Programación Científica y HPC

Programación concurrente

Índice

Esquema

PROGRAMACIÓN CONCURRENTE

CONCEPTOS BÁSICOS

- Programación concurrente
- GIL
- Entidades concurrentes

MÓDULO THREADING

- Clase `Thread`
- Estados de un hilo

COMUNICACIÓN ENTRE HILOS

- Monitores
- Colas (Queue)

MÉTODOS DE SINCRONIZACIÓN

- Cerrojos `Lock` y `Rlock`
- Semáforos
- Condiciones

PROBLEMAS EN PC

- Problemas de seguridad
- Problemas de vivacidad
- Prevención de problemas

EJEMPLOS CLÁSICOS DE LA PROGRAMACIÓN CONCURRENTE

- La cena de los filósofos
- El problema de los lectores-escritores

Ideas clave

7.1. Introducción y objetivos

La concurrencia en Python puede ser confusa, debido a que hay múltiples opciones para poder implementarla. Pero antes de afrontar este tema es necesario tratar algunos conceptos importantes.

En primer lugar, se presentan las siguientes definiciones:

Concurrencia es la ejecución simultánea de conjuntos de instrucciones que guardan cierta independencia.

Programación concurrente es el paradigma de programación que permite la creación de programas con ejecución simultánea de múltiples tareas.

Se debe tener en cuenta que **esta ejecución simultánea no es real** dado que, si existe un único procesador, las tareas deben compartirlo y, por tanto, en cada instante solo se encuentra una de ellas ejecutándose. En este tipo de programación, cuando una tarea está a la espera de una operación de entrada/salida se libera el procesador que puede ser usado por otra. Los problemas de compartir procesador y la falsa simultaneidad se resolverían con tantos procesadores como tareas concurrentes se quieran ejecutar.

Para entender las características y las diferencias de la programación concurrente, con los otros tipos de programación, es importante tener en cuenta que, de acuerdo con la forma de procesar las tareas, **existen distintos entornos de programación y formas de programar:**

- **Multiprogramación:** gestión de procesos (o hilos) en un sistema monoprocesador. Da lugar a lo que se conoce como **programación concurrente** que, como se ha visto, consiste en la ejecución «simultanea» de tareas.

- **Multiprocesamiento:** gestión de procesos en un sistema multiprocesador en el que puede existir memoria común. Da lugar a lo que se conoce como **programación paralela,** en la que se ejecutan tareas de forma simultánea, pero sobre distintos procesadores.

- **Procesamiento distribuido:** gestión de procesos en procesadores separados con memoria no compartida. Da lugar a lo que se conoce como **programación distribuida,** en la que se ejecutan tareas de forma simultánea sobre distintos procesadores pero que no comparten memoria.

Entidades concurrentes

En primer lugar, se debe considerar que tipos de entidades concurrentes están disponibles en los distintos lenguajes de programación.

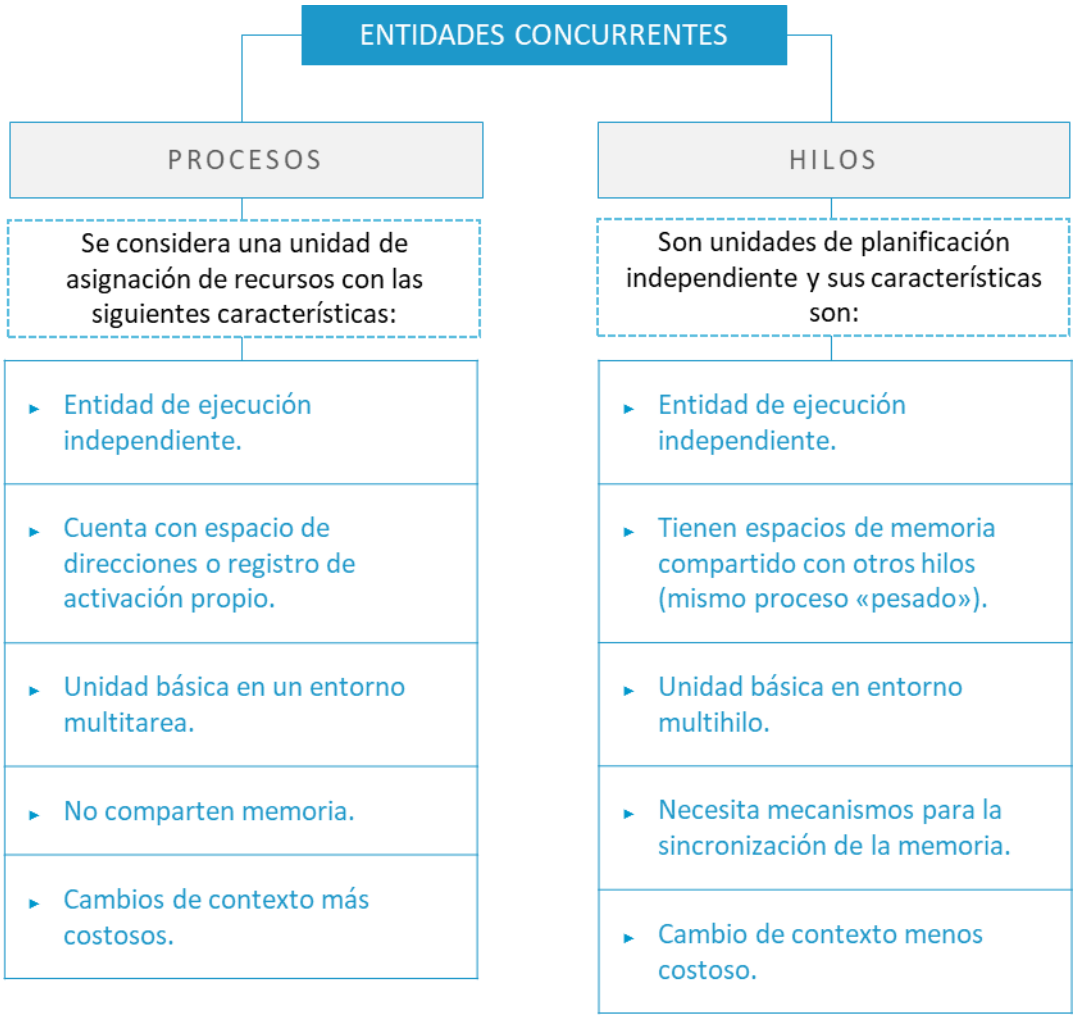

Figura1. Tipos de entidades concurrentes. Fuente: elaboración propia.

Las terminologías que se suelen usar son las de **proceso pesado,** para denominar a los procesos, y **proceso ligero,** para denominar a los hilos. Python no cuenta con procesos pesados, exceptuando el propio proceso principal.

Un esquema general de un programa con procesos e hilos se muestra en la siguiente Figura:

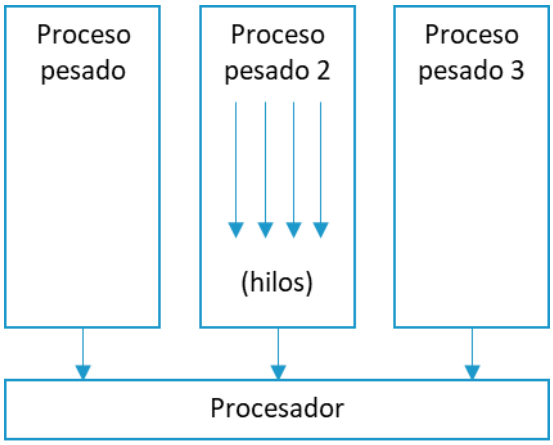

Figura 2. Programa concurrente con procesos e hilos. Fuente: elaboración propia.

Por tanto, el modelo de programación concurrente que usa Python es un modelo con hilos que está basado en el modelo de programación concurrente de Java.

Se verán características propias de la programación concurrente como la sincronización y la comunicación entre hilos y los problemas de vivacidad que se producen en los programas concurrentes.

Global Interpreter Lock (GIL)

El bloqueo o cerrojo global del interprete es un bloqueo asociado a un proceso o hilo. Mediante este bloqueo se asegura que solo un hilo puede acceder a un recurso particular, e impide el uso de objetos y `bytecodes` a la vez.

Por tanto, en Python se usa un bloqueo global en cada proceso que se está interpretando y, de esta forma, se ejecuta sobre el procesador. Esto significa que **cada proceso trata al propio intérprete de Python como un recurso.**

Cuando un hilo comienza a ejecutarse adquiere el GIL, de forma que no puedo ser adquirido por otro hilo. Si otro hilo quiere ejecutarse debe esperar a que se libere el bloqueo. Esto solo ocurre cuando el hilo termina, o bien queda a la espera, por una operación de entrada/salida o queda suspendido por una espera de tiempo.

La existencia de GIL no supone un problema si existe un procesador de un solo núcleo, porque se produce lo que se conoce como el *time slicing* en la planificación de los hilos. Mediante esta técnica, **el procesador es compartido por varios hilos.**

Cada diez instrucciones de `bytecode` por defecto (se puede modificar), cuando el hilo que se encontraba en el procesador debe realizar una operación de entrada y salida o se pone a dormir por la ejecución de una instrucción, **se produce un cambio de contexto** y, por tanto, el procesador será abandonado por el hilo que se estaba ejecutando y pasará a ejecutarse otro. Cuanto más optimizado este el `bytecode`, menor será el número de cambios de contexto.

Sin embargo, en el caso de los procesadores multinúcleo, aunque haya núcleos disponibles, solo se puede ejecutar el hilo que adquirió el bloqueo y, por tanto, no se estarán utilizando todos los núcleos disponibles al mismo tiempo.

La razón de la implementación del GIL en Python se debe, entre otras cosas, al **recolector de basura.** Este está implementado mediante un contador de referencias a cada objeto de forma que, cuando este esté a cero, la memoria asignada a un objeto se liberará. El problema es que este **contador es una variable global y es susceptible de verse afectada por condiciones de carrera,** que podrían ocasionar valores inconsistentes para el mismo.

