EJERCICIO 1

Para resolver este ejercicio he uso los módulos threading para poder trabajar con semáforos e hilos, time para poder crear pausas y random para generar números aleatorios.

```python
import threading #Este módulo nos permite trabajar con hilos
import time #Este módulo nos permite crear tiempos de espera
import random #Este módulo nos permite generar números aleatorios

n = threading.Semaphore(0)  # Inicialmente no hay elementos en el buffer
s = threading.Semaphore(1)  # Semáforo para controlar el acceso al buffer

buffer = [] #Creamos un buffer que va a contener los distintos elementos
```

Como he comentado importamos los 3 módulos para poder completar este ejercicio, creamos los semáforos n y s que controlan el acceso y el estado del buffer. n cuenta los elementos en el buffer, y s asegura el acceso exclusivo al mismo. Con esto vamos a conseguir que haya un único consumidor que esté extrayendo datos del buffer a la vez que impedimos que ocurra las superposiciones al acceder al buffer.

```python
def productor(): #Creamos la función del productor
    while True:
        item = random.randint(1, 100)  # Producimos un elemento
        s.acquire()  # Iniciamos la sesión crítica del recurso compartido
        buffer.append(item)  # Añadimos elemento al buffer
        print(f"Productor produjo: {item}")
        s.release() #Finalizamos la sección crítica del recurso compartido
        n.release()  # Indicamos que hay un nuevo elemento en el buffer
        time.sleep(random.random())  #Generamos un tiempo de espera aleatorio
```

En este segmento de código creamos la función productor, se va a encargar de que siempre se cree un nuevo elemento después de que se haya consumido el elemento que había puesto en el buffer previamente.

Vamos a crear un elemento nuevo que tras iniciar la sección crítica del recurso compartido va a ser añadido al buffer. Finalizamos la sección crítica para indicar que hay un nuevo elemento en el buffer y que el consumidor pueda tomar ese elemento. Generamos un tiempo de espera para que el consumidor cumpla su función.

```python
def consumidor(): #Creamos la función del consumidor
    while True:
        n.acquire()  # Esperamos si no hay elementos en el buffer
        s.acquire()  # Esperamos si otro hilo está accediendo al buffer
        item = buffer.pop(0)  # Extraeremos un elemento del buffer
        print(f"Consumidor consumió: {item}")
        s.release()#Finalizamos la sección crítica del recurso compartido
        time.sleep(random.random())  # Generamos un tiempo de espera aleatorio
```

En este segmento de código creamos la función consumidor, se va a encargar de consumir los elementos que previamente el productor ha añadido al buffer. Tras esperar mientras el productor añade elementos al buffer el consumidor va a dejar el buffer sin elementos y volverá a esperar a que el productor añada otro elemento entrando en un bucle que no tiene final.

```python
def main():# Creación y ejecución de hilos para el productor y el consumidor
    productor_thread = threading.Thread(target=productor) #Creamos un hilo para la función productor
    consumidor_thread = threading.Thread(target=consumidor) #Creamos un hilo para la función consumidor

    productor_thread.start() #Empezamos la ejecución de ambos hilos
    consumidor_thread.start()

    productor_thread.join() #Esperamos a que los hilos terminen
    consumidor_thread.join()

if __name__ == "__main__":
    main()
```

En este segmento de código creamos los hilos que corresponden a las funciones productor y consumidor. Vamos a iniciar ambos hilos y esperar a que finalicen.

EJERCICIO 2

Nos piden un área de datos compartida entre un número de procesos donde convivan escritores y lectores. Dónde los lectores pueden leer del área de datos de forma simultánea. Sólo un escritor al mismo tiempo puede escribir en el área de datos. Y si un escritor está escribiendo en el área de datos, ningún lector puede leerla.

