Universidad Tecnológica del Uruguay

ITR Suroeste – Fray Bentos

Ingeniería en Mecatrónica

UNIDAD CURRICULAR DE PROGRAMACIÓN 3

PROFESOR GIOVANI BOLZAN COGO

2024/I

CONCURRENCIA

Introducción a la Programación Concurrente

Objetivos de aprendizaje:

- Comprender los conceptos básicos de la programación concurrente.
- Conocer los problemas de la concurrencia y los métodos de resolución.
- Aplicar la concurrencia en el desarrollo de aplicaciones en Python.

CONCURRENCIA

Introducción a la Programación Concurrente

- La programación concurrente se refiere a la ejecución simultánea de múltiples tareas en un programa. Permite mejorar la eficiencia y la capacidad de respuesta de las aplicaciones. Sin embargo, también plantea desafíos, como las condiciones de carrera y las secciones críticas.
- Ejemplo práctico: Creación de hilos en Python y ejecución concurrente de tareas.

- Las condiciones de carrera ocurren cuando varios hilos intentan acceder y modificar los mismos recursos compartidos sin una sincronización adecuada. Las secciones críticas son regiones de código donde se manipulan estos recursos compartidos.
- Ejemplo práctico: Identificación y resolución de problemas de condición de carrera en un programa en Python. También, procesos cooperativos: afectan o son afectados por el estado de otros procesos.

CONCURRENCIA

Introducción a la Programación Concurrente

- Algunas técnicas para resolver los problemas de concurrencia incluyen el uso de mecanismos de sincronización, como semáforos y monitores. Estos mecanismos permiten controlar el acceso a los recursos compartidos y evitar condiciones de carrera.

- *En resumen, en esta clase hemos explorado los fundamentos de la programación concurrente en Python. Hemos aprendido sobre las condiciones de carrera, las secciones críticas y los mecanismos de sincronización disponibles en Python.*
- *Referencia bibliográfica: "Python Concurrency in Practice" de Brian Okken.*

CONCURRENCIA

Introducción a la Programación Concurrente

- **Ejercicio**: Control de acceso a una sala de espera (semáforo)

Descripción:

- Implementa un programa en Python que simule el control de acceso a una sala de espera utilizando semáforos como mecanismo de sincronización. La sala de espera tiene una capacidad máxima de 5 personas y se requiere que las personas esperen su turno para ingresar.

El programa debe tener las siguientes características:

1. Definir una clase SalaEspera que controle el acceso a la sala de espera.
2. Utilizar un semáforo para controlar la capacidad máxima de la sala.
3. Implementar los siguientes métodos en la clase SalaEspera:
   - entrar(): Permite a una persona entrar a la sala de espera. Si la capacidad máxima ha sido alcanzada, la persona debe esperar hasta que haya espacio disponible.
   - salir(): Permite a una persona salir de la sala de espera, liberando un espacio.
   - obtener_cantidad_personas(): Devuelve la cantidad de personas actualmente en la sala de espera.