Una condición de carrera ocurre cuando dos o más hilos, o procesos, acceden a un recurso en el que el acceso depende del orden de llegada y el tipo de operación se completa según una secuencia. Durante estas etapas otro hilo puede acceder al recurso, dejándolo en estado inconsistente.

Estas **condiciones de carrera son habituales en programas concurrente**s y, para evitar los problemas que produce, se debe acudir al uso de mecanismos de sincronización que permitirán el acceso adecuado al recurso compartido.

No obstante, el contador de referencia no es la única razón para el uso de GIL, también **ha servido para que la implementación de Python sea mucho más sencilla e incrementa la velocidad de ejecución cuando se usa un único hilo.**

Como conclusión **el GIL es una característica de implementación del intérprete.** CPython y PyPy lo tienen, pero Jython y IronPython no lo usan.

Aspectos que se deben considerar en la concurrencia

Tras lo expuesto, se enumeran algunos aspectos que se deben tener en cuenta cuando se usa la concurrencia:

- Planificación de los hilos o procesos, que determinará que hilo o proceso estará activo en cada momento.
- Asignación de memoria a los hilos o procesos.
- Sincronización de acceso a recursos compartidos o la falta de recursos.
- Prevención de problemas de falta de vitalidad asociados a la programación concurrente como, por ejemplo, los bloqueos entre hilos o procesos.

7.2. El módulo `threading`

Para implementar hilos en Python, se usa el módulo `threading`, que es un módulo de implementación de hilos de alto nivel. También existe el módulo `thread` pero este se considera obsoleto y, por tanto, no lo veremos, aunque sigue disponible por compatibilidad con desarrollos en versiones previas de Python.

El modelo de programación concurrente con hilos de Python está basado en el modelo de hilos de Java.

A continuación, se pasa a describir algunas de sus características más importantes.

El módulo `threading` cuenta con 4 tipos de elementos:

- **Una serie de clases,** entre las que cabe destacar la clase `Thread` para definir los hilos propiamente dichos.
- **Métodos para la creación y activación** de hilos y otros de consulta sobre estados y otras cuestiones de interés.
- **Objetos para sincronización** que permiten adquirir y liberar cerrojos sobre recursos compartidos.
- **Excepciones para informar de los problemas** que se producen en tiempo de ejecución sobre los hilos, para que puedan ser manejados de forma conveniente.

La clase `Thread`

Un hilo en Python es un objeto de la clase `Thread` o de una clase derivada de la misma. En esta se cuenta con las operaciones básicas que se deben implementar o sobrecargar en las clases derivadas que se definan.

Los objetos de estas clases representan secuencias de código que se ejecutan mediante un hilo de control propio.

Los pasos que hay que seguir para crear hilos en Python son:

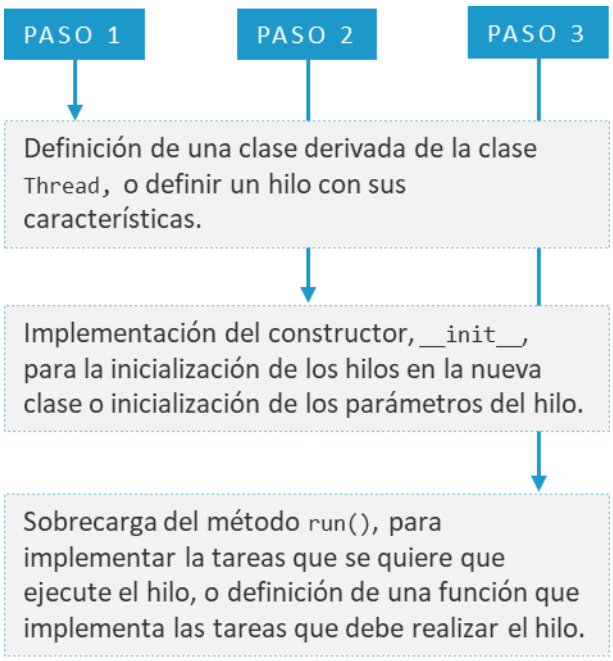

Figura 3. Pasos para la creación de hilos en Python. Fuente: elaboración propia.

Para la ejecución de un hilo se deben realizar, al menos, los siguientes pasos:

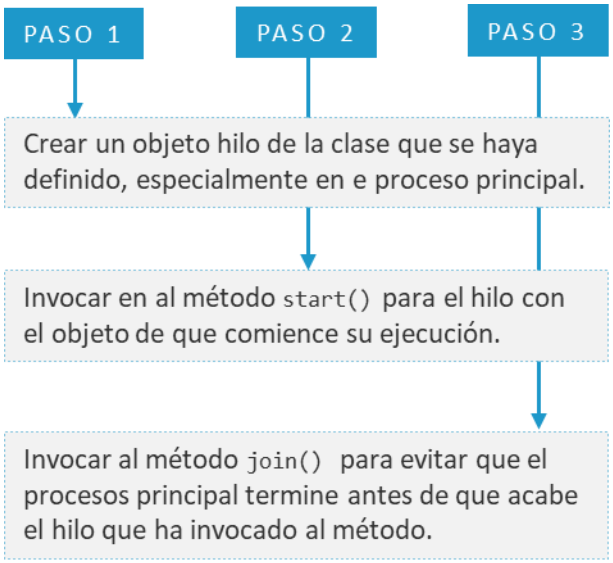

Figura 4. Pasos para la ejecución de un hilo. Fuente: elaboración propia.

En la siguiente Tabla se enumeran algunas de las principales funciones para manejar hilos:

Métodos	Tipo	Descripción
`isAlive()`	Método de objeto	Devuelve *True* si el hilo está vivo.
`start()`	Método de clase	Es un método que se invoca una única vez para que el hilo comience su ejecución.
`run()`	Método de clase	Este método define las tareas que se van a ejecutar para un hilo en concreto. Este método se sobrecarga para cada clase de hilo que se define.
`join()`	Método de clase	Bloquea la finalización de otro código hasta que el hilo en el que se llamó al método `join()` finalice.

Tabla 1. Métodos del módulo `threading` para hilos. Fuente: elaboración propia.

Ejemplo 1. Creación de hilos

En el siguiente ejemplo se verá la creación de hilos de la clase `Thread`.

```
import threading
import time
import logging

def tareaHilo(tiempo):
    print ("Comienzo de la ejecución del %s \n" %threading.currentThread().name)
    i=5
    while i:
        time.sleep(tiempo)
        print ("Ejecutándose %s \n" %threading.currentThread().name)
        i = i - 1
    print ("%s terminado " %threading.currentThread().name)

if __name__=="__main__":
    hilo1 = threading.Thread(name="Hilo Uno", target=tareaHilo,  args=(3,))
    hilo2 = threading.Thread(name="Hilo Dos", target=tareaHilo,  args=(2,))
    hilo3 = threading.Thread(name="Hilo Tres",target=tareaHilo,  args=(5,))
    hilo1.start()
    hilo2.start()
    hilo3.start()

    hilo1.join()
    hilo2.join()
    hilo3.join()
```

```
-------------------------------------------------------
Ejecución:
Comienzo de la ejecución del Hilo Uno
Comienzo de la ejecución del Hilo Dos
Comienzo de la ejecución del Hilo Tres
Ejecutándose Hilo Dos
Ejecutándose Hilo Uno
Ejecutándose Hilo Dos
Ejecutándose Hilo Tres
Ejecutándose Hilo Uno
Ejecutándose Hilo Dos
Ejecutándose Hilo Dos
Ejecutándose Hilo Uno
Ejecutándose Hilo Tres
Ejecutándose Hilo Dos
Hilo Dos terminado
Ejecutándose Hilo Uno
Ejecutándose Hilo Tres
Ejecutándose Hilo Uno
Hilo Uno terminado
Ejecutándose Hilo Tres
Ejecutándose Hilo Tres
Hilo Tres terminado
```

Figura 5. Código del Ejemplo 1. Fuente: elaboración propia.

En el ejemplo anterior se han creado tres hilos a los que se les ha puesto un nombre, y para los que se ha determinado el tiempo que simula el retardo de la ejecución, para lo que se ha usado la función `time.sleep()`. Esto produce un efecto sobre el estado del hilo, que hace que esté suspendido y pueda asignarse el procesador a otro hilo que lo espere.

Después de la creación se ha invocado el método `start()` para activar el hilo, y con el método `join()` se impide la finalización del proceso principal hasta que acabe la ejecución de los hilos que se han unido.

Ejemplo 2. Creación de clases hilo

En el siguiente ejemplo se verá la declaración de una clase que deriva de la clase `Thread`, para crear una serie de hilos.

```
import time
import threading

class HiloEjemplo (threading.Thread):
    def __init__(self, id, nombre, tiempo):
       threading.Thread.__init__(self)
       self.id = id
       self.name = nombre
       self.tiempo = tiempo

    def run(self):
        print ("Comienzo de %s \n" %self.name)
        i=5
        while i:
           time.sleep(self.tiempo)
           print ("Ejecutándose %s \n" %self.name)
           i = i - 1
        print ("%s terminado " %self.name)

if __name__=="__main__":
    hilo1 = HiloEjemplo(1, "Hilo Uno", 5)
    hilo2 = HiloEjemplo(2, "Hilo Dos", 2)
    hilo3 = HiloEjemplo(3, "Hilo Tres", 3)

    hilo1.start()
    hilo2.start()
    hilo3.start()

    hilo1.join()
    hilo2.join()
    hilo3.join()
```

```
------------------------------------------------
Ejecución:
Comienzo de Hilo Uno
Comienzo de Hilo Dos
Comienzo de Hilo Tres
Ejecutándose Hilo Dos
Ejecutándose Hilo Tres
Ejecutándose Hilo Dos
Ejecutándose Hilo Uno
Ejecutándose Hilo Tres
Ejecutándose Hilo Dos
Ejecutándose Hilo Dos
Ejecutándose Hilo Tres
Ejecutándose Hilo Uno
Ejecutándose Hilo Dos
Hilo Dos terminado
Ejecutándose Hilo Tres
Ejecutándose Hilo Uno
Ejecutándose Hilo Tres
Hilo Tres terminado
Ejecutándose Hilo Uno
Ejecutándose Hilo Uno
Hilo Uno terminado
```

Figura 6. Código del Ejemplo 2. Fuente: elaboración propia.

