Enrique Ulises Báez Gómez Tagle
0241823@up.edu.mx

Lectura 1 Concurrencia una introducción suave

1 Preguntas

1.1 ¿Qué problemas resuelven las GoRoutines en comparación a las Coroutines específicas del framework asyncio?

Característica	**GoRoutines (Go)**	**Coroutines (asyncio en Python)**
Modelo de concurrencia	Modelo CSP (Comunicación a través de Canales)	Modelo basado en un ciclo de eventos
Creación de hilos	Livianas, basadas en el propio runtime de Go	No crea nuevos hilos; se ejecutan en el mismo hilo de manera cooperativa
Tareas que manejan	Tareas de I/O y CPU. Diseñadas para cualquier tipo de tarea concurrente	Principalmente I/O asincrónico; menos eficientes para tareas pesadas de CPU
Bloqueo	No se bloquean entre sí; el uso de canales facilita la comunicación segura	No bloquean el hilo principal en operaciones de I/O, pero las tareas pesadas de CPU pueden bloquear el hilo
Escalabilidad	Permiten crear millones de GoRoutines sin sobrecargar el sistema	Menos escalables en comparación; manejan miles de coroutines
Comunicación entre tareas	Usan canales para la comunicación entre GoRoutines	Se basan en await/async para gestionar las suspensiones y reanudaciones
Paralelismo	Soportan paralelismo verdadero en sistemas con múltiples núcleos	No soportan paralelismo verdadero, ya que el GIL limita la ejecución a un solo hilo a la vez

- **Concurrencia ligera:** Las GoRoutines nos permiten crear varias tareas concurrentes con un uso muy bajo de recursos, mientras que las Coroutines en asyncio están más limitadas en escalabilidad.
- **Manejo de I/O y CPU:** Las GoRoutines sirven tanto para tareas de I/O como de CPU, mientras que las Coroutines están principalmente diseñadas para manejar operaciones de I/O asincrónicas.
- **Comunicación segura entre tareas:** Las GoRoutines usan canales para facilitar la comunicación y sincronización entre tareas concurrentes, mientras que las Coroutines utilizan ciclo de eventos y await/async.

- **Paralelismo real:** Las GoRoutines permiten un paralelismo en múltiples núcleos, mientras que las Coroutines están limitadas por el GIL de Python, entonces tienen restricciones para manejar múltiples hilos de ejecución simultáneos.

1.2 ¿Qué usos tienen los event-loops al momento de manejar loa paralelismos y concurrencia y cuál es su diferencia con usar threads?

1.2.1 Descripción:

- **Event-loops:** Los usamos para manejar concurrencia en tareas de I/O, no bloquean el sistema y son eficientes en términos de recursos. No permiten paralelismo real, solo usan un hilo.
- **Threads:** Nos otorgan paralelismo real para tareas intensivas de CPU, pero requieren mayor manejo de sincronización y llegan a consumir más recursos debido a la creación y gestión de múltiples hilos.

1.2.2 Event Loops:

- **Manejo eficiente de I/O:**
  - Son fundamentales para la programación concurrente, se usan en tareas de I/O intensivo (como operaciones de red o lectura de archivos).
  - Permiten que el sistema siga ejecutando otras tareas mientras espera la finalización de las operaciones, sin bloquear el hilo principal.
- **Concurrencia sin bloqueo:**
  - En Node.js o asyncio en Python, nos permiten ejecutar múltiples tareas asíncronas al mismo tiempo, sin crear hilos adicionales.
  - Es ideal en casos de alta concurrencia que manejan muchas conexiones, como servidores web o sistemas de notificaciones en tiempo real.
- **Escalabilidad:**
  - Como no crean nuevos hilos, permiten que un solo hilo administre muchas operaciones asíncronas de forma eficiente, así mejorando la escalabilidad de aplicaciones que manejan cientos o miles de conexiones simultáneas.