CONCURRENCIA

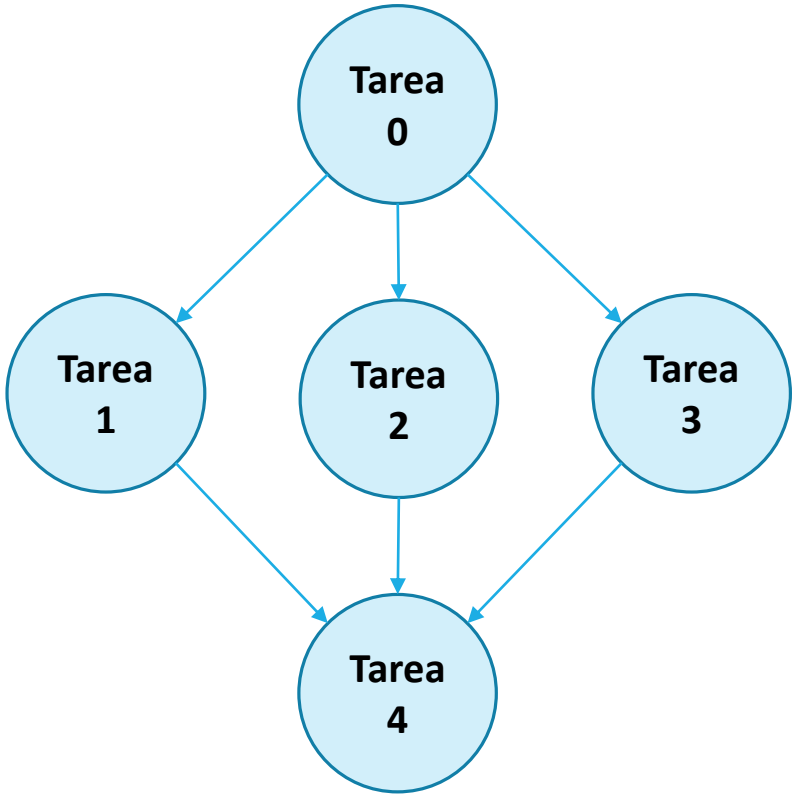
Cobegin-Coend y Grafos de Precedencia

Objetivos de aprendizaje:

- Comprender cómo utilizar la programación basada en eventos como alternativa a Cobegin-Coend.
- Aprender a utilizar eventos y tareas asíncronas para la programación concurrente en Python.
- Comparar las ventajas y desventajas de Cobegin-Coend y la programación basada en eventos.

CONCURRENCIA

El grafo de precedencia



Pseudocodigo cobegin-coend que resuelve el grafo

```
begin
    Tarea0
    cobegin
        Tarea1
        Tarea2
        Tarea3
    coend
    Tarea4
end
```

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- La problemática involucrada es garantizar que los recursos sean coherentes con la necesidad.
- En un caso en que se necesita resultado dependiente de otro proceso, se debe garantizar la precedencia de ejecución de tareas.
- El código al lado ilustra ese problema.

```
def task1():
    global a
    a = 1
    time.sleep(randint(0,3))
    print(f"Tarea 1 valor: {a}\n")
def task2():
    global a
    a = 2
    time.sleep(randint(0,3))
    print(f"Tarea 2 valor: {a}\n")
def task3():
    global a
    a = 3
    time.sleep(randint(0,3))
    print(f"Tarea 3 valor: {a}\n")
# Crear hilos para cada tarea
thread1 = threading.Thread(target=task1)
thread2 = threading.Thread(target=task2)
thread3 = threading.Thread(target=task3)
threads = [thread1,thread2,thread3]
# Iniciar los hilos
for t in threads:  t.start()
# Esperar a que los hilos terminen
for t in threads:  t.join()
```

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- Cobegin-Coend es una técnica que permite ejecutar varias tareas de forma concurrente y esperar a que todas finalicen antes de continuar. En Python, esto se puede lograr utilizando hilos y técnicas de sincronización.

- Ejemplo práctico: Implementación de Cobegin-Coend en Python utilizando hilos.

```
import threading
# Función para una tarea específica
def task1():
    print("Tarea 1 en ejecución")
def task2():
    print("Tarea 2 en ejecución")
def task3():
    print("Tarea 3 en ejecución")
# Crear hilos para cada tarea
thread1 = threading.Thread(target=task1)
thread2 = threading.Thread(target=task2)
thread3 = threading.Thread(target=task3)
# Iniciar los hilos
thread1.start()
thread2.start()
thread3.start()
# Esperar a que los hilos terminen
thread1.join()
thread2.join()
thread3.join()
print("Finalizado")
```