En este ejemplo se define una clase derivada de la clase `Thread` del módulo `threading`, en la que se ha implementado el constructor de la nueva clase invocando al constructor de la clase padre. En este se cambia el nombre del hilo. Para establecer

los hilos se crearán objetos de la clase que se activan con `start()` como en el Ejemplo 1. El uso de `join()` también tiene el mismo efecto que en el primer ejemplo.

Otras funciones de `threading`, que son de interés para el uso con hilos, son mostradas en la siguiente tabla:

Métodos	Descripción
`activeCount()`	Devuelve el número de hilos que siguen vivos.
`currentThread()`	Retorna el hilo actual.
`enumerate()`	Lista todos los hilos vivos.

Tabla 2. Funciones para consulta sobre hilos. Fuente: elaboración propia.

Consulte en el aula virtual el notebook. CreaciondeHilos.ipynb

A continuación, se presentan los métodos de sincronización y comunicación de hilos con los que cuenta Python y los problemas que pueden plantearse en la programación concurrente

De acuerdo con los métodos y las formas de planificación y ejecución de los hilos es importante introducir los estados por los que puede pasar un hilo.

Estados de un hilo

Como conclusión de todo lo visto, se puede determinar que un hilo pasa por cuatro estados:

- **Creado:** se ha invocado al constructor para construir el hilo, pero todavía no ha empezado su ejecución.

- **Ejecutable:** se invoca al método `start()` para el hilo. Puede comenzar su ejecución, pero depende de la planificación del procesador. Este estado se compone de dos subestados entre los que el hilo transita varias veces, dependiendo de si tiene asignado el procesador o no. Estos son:
  - **Preparado:** puede empezar su ejecución, pero no cuenta con el procesador.
  - **En ejecución:** tiene asignado el procesador.

- **Parado:** el hilo no avanza en la ejecución y libera el procesador. Hay básicamente cuatro estados posibles:
  - **En espera:** el hilo está en la cola de una condición por la ejecución de un `wait()`, solo sale de esta si se produce un `notify()` o `notifyAll()`.
  - **Dormido:** se ejecuta un `sleep()`, volverá a estar ejecutable cuando pase el tiempo.
  - **Bloqueado:** cuando el hilo está a la espera de poder adquirir un cerrojo o viene a la espera de una operación de entrada/salida (E/S).
  - **Terminado:** el hilo acaba su ejecución porque completa el código de su método asociado, o bien, se produce una excepción no controlada o una interrupción.

En la Figura 7 se muestran los estados de un hilo:

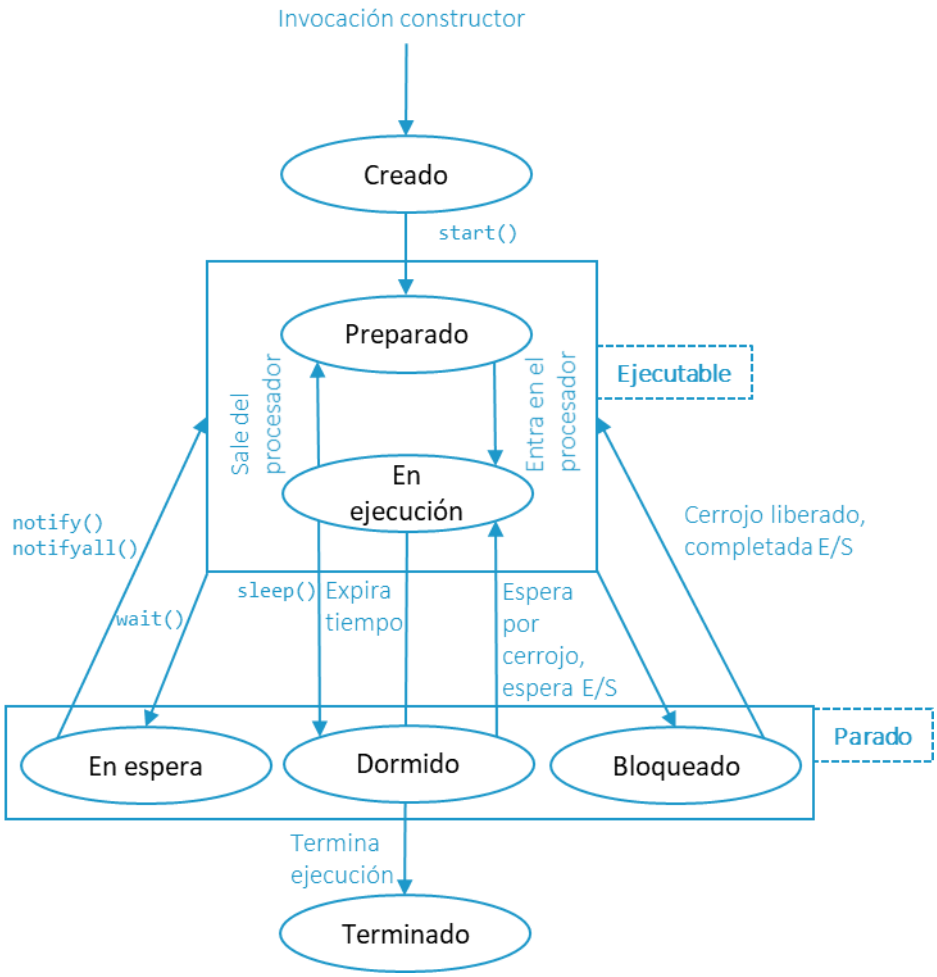

Figura 7. Estados de un hilo. Fuente: elaboración propia.

7.3. Métodos de sincronización

La sincronización es la **acción que permite el acceso a recursos compartidos por varios hilos** y asegura la consistencia en el acceso a estos, mientras garantiza estados coherentes.

Figura 8. Gestión y mecanismos de los métodos de sincronización. Fuente: elaboración propia.

Para tratar las condiciones de carrera, los bloqueos y otros problemas basados en hilos, el módulo de `threading` incorpora distintos mecanismos para manejar la sincronización entre hilos de acuerdo con las condiciones que se necesiten establecer. A continuación, veremos el uso de algunos de ellos.

Lock

Cuando un hilo quiere acceder a un recurso específico, adquiere un cerrojo para ese recurso. Una vez que un hilo bloquea un recurso concreto, ningún otro puede acceder a él hasta que se libere el bloqueo. Como resultado, los cambios en el recurso serán atómicos y se evitarán las condiciones de carrera.

Un cerrojo es una primitiva de sincronización de bajo nivel. Se implementa en el módulo `_thread`. Estos soportan dos métodos, cuyo uso produce los cambios de estado en el mismo.

- **Método `acquire()`:** el hilo que lo invoca adquiere el cerrojo y se produce el cambio de estado de un cerrojo a estado bloqueado, si este estaba desbloqueado. En otro caso, se bloquea la llamada hasta que el cerrojo pueda ser adquirido.

- **Método `release()`:** se utiliza para cambiar el estado del cerrojo a desbloqueado, es decir, lo libera. Puede ser llamado por cualquier hilo, no necesariamente el que adquirió el bloqueo.

Ejemplo 3. Acceso compartido sin cerrojos

Se muestra un ejemplo en el que el acceso compartido no es protegido y se observan resultados inconsistentes. Hay que tener en cuenta que la salida varía con cada ejecución.

```
import threading
import time


def incrementar():
    global contador
    for i in range(1000000):
        contador+=1



def decrementar():
    global contador
    for i in range(1000000):
        contador-=1


if __name__=="__main__":
    contador=0

    hilo1 = threading.Thread(target=incrementar)
    hilo2 = threading.Thread(target=decrementar)

    hilo1.start()
    hilo2.start()

    hilo1.join()
    hilo2.join()

    print(contador)
```

```
------------------------------------------------
Salida:
300607
```

Figura 9. Código del Ejemplo 3. Fuente: elaboración propia.

Ejemplo 4. Uso de cerrojos

En el siguiente ejemplo se verá el uso de cerrojos para proteger el acceso compartido a un recurso por parte de varios hilos. El resultado obtenido, como se ve, es coherente.

```
import threading
import time

cerrojo = threading.Lock()
def incrementar():
    global contador

    for i in range(1000000):
        cerrojo.acquire()
        contador+=1
        cerrojo.release()


def decrementar():
    global contador
    for i in range(1000000):
        cerrojo.acquire()
        contador-=1
        cerrojo.release()


if __name__=="__main__":
    contador=0

    hilo1 = threading.Thread(target=incrementar)
    hilo2 = threading.Thread(target=decrementar)

    hilo1.start()
    hilo2.start()

    hilo1.join()
    hilo2.join()

    print(contador)
```

```
-----------------------------------------
Salida:
0
```

Figura 10. Código del Ejemplo 4. Fuente: elaboración propia.

Existen cerrojos recursivos, `Rlock`, que pueden ser adquiridos varias veces por un mismo hilo sin necesidad de que haya sido liberado previamente.