1.2.3 Diferencia contra los threads:

- **Event-loops (concurrencia cooperativa):**
  - Las tareas asíncronas deben ceder el control al sistema para que otras tareas puedan ejecutarse.
  - Eficiente para tareas de I/O, pero no es ideal para operaciones intensivas de CPU.
- **Threads (concurrencia preemptiva):**
  - Los threads permiten ejecutar múltiples tareas en paralelo, utilizando varios núcleos de CPU.
  - Ideal para tareas intensivas de CPU, ya que cada hilo puede ejecutarse en paralelo en un núcleo diferente.
  - Múltiples hilos introducen desafíos como condiciones de carrera y sincronización.

1.3 ¿Qué tipo de estructuras conoces en otros lenguajes (que no sea python) para implementar reactive programming o event-driven services?

1.3.1 Node.js y EventEmitter (JavaScript):

- **Node.js** está basado en un event-loop no bloqueante.
- **EventEmitter** es una API que sirve para crear y gestionar eventos.
- **Aplicaciones comunes:** Notificaciones en tiempo real, microservicios event-driven, y sistemas de monitoreo/logging.

1.3.2 Kafka (Apache Kafka):
- **Kafka** plataforma de mensajería distribuida que permite manejar big data en streaming.
- Usa productores y consumidores para crear pipelines de datos donde los eventos se procesan de manera reactiva.
- **Aplicaciones comunes:** Streaming de datos en tiempo real, pipelines de procesamiento de datos, integración de microservicios, y monitoreo de infraestructuras.

1.3.3 AWS Lambda y EventBridge:
- **AWS Lambda** opción serverless que permite ejecutar código en respuesta a eventos.
- **AWS EventBridge** se usa para la integración de eventos desde múltiples servicios y fuentes para activar funciones Lambda.
- **Aplicaciones comunes:** Automatización de infraestructura, procesamiento reactivo de datos, integración serverless en pipelines de **AI/ML**, y tareas de **DevOps** basadas en eventos.

1.3.4 RxJS (Reactive Extensions for JavaScript):
- **RxJS** biblioteca para flujos de datos asíncronos y eventos mediante programación funcional.
- Nos permite realizar transformaciones en streams de eventos en tiempo real.
- **Aplicaciones comunes:** Flujos de datos en tiempo real, interfaces reactivas, y sistemas de monitoreo y notificaciones en tiempo real.

1.3.5 Redis Streams:
- **Redis Streams** permite manejar eventos en tiempo real dentro de Redis, ideal para arquitecturas event-driven.
- Ofrece capacidades para almacenar y procesar big data con eventos distribuidos de forma reactiva.
- **Aplicaciones comunes:** Sistemas de colas de eventos en pipelines de datos, procesamiento en tiempo real de logs, sensores o mensajes entre microservicios.

1.4 ¿Cuál es la diferencia entre secuencial, paralelo y dónde entra la concurrencia en cada caso?

1.4.1 1. Programación Secuencial:
- **Definición:** Se ejecutan las tareas una tras otra, sin solaparse ni interrumpirse. Una tarea debe completarse antes de que comience la siguiente.
- **Características**
  - Se ejecuta en un solo hilo.
  - No puede aprovechar múltiples núcleos de CPU.
  - Es adecuada para tareas simples o en sistemas con pocas demandas de procesamiento.
- **Básicamente** Una tarea a la vez, en orden.
- **Ejemplo** Leer un archivo, luego procesarlo, y finalmente guardarlo en disco, todo en orden secuencial.

1.4.2 2. Programación Paralela:
- **Definición:** varias tareas se ejecutan simultáneamente en múltiples núcleos de CPU. Cada tarea tiene su propio hilo o proceso, así que las tareas se completan más rápido.
- **Características**
  - Utiliza múltiples hilos o procesos que se ejecutan al mismo tiempo en distintos núcleos de CPU.
  - Ideal para tareas intensivas en CPU, como cálculos matemáticos complejos, procesamiento de imágenes o entrenamiento de modelos de Machine Learning.
  - Implica el uso de varias unidades de hardware de procesamiento (núcleos o incluso varias máquinas en un clúster).
- **Básicamente** Varias tareas al mismo tiempo, en múltiples núcleos.

- **Ejemplo** Dividir una tarea de procesamiento de imágenes en varias partes y ejecutarlas al mismo tiempo en diferentes núcleos.

