Deadlock’s

1-

```
import threading
import time

# Simular uma situação de deadlock onde quatro processos competem por
dois recursos
recurso1 = threading.Lock()
recurso2 = threading.Lock()

def processo1():
    with recurso1:  # Primeiro bloqueia o recurso 1
        print("O recurso 1 está sendo utilizado pelo processo 1!")
        time.sleep(1)

        print("Recurso 2 solicitado pelo processo 1!")
        with recurso2:  # Depois bloqueia o recurso 2
            print("Processo 1 termina de utilizar os 2 recursos!")

def processo2():
    with recurso1:  # Todos os processos bloqueiam na mesma ordem
        print("O recurso 1 está sendo utilizado pelo processo 2!")
        time.sleep(1)

        print("Recurso 2 solicitado pelo processo 2!")
        with recurso2:
            print("Processo 2 termina de utilizar os 2 recursos!")

def processo3():
    with recurso1:
        print("O recurso 1 está sendo utilizado pelo processo 3!")
        time.sleep(1)

        print("Recurso 2 solicitado pelo processo 3!")
        with recurso2:
            print("Processo 3 termina de utilizar os 2 recursos!")

def processo4():
    with recurso1:
        print("O recurso 1 está sendo utilizado pelo processo 4!")
        time.sleep(1)
```

```
        print("Recurso 2 solicitado pelo processo 4!")
        with recurso2:
            print("Processo 4 termina de utilizar os 2 recursos!")

def main():
    t1 = threading.Thread(target=processo1)
    t2 = threading.Thread(target=processo2)
    t3 = threading.Thread(target=processo3)
    t4 = threading.Thread(target=processo4)

    t1.start()
    t2.start()
    t3.start()
    t4.start()

    t1.join()
    t2.join()
    t3.join()
    t4.join()

main()
```

2- Implementação utilizando o Timeout para evitar Deadlock:

```
import threading
import time

# Criando os recursos compartilhados
recurso_A = threading.Lock()
recurso_B = threading.Lock()

def processo(nome):
    while True:  # Tenta até conseguir os dois recursos sem deadlock
        print(f"{nome} tentando adquirir Recurso A")
        acquired_A = recurso_A.acquire(timeout=1)  # Tenta pegar
Recurso A com timeout

        if acquired_A:
            print(f"{nome} adquiriu Recurso A")
            time.sleep(0.5)  # Simula tempo de uso do recurso

            print(f"{nome} tentando adquirir Recurso B")
```

```
            acquired_B = recurso_B.acquire(timeout=1)  # Tenta pegar
Recurso B com timeout

            if acquired_B:
                print(f"{nome} adquiriu Recurso B")

                # Simulando trabalho com os dois recursos
                time.sleep(0.5)
                print(f"{nome} finalizando...")

                # Libera os recursos na ordem inversa
                recurso_B.release()
                recurso_A.release()
                break  # Sai do loop pois conseguiu executar

            else:
                print(f"{nome} não conseguiu Recurso B, liberando A e
tentando novamente...")
                recurso_A.release()  # Libera Recurso A e tenta
novamente
                time.sleep(0.5)  # Pequeno atraso antes de tentar de
novo
        else:
            print(f"{nome} não conseguiu Recurso A, tentando
novamente...")
            time.sleep(0.5)  # Pequeno atraso antes de tentar de novo

# Criando e iniciando as threads dos processos
t1 = threading.Thread(target=processo, args=("Processo 1",))
t2 = threading.Thread(target=processo, args=("Processo 2",))
t3 = threading.Thread(target=processo, args=("Processo 3",))
t4 = threading.Thread(target=processo, args=("Processo 4",))

t1.start()
t2.start()
t3.start()
t4.start()

t1.join()
t2.join()
t3.join()
t4.join()
```

```
print("Todos os processos foram executados sem deadlock.")
```

3- **Vantagens da Prevenção de Deadlock**

1. **Garantir a continuidade do sistema**
   - Evita que processos fiquem presos indefinidamente, garantindo que todos progridam.
2. **Melhora a previsibilidade do sistema**
   - Como os impasses são prevenidos, o comportamento do sistema é mais previsível e gerenciável.
3. **Redução de despesas gerais de recuperação**
   - A prevenção evita a necessidade de mecanismos complexos de **detecção e recuperação** de impasses.
4. **Facilita o design do software**
   - Os desenvolvedores não precisam se preocupar tanto com falhas inesperadas devido a bloqueios mútuos.

