Informaticae Abstracta
Año 2024, Vol. 2, No. 1, pp. 36-72 JUL | DIC 2024

Concurrencia en Python: Teoría y una Aplicación para la Gestión del Portapapeles de Microsoft Windows

Salas-García, Javier; Portillo-Rodríguez, Otniel; Valle-Cruz, David; Moran-Gonzalez, Miguel Armando

Información del Artículo

Recibido: 10-sep-2023
Aceptado: 18-dic-2023

Palabras clave: threading, python, sincronización de hilos, portapapeles mejorado

Resumen

Este artículo presenta un análisis del «threading» en Python, abordando su implementación, optimización y aplicaciones prácticas en el desarrollo de software moderno. Partiendo de los fundamentos de la programación concurrente, se examina el modelo de concurrencia específico de Python, incluyendo las implicaciones del Global Interpreter Lock (GIL). El estudio profundiza en las herramientas de sincronización como locks, semáforos, eventos y colas, proporcionando ejemplos prácticos de su aplicación. Se analizan las situaciones en las que el «threading» mejora significativamente el rendimiento, particularmente en operaciones intensivas de E/S y procesamiento paralelo de datos. El artículo culmina con un caso de estudio detallado de una aplicación de visor de historial del portapapeles de Microsoft Windows, demostrando la aplicación práctica de los conceptos discutidos. Este artículo ofrece puntos tanto para desarrolladores novatos como experimentados, contribuyendo al entendimiento y la implementación efectiva de la programación concurrente en Python.

1. Introducción

En el campo de la ciencia de la computación, coexisten dos paradigmas fundamentales de ejecución de programas: la programación secuencial y la programación paralela. Estos enfoques difieren significativamente en su metodología de ejecución de instrucciones y en su capacidad para utilizar los recursos computacionales disponibles [1].

La programación secuencial, también denominada programación lineal, se caracteriza por la ejecución de instrucciones en un orden estrictamente definido. En este modelo, cada instrucción se procesa de manera individual y consecutiva, siguiendo una secuencia predeterminada [2].

El tiempo de ejecución total de un programa secuencial (T_s) puede representarse como la suma de los tiempos de ejecución de cada instrucción individual:

$$T_s = \sum_{i=1}^{n} t_i \tag{1}$$

donde n es el número total de instrucciones y t_i es el tiempo de ejecución de la i-ésima instrucción.

En contraste, la programación paralela permite la ejecución simultánea de múltiples instrucciones o secuencias de instrucciones. Este enfoque se basa en el principio de descomposición de un problema en subproblemas que pueden resolverse concurrentemente [3].

El tiempo de ejecución de un programa paralelo (T_p) puede modelarse de la siguiente manera:

$$T_p = T_s/p + T_o \tag{2}$$

donde p es el número de procesadores y T_o es el tiempo de sobrecarga debido a la comunicación y sincronización entre procesos.

La eficiencia de la paralelización se puede cuantificar mediante la ley de Amdahl [4], que establece que la aceleración máxima (S) que se puede lograr está limitada por la fracción del programa que no se puede paralelizar:

$$S = \frac{1}{(1-f) + f/p} \tag{3}$$

donde f es la fracción del programa que se puede paralelizar y p es el número de procesadores.

En el contexto específico de Python, el módulo «threading» proporciona una interfaz para implementar programación paralela a nivel de hilos. Aunque el Intérprete Global de Python (GIL) impone ciertas limitaciones en la ejecución verdaderamente paralela de hilos en operaciones intensivas de CPU, el «threading» en Python sigue siendo una herramienta valiosa para mejorar el rendimiento en tareas limitadas por operaciones de entrada/salida [5].

De este modo, la elección entre programación secuencial y paralela depende de factores como la naturaleza del problema, la arquitectura del sistema y los requisitos de rendimiento. Un análisis cuidadoso de estos factores es esencial para determinar el enfoque más adecuado en cada situación específica.

2. Importancia del «Threading» en la Era de Procesadores Multinúcleo

La proliferación de procesadores multinúcleo ha revolucionado el panorama de la computación, haciendo que el «threading» se convierta en un aspecto crucial del desarrollo de software moderno. El «threading», como implementación de la programación concurrente a nivel de software, permite aprovechar eficazmente la capacidad de procesamiento paralelo que ofrecen estas arquitecturas avanzadas [6].

En el contexto de los sistemas multinúcleo, el «threading» adquiere una relevancia particular por varias razones:

1. **Utilización eficiente de recursos:** El «threading» permite que múltiples tareas se ejecuten simultáneamente en diferentes núcleos, maximizando la utilización de los recursos de hardware disponibles. Esto se traduce en una mejora significativa del rendimiento, especialmente en aplicaciones que manejan cargas de trabajo intensivas [7].

2. **Mejora de la capacidad de respuesta:** En aplicaciones interactivas, el «threading» puede mejorar significativamente la experiencia del usuario al permitir que las operaciones de larga duración se ejecuten en segundo plano, manteniendo la interfaz de usuario receptiva [8].

3. **Escalabilidad:** A medida que aumenta el número de núcleos en los procesadores modernos, las aplicaciones que utilizan «threading» de manera efectiva pueden escalar su rendimiento de forma casi lineal, un fenómeno conocido como escalabilidad de Gustafson [9].

4. **Eficiencia energética:** El «threading» permite una distribución más uniforme de la carga de trabajo entre los núcleos, lo que puede conducir a una mejor gestión térmica y eficiencia energética en comparación con la ejecución en un solo núcleo a alta frecuencia [10].

Sin embargo, es importante destacar que el «threading» también introduce desafíos significativos. La sincronización y comunicación entre hilos pueden ser complejas y propensas a errores, como las condiciones de carrera y los interbloqueos. Además, en lenguajes como Python, el Global Interpreter Lock (GIL) puede limitar los beneficios del «threading» en operaciones intensivas de CPU [5].

A pesar de estos desafíos, el «threading» sigue siendo una herramienta fundamental para el desarrollo de software eficiente en la era de los procesadores multinúcleo. Su importancia se magnifica en aplicaciones que requieren un alto grado de concurrencia, como servidores web, sistemas de bases de datos y aplicaciones de procesamiento de datos a gran escala.

3. Beneficios del «Threading» para la Optimización de Programas

El «threading», como técnica de programación concurrente, ofrece una serie de beneficios significativos para la optimización de programas. Estos beneficios se manifiestan en diversos aspectos del rendimiento y la eficiencia del software, especialmente en el contexto de sistemas con múltiples núcleos de procesamiento. A continuación se describen algunos de estos beneficios.

1. Mejora del tiempo de respuesta: Uno de los beneficios más notables del «threading» es la mejora en el tiempo de respuesta de las aplicaciones. Al permitir que múltiples tareas se ejecuten concurrentemente, el «threading» puede reducir significativamente el tiempo de espera del usuario.