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- La programación basada en eventos es una alternativa a Cobegin-Coend que se centra en la programación de eventos y tareas asíncronas.
- En lugar de crear hilos o procesos separados, se utilizan bibliotecas o *frameworks* como **asyncio** en Python para gestionar eventos y realizar operaciones de forma asíncrona.
- Ejemplo práctico con programación basada en eventos en Python utilizando **asyncio**.

```
import asyncio
# Función para una tarea específica
async def task1():
    print("Tarea 1 en ejecución")
async def task2():
    print("Tarea 2 en ejecución")
# Función que ejecuta las tareas en paralelo
async def execute_tasks():
    # Iniciar las tareas concurrentemente
    task1_coroutine = task1()
    task2_coroutine = task2()
    # Esperar a que todas las tareas finalicen
    await asyncio.gather(task1_coroutine,
task2_coroutine)
    print("Finalizado")
# Ejecutar las tareas
asyncio.run(execute_tasks())
```

CONCURRENCIA

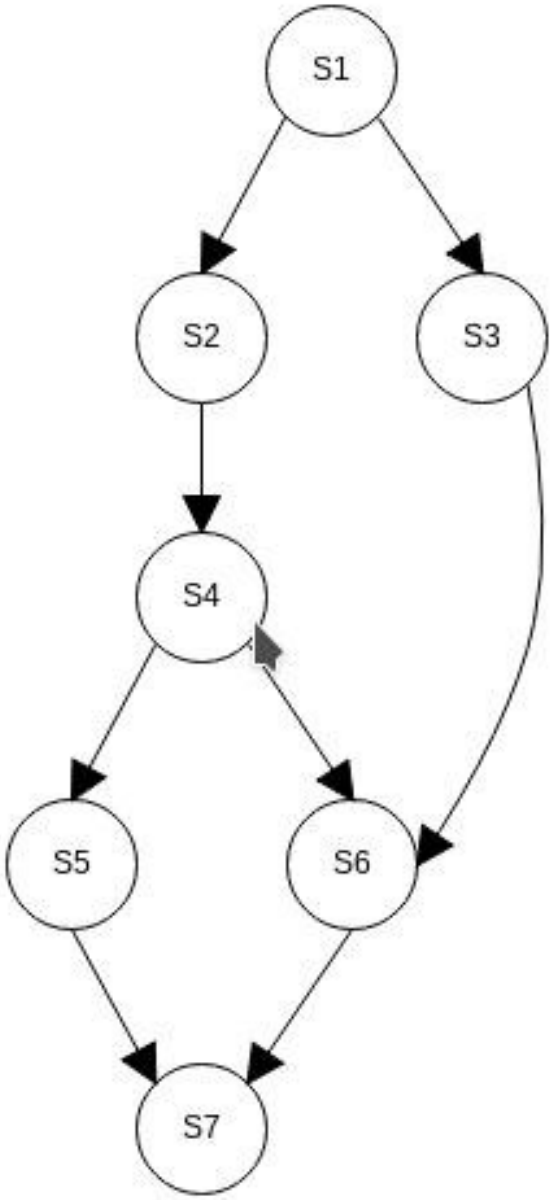
Cobegin-Coend y Grafos de Precedencia

- En este ejemplo, hemos definido dos tareas (*task1* y *task2*) como funciones asíncronas. Utilizamos la biblioteca **asyncio** para crear una función ***execute_tasks***() que ejecuta las tareas concurrentemente utilizando el mecanismo **await**.
- Las tareas se ejecutan de forma asíncrona y la función ***gather***() se encarga de esperar a que todas las tareas finalicen antes de continuar.
- La programación basada en eventos ofrece ventajas como un consumo eficiente de recursos y la posibilidad de manejar grandes volúmenes de tareas concurrentes. Sin embargo, también puede requerir un enfoque diferente y un mayor nivel de conocimiento para aprovechar al máximo su potencial.

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- Los grafos de precedencia son una representación gráfica de las tareas y sus dependencias en un programa concurrente.
- Ayudan a visualizar la planificación y la ejecución de las tareas de manera ordenada.
- Ejemplo práctico: Creación de un grafo de precedencia y planificación concurrente en Python.



CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- En este ejemplo, podemos ver un grafo de precedencia que representa la relación de dependencia entre diferentes tareas en un programa concurrente. Cada tarea se representa como un nodo en el grafo, y las flechas indican las dependencias entre ellas.
- Por ejemplo, la tarea S_{k+1} depende de la finalización de la tarea S_k.
- Utilizando el grafo de precedencia, se planifica la ejecución de las tareas de manera adecuada. Por ejemplo, se puede identificar tareas independientes que se pueden ejecutar simultáneamente y tareas que deben esperar a que otras finalicen antes de comenzar.
- Al utilizar grafos de precedencia en la programación concurrente, se mejora la eficiencia y evita problemas como condiciones de carrera.
- Estos grafos brindan una visión clara de las dependencias entre tareas y ayudan a coordinar su ejecución de manera adecuada.

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- Ejemplo: se crea una clase **Graph** que representa un grafo de precedencia utilizando un diccionario donde las tareas son las claves y las dependencias son las listas de valores.
- La función *add_dependency()* nos permite agregar dependencias entre tareas.
- Luego, se obtene las dependencias de una tarea específica utilizando la función *get_dependencies()*.
- Finalmente, se crea una lista *execution_order* que representa el orden de ejecución de las tareas según el grafo de precedencia y se recorre esa lista para ejecutar las tareas en el orden correcto utilizando la función *execute_task()*.

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_dependency(self, task, dependency):
        self.graph[task].append(dependency)
    def get_dependencies(self, task):
        return self.graph[task]
def execute_task(task):
    print("Ejecutando tarea:", task)
# Crear un grafo de precedencia
graph = Graph()
# Agregar dependencias
graph.add_dependency("A", "B")
graph.add_dependency("B", "C")
graph.add_dependency("B", "D")
graph.add_dependency("C", "E")
graph.add_dependency("D", "E")
# Obtener las dependencias de una tarea
task = "C"
dependencies = graph.get_dependencies(task)
print("Las dependencias de la tarea", task, "son:",
            dependencies)
# Ejecutar las tareas en el orden correcto según el
grafo de precedencia
execution_order = ["A", "B", "C", "D", "E"]
for task in execution_order:
    execute_task(task)
```

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- Algunas aplicaciones comunes de Cobegin-Coend y grafos de precedencia incluyen el procesamiento de lotes de datos, la ejecución paralela de algoritmos y la coordinación de tareas en sistemas distribuidos.

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

Ejercicio: Gestión de Pedidos en una Pizzería Cósmica (1h).

Descripción del problema: Eres el dueño de una pizzería cósmica y deseas optimizar la gestión de pedidos utilizando programación concurrente y grafos de precedencia. Cada pedido consta de varias etapas, como preparación de la masa, agregado de ingredientes y horneado. Quieres implementar un sistema que permita gestionar varios pedidos de forma concurrente y eficiente.

1. Define una lista de ingredientes cósmicos disponibles en la pizzería. Por ejemplo: ["queso lunar", "pepperoni estelar", "champiñones galácticos", "chorizo extraterrestre"].
2. Crea una lista de pedidos, donde cada pedido es una secuencia de etapas que deben ser realizadas en orden. Por ejemplo:

```
pedidos = [["preparar_masa", "agregar_salsa", "agregar_queso"],
           ["preparar_masa", "agregar_salsa", "agregar_queso", "agregar_pepperoni"],
           ["preparar_masa", "agregar_salsa", "agregar_queso", "agregar_champiñones"],
           ["preparar_masa", "agregar_salsa", "agregar_queso", "agregar_chorizo"]]
```

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

Ejercicio: Gestión de Pedidos en una Pizzería Cósmica (1h).

3. Implementa una función para cada etapa del pedido. Cada función debe recibir los ingredientes necesarios y realizar la tarea correspondiente, como "preparar_masa", "agregar_salsa", "agregar_queso", etc. Utiliza la biblioteca threading para ejecutar cada función en un hilo separado.
4. Utiliza un grafo de precedencia para definir las dependencias entre las etapas de cada pedido. Por ejemplo, si "agregar_queso" depende de "agregar_salsa", asegúrate de que la función "agregar_queso" se ejecute después de que la función "agregar_salsa" haya terminado.
5. Crea un hilo separado para cada pedido y ejecuta las etapas correspondientes en orden, respetando las dependencias definidas por el grafo de precedencia.
6. Espera a que todos los hilos finalicen y muestra el estado final de cada pedido, indicando si ha sido completado con éxito o si ha habido algún problema durante el proceso.

CONCURRENCIA

Cobegin-Coend y Grafos de Precedencia

- *En resumen, en esta clase hemos explorado Cobegin-Coend y grafos de precedencia como técnicas para la programación concurrente en Python. Hemos aprendido cómo ejecutar tareas de forma concurrente y coordinar su ejecución utilizando estos enfoques.*
- *En resumen, Cobegin-Coend y la programación basada en eventos son dos enfoques diferentes para la programación concurrente. Cada uno tiene sus propias ventajas y desventajas, y la elección depende del contexto y los requisitos del proyecto.*

- *Referencia bibliográfica:*
  - *"Python Concurrency: Performant Programming for the Modern Era" de Katherine Jarmul y Barry Warsaw.*
  - *"Python Parallel Programming Cookbook" de Giancarlo Zaccone.*
  - *"Operating Systems: Three Easy Pieces" de Remzi H. Arpaci-Dusseau y Andrea C. Arpaci-Dusseau.*

CONCURRENCIA

Problema de las secciones críticas

Definición:

- El problema de las secciones críticas se refiere a la situación en la que varios hilos o procesos comparten un recurso crítico y pueden acceder a él simultáneamente, lo que puede llevar a resultados inesperados o inconsistentes.

Características:

- Los hilos compiten por el acceso al recurso crítico.
- Si varios hilos acceden al recurso crítico al mismo tiempo, pueden producirse condiciones de carrera.
- La solución adecuada requiere la implementación de mecanismos de exclusión mutua para garantizar que solo un hilo pueda acceder al recurso crítico a la vez.

Aplicaciones:

- Sistemas de bases de datos: varios hilos pueden intentar acceder a la misma base de datos simultáneamente, lo que requiere mecanismos de exclusión mutua para garantizar la integridad de los datos.
- Sistemas operativos: los procesos en un sistema operativo pueden compartir recursos críticos como la memoria o los archivos, lo que requiere sincronización y exclusión mutua para evitar problemas de concurrencia.

CONCURRENCIA

Problema de las secciones críticas

Ejemplo de problema:

- El código presenta un recurso global **contador**, el cual es compartido por 2 hilos.
- Debido a la falta de exclusión mutua, los hilos pueden acceder al contador al mismo tiempo y causar resultados inconsistentes.
- El resultado esperado es 2M, pero no siempre se expresa de esa manera, porque 2 tareas pueden estar actualizando el recurso en simultáneo.

```
import threading
contador = 0
def incrementar():
    global contador
    for _ in range(1000000):
        contador += 1
def main():
    # Crear hilos
    hilo1 = threading.Thread(target=incrementar)
    hilo2 = threading.Thread(target=incrementar)
    # Iniciar hilos
    hilo1.start()
    hilo2.start()
    # Esperar a que los hilos terminen
    hilo1.join()
    hilo2.join()
    # Imprimir el valor final del contador
    print("Contador:", contador)
if __name__ == "__main__":
    main()
```

CONCURRENCIA

Problema de las secciones críticas

Solución de problema:

- El código agrega un objeto **Lock**, el cual bloquea acceso al recurso global **contador**.
- El bloque **with mutex** antes de incrementar el contador asegura que ningún otro hilo pueda acceder a él hasta que se libere el bloqueo.
- El resultado 2M está asegurado.