Semáforos

Los semáforos **son cerrojos, pero que pueden permitir el acceso de más de un hilo a un recurso a la vez, aunque limita el número total.** Se debe de inicializar a un valor que por defecto es 1. Por ejemplo, un *pool* de conexiones puede soportar un número fijo de conexiones simultáneas, o una aplicación de red puede soportar un número fijo de descargas simultáneas. Un semáforo es una **forma de gestionar esas conexiones.**

Ejemplo 5. Gestión de conexiones con semáforos

En el siguiente ejemplo se verá el uso de `Semaphore` para gestionar el máximo de conexiones activas en un sistema. Es importante indicar que se hace uso del cerrojo mediante el uso de `with`, que maneja el contexto en el que se aplica el cerrojo al tiempo que lo adquiere de forma implícita y lo libera cuando termina el contexto marcado. En este caso se hace uso de `logging` para rastrear la ejecución

```
import logging
import random
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format=' (%(threadName)-2s) %(message)s',
                    )

class GestorConexiones(object):
    def __init__(self):
        threading.Thread.__init__(self)
        self.activas = []
        self.cerrojo = threading.Lock()
    def activar(self, name):
        with self.cerrojo:
            self.activas.append(name)
            logging.debug('Conexiones activas: %s', self.activas)
    def desactivar(self, name):
        with self.cerrojo:
            self.activas.remove(name)
            logging.debug('Conexiones activas: %s', self.activas)

def conexion(s, gestor):
    logging.debug('Esperando para unirse al gestor')
    with s:
        nombre = threading.currentThread().getName()
        gestor.activar(nombre)
        time.sleep(0.1)
        gestor.desactivar(nombre)

gestor = GestorConexiones()
semaforo = threading.Semaphore(2)
for i in range(4):
    t = threading.Thread(target=conexion, name="Hilo "+str(i), args=(semaforo, gestor))
    t.start()
```

Ejecución:

```
(Hilo 0) Esperando para unirse al gestor
(Hilo 1) Esperando para unirse al gestor
(Hilo 0) Conexiones activas: ['Hilo 0']
(Hilo 2) Esperando para unirse al gestor
(Hilo 3) Esperando para unirse al gestor
(Hilo 1) Conexiones activas: ['Hilo 0', 'Hilo 1']
(Hilo 0) Conexiones activas: ['Hilo 1']
(Hilo 2) Conexiones activas: ['Hilo 1', 'Hilo 2']
(Hilo 1) Conexiones activas: ['Hilo 2']
(Hilo 3) Conexiones activas: ['Hilo 2', 'Hilo 3']
(Hilo 2) Conexiones activas: ['Hilo 3']
(Hilo 3) Conexiones activas: []
```

Figura 11. Código del Ejemplo 5. Fuente: elaboración propia.

Condiciones

Otra forma de sincronizar hilos es mediante el uso de un objeto de tipo `Condition`. Dado que las condiciones también usan cerrojos, se pueden vincular al uso de recursos compartido.

En este caso, cuando se adquiere el cerrojo, **si no se cumple una determinada condición, el hilo se pone a la espera en una cola asociada a este cerrojo** hasta que el recurso esté en condiciones de soportar la operación y sea activado el hilo de la cola.

Además de las operaciones de adquisición y liberación, las condiciones soportan las siguientes operaciones:

- `wait()`: detiene la ejecución del hilo hasta que se cumplen las condiciones necesarias para continuar.
- `notify()`: activa uno de los hilos que están en espera.
- `notifyAll()`: activa todos los hilos que están en espera y que se ejecutarán según la estrategia de planificación.

> Ejemplo 6. Condiciones
>
> En este ejemplo, hay unos hilos consumidores y un productor. Los consumidores esperan con `wait()` a que el productor ponga el recurso disponible, y activan los hilos en espera con `notify()`. En el ejemplo no se muestra cual es la condición que debe cumplirse.

```python
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format=' (%(threadName)-2s) %(message)s',
                    )

def consumir(cond):
    """Consumirá el recurso si  se cumplen  las condiciones de uso"""
    logging.debug('Espera para  consumir')
    t = threading.currentThread()

    with cond:
        cond.wait()
        logging.debug('Empieza a consumir')

def producir(cond):
    """Actualiza el recurso para  que  sea usado por  el consumidor"""
    logging.debug('Comienza a  producir')
    with cond:
        logging.debug('Pone disponible el recurso')
        cond.notifyAll()

condicion = threading.Condition()
consumidor1 = threading.Thread(name='Consumidor 1', target=consumir, args=(condicion,))
consumidor2 = threading.Thread(name='Consumidor 2', target=consumir, args=(condicion,))
productor1 = threading.Thread(name='Productor 1', target=producir, args=(condicion,))

consumidor1.start()
time.sleep(3)
consumidor2.start()
time.sleep(2)

productor1.start()
```

Ejecución:

```
(Consumidor 1) Espera para  consumir
(Consumidor 2) Espera para  consumir
(Productor 1) Comienza a  producir
(Productor 1) Pone disponible el recurso
(Consumidor 1) Empieza a consumir
(Consumidor 2) Empieza a consumir
```

Figura 12. Código del ejemplo 6. Fuente: elaboración propia.

7.4. Comunicación entre hilos

En la programación concurrente propiamente dicha, **la comunicación entre procesos de hilos se basa,** fundamentalmente, **en el uso de recursos compartidos.**

Monitores

Una buena práctica para la comunicación es la implementación de lo que se conocen como monitores, que son **objetos que representan recursos compartidos de acceso sincronizado y en exclusión mutua.** Permite que los hilos sean bloqueados si no se cumple una condición y queden en una cola de espera hasta que otro hilo, usando

otro de los métodos, le notifique que puede continuar la ejecución porque se ha alcanzado la condición.

En Python, un monitor puede ser implementado mediante una clase en la que los métodos tienen implementadas condiciones.

Ejemplo 7. Productor consumidor sin buffer de almacenamiento

En este ejemplo se implementará un monitor con una variable compartida y dos métodos de accesos para obtener el valor y generarlo.

Además, se implementa una clase productor y una clase consumidor que hacen al monitor compartido. Aunque los consumidores empiezan antes, deben esperar en una condición a que el productor genere el primer valor. El valor proporcionado por el productor se genera aleatoriamente.

Como el productor es más rápido que los consumidores, hay valores del productor que no serán consultados por los productores, por lo que los consumidores consumen varias veces el mismo valor. Por tanto, se trata de ajustar estos parámetros.

```
import random
import threading
import time

class RecursoMonitor:
    def __init__(self,cerrojo):
        self.valor=None
        self.cerrojo=cerrojo

    def consumir(self):
        with self.cerrojo:
            print("El "+threading.currentThread().getName()+" espera a que haya un valor de consulta  ")
            if self.valor==None:

                self.cerrojo.wait()
            texto="El{hilo} ha obtenido el valor {valor}.\n"
            print (texto.format(hilo=threading.currentThread().getName(), valor=self.valor))
            return self.valor

    def producir(self, valor):
        with self.cerrojo:
            self.valor=valor
            texto="El{hilo} ha producido el valor {valor}.\n"
            print (texto.format(hilo=threading.currentThread().getName(), valor=self.valor))
            self.cerrojo.notifyAll()

class Consumidor(threading.Thread):
    def __init__(self, nombre, num_consultas, recurso):
        threading.Thread.__init__(self)
        self.name=nombre
        self.valor=None
        self.num_consultas=num_consultas
        self.recurso=recurso

    def run(self):
        print("Comienza "+self.name)
        for i in range(self.num_consultas):
            self.valor=self.recurso.consumir()

            time.sleep(5)

class Productor(threading.Thread):
    def __init__(self, nombre, num_consultas, recurso):
        threading.Thread.__init__(self)
        self.name=nombre
        self.valor=None
        self.num_consultas=num_consultas
        self.recurso=recurso

    def run(self):
        print("Comienza ", self.name)
        for i in range(self.num_consultas):
            valor=random.random()
            self.recurso.producir(valor)
            time.sleep(1)


if __name__=="__main__":
    condicion=threading.Condition()
    monitor=RecursoMonitor (condicion)
    consumidor1 = Consumidor('Consumidor 1', 5, monitor)
    consumidor2 = Consumidor('Consumidor 2', 5, monitor)
    productor = Productor('Productor ', 5, monitor)

    consumidor1.start()
    time.sleep(3)
    consumidor2.start()
    time.sleep(2)

    productor.start()
```

Figura 13. Código del Ejemplo 7. Fuente: elaboración propia.

```
------------------------------------------
Ejecución:
Comienza Consumidor 1
El Consumidor 1 espera a que haya un valor de consulta
Comienza Consumidor 2
El Consumidor 2 espera a que haya un valor de consulta
Comienza  Productor
ElProductor  ha producido el valor 0.9956833761171444.

ElConsumidor 1 ha obtenido el valor 0.9956833761171444.

ElConsumidor 2 ha obtenido el valor 0.9956833761171444.

ElProductor  ha producido el valor 0.04463940701719671.

ElProductor  ha producido el valor 0.10411384300635951.

ElProductor  ha producido el valor 0.4186248999207327.

ElProductor  ha producido el valor 0.6641777654486484.

El Consumidor 2 espera a que haya un valor de consulta
ElConsumidor 2 ha obtenido el valor 0.6641777654486484.

El Consumidor 1 espera a que haya un valor de consulta
ElConsumidor 1 ha obtenido el valor 0.6641777654486484.

El Consumidor 2 espera a que haya un valor de consulta
ElConsumidor 2 ha obtenido el valor 0.6641777654486484.

El Consumidor 1 espera a que haya un valor de consulta
ElConsumidor 1 ha obtenido el valor 0.6641777654486484.

El Consumidor 2 espera a que haya un valor de consulta
ElConsumidor 2 ha obtenido el valor 0.6641777654486484.

El Consumidor 1 espera a que haya un valor de consulta
ElConsumidor 1 ha obtenido el valor 0.6641777654486484.

El Consumidor 2 espera a que haya un valor de consulta
ElConsumidor 2 ha obtenido el valor 0.6641777654486484.

El Consumidor 1 espera a que haya un valor de consulta
ElConsumidor 1 ha obtenido el valor 0.6641777654486484.
```

Figura 13. Código del Ejemplo 7. Fuente: elaboración propia.

Acceda al código en el notebook del aula virtual monitores.ipynb

Colas

Una característica importante para la comunicación son las colas de la biblioteca `Queue`, que incluyen colas «primero en entrar, primero en salir» (FIFO, por sus siglas en inglés), colas «último en entrar primero» (LIFO, por sus siglas en inglés), `LifoQueue`, y colas de prioridades, `PriorityQueue`. En todas estas colas **las operaciones de inserción y extracción están implementadas como operaciones atómicas para permitir su uso por parte de los hilos o procesos concurrentes.**

Para profundizar sobre las operaciones y características de este tipo de colas pueden acceder a la página oficial de Python.