2. Aumento del «throughput»: El «threading» puede aumentar el throughput de un programa, es decir, la cantidad de trabajo realizado por unidad de tiempo. En sistemas con múltiples procesadores o núcleos, los hilos pueden distribuirse entre estos recursos de hardware, permitiendo que se realicen más operaciones simultáneamente. Este paralelismo puede llevar a mejoras sustanciales en el rendimiento, especialmente en aplicaciones que manejan grandes volúmenes de datos o cálculos complejos [11].

3. Modelo de programación más «natural» para ciertos problemas: Para ciertos tipos de problemas, el «threading» proporciona un modelo de programación más natural y fácil de conceptualizar. Por ejemplo, en simulaciones de sistemas físicos o en modelado de entidades independientes, cada entidad puede

representarse como un hilo separado, lo que facilita la implementación y el mantenimiento del código [7].

4. Escalabilidad mejorada: El «threading» permite que las aplicaciones escalen de manera más eficiente con el aumento de los recursos de hardware. A medida que se añaden más núcleos o procesadores a un sistema, las aplicaciones bien diseñadas que utilizan «threading» pueden aprovechar automáticamente estos recursos adicionales sin necesidad de modificaciones significativas en el código [12].

5. Latencia reducida en sistemas distribuidos: En sistemas distribuidos, el «threading» puede ayudar a reducir la latencia al permitir que múltiples operaciones de red se realicen simultáneamente. Esto es especialmente beneficioso en aplicaciones cliente-servidor, donde múltiples solicitudes pueden procesarse en paralelo, mejorando el tiempo de respuesta global del sistema [13].

6. Optimización del Uso de Caché: El «threading» puede llevar a una mejor utilización de la memoria caché del procesador. Al dividir un problema en subproblemas más pequeños que se ejecutan en hilos separados, es posible que cada hilo opere en un conjunto de datos que cabe completamente en la caché, reduciendo los costosos accesos a la memoria principal [14].

Estos beneficios del «threading» contribuyen significativamente a la optimización de programas, permitiendo un mejor aprovechamiento de los recursos de hardware modernos y mejorando el rendimiento general de las aplicaciones en diversos escenarios.

4. Fundamentos del «Threading»

En el ámbito de la programación concurrente, el concepto de thread (hilo en español) es fundamental. Un thread puede definirse como la unidad más pequeña de ejecución dentro de un proceso que puede ser gestionada independientemente por el planificador de un sistema operativo [15]. En términos más sencillos, un thread representa una secuencia única de instrucciones que pueden ejecutarse de forma paralela con otras secuencias dentro del mismo programa.

Para comprender mejor este concepto, se puede imaginar un thread como un trabajador individual en una fábrica. Así como múltiples trabajadores pueden realizar diferentes tareas simultáneamente en una línea de producción, varios threads pueden ejecutar diferentes partes de un programa al mismo tiempo. Cada thread tiene su propio contador de programa, pila de ejecución y conjunto de registros, pero comparte el espacio de memoria y los recursos del proceso principal con otros threads [16, 17].

En el contexto de la programación, los threads se utilizan para lograr la concurrencia, permitiendo que múltiples operaciones se realicen simultáneamente. Esto es especialmente útil en sistemas con múltiples núcleos de procesamiento, donde diferentes threads pueden ejecutarse en paralelo en diferentes núcleos, mejorando potencialmente el rendimiento general del programa [11].

Es importante destacar que la creación y gestión de threads conlleva cierta sobrecarga computacional. Cada thread requiere recursos del sistema, como espacio en la pila y tiempo de CPU para la conmutación de contexto. Por lo tanto, el uso eficiente de threads implica un equilibrio entre el número de threads creados y los beneficios de rendimiento obtenidos [8].

En lenguajes de programación modernos como Python, Java o C++, existen bibliotecas y módulos específicos que facilitan la creación y gestión de threads. Estos proporcionan abstracciones de alto nivel que permiten a los desarrolladores implementar programas concurrentes sin necesidad de manejar directamente los detalles de bajo nivel de la creación y sincronización de threads [5].

En resumen, un thread es una unidad de ejecución ligera dentro de un proceso que permite la concurrencia en programas. Su comprensión y uso adecuado son muy importantes para el desarrollo de software eficiente y de alto rendimiento en la era de los sistemas multinúcleo y las aplicaciones complejas.

5. Diferencias entre Procesos y Threads

Para comprender plenamente el concepto de «threading», es crucial establecer una distinción clara entre procesos y threads. Aunque ambos son unidades de ejecución, difieren significativamente en su estructura, gestión de recursos y propósito dentro de un sistema operativo.

Un proceso, en el contexto de sistemas operativos, representa una instancia de un programa en ejecución. Cada proceso tiene su propio espacio de direcciones de memoria, lo que significa que opera de manera aislada y no puede acceder directamente a la memoria de otros procesos. Esta característica proporciona un alto grado de protección y aislamiento entre diferentes aplicaciones en ejecución [16].

En contraste, los threads existen dentro del contexto de un proceso. Múltiples threads de un mismo proceso comparten el espacio de direcciones de memoria y otros recursos del proceso padre. Esta característica permite una comunicación más eficiente entre threads, pero también requiere mecanismos de sincronización para evitar conflictos en el acceso a datos compartidos [17].

La creación y gestión de procesos implica una sobrecarga significativa para el sistema operativo. Cada nuevo proceso requiere la asignación de un nuevo espacio de direcciones, la inicialización de estructuras de datos del kernel y la copia de recursos del proceso padre. Por otro lado, la creación de threads es más ligera, ya que comparten los recursos del proceso existente, lo que resulta en un menor tiempo de creación y cambio de contexto [15].

En términos de comunicación, los procesos requieren mecanismos de comunicación entre procesos (IPC) proporcionados por el sistema operativo, como tuberías, colas de mensajes o memoria compartida. Estos mecanismos son relativamente costosos en términos de rendimiento. Los threads, al compartir el mismo espacio de memoria, pueden comunicarse de manera más eficiente simplemente accediendo a variables compartidas, aunque esto introduce la necesidad de sincronización [18].

La robustez es otra área de diferencia. Un fallo en un thread puede potencialmente afectar a todo el proceso y, por extensión, a todos los threads dentro de ese

proceso. En contraste, un fallo en un proceso generalmente no afecta directamente a otros procesos en ejecución, proporcionando un mayor nivel de aislamiento y estabilidad del sistema [19].

En cuanto a la escalabilidad, los threads ofrecen ventajas significativas en sistemas con múltiples núcleos o procesadores. Múltiples threads de un proceso pueden ejecutarse verdaderamente en paralelo en diferentes núcleos, mientras que los procesos, aunque pueden distribuirse entre núcleos, tienen una granularidad más gruesa y mayores costos de comunicación [11].