Primero definimos como deben comportarse los lectores

```
# Definimos una función para el lector
def lector(nombre, semlectores, semescritores):
    while True:
        with semlectores: #llamamos al semaforo de lectura
            print(f'Lector {nombre} leyendo.')# print que te dice el lector que esta leyendo
            time.sleep(2) #el semáforo permanece bloqueado para otros lectores 2 segundos.
```

Este código cuando sea true (cuando el semáforo este en verde) nos printea printea el número del lector que está leyendo y con un tiempo de descanso de 2 segundos.

Ahora definimos los escritores

```
# Función para el escritor
def escritor(nombre, semlectores, semescritores):
    while True:
        with semescritores: #llamamos al semaforo de escritura
            print(f'Escritor {nombre} escribiendo.') # print que te dice el escritor
            time.sleep(2) #tiempo bloqueado
```

Básicamente es lo mismo que los lectores, pero con el semáforo de escritores.

Posteriormente le ponemos bloqueos a los semáforos y definimos el número de escritores y lectores para el ejercicio en este caso 3 lectores y 2 escritores.

```
#Ya definidas las dos funciones con sus semaforos
if __name__ == "__main__": #verificamos si el script se está ejecutando como un programa principal y no siendo importado
    semlectores = threading.Lock() #creamos los bloqueos para los semaforos
    semescritores = threading.Lock()

    lectores = 3 #ponemos 3 lectores y 2 escritores
    escritores =  2
```

Por último, creamos dos hilos uno para los lectores y otro para los escritores, los iniciamos y esperamos a que terminen

```
# Creamos los hilos para lectores y escritores
hlectores = [threading.Thread(target=lector, args=(i, semlectores, semescritores)) for i in range(lectores)]
hescritores = [threading.Thread(target=escritor, args=(i, semlectores, semescritores)) for i in range(escritores)]

# Iniciamos los hilos
for hilo in hlectores + hescritores:
    hilo.start()

# Esperamos a que todos los hilos terminen
for hilo in hlectores + hescritores:
    hilo.join()
```

Este ejemplo serio donde los lectores tienen prioridad, pero el enunciado nos pide que creemos dos versiones una donde los lectores tengan prioridad y otra donde los escritores la tengan.

El caso es que la versión de escritores es imposible de hacer.

EJERCICIO 3

Para resolver este ejercicio he uso los módulos threading para poder trabajar con semáforos e hilos.

```
import threading #Este módulo nos permite trabajar con hilos

semaforo_1 = threading.Semaphore(0) #Creamos los dos semáforos necesarios
semaforo_2 = threading.Semaphore(0)
```

En este segmento de código importamos el módulo threading para poder trabajar con semáforos e hilos. Creamos dos semáforos que son los necesarios para poder cumplir los requisitos que nos pide este ejercicio. Un semáforo asegurará que el "1" se imprima antes del "4", y otro semáforo garantizará que el "6" se imprima antes del "3".

Semaforo_1 comienza en 0, lo que significa que inicialmente H2 tendrá que esperar hasta que H1 imprima "1" y libere el semáforo.

Semaforo_2 también comienza en 0 y se utiliza para asegurar que "6" se imprima antes que "3".

```
def hilo_H1(): #Creamos la función para el primer hilo
    print("1") #Imprimimos cada uno de los números en el orden indicado
    semaforo_1.release()  # Señal para permitir que H2 imprima "5"
    semaforo_2.acquire()  # Espera hasta que H2 imprima "6"
    print("3")
    print("2")
```

Esta función imprime "1" y luego libera semáforo_1, lo que permite que H2 continúe y pueda imprimir "4". Después, H1 espera (semáforo_2.acquire()) hasta que H2 haya impreso "6".

```
def hilo_H2(): #Creamos la función para el segundo hilo
    semaforo_1.acquire()  # Espera hasta que H1 imprima "1"
    print("5")
    print("4") #Imprimimos cada uno de los números en el orden indicado
    print("6")
    semaforo_2.release()  # Señal para permitir que H1 imprima "3"

H1 = threading.Thread(target=hilo_H1) #Creamos nuestros dos hilos
H2 = threading.Thread(target=hilo_H2)
```