7.5. Problemas en la programación concurrente

Los problemas más importantes que se producen en la programación concurrente son:

- **Problemas de seguridad:** la base de la seguridad en programación concurrente es garantizar que «nunca sucede nada malo a un objeto». Las amenazas a la seguridad se producen, fundamentalmente, al resolver los siguientes conflictos:

  - Conflictos de lectura/escritura.
  - Conflictos de escritura/escritura.

Estos conflictos se resuelven mediante los mecanismos que se pueden aplicar para garantizar la sincronización de acceso a los recursos por parte de los procesos o hilos.

- **Problemas de vivacidad:** la base para que no existan problemas de vivacidad es garantizar que «algo sucede eventualmente durante una actividad». La falta de vivacidad se puede producir si:

  - **Interbloqueo** (*deadlock*): ningún hilo puede continuar su ejecución porque espera un evento de otro proceso para poder avanzar.
  - **Bloqueo en vida** (*livelock*): un hilo cambia de estado reintentando una operación, pero no puede avanzar.
  - **Inanición** (*starvation*): se deniega siempre el recurso al hilo que lo necesita.

Las condiciones que se necesitan, en algunos programas concurrentes, y que producen interbloqueo son:

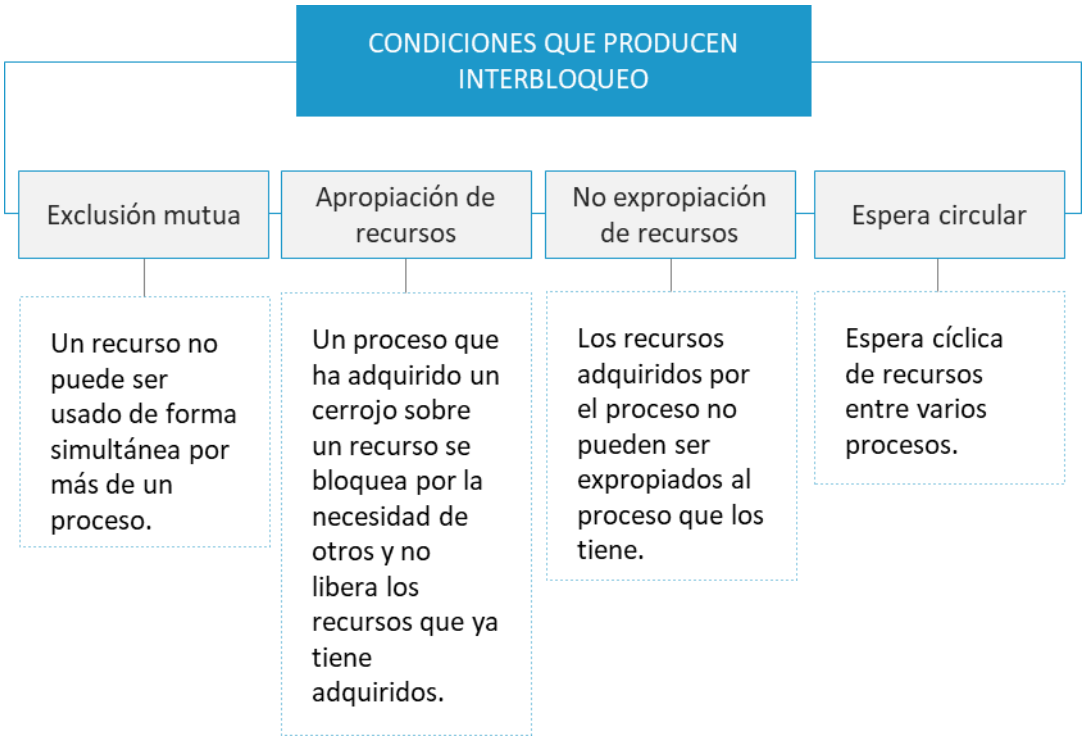

Figura 19. Condiciones que producen interbloqueo. Fuente: elaboración propia.

A continuación, se verá cuando se producen algunos de estos problemas y problemas clásicos de la programación concurrente en la que se producen.

Prevención de problemas

Para prevenir estos problemas se pueden realizar las siguientes estrategias:

- **Garantizar** que cada vez que un hilo solicite un recurso y pueda quedarse bloqueado, no retenga otro.

- **Requerir** que un hilo adquiera todos los recursos al principio, pero este no es muy viable.

- **Imponer** un orden en la solicitud de los recursos.

- **Liberar** los recursos adquiridos por un hilo si no consigue alguno de los que necesita para poder avanzar.

- **Asegurar** que al menos un hilo puede siempre adquirir los recursos que necesita, retrasando el acceso de alguno de los hilos.

Ejemplos de problemas de problemas de vivacidad

Los siguientes ejemplos muestran situaciones en las que los hilos no progresan en su ejecución.

> Ejemplo 8. Problema de interbloqueo
>
> Los hilos 1 y 2 quieren intercambiar los valores de dos celdas. Cada uno de ellos consulta primero el valor de una celda y luego el de la otra, pero son celdas contrarias. Se observa que el monitor celda está totalmente sincronizado y, por ello, se produce un interbloqueo, porque ninguno de los hilos libera el cerrojo sobre su celda original hasta que no ha hecho el intercambio. Si antes de que haya hecho el intercambio llega un hilo a la celda contraria y adquiere su cerrojo, el primer hilo no podrá continuar el intercambio.

```python
import threading
import time
class Celda:
    def __init__(self, valor,cerrojo):
        self.valor=valor
        self.cerrojo=cerrojo

    def obtenerValor(self):

        self.cerrojo.acquire()
        valor=self.valor
        print(threading.currentThread().getName()+" obtiene valor "+str(self.valor))
        self.cerrojo.release()

        return valor


    def ponerValor(self, valor):

        self.cerrojo.acquire()
        self.valor=valor
        self.cerrojo.release()
        print(threading.currentThread().getName()+" pone valor "+str(valor))

    def intercambiarValor(self, celda):
        self.cerrojo.acquire()
        val1=self.obtenerValor()
        val2=celda.obtenerValor()
        time.sleep(3)
        self.ponerValor(val2)
        celda.ponerValor(val1)
        self.cerrojo.release()

if __name__=="__main__":
    cerrojo1=threading.RLock()
    cerrojo2=threading.RLock()
    celda1=Celda(2,cerrojo1)
    celda2=Celda(3,cerrojo2)
    print("inicio programa")
    thread1 = threading.Thread(target=celda1.intercambiarValor, args=(celda2,))
    thread2 = threading.Thread(target=celda2.intercambiarValor, args=(celda1,))

    thread1.start()

    thread2.start()

    thread1.join()
    thread2.join()

    print("final programa"+str(celda1.obtenerValor())+" "+str(celda2.obtenerValor()))
```

Ejecución:

```
inicio programa
Thread-4 obtiene valor 2
Thread-4 obtiene valor 3
Thread-5 obtiene valor 3
Thread-4 pone valor 3
```

Figura 20. Código del ejemplo 8. Fuente: elaboración propia.

El primer hilo (`Thread-4`) llega hasta poner el valor de la segunda celda en la primera, pero el segundo hilo (`Thread-5`) entra y adquiere el cerrojo de la segunda celda y ya no avanzan ninguno de los dos, porque cada hilo tiene el cerrojo sobre una celda y no lo liberan. Como se ve, se usa `Rlock` porque el mismo hilo adquiere varias veces el mismo cerrojo en las distintas funciones.

Solución:

Esto se debe a un error de programación, por lo que hay que eliminar los bloqueos de alguno de los métodos.

Bloqueo en vida

En el siguiente código de ejemplo se presenta el problema de los monitores anidados, en el que se produce un problema de bloqueo en un hilo a la espera de un evento por parte de otros, que se quedan bloqueados en vida porque reintentan adquirir el cerrojo del monitor externo y no lo consiguen. En este caso se produce un anidamiento que da lugar a que si el hilo se bloquea en el nivel interno, deja con un bloqueo el monitor externo, cuando en realidad se podría seguir usando para otras funciones.

> Ejemplo 9. Problema de los monitores anidados
>
> En este problema se presentan dos monitores (clases) que en todos sus métodos se adquieren cerrojos.
>
> La clase externa es el monitor que controla un objeto de la clase interna. En esta última clase, en el método `esperarCondicion()`, se usa un `wait()` que suspende el hilo hasta que se cumpla la condición. El `wait()` libera el cerrojo del objeto de la clase interna, pero no el de la externa, por ello, el segundo hilo que podría liberar el hilo suspendido no puede hacerlo, porque el cerrojo de la externa sigue en poder del hilo suspendido y no se puede expropiar.

```python
import threading
class Interna:
    def __init__(self,condicion):
        self.condicion=condicion

    def esperarCondicion(self):
        with self.condicion:
            self.condicion.wait()

    def liberarCondicion(self):
        with self.condicion:
            self.condicion.notifyAll()

class Externa:
    def __init__(self,cerrojo,objetoInterno):
        self.objetoInterno=objetoInterno
        self.cerrojo=cerrojo

    def esperar(self):
        with self.cerrojo:
            self.objetoInterno.esperarCondicion()

    def liberar(self):
        with self.cerrojo:
            self.objetoInterno.liberarCondicion()

if __name__=="__main__":
    cerrojo=threading.Lock()
    condicion=threading.Condition()
    objInterno=Interna(condicion)
    objExterno=Externa(cerrojo,objInterno)
    print("inicio programa")
    thread1 = threading.Thread(target=objExterno.esperar)
    thread2 = threading.Thread(target=objExterno.liberar)

    thread1.start()

    thread2.start()

    thread1.join()
    thread2.join()

    print("final programa")
```

```
-----------------------------------------
Ejecución:
Inicio programa
Solución:
Eliminar los bloqueos de la clase externa
```

Figura 21. Código del Ejemplo 9. Fuente: elaboración propia.

Acceder al código en el notebook del aula virtual. problemasenPC.ipynb

7.6. Ejemplos clásicos de la programación concurrente

En esta oportunidad veremos algunos ejemplos clásicos de la programación concurrente.

Problema 1. La cena de los filósofos

Este problema, concebido por Dijkstra (1965), resuelve un problema de sincronización y sirve como ejemplo para modelar procesos que compiten por el acceso exclusivo a un número limitado de recursos, como una unidad de cinta u otro dispositivo de E/S.