Es importante señalar que algunos sistemas operativos modernos implementan modelos híbridos, como los «lightweight processes» o procesos ligeros, que combinan características de procesos y threads para optimizar el rendimiento y la flexibilidad [20].

Comprender estas diferencias es fundamental para los desarrolladores de software, ya que influye directamente en las decisiones de diseño relacionadas con la concurrencia, la paralelización y la arquitectura general de las aplicaciones.

6. Modelo de Concurrencia en Python y el Global Interpreter Lock

El modelo de concurrencia en Python presenta características únicas que lo distinguen de otros lenguajes de programación. Python implementa la concurrencia principalmente a través de su módulo `threading`, que permite la creación y gestión de hilos. Sin embargo, la ejecución de estos hilos está fuertemente influenciada por una característica fundamental de la implementación de referencia de Python (CPython): el Global Interpreter Lock (GIL) [5].

El GIL es un mecanismo de bloqueo que asegura que sólo un hilo ejecute código de bytes de Python en un momento dado. Esta restricción se implementó originalmente para simplificar el manejo de la memoria y garantizar la seguridad de los hilos en operaciones que no son atómicas en la implementación subyacente en C de Python [21].

El funcionamiento del GIL tiene implicaciones significativas en el rendimiento de programas Python multiproceso:

1. **Limitación del paralelismo real**: Aunque Python permite la creación de múltiples hilos, el GIL impide que estos se ejecuten verdaderamente en paralelo en múltiples núcleos de CPU para operaciones intensivas de CPU [22].

2. **Eficiencia en operaciones I/O**: El GIL se libera durante operaciones de I/O, permitiendo que otros hilos se ejecuten mientras un hilo está bloqueado en I/O. Esto hace que el «threading» en Python sea particularmente eficiente para aplicaciones limitadas por I/O [26].

3. **Impacto en el rendimiento**: En aplicaciones intensivas en CPU, el GIL puede llevar a una disminución del rendimiento en sistemas multinúcleo, ya que los hilos compiten por adquirir el GIL [5].

A pesar de estas limitaciones, Python ofrece varias alternativas para lograr concurrencia y paralelismo:

1. **Multiprocessing**: El módulo `multiprocessing` permite crear procesos separados, cada uno con su propio intérprete Python y GIL, permitiendo un verdadero paralelismo en sistemas multinúcleo [23].

2. **Asyncio**: Introducido en Python 3.4, este módulo proporciona un framework para programación asíncrona basada en corrutinas, ideal para aplicaciones con un alto grado de concurrencia en I/O [24].

3. **Implementaciones alternativas**: Algunas implementaciones de Python, como Jython o IronPython, no tienen GIL, permitiendo un verdadero multithreading [5].

Es importante destacar que, a pesar de las limitaciones impuestas por el GIL, el modelo de concurrencia de Python sigue siendo efectivo para muchas aplicaciones, especialmente aquellas limitadas por I/O. Además, el debate sobre el futuro del GIL en Python continúa, con propuestas y experimentos en curso para su posible eliminación en futuras versiones del lenguaje [25].

Comprender el modelo de concurrencia de Python y las implicaciones del GIL es crucial para los desarrolladores que buscan optimizar el rendimiento de aplicaciones Python en entornos concurrentes y paralelos. Este conocimiento permite tomar decisiones informadas sobre la arquitectura de la aplicación y la elección de las herramientas de concurrencia más apropiadas para cada caso de uso específico.

7. Implementación Básica de «Threading» en Python

Introducción al Módulo «threading»

El módulo «threading» es una parte fundamental de la biblioteca estándar de Python, diseñado para facilitar la implementación de programación concurrente basada en hilos. Este módulo proporciona una capa de abstracción de alto nivel sobre las primitivas de threading del sistema operativo, permitiendo a los desarrolladores crear y gestionar hilos de manera eficiente y portable [27].

El propósito principal del módulo «threading» es permitir que múltiples operaciones se ejecuten concurrentemente dentro de un mismo proceso de Python. Esto es particularmente útil para mejorar el rendimiento de aplicaciones que realizan operaciones de entrada/salida intensivas o que se benefician de la ejecución paralela en sistemas con múltiples núcleos de procesamiento [5].

Entre las funcionalidades clave que ofrece el módulo «threading» se encuentran:

1. Creación y gestión de hilos: Proporciona mecanismos para iniciar, pausar y detener hilos.

2. Sincronización: Ofrece primitivas como locks, semáforos y condiciones para coordinar la ejecución entre hilos.

3. Comunicación entre hilos: Facilita el intercambio seguro de datos entre hilos concurrentes.

4. Agrupación de hilos: Permite organizar y gestionar grupos de hilos relacionados.

5. Temporizadores: Ofrece la capacidad de ejecutar funciones en hilos separados después de un intervalo de tiempo especificado.

Es importante destacar que el módulo «threading» opera dentro de las limitaciones impuestas por el Global Interpreter Lock (GIL) de Python, lo que significa que, aunque puede mejorar significativamente el rendimiento en tareas limitadas por I/O, puede no proporcionar ganancias de rendimiento significativas para tareas intensivas en CPU en implementaciones estándar de Python como CPython [22].

El módulo «threading» se basa en gran medida en la API de «threading» de bajo nivel proporcionada por el módulo «thread», pero ofrece una interfaz más amigable y de alto nivel. Esto hace que sea más accesible para los desarrolladores, al tiempo que proporciona un mayor control sobre la gestión de hilos en comparación con otras alternativas como el módulo «concurrent.futures» [21].

En el contexto del desarrollo de software moderno, el módulo «threading» de Python juega un papel crucial en la implementación de aplicaciones concurrentes, desde servidores web y sistemas de procesamiento de datos hasta interfaces de usuario responsivas y sistemas de automatización. Su integración con otras bibliotecas y frameworks de Python lo convierte en una herramienta versátil para abordar una amplia gama de desafíos de programación concurrente [26].

8. Creación y Gestión de Threads Básicos en Python

La implementación de threads en Python utilizando el módulo «threading» es relativamente sencilla. A continuación, se presentan ejemplos de código que demuestran la creación y gestión básica de threads.

Creación de un Thread Básico

El siguiente ejemplo muestra cómo crear y ejecutar un thread básico:

```python
import threading
import time

def worker():
    print(f"Thread {threading.current_thread().name} iniciando")
    time.sleep(2)
    print(f"Thread {threading.current_thread().name} finalizando")

# Creación del thread
thread = threading.Thread(target=worker, name="WorkerThread")

# Inicio del thread
thread.start()

# Espera a que el thread termine
thread.join()

print("Programa principal finalizando")
```

En este ejemplo, se define una función `worker` que será ejecutada por el thread. Se crea un objeto `Thread`, especificando la función objetivo y un nombre para el thread. El método `start()` inicia la ejecución del thread, y `join()` espera a que el thread termine antes de continuar con la ejecución del programa principal [27].