Esta función Comienza con semáforo_1.acquire(), lo que significa que esperará a que H1 imprima "1" antes de continuar. Después de imprimir "4", "5" y "6", libera semáforo_2, permitiendo que H1 imprima "3".Este diseño asegura que el "1" siempre se imprima antes del "4", y que el "6" siempre se imprima antes del "3", cumpliendo con los requisitos del ejercicio.

```
H1 = threading.Thread(target=hilo_H1) #Creamos nuestros dos hilos
H2 = threading.Thread(target=hilo_H2)


H1.start() #Inicamos los dos hilos
H2.start()

H1.join() #Esperamos a que los dos hilos terminen
H2.join()
```

En este segmento de código vamos a crear nuestros hilos para iniciarlos y que cumplan su función.

EJERCICIO 4

Este ejercicio es bastante similar al anterior, en este caso tenemos dos hilos. El hilo 1 contiene las letras C y E y el hilo 2 contiene las letras A, R, O.

El enunciado nos pide crear con esos dos hilos y utilizando los menores semáforos posibles las palabras ACERO o ACREO.

Primero como es lógico importamos el módulo threading para simular hilos y el módulo random para alternar entre letras

```
import threading
import random
```

Ahora creamos dos semáforos que nos servirán para elegir el orden de impresión de las letras correctamente.

También definimos una variable llamada alternate que elegirá entre dos opciones en este caso entre la letra R o la letra E.

```
# definimos dos semaforos para que bloqueen ciertas letras
semA = threading.Semaphore(0)
semC = threading.Semaphore(0)

# creamos una variable que decide si las letras "E" y "R" se imprimen en un orden específ
alternate = random.choice([True, False])
```

Definimos dos Hilos que tienen que seguir un orden de impresión de las letras y por ello usamos los Sem.acquire y el Sem.release

Hay que crear la palabra ACREO o ACERO y para ello el orden debe ser: las letras A y la C siempre sean las primeras en imprimir. Luego un semáforo obligara a que después de la C se imprima la letra R o la letra E intercalándose las posiciones entre ellas y por último que se imprima la letra O

```python
# creamos una variable que decide si las letras "E" y "R" se imprimen en un orden específico o de mane
alternate = random.choice([True, False])

# Hilo H1
def hilo_H1():
    semA.acquire()  #imprime la letra C después de que hilo 2 imprime la letra A.
    print("C")
    semC.release()

# Hilo H2
def hilo_H2():
    print("A")
    semA.release()  # imprime la letra A, luego espera a que el hilo h1 imprima la letra C  Después de
    semC.acquire()
    if alternate:
        print("E")
        print("R") #imprime E o R
    else:
        print("R")
        print("E")
    print("O") #imprime la letra O.
```

Ahora los hilos los metemos en 2 variables llamadas H1 y H2. Los iniciamos y esperamos a que terminen.

```python
# Ccreamos e iniciamos los hilos
thread1 = threading.Thread(target=hilo_H1)
thread2 = threading.Thread(target=hilo_H2)

thread1.start() #se inician los hilos
thread2.start()

thread1.join() #terminan los hilos
thread2.join()
```

Al ejecutarlo nos saldrá ACREO o ACERO

```
A
C
R
E
O
PS C
Driv
A
C
E
R
O
```

EJERCICIO 5

Para resolver este ejercicio he uso los módulos threading para poder trabajar con semáforos e hilos.

```
import threading

semaforo_A_F = threading.Semaphore(0)  # Controla la relación entre A y F
semaforo_E_H = threading.Semaphore(0)  # Controla la relación entre E y H
semaforo_C_G = threading.Semaphore(0)  # Controla la relación entre C y G
```