Enunciado

Cinco filósofos dedican sus vidas a pensar y comer. Los filósofos comparten una mesa circular rodeada por cinco sillas, cada una de las cuáles pertenece a un filósofo. En la mesa hay cinco platos de arroz y cinco palillos (en otros enunciados se habla de espaguetis y tenedores, pero no es relevante).

Los filósofos necesitan dos palillos para comer y los comparten con su par de la izquierda y el de la derecha, de forma que no todos pueden comer al mismo tiempo. Por tanto, los palillos son recursos en exclusión mutua. Si se da el caso de que todos los filósofos se sienten a la vez y tomen el palillo de la izquierda a la vez, se produce una situación de interbloqueo.

En este problema se da:

- **Acceso en exclusión mutua a recursos.** Cada palillo solo puede ser usado por un filósofo a la vez.

- **Apropiación de recursos.** El filósofo coge el palillo de la izquierda y no lo libera, esperando por el de la derecha.
- **No expropiación de recursos.** No se expropia el palillo al filósofo que no está comiendo.
- **Espera circular.** Los palillos se adquieren de forma circular. Cada filósofo espera por el palillo del de su derecha.

Se pueden usar distintas estrategias para evitar este interbloqueo:

- **Liberación de recursos.** Si el filósofo no consigue su segundo palillo libera el que ya tiene y hace un reintento posterior.
- **Limitar el número de hilos que compiten por los recursos.** Se usa un método que impida el acceso a más de cuatro filósofos al comedor, de esta forma se asegura que al menos un filósofo podrá comer.

En el vídeo **La cena de los filósofos. Programación concurrente en Python** se puede ver un ejemplo clásico de programación concurrente, con distintas soluciones en las que se verán las herramientas de sincronización y los problemas que se pueden producir.

Problema 2. El problema de los lectores-escritores

El problema de los lectores y escritores fue enunciado por Courtois *et al*. (1971), que modela el acceso a una base de datos.

En este ejercicio se cuenta con una información compartida a la que pueden acceder varios hilos. Hay dos tipos hilos en el sistema, lectores y escritores. Cualquier número de lectores pueden leer del recurso compartido de forma simultánea, pero solo un

escritor puede estar escribiendo a la vez, y solo si no hay lectores. Cuando un escritor está redactando datos en el recurso, ningún otro proceso puede acceder a este.

En el problema tradicional, los lectores tienen prioridad sobre los escritores, menos en el momento inicial, ya que los lectores no pueden comenzar si el recurso accedido está vacío.

7.7. Referencias bibliográficas

Courtois, P. J., Heymans, F. y Parnas, D. L. (1971). Concurrent control with “readers” and “writers”. *Communications of the ACM, 14*(10), pp. 667-668. https://doi.org/10.1145/362759.362813

Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM, 8*(9), p. 569. https://doi.org/10.1145/365559.365617

7.8. Cuaderno de ejercicios

Ejercicio 1. Productor-consumidor con cola de almacenamiento

En este ejemplo se usa una cola de almacenamiento de la biblioteca `queue`. Como puede verse no se usan cerrojos ni condiciones, porque las colas tienen implementados estos mecanismos de forma implícita en los métodos `get()` y `put()`.

Se han invocado, con el parámetro `block`, a *True* para que, en el caso de que no se cumplan las condiciones, se bloquee el hilo hasta que se cumplan.

Se ha implementado, de forma premeditada, que el productor solo produzca cinco valores, cuando los consumidores necesitan un total de seis, porque cada uno lee tres veces. Para gestionar esto, en `get()` se ha inicializado el parámetro `timeout` a veinte segundos para que, en el caso de que pasado ese tiempo, no haya valores en la cola, se eleva la excepción `queue.Empty` y se captura, emitiendo el mensaje correspondiente y retornando el valor `None`. De forma similar se puede tratar el `timeout` de `put()`, con la excepción `queue.Full`.

```
import random
import threading
import time
import queue



class Buffer:
    def __init__(self,buffer):
        self.buffer=buffer


    def consumir(self):
        try:
            valor=buffer.get(block=True,timeout=20) #operación atómica,
            #el tim-out limita la espera
            #pasado el tiempo se produce una excepcion
            texto="El {hilo} ha obtenido el valor {valor}.\n"
            print (texto.format(hilo=threading.currentThread().getName(), valor=valor))
        except queue.Empty:
            texto="El {hilo} se ha quedado sin valores.\n"
            print (texto.format(hilo=threading.currentThread().getName()))
            return None
        return valor

    def producir(self, valor):
        try:
            buffer.put(valor,block=True,timeout=3)# operación atómica
            texto="El {hilo} ha producido el valor {valor}.\n"
            print (texto.format(hilo=threading.currentThread().getName(), valor=valor))
        except queue.Full:
            texto="No ha espacio para guardar más valores, el {hilo} queda a la espera.\n"
            print (texto.format(hilo=threading.currentThread().getName()))
            buffer.put(valor,block=True)
            texto="El {hilo} ha producido el valor {valor}.\n"
            print (texto.format(hilo=threading.currentThread().getName(), valor=valor))

class ConsumidorBuffer(threading.Thread):
    def __init__(self, nombre, num_consultas, recurso):
        threading.Thread.__init__(self)
        self.name=nombre
        self.valor=None
        self.num_consultas=num_consultas
        self.recurso=recurso

    def run(self):
        print("Comienza "+self.name)
        for i in range(self.num_consultas):
            self.valor=self.recurso.consumir()
            time.sleep(20)

class ProductorBuffer(threading.Thread):
    def __init__(self, nombre, num_consultas, recurso):
        threading.Thread.__init__(self)
        self.name=nombre
        self.valor=None
        self.num_consultas=num_consultas
        self.recurso=recurso

    def run(self):
        print("Comienza ", self.name)
        for i in range(self.num_consultas):
            valor=random.random()
            self.recurso.producir(valor)
            time.sleep(1)
```

Figura 22. Código del Ejercicio1. Fuente: elaboración propia.

```
if __name__=="__main__":

    buffer=queue.Queue(maxsize=2)

    monitor=Buffer (buffer)
    consumidor1 = ConsumidorBuffer('Consumidor 1', 3, monitor)
    consumidor2 = ConsumidorBuffer('Consumidor 2', 3, monitor)
    productor = ProductorBuffer('Productor ', 5, monitor)

    consumidor1.start()
    time.sleep(3)
    consumidor2.start()
    time.sleep(2)

    productor.start()
```

```
-----------------------------------------
Ejecución:
Comienza Consumidor 1
Comienza Consumidor 2
Comienza  Productor
El Productor  ha producido el valor 0.7325920954295542.

El Consumidor 1 ha obtenido el valor 0.7325920954295542.

El Productor  ha producido el valor 0.6766503505662871.
El Consumidor 2 ha obtenido el valor 0.6766503505662871.


El Productor  ha producido el valor 0.11674873424897558.

El Productor  ha producido el valor 0.8465429461184446.

El Consumidor 1 ha obtenido el valor 0.11674873424897558.
El Productor  ha producido el valor 0.41603373042022396.


El Consumidor 2 ha obtenido el valor 0.8465429461184446.

El Consumidor 1 ha obtenido el valor 0.41603373042022396.

El Consumidor 2 se ha quedado sin valores.
```

Figura 22. Código del Ejercicio1. Fuente: elaboración propia.

Consulte el notebook en el aula virtual. productor-consumidorconCola.ipynb

Comprueben que pasaría si en lugar de usar el `put()` y `get()` con bloqueo, se usa el `put_nowait()` y el `get_nowait()`.

Ejercicio 2. Problema de los filósofos con tendencia a interbloqueo

Los palillos son representados por objetos con dos métodos para controlar el acceso en exclusión mutua. En uno se adquiere el cerrojo y en otro se libera sobre el palillo

Los filósofos, en este caso, son hilos. Se crean cinco objetos palillo, con su cerrojo, y cinco filósofos a los que se les pasan los palillos que tienen que conseguir.

```python
import threading
import time

class Palillo:
    def __init__(self,numero,cerrojo):
        self.numero=numero
        self.cerrojo=cerrojo

    def obtenerPalillo(self):
        self.cerrojo.acquire()
        print("El palillo "+str(self.numero)+" lo tiene el filosofo "+threading.currentThread().getName()+"\n")

    def soltarPalillo(self):
        #print("El filosofo "+threading.currentThread().getName()+" suelta el palillo "+str(self.numero)+"\n")
        self.cerrojo.release()

class Filosofo(threading.Thread):

    def __init__(self,nombre, id, izdo,dcho):
        threading.Thread.__init__(self)
        self.name=nombre
        self.id=id
        self.izdo=izdo
        self.dcho=dcho


    def run(self):

        for i in range(2):
            print("El filosofo "+threading.currentThread().getName()+" ha entrado a comer\n ")
            self.izdo.obtenerPalillo()
            time.sleep(0.001)
            self.dcho.obtenerPalillo()
            print("El filosofo "+threading.currentThread().getName()+" está comiendo\n ")
            self.izdo.soltarPalillo()
            self.dcho.soltarPalillo()
            print("El filosofo "+threading.currentThread().getName()+" está pensando\n ")

if __name__=="__main__":
    palillos=[]

    for i  in range(5):
        cerrojo=threading.Lock()
        palillo=Palillo(i,cerrojo)
        palillos.append(palillo)

    filosofos=[]
    for i in range(5):
        nombre="Filosofo "+str(i)
        filosofo=Filosofo(nombre,i,palillos[i],palillos[(i+1)%5])
        filosofos.append(filosofo)

    for i in range(5):
        filosofos[i].start()

    for i in range(5):
        filosofos[i].join()
```

```
-----------------------------------------
Ejecución:
El filosofo Filosofo 0 ha entrado a comer

El palillo 0 lo tiene el filosofo Filosofo 0

El filosofo Filosofo 1 ha entrado a comer

El palillo 1 lo tiene el filosofo Filosofo 1

El filosofo Filosofo 2 ha entrado a comer

El palillo 2 lo tiene el filosofo Filosofo 2
El filosofo Filosofo 3 ha entrado a comer

El palillo 3 lo tiene el filosofo Filosofo 3


El filosofo Filosofo 4 ha entrado a comer

El palillo 4 lo tiene el filosofo Filosofo 4
```

Figura 23. Código del Ejercicio 2. Fuente: elaboración propia.