Creación de Múltiples Threads

Para demostrar la ejecución concurrente, se pueden crear múltiples threads:

```
import threading
import time

def worker(id):
    print(f"Thread {id} iniciando")
    time.sleep(2)
    print(f"Thread {id} finalizando")

threads = []
for i in range(5):
    thread = threading.Thread(target=worker, args=(i,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Todos los threads han finalizado")
```

Este ejemplo crea cinco threads que se ejecutan concurrentemente. Cada thread ejecuta la función `worker` con un identificador único [5].

Uso de una Subclase de Thread

Alternativamente, se puede crear una subclase de `Thread` para encapsular la funcionalidad:

```
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, id):
        super().__init__()
        self.id = id

    def run(self):
        print(f"Thread {self.id} iniciando")
        time.sleep(2)
        print(f"Thread {self.id} finalizando")

threads = []
for i in range(5):
    thread = MyThread(i)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Todos los threads han finalizado")
```

En este enfoque, se define una clase `MyThread` que hereda de `threading.Thread`. El método `run()` se sobrescribe para definir el comportamiento del thread [22].

Estos ejemplos demuestran los conceptos básicos de creación y gestión de threads en Python. Es importante notar que, debido al Global Interpreter Lock (GIL), estos threads no se ejecutarán en paralelo verdadero para operaciones intensivas de CPU en CPython, pero pueden ser muy efectivos para tareas limitadas por I/O [21].

9. Métodos Principales de la Clase Thread

La clase Thread en Python proporciona varios métodos para controlar el ciclo de vida y el comportamiento de los hilos. Entre estos, tres métodos destacan por su importancia y uso frecuente: `start()`, `join()`, y `is_alive()`. A continuación, se explica en detalle cada uno de estos métodos.

Método start()

El método `start()` se utiliza para iniciar la ejecución de un hilo. Cuando se llama a este método, Python realiza las siguientes acciones:

1. Crea un nuevo hilo en el sistema operativo.
2. Invoca el método `run()` del objeto Thread en el nuevo hilo.

Es importante notar que `start()` debe ser llamado solo una vez por objeto Thread. Intentar llamarlo más de una vez resultará en una excepción `RuntimeError` [27].

Ejemplo de uso:

```
        thread = threading.Thread(target=worker_function)
        thread.start()  # Inicia la ejecución del hilo
```

El método `start()` retorna inmediatamente, permitiendo que el programa continúe su ejecución sin esperar a que el hilo complete su tarea.

Método join()

El método `join()` se utiliza para esperar a que un hilo complete su ejecución. Este método bloquea el hilo que lo llama hasta que el hilo sobre el que se invoca termine su ejecución o hasta que transcurra un tiempo de espera opcional.

Características principales de `join()`:

- Puede especificar un tiempo de espera máximo como argumento (en segundos).
- Si no se especifica un tiempo de espera, esperará indefinidamente.
- Retorna `None` si el hilo terminó dentro del tiempo de espera, o un booleano indicando si el hilo aún está vivo.

Ejemplo de uso:

```
thread.join()  # Espera indefinidamente
thread.join(timeout=5)  # Espera hasta 5 segundos
```

El uso de `join()` es crucial para sincronizar la ejecución entre hilos y asegurar que ciertos hilos hayan completado su tarea antes de continuar con el programa principal [5].

Método is_alive()

El método `is_alive()` se utiliza para verificar si un hilo está actualmente en ejecución. Retorna `True` si el hilo ha sido iniciado y no ha terminado aún, y `False` en caso contrario.

Este método es útil para:

- Verificar el estado de un hilo antes de realizar ciertas operaciones.
- Implementar lógica de espera o reintento basada en el estado de los hilos.
- Depurar y monitorear el comportamiento de los hilos en una aplicación.

Ejemplo de uso:

```
while thread.is_alive():
    print("El hilo aún está en ejecución")
    time.sleep(1)
```

Es importante tener en cuenta que el estado de un hilo puede cambiar inmediatamente después de llamar a `is_alive()`, por lo que su valor debe tratarse como una aproximación en escenarios de alta concurrencia [22].

La comprensión y el uso adecuado de estos métodos son fundamentales para el manejo efectivo de hilos en Python. Permiten un control preciso sobre el ciclo de vida de los hilos, facilitando la implementación de patrones de concurrencia robustos y eficientes.

10. Sincronización y Comunicación entre Threads

Desafíos de la Programación Concurrente

La programación concurrente, aunque poderosa, introduce una serie de desafíos significativos que los desarrolladores deben abordar para garantizar la corrección y eficiencia de sus programas. Dos de los problemas más críticos y frecuentes en la programación concurrente son las condiciones de carrera (race conditions) y los interbloqueos (deadlocks).

Condiciones de Carrera (Race Conditions)

Las condiciones de carrera ocurren cuando múltiples threads acceden y manipulan datos compartidos de manera concurrente, y el resultado final depende del orden de ejecución de los threads [17]. Este fenómeno puede llevar a comportamientos impredecibles y errores difíciles de reproducir y depurar.

Considere el siguiente ejemplo conceptual:

```
1 contador = 0
2
3 def incrementar():
4     global contador
5     temp = contador
6     temp = temp + 1
7     contador = temp
8
9 # Suponga que dos threads ejecutan esta función concurrentemente
```

En este escenario, si dos threads ejecutan la función `incrementar()` simultáneamente, podrían leer el mismo valor inicial, incrementarlo y escribirlo de vuelta, resultando en un incremento único en lugar de dos. Este tipo de errores sutiles pueden tener consecuencias graves en sistemas de producción.

Las condiciones de carrera pueden manifestarse de diversas formas, incluyendo:

- Pérdida de actualizaciones
- Lecturas sucias (cuando un thread lee datos parcialmente actualizados por otro thread)
- Inconsistencias en estructuras de datos complejas

La prevención de condiciones de carrera generalmente implica el uso de mecanismos de sincronización para garantizar el acceso exclusivo a los recursos compartidos, un tema que se abordará en detalle más adelante [11].

Interbloqueos (Deadlocks)

Los interbloqueos representan otro desafío crítico en la programación concurrente. Un interbloqueo ocurre cuando dos o más threads se bloquean mutuamente, cada uno esperando que el otro libere un recurso, resultando en un estancamiento permanente [15].