En este segmento de código vamos a crear tres semáforos, cada semáforo comienza en 0, lo que significa que inicialmente los hilos H2 y H3 tendrán que esperar hasta que H1 y H2 respectivamente liberen los semáforos.

```
#Tras crear nuestros tres semaforos vamos a crear una función para cada uno de ellos
def hilo_H1(): #Función para nuestro primer hilo
    while True:
        print("A")
        semaforo_A_F.release()  # Permite imprimir un F
        print("B")
        print("C")
        semaforo_C_G.release()  # Permite imprimir un G
        print("D")
```

Vamos a crear una función para cada uno de nuestros semáforos que posteriormente van a convertirse en hilos. Esta función imprime "A", liberando semaforo_A_F para permitir a H2 imprimir "F", y luego imprime "C", liberando semaforo_C_G para permitir a H2 imprimir "G".

```
def hilo_H2(): #Función para nuestro segundo hilo
    while True:
        semaforo_A_F.acquire()  # Espera a que se imprima un A antes de imprimir F
        print("E")
        semaforo_E_H.release()  # Permite imprimir un H
        print("F")
        semaforo_C_G.acquire()  # Espera a que se imprima un C antes de imprimir G
        print("G")
```

Esta función espera (semaforo_A_F.acquire()) hasta que H1 imprima "A" para imprimir "F", luego imprime "E", liberando semaforo_E_H para permitir a H3 imprimir "H", y espera (semaforo_C_G.acquire()) hasta que H1 imprima "C" para imprimir "G".

```python
def hilo_H3(): #Función para nuestro tercer hilo
    while True:
        semaforo_E_H.acquire()  # Espera a que se imprima un E antes de imprimir H
        print("H")
        print("I")
```

Esta función espera (sem_E_H.acquire()) hasta que H2 imprima "E" para imprimir "H".

```python
H1 = threading.Thread(target=hilo_H1) #Creamos un hilo para cada uno de nuestros semaforos
H2 = threading.Thread(target=hilo_H2)
H3 = threading.Thread(target=hilo_H3)


H1.start() #Inciamos los hilos
H2.start()
H3.start()
```

Creamos un hilo para cada una de la funciones que hemos creado anteriormente y los iniciamos.

EJERCICIO 6

En este ejercicio vamos a aplicar el método del banquero que consiste en una estrategia para evitar el interbloqueo al gestionar la asignación de recursos. Donde se evalúa la seguridad antes de aprobar solicitudes de recursos, asegurándose de que la asignación propuesta no conduzca a un estado inseguro donde los procesos no puedan completarse debido a la falta de recursos. Esto contribuye a la estabilidad del sistema al evitar situaciones en las que los procesos quedan atrapados esperando recursos indefinidamente.

 # Nota algunas partes me he ayudado de una ia concretamente la parte de la función validar_matrices

Primero necesitamos crear un archivo.txt llamado matrices con este contenido.

```
3 2 2
6 1 3
3 1 4
4 2 2
-----------------------------
1 0 0
5 1 1
2 1 1
0 0 2
-----------------------------
9 3 6
-----------------------------
1 1 2
```

Ahora explicaremos el código.

Primero como hay que trabajar con matrices importamos la librería numpy.

```
import numpy as np #Importamos numpy para el manejo de matrices
```

Ahora vamos a definir 4 funciones para que nuestro código funcione.

La primera sirve para que nos lea el archivo de las matrices.

En concreto abre el archivo, dice que elemento delimitan entre una matriz y otra y separa las matrices

```
def leer_archivo(archivo): # esta funcion lee un archivo que contiene bloques de matrices y con numpy las transforma en matrices
    with open(archivo) as file:
        raw_data = file.read().split('----------\n')
    return [np.array([list(map(int, row.split())) for row in matrix.split('\n') if row]) for matrix in raw_data]
```

Ahora definimos otra función que nos dirá cuando es un estado seguro o no.

```
def es_estado_seguro(need, allocation, available): #es
                                                    #
    work = available.copy() #creamos una copia de vect
    finish = [False] * len(need) #vemos si el cada pro
    sequence = []

    while False in finish:
        for i, (n, a, f) in enumerate(zip(need, alloca
            if f == False and all(n <= work):
                work += a
                finish[i] = True
                sequence.append(i)
                break
        else:
            return False, []

    return True, sequence
```

En resumen, utilizamos un bucle tipo while que pasa por todos los procesos que no han terminado una y otra vez hasta que estén todos terminados.