```
import threading
contador = 0
mutex = threading.Lock()
def incrementar():
    global contador
    for _ in range(1000000):
        with mutex:
            contador += 1
def main():
    # Crear hilos
    hilo1 = threading.Thread(target=incrementar)
    hilo2 = threading.Thread(target=incrementar)
    # Iniciar hilos
    hilo1.start()
    hilo2.start()
    # Esperar a que los hilos terminen
    hilo1.join()
    hilo2.join()
    # Imprimir el valor final del contador
    print("Contador:", contador)
if __name__ == "__main__":
    main()
```

CONCURRENCIA

Problema de la mutua exclusión

Definición:

- El problema de la mutua exclusión se refiere a la necesidad de garantizar que solo un hilo o proceso pueda acceder a un recurso compartido a la vez, evitando condiciones de carrera y garantizando la consistencia de los datos.

Características:

- Los hilos o procesos compiten por el acceso a un recurso compartido.
- Si varios hilos acceden al recurso compartido al mismo tiempo, pueden producirse resultados inconsistentes o incorrectos.
- Se requiere la implementación de técnicas de exclusión mutua para garantizar que solo un hilo o proceso pueda acceder al recurso compartido a la vez.

Aplicaciones:

- Sistemas de control de acceso: varios usuarios pueden intentar acceder a un recurso compartido, como una base de datos o un archivo, lo que requiere exclusión mutua para evitar problemas de concurrencia.
- Sistemas de cola o gestión de recursos: cuando varios hilos o procesos necesitan acceder a un recurso limitado, como una impresora o un servidor, es necesario garantizar la exclusión mutua para un uso correcto y eficiente del recurso.

CONCURRENCIA

Problema de la mutua exclusión

Ejemplo de problema:

- En este ejemplo, ambos hilos intentan agregar elementos a la lista compartida al mismo tiempo, lo que puede resultar en un comportamiento inesperado debido a la falta de exclusión mutua.
- Puede haber condiciones de carrera donde los hilos se intercalan y modifican la lista simultáneamente, lo que lleva a resultados incorrectos o inconsistentes.
- Se utiliza la función **time.sleep()** con el fin de simular un escenario donde cada hilo tiene diferentes tiempos de procesamiento antes de realizar su operación en la lista.

```
import threading, time, random
recurso_compartido = []
def agregar(elemento):
    time.sleep(random.uniform(0.5, 1.5))  #Espera
    recurso_compartido.append(elemento)
def main():
    # Crear hilos
    hilo1 = threading.Thread(target=agregar,
args=("Elemento 1",))
    hilo2 = threading.Thread(target=agregar,
args=("Elemento 2",))
    # Iniciar hilos
    hilo1.start()
    hilo2.start()
    # Esperar a que los hilos terminen
    hilo1.join()
    hilo2.join()
    # Imprimir el recurso compartido
    print("Recurso compartido:",
recurso_compartido)
if __name__ == "__main__":
    main()
```

CONCURRENCIA

Problema de la mutua exclusión

Solución de problema:

- El código agrega un objeto **Lock**, el cual bloquea acceso al recurso global **contador**.
- El bloque **with mutex** antes de incrementar el contador asegura que ningún otro hilo pueda acceder a él hasta que se libere el bloqueo.
- El resultado 2M está asegurado.