Para que se produzca un interbloqueo, deben cumplirse cuatro condiciones (conocidas como las condiciones de Coffman):

1. Exclusión mutua: Al menos un recurso debe ser retenido en modo no compartible.
2. Retención y espera: Un thread debe retener al menos un recurso mientras espera adquirir recursos adicionales retenidos por otros threads.
3. No apropiación: Los recursos no pueden ser forzosamente quitados de un thread; deben ser liberados voluntariamente.
4. Espera circular: Debe existir un conjunto circular de threads, cada uno esperando un recurso retenido por el siguiente thread en el ciclo.

Un ejemplo clásico de escenario propenso a interbloqueos es el «problema de la cena de los filósofos», donde múltiples filósofos («threads») compiten por un número limitado de tenedores (recursos compartidos) [28].

La prevención y detección de interbloqueos son áreas críticas en el diseño de sistemas concurrentes. Las estrategias para abordar los interbloqueos incluyen:

- Prevención: Diseñar el sistema para que al menos una de las condiciones de Coffman no pueda ocurrir.
- Evasión: Tomar decisiones dinámicas sobre la asignación de recursos para evitar estados inseguros.

- Detección y recuperación: Permitir que los interbloqueos ocurran, pero implementar mecanismos para detectarlos y recuperarse de ellos.

Estos desafíos subrayan la complejidad inherente a la programación concurrente y la necesidad de un diseño cuidadoso y una implementación rigurosa al trabajar con threads. La comprensión profunda de estos problemas es crucial para desarrollar sistemas concurrentes robustos y fiables.

11. Herramientas de Sincronización

Locks (Cerrojos)

Los locks, también conocidos como cerrojos o mutex (exclusión mutua), son una de las herramientas fundamentales de sincronización en la programación concurrente. Su función principal es proteger recursos compartidos, asegurando que solo un thread a la vez pueda acceder a un recurso crítico [8].

Funcionamiento de los Locks

Un lock opera como un mecanismo de bloqueo binario que puede estar en uno de dos estados: bloqueado o desbloqueado. Cuando un thread adquiere un lock, este pasa al estado bloqueado, impidiendo que otros threads lo adquieran hasta que sea liberado. Los locks proporcionan las siguientes operaciones básicas:

- Adquirir (lock): Intenta obtener el lock. Si está disponible, lo adquiere y continúa la ejecución. Si no está disponible, el thread se bloquea hasta que pueda adquirirlo.
- Liberar (unlock): Libera el lock, permitiendo que otros threads lo adquieran.

Implementación en Python

En Python, los locks se implementan a través de la clase `threading.Lock`. He aquí un ejemplo básico de su uso:

```python
import threading

# Recurso compartido
counter = 0
# Creación del lock
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

# Creación de threads
threads = [threading.Thread(target=increment) for _ in range(5)]

# Inicio de threads
for thread in threads:
    thread.start()

# Espera a que todos los threads terminen
```

```
22 for thread in threads:
23     thread.join()
24
25 print(f"Contador final: {counter}")
```

En este ejemplo, el lock protege la operación de incremento del contador, asegurando que solo un thread a la vez pueda modificar el valor [22].

Ventajas de los Locks

- Simplicidad: Los locks son conceptualmente simples y fáciles de usar.
- Eficiencia: Proporcionan una sobrecarga mínima en comparación con otras herramientas de sincronización más complejas.
- Prevención de condiciones de carrera: Garantizan el acceso exclusivo a recursos compartidos.

Desafíos y Consideraciones

A pesar de su utilidad, los locks presentan algunos desafíos:

- Deadlocks: Si no se gestionan adecuadamente, pueden llevar a situaciones de interbloqueo.
- Granularidad: Un lock demasiado amplio puede reducir el paralelismo, mientras que uno demasiado fino puede aumentar la complejidad y la sobrecarga.
- Inversión de prioridad: Puede ocurrir cuando un thread de baja prioridad mantiene un lock necesario para un thread de alta prioridad.

Variantes de Locks

Python también ofrece variantes más especializadas de locks:

- `RLock` (Reentrant Lock): Permite que un mismo thread adquiera el lock múltiples veces sin bloquearse a sí mismo.
- `Condition`: Combina la funcionalidad de un lock con la capacidad de esperar por una condición específica.

El uso efectivo de locks requiere una comprensión profunda de la arquitectura del sistema y los patrones de acceso a los recursos compartidos. Cuando se implementan correctamente, los locks son una herramienta poderosa para garantizar la consistencia y prevenir condiciones de carrera en programas concurrentes [21].

Semáforos

Los semáforos son otra herramienta fundamental de sincronización en la programación concurrente. Introducidos por Edsger Dijkstra en 1965, los semáforos proporcionan un mecanismo más flexible que los locks para controlar el acceso a recursos compartidos y coordinar la ejecución de múltiples threads [29].

Concepto y Funcionamiento

Un semáforo es esencialmente un contador que puede ser decrementado o incrementado por threads concurrentes. El valor del semáforo representa el número de unidades de un recurso disponible. Los semáforos soportan dos operaciones principales:

- `acquire()` (también conocido como `wait()` o `P()`): Decrementa el contador. Si el contador es cero, el thread se bloquea hasta que el contador sea mayor que cero.
- `release()` (también conocido como `signal()` o `V()`): Incrementa el contador y desbloquea un thread en espera, si lo hay.

Tipos de Semáforos

Existen dos tipos principales de semáforos:

1. **Semáforos binarios**: Solo pueden tomar los valores 0 o 1. Son similares a los locks, pero permiten que un thread libere un semáforo que fue adquirido por otro thread.
2. **Semáforos de conteo**: Pueden tomar cualquier valor entero no negativo. Son útiles para representar recursos con múltiples instancias.

Implementación en Python

En Python, los semáforos se implementan a través de la clase `threading.Semaphore`. Aquí hay un ejemplo de su uso:

```python
import threading
import time

# Creación de un semáforo con valor inicial 2
semaphore = threading.Semaphore(2)

def worker(id):
    print(f"Thread {id} intentando adquirir el semáforo")
    with semaphore:
        print(f"Thread {id} adquirió el semáforo")
        time.sleep(2)  # Simula algún trabajo
    print(f"Thread {id} liberó el semáforo")

# Creación y ejecución de 5 threads
threads = [threading.Thread(target=worker, args=(i,)) for i in range(5)]
for thread in threads:
    thread.start()

for thread in threads:
    thread.join()
```

En este ejemplo, el semáforo limita el número de threads que pueden ejecutar la sección crítica simultáneamente a dos [22].

Aplicaciones de los Semáforos

Los semáforos son particularmente útiles en varios escenarios:

- **Control de acceso a recursos limitados**: Por ejemplo, controlar el número de conexiones simultáneas a una base de datos.
- **Sincronización de productor-consumidor**: Coordinar threads productores y consumidores que operan sobre un buffer compartido.
- **Implementación de barreras**: Sincronizar múltiples threads para que esperen en un punto específico hasta que todos hayan llegado.

