	P(SC)
2	P(SA)	P(SA)	P(SC)
3	V(SA)	.	P(SB)
4	.	.	.
5	.	.	.
6	.	P(SC)	V(SB)
7	V(SC)	V(SA)	V(SB)
8	END	END	V(SA)
9			END

Semáforos	a)	b)	c)	d)	e)	f)	g)
SA	2	1	2	0	3	2	1
SB	0	0	1	0	1	2	1
SC	2	1	1	2	1	2	1

a)
SA: 2 , 1 , 0 , 1
SB: 0 , 0
SC: 2 , 1 , 0 , 1
TA:

P(SA)	P(SA)	V(SA)	.	.	.	V(SC)	END		
0 1	2	3	4	5	6	7	8	9	10

TB:

P(SB)*	*	*	*	*	*	*	*		
0 1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)	P(SB)*	*	*	*	*	*		
0 1	2	3	4	5	6	7	8	9	10

DEADLOCK, As tarefas B e C travaram.

b)
SA: 1 , 0
SB: 0
SC: 1 , 0
TA:

P(SA)	P(SA)*								
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)*	*								
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)*								
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Todas tarefas travaram.
c)
SA: 2 , 1 , 0 , 1 , 2
SB: 1 , 0
SC: 1 , 0 , 1 , 0
TA:

P(SA)	P(SA)	V(SA)	.	.	.	V(SC)	END		
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)	P(SA)*	*	.	.	.	P(SC)	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)*	*	*	*	*	*	*	*	
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Tarefas C travou.
d)SA: 0
SB: 0
SC: 2 , 1 , 0
TA:

P(SA)*	*	*							
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)*	*	*							
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)	P(SB)*							
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Todas tarefas travaram.

e)
SA: 3 , 2 , 1 , 0 , 1 , 2
SB: 1 , 0
SC: 1 , 0 , 1
TA:

P(SA)	P(SA)	V(SA)	.	.	.	V(SC)	END		
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)	P(SA)	.	.	.	P(SC)*	*	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)*	*	*	*	*	*	*	*	
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Tarefas C travou.
f)
SA: 2 , 1 , 0 , 1 , 0 , 1 , 2
SB: 2 , 1 , 0 , 1
SC: 2 , 1 , 0 , 1
TA:

P(SA)	P(SA)	V(SA)	.	.	.	V(SC)	END		
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)	P(SA)*	*	.	.	.	P(SC)*	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	(PSC)	P(SB)	.	.	V(SB)	V(SB)	V(SA)	END	
0	1	2	3	4	5	6	7	8	9	10

Não deu DEADLOCK.
g)SA: 1 , 0
SB: 1 , 0
SC: 1 , 0
TA:

P(SA)	P(SA)*								
0	1	2	3	4	5	6	7	8	9	10

TB:

P(SB)	P(SA)*								
0	1	2	3	4	5	6	7	8	9	10

TC:

P(SC)	P(SC)*								
0	1	2	3	4	5	6	7	8	9	10

DEADLOCK, Todas tarefas travaram.

2)

#A

O código Python cria duas threads utilizando o módulo threading e a classe Thread. As threads estão executando a função wish, que é definida para imprimir uma mensagem de saudação contendo o nome passado como argumento, pausar por 2 segundos e, em seguida, imprimir a idade também passada como argumento.

1. **Duas threads são criadas, 't1' e 't2', e ambas são iniciadas.**
2. **Cada thread executa a função 'wish' com os argumentos apropriados.**
3. **Quando uma thread chama l.acquire(), ela bloqueia o objeto de bloqueio, o que significa que qualquer outra thread que tente adquirir o bloqueio ficará bloqueada até que o bloqueio seja liberado.**
4. **Portanto, enquanto uma thread estiver dentro do bloco 'l.acquire()' e 'l.release()', a outra thread não pode executar o mesmo bloco de código, pois o bloqueio está em uso.**
5. **Isso garante que as mensagens de saudação e idade sejam impressas de forma coerente para cada thread, sem misturar as saídas.**

Conteúdo que será exibido após o final da execução:

Hi Sireesh
Your age is 15
Hi Nitya
Your age is 20
Hi Sireesh
Your age is 15
Hi Nitya
Your age is 20
Hi Sireesh
Your age is 15
Hi Nitya
Your age is 20

#B

Este código Python faz uso do módulo ‘threading’ para criar threads e de um semáforo para controlar o acesso concorrente a uma seção crítica. Aqui está uma explicação passo a passo do comportamento do código e o que será exibido ao final de sua execução:

1. **Um semáforo com uma contagem inicial de 2 é criado. Isso significa que até dois threads podem adquirir o semáforo simultaneamente.**
2. **Uma função wish é definida para imprimir uma mensagem de saudação e, em seguida, dormir por 2 segundos.**
3. **Quatro threads (‘t1’, ‘t2’, ‘t3’ e ‘t4’) são criados, cada um apontando para a função wish com argumentos diferentes.**
4. **Os quatro threads são iniciados (‘start()’).**
5. **Cada thread tentará adquirir o semáforo antes de imprimir a mensagem de saudação. Se o semáforo já estiver sendo usado por dois threads, o terceiro e o quarto threads terão que esperar até que pelo menos um dos outros threads libere o semáforo com release().**
6. **Cada thread imprime a mensagem "Hi" seguido pelo nome passado como argumento para a função wish.**
7. **Cada thread dorme por 2 segundos após imprimir a mensagem.**
8. **Depois que o thread termina, ele libera o semáforo para permitir que outros threads o utilizem.**
9. **Após o término de todos os threads, o programa principal termina.**

Conteúdo que será exibido após a execução:

Hi Sireesh
Hi Nitya
Hi Shiva
Hi Sireesh
Hi Ajay
Hi Nitya

#C

Este código Python faz uso de threads e um bloqueio (lock) para evitar condições de corrida enquanto várias threads tentam modificar a mesma variável global g. Aqui está uma explicação do comportamento do código e o resultado esperado:

1. **Importações e Inicialização: O código começa importando a classe ‘Lock’ e a classe ‘Thread’ do módulo ‘threading’. Em seguida, uma instância de ‘Lock’ é criada e uma variável global ‘g’ é inicializada com 0.**

2. **Definições de Funções: Duas funções são definidas: ‘add_one()’ e ‘add_two()’. Ambas as funções modificam a variável global ‘g’, uma adicionando 1 e outra adicionando 2 ao seu valor. Cada função utiliza o método ‘acquire()’ do objeto ‘lock’ antes de modificar ‘g’ e o método ‘release()’ após a modificação para garantir que apenas uma thread por vez possa modificar ‘g’.**
3. **Criação e Inicialização de Threads: Um loop itera sobre uma lista de referências às funções ‘add_one()’ e ‘add_two()’. Para cada função na lista, uma nova thread é criada com o alvo definido como a função. Em seguida, a thread é iniciada.**
4. **Aguardando a Conclusão das Threads: Outro loop itera sobre a lista de threads e chama o método join() em cada uma. Isso faz com que o programa principal aguarde a conclusão de todas as threads antes de prosseguir.**
5. **Impressão do Resultado: Após todas as threads serem concluídas, o programa imprime o valor final da variável global g.**

Resultado Esperado:
O valor final impresso dependerá da ordem em que as threads são executadas e das condições de corrida que ocorrem. Como o uso do lock garante que apenas uma thread pode modificar g de cada vez, o resultado esperado será consistente.

Dado o padrão de chamadas de função na lista de threads, o valor final de g será determinado pelo número total de vezes que cada função é chamada e pelo valor adicionado por cada função:

add_one() é chamada três vezes.
add_two() é chamada três vezes.
Então, g será incrementado 3 vezes por 1 (de add_one()) e 3 vezes por 2 (de add_two()).
o resultado final será 3 * 1 + 3 * 2 = 3 + 6 = 9.

3)

I. Finalidade do Código:
O código simula transferências de saldo entre duas contas bancárias (conta1 e conta2) usando threads. Cada thread representa uma transferência de valor fixo (1 unidade) da conta1 para a conta2.

II. Resultado Após Execução:
Após a execução do código fornecido, as contas serão exibidas com seus saldos atuais. O saldo da conta1 será o saldo inicial menos o valor transferido (100 - 100 = 0) e o saldo da conta2 será o saldo inicial mais o valor transferido (0 + 100 = 100). Isso ocorre porque o código executa 100 transferências de 1 unidade da conta1 para a conta2.

III. Execução do Código 10 Vezes:
Se você executar o código várias vezes, é provável que os resultados não sejam sempre os mesmos. Isso ocorre devido à natureza concorrente das threads. Como não há mecanismos de sincronização no código fornecido, várias threads podem tentar acessar e modificar as mesmas variáveis (saldo) ao mesmo tempo, o que pode levar a resultados inconsistentes. Portanto, é possível que em algumas execuções, as threads terminem em uma ordem diferente, resultando em saldos diferentes para as contas.

IV. Utilizando Mecanismos de Sincronização:
Para garantir que conta2 tenha um saldo final de 100 e conta1 tenha um saldo final de 0, podemos usar um mecanismo de sincronização, como um bloqueio (lock). modificando o código para incluir um lock e garantir que as operações de transferência sejam realizadas atomicamente, evitando condições de corrida:

```
class ContaBancaria():
    def __init__(self, nome, saldo):
        self.nome = nome
        self.saldo = saldo
        self.lock = threading.Lock()
 def __str__(self):
        return self.nome


def run(self):
        with self.origem.lock:
            origem_saldo_inicial = self.origem.saldo
            origem_saldo_inicial -= self.valor
            time.sleep(0.001)
            self.origem.saldo = origem_saldo_inicial
        with self.destino.lock:
            destino_saldo_inicial = self.destino.saldo
            destino_saldo_inicial += self.valor
            time.sleep(0.001)
            self.destino.saldo = destino_saldo_inicial
```