En este problema se da el interbloqueo si todos los filósofos llegan simultáneamente y cogen su primer palillo a la vez, con lo que ya no pueden coger el segundo. Para

simular esta situación se ha insertado un retardo después de que los filósofos cogen el palillo izquierdo.

Ejercicio 3. Problema de los filósofos con liberación de recursos

Si el filósofo, después de coger el primer palillo, no puede coger el segundo, libera el primero de ellos. Esta solución, sin embargo, puede que produzca un problema de inanición en alguno de los filósofos si nunca puede llegar a coger los dos.

```
import threading
import time

class PalilloConEstado:
    def __init__(self,numero):
        self.numero=numero
        self.cerrojo=threading.Lock()
        self.ocupado=False # se necesita para poder comprobar previamente el estado del segundo palillo

    def obtenerPalillo(self):
        self.cerrojo.acquire()
        self.ocupado=True
        print("El palillo "+str(self.numero)+" lo tiene el filosofo "+threading.currentThread().getName()+"\n"

    def soltarPalillo(self):
        print("El filosofo "+threading.currentThread().getName()+" suelta el palillo "+str(self.numero)+"\n")
        self.ocupado=False
        self.cerrojo.release()

class FilosofoConLiberacion(threading.Thread):

    def __init__(self,nombre, id, izdo,dcho):
        threading.Thread.__init__(self)
        self.name=nombre
        self.id=id
        self.izdo=izdo
        self.dcho=dcho


    def run(self):

        for i in range(2):
            print("El filosofo "+threading.currentThread().getName()+" ha entrado a comer\n ")
            comer=False
            while not comer:
                self.izdo.obtenerPalillo()
                if self.dcho.ocupado:
                    self.izdo.soltarPalillo()
                    time.sleep(2)
                else:
                    self.dcho.obtenerPalillo()
                    comer=True

            print("El filosofo "+threading.currentThread().getName()+" está comiendo\n ")
            time.sleep(2)
            self.izdo.soltarPalillo()
            self.dcho.soltarPalillo()
            print("El filosofo "+threading.currentThread().getName()+" está pensando\n ")
            time.sleep(2)
if __name__=="__main__":
    palillosConEstado=[]

    for i  in range(5):
        palillo=PalilloConEstado(i)
        palillosConEstado.append(palillo)

    filosofosCL=[]
    for i in range(5):
        nombre="Filosofo "+str(i)
        filosofo=FilosofoConLiberacion(nombre,i,palillosConEstado[i],palillosConEstado[(i+1)%5])
        filosofosCL.append(filosofo)

    for i in range(5):
        filosofosCL[i].start()

    for i in range(5):
        filosofosCL[i].join()
```

Figura 24. Código del Ejercicio 3. Fuente: elaboración propia.

Uno de los problemas que pueden ocurrir es que, si un filósofo reintenta continuamente coger los palillos, puede que sea adelantado por todos y nunca pueda comer, pero además se apropia del procesador sin permitir que se liberen recursos y se produzca el bloqueo en vida, porque nunca sale del bucle ni los demás avanzan. Por eso se usan los tiempos de retardo en el reintento, sin ellos se produciría el problema recién mencionado.

Ejercicio 4. Problema de los filósofos con limitación de acceso

La última solución consiste en limitar el acceso a la sala a más de cuatro filósofos. Para esto se va a usar un semáforo inicializado a cuatro, que controla el acceso al comedor como si hubiese un camarero.

Uno de los problemas que pueden ocurrir es que, si un filósofo reintenta continuamente coger los palillos, puede que sea adelantado por todos y nunca pueda comer, pero además se apropia del procesador sin permitir que se liberen recursos y se produzca el bloqueo en vida, porque nunca sale del bucle ni los demás avanzan. Por eso se usan los tiempos de retardo en el reintento.

Prueben quitarlo y observen lo que ocurre.

```
import threading
import time

camarero=threading.Semaphore(4)

class PalilloCamarero:
    def __init__(self,numero):
        self.numero=numero
        self.cerrojo=threading.Lock()
        self.ocupado=False # se necesita para poder comprobar previamente el estado del segundo palillo

    def obtenerPalillo(self):
        self.cerrojo.acquire()
        self.ocupado=True
        print("El palillo "+str(self.numero)+" lo tiene el filosofo "+threading.currentThread().getName()+"\n")

    def soltarPalillo(self):
        print("El filosofo "+threading.currentThread().getName()+" suelta el palillo "+str(self.numero)+"\n")
        self.ocupado=False
        self.cerrojo.release()

class FilosofoCamarero(threading.Thread):

    def __init__(self,nombre, id, izdo,dcho):
        threading.Thread.__init__(self)
        self.name=nombre
        self.id=id
        self.izdo=izdo
        self.dcho=dcho


    def run(self):

        for i in range(2):
            camarero.acquire()
            print("El filosofo "+threading.currentThread().getName()+" ha entrado a comer\n ")
            self.izdo.obtenerPalillo()
            self.dcho.obtenerPalillo()

            print("El filosofo "+threading.currentThread().getName()+" está comiendo\n ")
            time.sleep(2)
            self.izdo.soltarPalillo()
            self.dcho.soltarPalillo()
            camarero.release()
            print("El filosofo "+threading.currentThread().getName()+" está pensando\n ")
            time.sleep(2)
if __name__=="__main__":
    #limita el acceso a 4 filósofos
    palillosCamarero=[]

    for i  in range(5):
        palillo=PalilloCamarero(i)
        palillosCamarero.append(palillo)

    filosofosCamarero=[]
    for i in range(5):
        nombre="Filosofo "+str(i)
        filosofo=FilosofoCamarero(nombre,i,palillosConEstado[i],palillosConEstado[(i+1)%5])
        filosofosCamarero.append(filosofo)

    for i in range(5):
        filosofosCamarero[i].start()

    for i in range(5):
        filosofosCamarero[i].join()
```

Figura 25. Código del Ejercicio 4. Fuente: elaboración propia.

El código de estas está disponible en el aula en el notebook filosofos.ipynb

Ejercicio 5. Problema de los lectores-escritores

Se implementará una clase `LectoresEscritores` con dos métodos, leer y escribir. Además, contará con dos semáforos:

- Uno para la modificación del contador de lectores en exclusión mutua.
- Otro para los escritores que bloquean el acceso a los escritores, desde la función lector, o a otros escritores, desde la función escritor.

Los lectores usan el semáforo para modificar la variable contador y adquieren el semáforo de los escritores cuando hay un lector, y no lo liberan hasta que no quede ninguno.

Se da prioridad a los lectores, excepto si se da la situación de que la información esté vacía, para lo que deberá esperar al escritor.

Las instrucciones propiamente de escritura se realizan en exclusión mutua, las de lectura no, porque pueda haber varios lectores a la vez.

```
import threading
import time
import random

class LectoreEscritores():
    def __init__(self):
        self.semaforoExclusionMutua = threading.Semaphore()
        #se crea un semaforo para garantizar que el contador de lectores se modifica en exclusión mutua
        self.semaforoEscritores = threading.Semaphore()
        #semaforo que bloquea el acceso a más de un escritor
        self.numLectores = 0   #contador de lectores activos
        self.dato=None
        self.condicionVacio=threading.Condition()
        #la condición se usa para que asegurar que un lector no lee uun valor nulo


    def lector(self,dato):
        while True:
            with self.condicionVacio:
                if self.dato==None:
                    self.condicionVacio.wait()#el hilo no avanza si el dato está vacío
            self.semaforoExclusionMutua.acquire()      #espera a leer en el semaforo
            self.numLectores+=1       #incremento del contador de lectores
            if self.numLectores == 1: #se toma el semaforo de los escritores para que no entre ninguno
                self.semaforoEscritores.acquire()
            self.semaforoExclusionMutua.release()     #se LIBERA el semaforo para dar paso a más lectores

            dato=self.dato
            print(f"El {threading.currentThread().getName()} leyendo\n")
            print(f"Hay {self.numLectores} leyendo {dato}\n")

            self.semaforoExclusionMutua.acquire()   #se vuelve a adquirir el  semaforo para decrementar los lectores

            self.numLectores-=1   #se decrementa el contador de lectores
            print(f"Hay {self.numLectores} leyendo\n")
            if self.numLectores == 0: #si no quedan lectores se libera  el semaforo de los escritores
                self.semaforoEscritores.release()

            self.semaforoExclusionMutua.release()      #se libera el semaforo de los lectores
            time.sleep(3)

    def escritor(self):
        while True:
            self.semaforoEscritores.acquire()
            #adquiere el semaforo y ejecuta las acciones en exclusión mutua

            dato=random.random()
            print(f"El {threading.currentThread().getName()} escribiendo\n")
            print("Escribiendo datos "+str(dato))
            with self.condicionVacio:
                if self.dato==None:
                    self.condicionVacio.notifyAll()
            self.dato=dato

            self.semaforoEscritores.release()      #se libera el semáforo
            time.sleep(3)
```

Figura 26. Código del ejercicio 5. Fuente: elaboración propia.

```
if __name__=="__main__":
        lecEsc= LectoreEscritores()
        dato=0

        t1 = threading.Thread(target = lecEsc.lector, args=((dato),), name="Lector 1")
        t1.start()
        t2 = threading.Thread(target = lecEsc.escritor,  name="Escritor 1")
        t2.start()
        t3 = threading.Thread(target = lecEsc.lector, args=((dato),), name="Lector 2")
        t3.start()
        t4 = threading.Thread(target = lecEsc.lector, args=((dato),), name="Lector 3")
        t4.start()
        t5 = threading.Thread(target = lecEsc.escritor, name="Escritor 2")
        t5.start()
        t6 = threading.Thread(target = lecEsc.lector, args=((dato),), name="Lector 4")
        t6.start()
```

```
-----------------------------------------
Ejecución:
El Escritor 1 escribiendo

Escribiendo datos 0.3009150095103653
El Lector 2 leyendo

Hay 1 leyendo 0.3009150095103653

Hay 0 leyendo

El Lector 3 leyendo

Hay 1 leyendo 0.3009150095103653

El Lector 1 leyendo

Hay 2 leyendo 0.3009150095103653

Hay 1 leyendo
```

Figura 26. Código del ejercicio 5. Fuente: elaboración propia.