❌ **Desvantagens da Prevenção de Deadlock**

1. **Uso ineficiente de recursos**
   - Técnicas de **alocação ordenada de recursos** podem deixar recursos ociosos por mais tempo, reduzindo a eficiência.
2. **Impacto no desempenho**
   - Algumas abordagens (como evitar espera circular) podem forçar processos a liberar recursos e tentar novamente, causando atrasos e consumo extra de CPU.
3. **Dificuldade de implementação**
   - Em sistemas grandes e distribuídos, garantir a **ordem de requisição de recursos** ou evitar alocação simultânea pode ser complicado.
4. **Podemos restringir a concorrência**
   - Estratégias de **preempção de recursos** podem fazer com que um processo tenha que liberar um recurso antes de concluir sua execução, diminuindo a eficiência geral.

🎯 **Quando usar a prevenção de deadlock?**

- **Sistemas críticos** (exemplo: aviação, hospitais, bancos) → onde **não pode haver falhas ou paradas** .
- **Ambientes com poucos recursos** → evitar impasses melhora o uso eficiente dos recursos disponíveis.

Já em sistemas de alto desempenho, pode ser melhor usar **detecção e recuperação** para não reduzir a concorrência.

4- A detecção de Deadlock pode ser renovada utilizando o gráfico de alocação de recursos ou a matriz de alocação de recursos . O algoritmo mais comum para essa detecção é baseado no detector de ciclos no gráfico de alocação.

5- Ele é chamado de "algoritmo do avestruz" porque, assim como o mito de que o avestruz enterra a cabeça na areia diante do perigo, ele não tenta evitar, prevenir ou resolver o impasse .

Em vez disso, o sistema simplesmente assume que o impasse não ocorrerá com frequência suficiente para soluções mais complexas.
Implementação:

```
import random
import time

def avestruz_strategy():
    print("🦩 Estratégia do avestruz ativada: Ignorando possíveis deadlocks...")

def simulate_deadlock():
    processes = ["P1", "P2", "P3", "P4"]
    resources = ["R1", "R2"]

    # Cada processo tenta pegar um recurso aleatoriamente
    allocation = {p: None for p in processes}

    # Simular processos pegando recursos
    for p in processes:
        allocation[p] = random.choice(resources)
        print(f"{p} pegou o recurso {allocation[p]}")
        time.sleep(0.5)

    print("\n❌ Deadlock potencial detectado! Mas o sistema segue ignorando...\n")
    avestruz_strategy()

# Rodar a simulação
simulate_deadlock()
```