Ahora definimos una función que nos dirá si hay algún problema con las matrices ya sea distinta dimensión, que haya valores negativos, o que no hayas recursos asignados.

Les ponemos los requerimientos que necesitan y ponemos tres prints en caso de que se den.

Si hay uno de esos tres problemas el programa lo detectara y printeara el que sea conveniente.

```python
def validar_matrices(need, allocation, resources, available):
    # Verificamos la congruencia de las matrices
    if len(need) != len(allocation) or len(need[0]) != len(allocation[0]):
        raise ValueError("Las matrices de Necesidad y Asignación no son congruentes.")

    # Verificamos que no haya valores negativos en las matrices
    if np.any(need < 0) or np.any(allocation < 0) or np.any(resources < 0) or np.any(available < 0):
        raise ValueError("Las matrices y vectores no pueden contener valores negativos.")

    # Verificamos que no haya más recursos asignados o necesitados que los que existen en total
    if np.any(allocation > resources) or np.any(need > resources):
        raise ValueError("Hay más recursos asignados o necesitados que los disponibles en total.")
```

Ahora va nuestra función algoritmo_banquero utilizara las funciones ya explicadas antes y printeara el número total de procesos y recursos, la matriz de necesidad, la de asignación el vector de recursos y el de disponibles.

```python
def algoritmo_banquero(archivo): #ahora aplicamos loa tres funciones ya explicadas.
    need, allocation, resources, available = leer_archivo(archivo) #leemos las matrices

    try:
        validar_matrices(need, allocation, resources, available) #ver si tiene algun problema
    except ValueError as e:
        print(f"Error de validación: {e}")
        return
# ahora el sistema imprime el numero de procesos recursos la matriz de asignacion de necesidad
# el vector de recursos y el vector de disponibles ademas de decirnos si el estado es seguro o
    print("Número total de procesos:", len(need))
    print("Número total de recursos:", len(resources))

    print("Matriz de Necesidad:")
    print(need)

    print("Matriz de Asignación:")
    print(allocation)

    print("Vector de Recursos:")
    print(resources)

    print("Vector de Disponibles:")
    print(available[0])
```

Por último, si el vector de recursos puede ir limpiando la matriz y dejarla a 0 el programa imprimirá que es un estado seguro, si por otra parte no puede dirá que no es un estado seguro.

```
estado_seguro, secuencia_ejecucion = es_estado_seguro(need, allocation, available[0])

if not estado_seguro:
    print("El estado no es seguro.")
else:
    print("El estado es seguro.")
    print("Secuencia de ejecución segura:", secuencia_ejecucion)
```

En este caso el estado no es seguro y lo he comprobado manualmente ya que el vector de recursos no puede eliminar ningún proceso de la matriz N-A

El output del sistema es el siguiente:

```
Número total de procesos: 4
Número total de recursos: 1
Matriz de Necesidad:
[[3 2 2]
 [6 1 3]
 [3 1 4]
 [4 2 2]]
Matriz de Asignación:
[[1 0 0]
 [5 1 1]
 [2 1 1]
 [0 0 2]]
Vector de Recursos:
[[9 3 6]]
Vector de Disponibles:
[1 1 2]
El estado no es seguro.
PS C:\Users\pablo\OneDrive\Escritorio\sistemas operativos>
```

La práctica ya estaría terminada nos has hecho sufrir bastante Carlos, pero te queremos aun así :)