Ventajas de los Semáforos

- **Flexibilidad**: Pueden modelar una variedad de problemas de sincronización más allá de la exclusión mutua simple.
- **Eficiencia**: Proporcionan una forma eficiente de gestionar múltiples instancias de un recurso.
- **Prevención de condiciones de carrera**: Al igual que los locks, ayudan a prevenir condiciones de carrera en el acceso a recursos compartidos.

Desafíos y Consideraciones

El uso de semáforos también presenta algunos desafíos:

- **Complejidad**: Pueden ser más difíciles de razonar y depurar que los locks simples, especialmente en sistemas complejos.
- **Riesgo de deadlock**: El uso incorrecto de semáforos puede llevar a situaciones de interbloqueo.
- **Inversión de prioridad**: Al igual que con los locks, puede ocurrir inversión de prioridad si no se maneja adecuadamente.

Los semáforos son una herramienta poderosa en el arsenal de sincronización del programador concurrente. Su versatilidad los hace adecuados para una amplia gama de problemas de sincronización, pero requieren una comprensión cuidadosa y una implementación precisa para evitar errores sutiles [17].

Eventos

Los eventos son otra herramienta importante en el arsenal de sincronización para la programación concurrente. Proporcionan un mecanismo simple pero poderoso para la comunicación entre threads, permitiendo que un thread señalice a otros threads sobre la ocurrencia de una condición o estado específico [8].

Concepto y Funcionamiento

Un evento es esencialmente un objeto que actúa como una señal entre threads. Tiene dos estados principales: establecido (set) o no establecido (unset). Los threads pueden esperar a que un evento se establezca, o pueden establecer el evento para notificar a otros threads. Las operaciones principales de un evento son:

- `wait()`: Bloquea el thread hasta que el evento se establezca.
- `set()`: Establece el evento, despertando a todos los threads que están esperando.
- `clear()`: Restablece el evento a su estado no establecido.
- `is_set()`: Comprueba si el evento está establecido.

Implementación en Python

En Python, los eventos se implementan a través de la clase `threading.Event`. Aquí hay un ejemplo de su uso:

```python
import threading
import time

def waiter(event, name):
    print(f"{name} esperando el evento")
    event.wait()
    print(f"{name} recibió el evento")

def setter(event, name, delay):
    print(f"{name} esperando {delay} segundos antes de establecer el evento")
    time.sleep(delay)
    event.set()
    print(f"{name} estableció el evento")

# Creación del evento
event = threading.Event()

# Creación de threads
threads = [
    threading.Thread(target=waiter, args=(event, "Waiter 1")),
    threading.Thread(target=waiter, args=(event, "Waiter 2")),
    threading.Thread(target=setter, args=(event, "Setter", 3))
]

# Inicio de threads
for thread in threads:
    thread.start()

# Espera a que todos los threads terminen
for thread in threads:
    thread.join()
```

En este ejemplo, dos threads esperan a que el evento se establezca, mientras que un tercer thread lo establece después de una espera de 3 segundos [22].

Aplicaciones de los Eventos

Los eventos son particularmente útiles en varios escenarios:

- **Señalización de inicio**: Sincronizar el inicio de múltiples threads.
- **Notificación de finalización**: Indicar a otros threads que una tarea ha sido completada.
- **Control de flujo**: Pausar y reanudar la ejecución de threads basándose en ciertas condiciones.

- **Implementación de timeouts**: Combinar eventos con temporizadores para implementar esperas con límite de tiempo.

Ventajas de los Eventos

- **Simplicidad**: Los eventos proporcionan una interfaz simple y fácil de entender para la sincronización.
- **Eficiencia**: Son eficientes para escenarios de «esperar y notificar» con múltiples oyentes.
- **Flexibilidad**: Pueden ser utilizados en una variedad de patrones de sincronización.

Consideraciones y Mejores Prácticas

Al trabajar con eventos, es importante tener en cuenta:

- **Granularidad**: Usar eventos para sincronización de grano grueso, no para proteger accesos a recursos compartidos (para eso son más apropiados los locks).
- **Orden de ejecución**: Los eventos no garantizan un orden específico en el que los threads esperando serán notificados.
- **Condiciones de carrera**: Tener cuidado con las condiciones de carrera entre la comprobación del estado del evento y la espera.

Variantes y Extensiones

Algunas implementaciones de eventos ofrecen características adicionales:

- **Eventos con auto-reset**: Se restablecen automáticamente después de despertar a un thread.
- **Eventos con conteo**: Combinan características de eventos y semáforos, permitiendo un número específico de threads para proceder.

Los eventos son una herramienta valiosa en el diseño de sistemas concurrentes, proporcionando un mecanismo sencillo pero potente para la coordinación entre threads. Su uso efectivo puede simplificar significativamente la lógica de sincronización en muchos escenarios de programación concurrente [21].

Colas

Las colas son estructuras de datos fundamentales en la programación concurrente que facilitan la comunicación y sincronización entre threads. Proporcionan un mecanismo seguro para threads para intercambiar datos y coordinar su ejecución, especialmente útil en escenarios de productor-consumidor [8].

Concepto y Funcionamiento

Una cola en el contexto de la programación concurrente es una estructura de datos que permite a los threads añadir elementos (enqueue) y retirar elementos (dequeue) de manera segura para threads. Las operaciones principales son:

- `put()`: Añade un elemento a la cola. Si la cola está llena, el thread se bloquea hasta que haya espacio disponible.
- `get()`: Retira y devuelve un elemento de la cola. Si la cola está vacía, el thread se bloquea hasta que haya un elemento disponible.
- `task_done()`: Indica que una tarea ha sido completada.
- `join()`: Bloquea hasta que todos los elementos en la cola hayan sido procesados.

Implementación en Python

En Python, las colas thread-safe se implementan a través del módulo `queue`. Aquí hay un ejemplo de su uso:

```python
import threading
import queue
import time

def producer(q):
    for i in range(5):
        item = f"item-{i}"
        q.put(item)
        print(f"Producido: {item}")
        time.sleep(1)

def consumer(q):
    while True:
        item = q.get()
        if item is None:
        break
        print(f"Consumido: {item}")
        q.task_done()

# Creación de la cola
q = queue.Queue()

# Creación de threads
producer_thread = threading.Thread(target=producer, args=(q,))
consumer_thread = threading.Thread(target=consumer, args=(q,))

# Inicio de threads
producer_thread.start()
consumer_thread.start()

# Espera a que el productor termine
producer_thread.join()

# Señal de finalización al consumidor
q.put(None)

# Espera a que el consumidor termine
consumer_thread.join()
```

En este ejemplo, un thread productor genera elementos y los coloca en la cola, mientras que un thread consumidor los retira y procesa [22].