1.4.3 3. Programación Concurrente:

- **Definición:** Permite que múltiples tareas parezcan ejecutarse al mismo tiempo, pero en realidad el sistema alterna entre ellas de forma rápida. No implica necesariamente paralelismo.
- **Características**
  - Se puede ejecutar en un solo núcleo de CPU o varios.
  - Maneja múltiples tareas intercalando su ejecución, aprovechando los tiempos de espera (por ejemplo, en operaciones de I/O).
  - Es más eficiente en tareas de I/O que pueden esperar a recursos externos (lectura de archivos, solicitudes de red), permitiendo que otras tareas se ejecuten mientras tanto.
- **Básicamente** Múltiples tareas parecen ejecutarse al mismo tiempo, aunque el sistema puede alternar rápidamente entre ellas (no necesariamente en paralelo).
- **Ejemplo** Un servidor web que maneja múltiples solicitudes de clientes. Aunque está basado en un solo hilo, puede intercalar la gestión de diferentes conexiones.

1.4.4 ¿Dónde entra la concurrencia?

- **Secuencial vs Concurrencia** En un sistema secuencial, no hay concurrencia. Todo se ejecuta en un orden fijo. En un sistema concurrente, múltiples tareas parecen ejecutarse al mismo tiempo, aunque no necesariamente lo hacen en paralelo.
- **Paralelo vs Concurrencia** Un sistema paralelo puede ejecutar múltiples tareas al mismo tiempo, utilizando varios núcleos de CPU. La concurrencia puede ocurrir dentro de un sistema paralelo (por ejemplo, múltiples tareas concurrentes ejecutándose en diferentes hilos o procesos), pero también puede ocurrir en un sistema con un solo núcleo, alternando entre tareas.

1.5 Menciona frameworks o estructuras para implementar paralelismo y concurrencia en python

1.5.1 Asyncio:

- Framework para implementar concurrencia asíncrona sin bloquear el flujo principal.
- Utiliza coroutines (async/await) para manejar múltiples tareas de I/O de forma no bloqueante.
- **Ideal para:** Aplicaciones de red y tareas I/O intensivas como operaciones en bases de datos o solicitudes HTTP.

1.5.2 Joblib:

- Facilita el paralelismo de tareas costosas en CPU, dividiendo el trabajo en paralelo.
- Utilizado en machine learning y procesamiento de big data.
- **Ideal para:** Paralelizar bucles o tareas pesadas, entrenar modelos o preprocesar datos en paralelo.

1.5.3 Dask:

- Permite el procesamiento paralelo de big data en múltiples núcleos o nodos.
- Maneja DataFrames, arrays y tareas complejas de manera distribuida y paralela.
- **Ideal para:** Big Data, cálculos distribuidos en clústeres o múltiples núcleos.

1.5.4 Threading:

- Permite la ejecución concurrente de tareas ligeras utilizando hilos en lugar de procesos.
- No ofrece paralelismo *verdadero* debido al GIL, pero es útil para tareas de I/O donde el bloqueo es común.
- **Ideal para:** Manejo de múltiples conexiones de red o tareas que dependen de I/O.

1.5.5 Twisted:

- Framework de red basado en eventos para manejar conexiones concurrentes de manera eficiente.
- Facilita la creación de servidores y aplicaciones de red en tiempo real.
- **Ideal para:** Aplicaciones concurrentes de red como servidores HTTP, chat en tiempo real y websockets.

1.5.6 Celery:
- Sistema de tareas distribuidas que permite ejecutar procesos en segundo plano, utilizando colas de mensajes como RabbitMQ o Redis.
- Utilizado para tareas asíncronas y distribuidas en aplicaciones web.
- **Ideal para:** Enviar correos electrónicos, procesamiento en segundo plano y tareas que no necesitan ser ejecutadas en tiempo real.

1.5.7 Multiprocessing:
- Permite ejecutar tareas en múltiples procesos independientes, cada uno en su propio núcleo de CPU, lo que habilita el paralelismo.
- Utiliza la creación de procesos, no hilos, así evita los problemas con el Global Interpreter Lock de Python.
- **Ideal para:** Tareas que consumen mucha CPU (procesamiento de imágenes, cálculos intensivos).