Ejercicio 6. Problema de los lectores-escritores con prioridad para los escritores

Se implementa una solución en la que los escritores tienen prioridad. Se necesitan más semáforos, dos de ellos son equivalentes a los que existían en el caso anterior, pero enfocados en dar prioridad a los escritores.

- Ahora los lectores bloquearán a los lectores y adquirirán el semáforo nuevo para escritores, pero además los escritores necesitan un semáforo para garantizar la exclusión mutua en el incremento del contador de escritores y el bloqueo de los lectores. Por ello, habrá un `semaforoLectores` que se necesita para que puedan ser bloqueados cuando aparece un escritor.

- Además, se añade un semáforo de prioridades para asegurar que solo hay un lector que accede a adquirir inicialmente su semáforo, con esto se garantiza que el escritor tiene la prioridad.

En el caso de que no haya escritores, los distintos lectores pueden tener acceso simultáneo a la zona de lectura después de pasar por la región de incremento del contador, en exclusión mutua.

```
import threading
import time
import random

class LectoreEscritoresPrioridadEscritor():
    def __init__(self):
        self.semaforoExclusionMutuaLectores = threading.Semaphore()
        #se crea un semaforo para garantizar que el contador de lectores se modifica en exclusión mutua
        self.semaforoExclusionMutuaEscritores = threading.Semaphore()
        #se crea un semaforo para garantizar que el contador de escritores se modifica en exclusión mutua
        self.semaforoPrioridadEscritores=threading.Semaphore()
        #garantiza que a la zona para control inical solo accede un lector
        self.semaforoEscritores = threading.Semaphore()
        #semaforo que bloquea el acceso a más de un escritor
        self.semaforoLectores = threading.Semaphore()
        #semaforo que bloquea el acceso a lectores para dar prioridad a los escritores
        self.numLectores = 0   #contador de lectores activos
        self.numEscritores=0
        self.dato=None
        self.condicionVacio=threading.Condition()

    def lectorPrioridad(self,dato):
        while True:
            with self.condicionVacio:
                if self.dato==None:
                    self.condicionVacio.wait()#el hilo no avanza si el dato está vacío
            self.semaforoPrioridadEscritores.acquire()
            self.semaforoLectores.acquire()
            self.semaforoExclusionMutuaLectores.acquire()
            #espera a leer en el semaforo
            self.numLectores+=1       #incremento del contador de lectores
            if self.numLectores == 1: #se toma el semaforo de los escritores para que no entre ninguno
                self.semaforoEscritores.acquire()
            self.semaforoExclusionMutuaLectores.release()
            self.semaforoLectores.release()
            self.semaforoPrioridadEscritores.release()
            #se LIBERA el semaforo para dar paso a más lectores

            dato=self.dato
            print(f"El {threading.currentThread().getName()} leyendo\n")
            print(f"Hay {self.numLectores} leyendo {dato}\n")
            time.sleep(2)

            self.semaforoExclusionMutuaLectores.acquire()
            #se vuelve a adquirir el  semaforo para decrementar los lectores
            self.numLectores-=1
            #se decrementa el contador de lectores
            print(f"Hay {self.numLectores} leyendo\n")
            if self.numLectores == 0: #si no quedan lectores se libera  el semaforo de los escritores
                self.semaforoEscritores.release()

            self.semaforoExclusionMutuaLectores.release()      #se libera el semaforo de los lectores
```

Figura 27. Código del Ejercicio 6. Fuente: elaboración propia.

```
    def escritorPrioridad(self):
        while True:
            self.semaforoExclusionMutuaEscritores.acquire()
            self.numEscritores+=1
            if self.numEscritores==1:
                self.semaforoLectores.acquire()
            self.semaforoExclusionMutuaEscritores.release()
            self.semaforoEscritores.acquire()
            #adquiere el semaforo y ejecuta las acciones en exclusión mutua

            dato=random.random()
            print(f"El {threading.currentThread().getName()} escribiendo\n")
            print("Escribiendo datos "+str(dato))
            with self.condicionVacio:
                if self.dato==None:
                    self.condicionVacio.notifyAll()
            self.dato=dato
            time.sleep(3)

            self.semaforoEscritores.release()      #se libera el semáforo
            self.semaforoExclusionMutuaEscritores.acquire()
            self.numEscritores-=1
            if self.numEscritores==0:
                self.semaforoLectores.release()
            self.semaforoExclusionMutuaEscritores.release()
            time.sleep(4)

if __name__=="__main__":
        lecEscPrioridad= LectoreEscritoresPrioridadEscritor()
        dato=0

        t1 = threading.Thread(target = lecEscPrioridad.lectorPrioridad, args=((dato),), name="Lector 1")
        t1.start()
        t2 = threading.Thread(target = lecEscPrioridad.escritorPrioridad,   name="Escritor 1")
        t2.start()
        t3 = threading.Thread(target = lecEscPrioridad.lectorPrioridad,args=((dato),), name="Lector 2")
        t3.start()
        t4 = threading.Thread(target = lecEscPrioridad.lectorPrioridad,args=((dato),), name="Lector 3")
        t4.start()
        t6 = threading.Thread(target = lecEscPrioridad.escritorPrioridad, name="Escritor 2")
        t6.start()
        t5 = threading.Thread(target = lecEscPrioridad.lectorPrioridad, args=((dato),), name="Lector 4")
        t5.start()
```

Ejecución:

```
El Escritor 1 escribiendo

Escribiendo datos 0.5451703674910571
El Escritor 1 escribiendo

Escribiendo datos 0.19580198901897905
El Escritor 2 escribiendo

Escribiendo datos 0.3369287608258196
El Lector 4 leyendo

Hay 1 leyendo 0.3369287608258196

Hay 0 leyendo

El Lector 2 leyendo

Hay 1 leyendo 0.3369287608258196
```

Figura 27. Código del Ejercicio 6. Fuente: elaboración propia.

Puede consultar el código en el notebook del aula virtual lectoresescritores.ipynb

A fondo

Colas con el módulo queue

Recursos Python (7 de septiembre de 2013). *Colas con el módulo queue*. Colas con el módulo queue - Recursos Python

Profundiza en el uso de las colas con las funciones de modificación de las mismas como operaciones atómicas.

Concurrency in Python – Quick guide

Tutorials Point (s. f.). *Concurrency in Python- Quick Guide*. Concurrency in Python - Quick Guide (tutorialspoint.com)

Este tutorial cubre de forma concisa todos los aspectos sobre programación concurrente en Python

Ejecución concurrente

Python (s. f.). *Ejecución concurrente*. Ejecución concurrente — documentación de Python - 3.9.6

En este documento se detallan todos los elementos con los que cuenta el módulo `threading` de Python.

Test

1. En la programación concurrente:

A. Los hilos se ejecutan de forma simultánea de manera real.

B. Los hilos se ejecutan, aparentemente, de forma simultánea.

C. Los hilos se ejecutan uno detrás de otro.

D. Ninguna de las anteriores es verdadera

2. Un entorno de multiprogramación:

A. Cuenta con múltiples procesadores con memoria compartida.

B. Cuenta con múltiples procesadores sin memoria compartida.

C. Cuenta con un único procesador.

D. Ninguna de las anteriores es verdadera.

3. Un hilo es:

A. Una entidad de ejecución independiente que comparte memoria con otros hilos.

B. Una entidad de ejecución independiente que no comparte memoria con otros hilos.

C. Una entidad que no se ejecuta de forma independiente y que comparte memoria con otros hilos.

D. Ninguna de las anteriores es verdadera.

4. El GIL de CPhyton y PyPy:

A. Es un problema en entornos de monoprocesador.

B. Es un problema en un entorno multiprocesador.

C. No afecta al uso de los procesadores.

D. Ninguna de las anteriores es verdadera.

5. Los hilos en Python:

A. Solo se crean como objetos de la clase `Thread` del módulo `threading`.

B. Solo se crean como objetos de una clase derivada de `Thread` del módulo `threading`.

C. Se crean como objetos de la clase `Thread` o de clases derivadas de esta.

D. Ninguna de las anteriores es verdadera.

6. Los estados principales por los que puede pasar un hilo son:

A. Creado, ejecutable y terminado.

B. Ejecutable, parado y terminado.

C. Creado, ejecutable, parado y terminado.

D. Ninguna de las anteriores es verdadera,

7. Los semáforos son:

A. Cerrojos que solo pueden ser adquiridos por un hilo cada vez.

B. Cerrojos que pueden ser adquiridos por más de un hilo a la vez hasta un número limitado de hilos.

C. Cerrojos que pueden ser adquiridos por un número ilimitado de hilos a la vez.

D. Ninguna de las anteriores es verdadera,

8. Los cerrojos de tipo `Lock`:

A. Puede ser adquirido varias veces por un mismo hilo, si este lo ha liberado antes de que se vuelva a adquirir.

B. Puede ser adquirido varias veces por un mismo hilo.

C. Puede ser adquirido varias veces por un mismo hilo, si este lo ha liberado antes de que se vuelva a adquirir.

D. Ninguna de las anteriores es verdadera.

9. Para implementar un monitor en Python:

A. Se debe implementar un método con un `Lock`.

B. Se debe implementar una clase con al menos un método con un semáforo.

C. Se debe implementar una clase en la que todos los métodos se ejecuten en exclusión mutua mediante el bloqueo de los hilos mediante cerrojo `Condition`.

D. Ninguna de las anteriores es verdadera.

10. La inanición es:

A. Un problema de seguridad.

B. Un problema de vivacidad.

C. No es un problema de la programación concurrente.

D. Ninguna de las anteriores es verdadera.