Tipos de Colas en Python

Python ofrece varios tipos de colas en el módulo `queue`:

- `Queue`: Cola FIFO (First-In-First-Out) estándar.
- `LifoQueue`: Cola LIFO (Last-In-First-Out) o pila.
- `PriorityQueue`: Cola donde los elementos se ordenan por prioridad.

Aplicaciones de las Colas

Las colas son particularmente útiles en varios escenarios:

- **Patrón productor-consumidor**: Ideal para situaciones donde unos threads producen datos y otros los consumen.
- **Balanceo de carga**: Distribuir tareas entre múltiples threads trabajadores.
- **Buffering**: Manejar flujos de datos asincrónicos o de velocidad variable.
- **Desacoplamiento de componentes**: Permitir que diferentes partes de un programa operen de manera independiente.

Ventajas de las Colas

- **Thread-safety**: Las operaciones de cola son atómicas y seguras para threads.
- **Bloqueo automático**: Manejan automáticamente el bloqueo y desbloqueo de threads.
- **Flexibilidad**: Pueden adaptarse a diversos patrones de comunicación entre threads.
- **Escalabilidad**: Facilitan la implementación de sistemas escalables con múltiples productores y consumidores.

Consideraciones y Mejores Prácticas

Al trabajar con colas, es importante tener en cuenta:

- **Tamaño de la cola**: Considerar limitar el tamaño de la cola para evitar el consumo excesivo de memoria.
- **Manejo de excepciones**: Implementar un manejo adecuado de excepciones, especialmente para operaciones que puedan bloquearse.
- **Señalización de finalización**: Utilizar un valor centinela (como `None`) para señalar el fin de la producción.
- **Monitoreo**: Considerar implementar mecanismos para monitorear el tamaño y el rendimiento de la cola.

Las colas proporcionan un mecanismo robusto y flexible para la comunicación entre threads, simplificando significativamente la implementación de patrones comunes en programación concurrente. Su uso efectivo puede llevar a diseños más limpios y eficientes en sistemas «multithreading» complejos [21].

12. Ejemplos Prácticos de Sincronización

Para ilustrar el uso práctico de las herramientas de sincronización discutidas, se presentan una serie de ejemplos que abordan problemas comunes en programación concurrente.

Ejemplo 1: Protección de Recursos Compartidos con Locks

Este ejemplo muestra cómo utilizar un lock para proteger un contador compartido:

```
import threading

class SharedCounter:
    def __init__(self):
        self.count = 0
        self.lock = threading.Lock()

    def increment(self):
        with self.lock:
        self.count += 1

    def get_count(self):
        with self.lock:
        return self.count

def worker(counter, num_increments):
    for _ in range(num_increments):
        counter.increment()

counter = SharedCounter()
threads = []
for _ in range(5):
    t = threading.Thread(target=worker, args=(counter, 10000))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Conteo final: {counter.get_count()}")
```

Este ejemplo demuestra cómo un lock puede prevenir condiciones de carrera al asegurar que solo un thread a la vez pueda modificar el contador [8].

Ejemplo 2: Control de Acceso con Semáforos

Este ejemplo utiliza un semáforo para limitar el número de threads que pueden acceder a un recurso simultáneamente:

```
import threading
import time

class LimitedResource:
    def __init__(self, max_workers):
```

```
        self.semaphore = threading.Semaphore(max_workers)

    def use_resource(self, worker_id):
        with self.semaphore:
            print(f"Worker {worker_id} está usando el recurso")
            time.sleep(1)  # Simula el uso del recurso
            print(f"Worker {worker_id} ha terminado de usar el recurso")

def worker(resource, worker_id):
    for _ in range(3):
        resource.use_resource(worker_id)

resource = LimitedResource(max_workers=3)
threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(resource, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Este ejemplo muestra cómo un semáforo puede controlar el acceso a un recurso limitado, permitiendo que solo un número específico de threads lo utilicen simultáneamente [22].

Ejemplo 3: Sincronización de Inicio con Eventos

Este ejemplo utiliza un evento para sincronizar el inicio de múltiples threads:

```
import threading
import time
import random

def worker(start_event, worker_id):
    print(f"Worker {worker_id} está listo y esperando")
    start_event.wait()
    print(f"Worker {worker_id} ha comenzado")
    time.sleep(random.random())
    print(f"Worker {worker_id} ha terminado")

start_event = threading.Event()
threads = []

for i in range(5):
    t = threading.Thread(target=worker, args=(start_event, i))
    threads.append(t)
    t.start()

print("Preparando para iniciar los workers...")
time.sleep(2)
print("¡Iniciando todos los workers!")
start_event.set()

for t in threads:
    t.join()
```

Este ejemplo demuestra cómo un evento puede ser utilizado para sincronizar el inicio de múltiples threads, asegurando que todos comiencen su tarea principal al mismo tiempo [21].

Ejemplo 4: Patrón Productor-Consumidor con Colas

El siguiente ejemplo implementa el patrón productor-consumidor utilizando una cola. Este patrón es útil en la programación concurrente y se utiliza para coordinar la comunicación entre threads que producen datos (productores) y threads que procesan esos datos (consumidores).