```
import threading, time, random
recurso_compartido = []
semaphore = threading.Semaphore()
def agregar(elemento):
    semaphore.acquire()
    try:
        time.sleep(random.uniform(0.5, 1.5))
        recurso_compartido.append(elemento)
    finally:
        semaphore.release()
def main():
    # Crear hilos
    hilo1 = threading.Thread(target=agregar,
args=("Elemento 1",))
    hilo2 = threading.Thread(target=agregar,
args=("Elemento 2",))
    hilo1.start(), hilo2.start()
    hilo1.join(), hilo2.join()
    print("Recurso compartido:",
recurso_compartido)
if __name__ == "__main__":
    main()
```

CONCURRENCIA

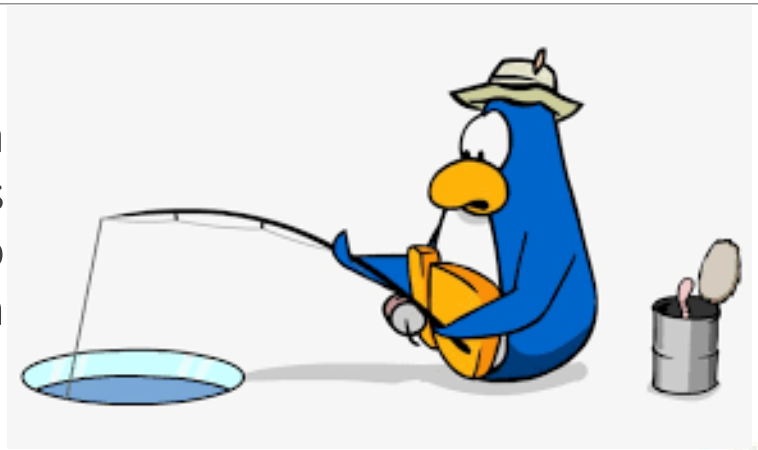
Problema de la mutua exclusión

- *En resumen, las secciones críticas y la mutua exclusión son conceptos fundamentales en la programación concurrente y se utilizan para garantizar la integridad de los datos compartidos en un entorno multi-hilo.*
- *Una sección crítica es una parte del código en la que se acceden o modifican datos compartidos por múltiples hilos de ejecución. Es importante garantizar que solo un hilo pueda ejecutar la sección crítica a la vez para evitar problemas como condiciones de carrera y resultados inconsistentes.*
- *La mutua exclusión es el principio que garantiza que solo un hilo pueda ejecutar una sección crítica en un momento dado. Se utiliza para evitar conflictos y garantizar la consistencia de los datos compartidos. La implementación de la mutua exclusión puede lograrse a través de diversas técnicas, como el uso de semáforos, cerraduras (locks) o variables condicionales.*
- *Referencia bibliográfica:*
  - *Tanenbaum, A. S., & van Steen, M. (2007). Distributed systems: principles and paradigms. Pearson Education.*
  - *Andrews, G. R. (2000). Foundations of multithreaded, parallel, and distributed programming. Addison-Wesley Professional.*

CONCURRENCIA

Ejercicio: La competencia de los pingüinos glotones

En un mundo helado, se está llevando a cabo una competencia entre pingüinos para ver quién come más peces en un tiempo determinado. Sin embargo, solo hay un área de pesca disponible y los pingüinos deben competir por acceder a ella de manera segura y justa.



En este ejercicio, se pide implementar un programa que simule la competencia de los pingüinos glotones, aplicando los conceptos de secciones críticas y mutua exclusión.

Cada pingüino es representado por un hilo y la sección crítica es el área de pesca, donde solo un pingüino puede acceder a la vez.

CONCURRENCIA

Ejercicio: La competencia de los pingüinos glotones

El programa debe cumplir con las siguientes especificaciones:

1. Cada pingüino debe ser representado por una clase Pingüino, que herede de la clase Thread.
2. Cada pingüino debe tener un nombre único y una cantidad de peces que ha comido.
3. La cantidad total de peces disponibles en el área de pesca es limitada.
4. Un pingüino puede comer un pez a la vez y debe esperar su turno para acceder al área de pesca.
5. Se debe garantizar la exclusión mutua para evitar que dos pingüinos accedan al área de pesca al mismo tiempo.
6. Al finalizar el tiempo establecido, se debe mostrar el nombre del pingüino ganador, es decir, aquel que haya comido la mayor cantidad de peces.

Nota: Puedes utilizar semáforos o mutex para implementar la solución al problema de la sección crítica y la mutua exclusión.