6- O algoritmo do banqueiro é um método de evitar impasses em sistemas operacionais que gerenciam múltiplos processos compartilhando recursos. Foi proposto por Edsger Dijkstra e funciona de maneira semelhante a um banco que concede empréstimos, garantindo que sempre haja recursos suficientes para atender a demanda mínima dos clientes antes de liberar novos recursos.
Implementação:

```
import numpy as np

class BankersAlgorithm:
    def __init__(self, total_resources, allocation, maximum):
        self.total_resources = np.array(total_resources)  # Recursos totais disponíveis
```

```
        self.allocation = np.array(allocation)              # Recursos
atualmente alocados
        self.maximum = np.array(maximum)                    #
Necessidades máximas de cada processo
        self.available = self.total_resources -
self.allocation.sum(axis=0)  # Recursos disponíveis
        self.need = self.maximum - self.allocation  # Necessidades
restantes de cada processo

    def is_safe_state(self):
        work = self.available.copy()
        finish = [False] * len(self.allocation)
        safe_sequence = []

        while True:
            allocated = False
            for i in range(len(self.allocation)):
                if not finish[i] and all(self.need[i] <= work):  # Se o
processo pode ser concluído
                    work += self.allocation[i]  # Libera os recursos do
processo concluído
                    finish[i] = True
                    safe_sequence.append(i)
                    allocated = True
            if not allocated:
                break  # Se não há mais processos a serem atendidos,
saia do loop

        return all(finish), safe_sequence

    def request_resources(self, process_id, request):
        request = np.array(request)

        if any(request > self.need[process_id]):  # Pedido maior que a
necessidade do processo
            print(f"❌ ERRO: O processo {process_id} está pedindo mais
recursos do que precisa.")
            return False

        if any(request > self.available):  # Pedido maior que os
recursos disponíveis
            print(f"⏳ O processo {process_id} deve esperar por
recursos disponíveis.")
```

```
            return False

        # Aloca temporariamente os recursos
        self.available -= request
        self.allocation[process_id] += request
        self.need[process_id] -= request

        # Verifica se o estado ainda é seguro
        safe, sequence = self.is_safe_state()
        if safe:
            print(f"✅ Pedido concedido! Nova sequência segura:
{sequence}")
            return True
        else:
            # Reverte a alocação se for um estado inseguro
            self.available += request
            self.allocation[process_id] -= request
            self.need[process_id] += request
            print(f"❌ Pedido negado para evitar um estado inseguro!")
            return False


# Simulação de um sistema com 5 processos e 3 recursos
total_resources = [10, 5, 7]  # Recursos totais do sistema

allocation = [  # Recursos já alocados para cada processo
    [0, 1, 0],
    [2, 0, 0],
    [3, 0, 2],
    [2, 1, 1],
    [0, 0, 2]
]

maximum = [  # Necessidades máximas de cada processo
    [7, 5, 3],
    [3, 2, 2],
    [9, 0, 2],
    [2, 2, 2],
    [4, 3, 3]
]

# Criar uma instância do algoritmo do Banqueiro
bank = BankersAlgorithm(total_resources, allocation, maximum)
```

```
# Testar um pedido de recursos do processo 1
request = [1, 0, 2]  # Processo 1 pede mais recursos
print("\n🔍 Processo 1 solicita recursos:", request)
bank.request_resources(1, request)

# Testar um pedido inseguro do processo 2
request = [6, 0, 0]  # Processo 2 pede mais do que o disponível
print("\n🔍 Processo 2 solicita recursos:", request)
bank.request_resources(2, request)
```

Simulação Hipotética:
Suponha que tenhamos **5 processos (P0 a P4)** e **3 tipos de recursos (R0, R1, R2)** .
O sistema começa com **recursos disponíveis** e uma certa **alocação de recursos** para cada processo.

Cenários Testados:

1. O **processo 1** pede `[1, 0, 2]`– O sistema verifica que pode atender ao pedido **sem risco de impasse** e conceder os recursos.
2. O **processo 2** pede `[6, 0, 0]`– O sistema detecta que isso **deixaria o sistema inseguro** e negaria o pedido.

📌 Vantagens do Algoritmo do Banqueiro

✔ **Evite impasses** – O sistema só concede recursos se puder continuar seguro.
✔ **Garantir segurança** – Sempre verifique antes de alocar um recurso.
✔ **Útil para sistemas críticos** – Bancos, hospitais e redes de servidores podem usá-lo para garantir a operação contínua.

📌 Desvantagens do Algoritmo do Banqueiro

❌ **Alta sobrecarga computacional** – É necessário verificar o estado seguro a cada solicitação.
❌ **Não é adequado para sistemas dinâmicos** – Precisa de um número fixo de processos e recursos.
❌ **Nem sempre é prático** – Pode rejeitar pedidos que, na prática, poderiam ser atendidos sem problema.