```
1  import threading
2  import queue
3  import time
4  import random
5
6  def producer(q, num_items):
7      for i in range(num_items):
8          item = f"Item {i}"
9          q.put(item)
10         print(f"Producido: {item}")
11         time.sleep(random.random())
12     q.put(None)  # Señal de finalización
13
14 def consumer(q):
15     while True:
16         item = q.get()
17         if item is None:
18         break
19         print(f"Consumido: {item}")
20         time.sleep(random.random() * 2)
21         q.task_done()
22
23 q = queue.Queue(maxsize=5)
24 producer_thread = threading.Thread(target=producer, args=(q, 10))
25 consumer_thread = threading.Thread(target=consumer, args=(q,))
26
27 producer_thread.start()
28 consumer_thread.start()
29
30 producer_thread.join()
31 consumer_thread.join()
32
33 print("Producción y consumo completados")
```

1. **Importaciones (líneas 1-4):**
   - `threading`: Para crear y gestionar threads.
   - `queue`: Proporciona la implementación thread-safe de una cola.
   - `time`: Para simular el tiempo de procesamiento.
   - `random`: Para generar tiempos de espera aleatorios.
2. **Función productor (líneas 6-12):**
   - Recibe como argumentos la cola `q` y el número de items a producir `num_items`.
   - Genera items y los coloca en la cola usando `q.put(item)`.
   - Simula tiempo de producción variable con `time.sleep(random.random())`.
   - Al finalizar, envía una señal de terminación (`None`) a la cola.
3. **Función consumidor (líneas 14-21):**
   - Ejecuta un bucle infinito que obtiene items de la cola con `q.get()`.

- Si el item es `None`, termina el bucle (señal de finalización).
- Simula el tiempo de procesamiento con `time.sleep(random.random() * 2)`.
- Llama a `q.task_done()` para indicar que ha terminado de procesar el item.

4. **Creación de la cola (línea 23):**
   - Se crea una cola con un tamaño máximo de 5 items.
   - Esto introduce un límite en la producción, haciendo que el productor espere si la cola está llena.
5. **Creación de threads (líneas 24-25):**
   - Se crean dos threads: uno para el productor y otro para el consumidor.
   - El thread productor se configura para producir 10 items.
6. **Inicio de threads (líneas 27-28):**
   - Se inician ambos threads con el método `start()`.
7. **Espera de finalización (líneas 30-31):**
   - Se usa `join()` para esperar a que ambos threads terminen su ejecución.

Funcionamiento del Patrón

- El productor genera items y los coloca en la cola. Si la cola está llena (5 items), el productor se bloquea hasta que haya espacio disponible.
- El consumidor continuamente intenta obtener items de la cola. Si la cola está vacía, se bloquea hasta que haya un item disponible.
- El uso de `queue.Queue` garantiza la sincronización thread-safe entre el productor y el consumidor.
- Los tiempos de espera aleatorios simulan variaciones en los tiempos de producción y consumo, demostrando cómo la cola actúa como un buffer entre productor y consumidor.
- La señal de finalización (`None`) permite que el consumidor sepa cuándo terminar su ejecución.

Ventajas del Patrón

- **Desacoplamiento:** El productor y el consumidor no necesitan conocerse mutuamente, solo interactúan a través de la cola.
- **Control de flujo:** La cola actúa como un buffer, permitiendo que productor y consumidor trabajen a ritmos diferentes.
- **Sincronización implícita:** La implementación thread-safe de `queue.Queue` maneja automáticamente los problemas de sincronización.

Estos ejemplos prácticos ilustran cómo las diferentes herramientas de sincronización pueden ser aplicadas para resolver problemas comunes en programación concurrente. Cada herramienta tiene sus propias fortalezas y es más adecuada para ciertos escenarios. La elección de la herramienta correcta depende de las necesidades específicas del problema y de las características del sistema concurrente que se está desarrollando.

13. Optimización de Programas con «Threading»

Situaciones en las que el «Threading» Mejora el Rendimiento

El «threading» puede mejorar significativamente el rendimiento de los programas en ciertas situaciones. Es crucial entender estos escenarios para aprovechar eficazmente la programación concurrente. A continuación, se analizan las principales situaciones en las que el «threading» puede ofrecer beneficios de rendimiento notables.

Operaciones Intensivas en E/S

Una de las situaciones más comunes donde el «threading» mejora el rendimiento es en programas con operaciones intensivas en entrada/salida (E/S). Estas operaciones incluyen:

- Lectura y escritura de archivos
- Comunicaciones de red
- Consultas a bases de datos
- Llamadas a APIs externas

En estos casos, un thread puede iniciar una operación de E/S y luego ceder el control, permitiendo que otros threads se ejecuten mientras se espera la finalización de la E/S. Esto maximiza la utilización del CPU y reduce el tiempo total de ejecución [8].

Ejemplo conceptual:

```
import threading
import requests

def fetch_url(url):
    response = requests.get(url)
    print(f"Fetched: {url}, Status: {response.status_code}")

urls = ["http://example.com", "http://example.org", "http://example.net"]
threads = []

for url in urls:
    thread = threading.Thread(target=fetch_url, args=(url,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

En este ejemplo, múltiples solicitudes HTTP se realizan concurrentemente, mejorando significativamente el tiempo total de ejecución en comparación con un enfoque secuencial.

Procesamiento Paralelo de Datos

Para tareas que involucran el procesamiento de grandes cantidades de datos independientes, el «threading» puede distribuir el trabajo entre múltiples núcleos de CPU, acelerando significativamente el procesamiento [22].

Escenarios típicos incluyen:

- Procesamiento de imágenes o video
- Análisis de grandes conjuntos de datos
- Simulaciones científicas
- Operaciones matriciales y vectoriales

Ejemplo conceptual:

```
import threading
import numpy as np

def process_chunk(chunk):
    # Simula algún procesamiento intensivo
    result = np.mean(chunk ** 2)
    print(f"Processed chunk, result: {result}")

data = np.random.rand(1000000)
chunk_size = len(data) // 4
threads = []

for i in range(4):
    start = i * chunk_size
    end = start + chunk_size
    chunk = data[start:end]
    thread = threading.Thread(target=process_chunk, args=(chunk,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

Este ejemplo divide un gran array en chunks y procesa cada chunk en un thread separado, aprovechando múltiples núcleos de CPU.

Aplicaciones de Interfaz de Usuario

En aplicaciones con interfaces de usuario gráficas (GUI), el «threading» puede mejorar significativamente la capacidad de respuesta. Permite que las operaciones de larga duración se ejecuten en threads separados, manteniendo la interfaz de usuario receptiva [26].

Beneficios clave:

- Prevención de «congelamiento» de la interfaz
- Actualización suave de la interfaz durante operaciones largas
- Mejor experiencia de usuario en general

Servidores y Aplicaciones de Red

Los servidores y aplicaciones de red se benefician enormemente del «threading», permitiendo manejar múltiples conexiones simultáneamente. Esto es crucial para:

- Servidores web
- Servidores de bases de datos
- Aplicaciones de chat o mensajería
- Sistemas de monitoreo en tiempo real

El «threading» permite que cada conexión o solicitud de cliente sea manejada en un thread separado, maximizando el throughput y la capacidad de respuesta del servidor [21].

Cálculos Asíncronos y Tareas en Segundo Plano

El «threading» es ideal para ejecutar cálculos asíncronos o tareas en segundo plano que no requieren interacción inmediata con el usuario. Ejemplos incluyen:

- Generación de informes
- Cálculos predictivos
- Actualización de cachés
- Tareas de mantenimiento programadas

Estas tareas pueden ejecutarse en threads separados, permitiendo que el programa principal continúe su ejecución sin interrupciones.

14. Visor del Historial del Portapapeles: Una Herramienta para Mejorar la Productividad

El Visor del Historial del Portapapeles es una aplicación diseñada para mejorar significativamente la eficiencia y comodidad en la gestión de información copiada en su computadora. Esta herramienta ofrece una solución práctica para uno de los desafíos comunes en el trabajo diario con ordenadores: la necesidad de acceder rápidamente a múltiples elementos copiados previamente.

Funcionalidad Principal

- **Historial de Copiado**: La aplicación mantiene un registro de los últimos 20 elementos copiados al portapapeles de su sistema. Esto significa que ya no tendrá que preocuparse por perder información importante que copió hace unos momentos.

- **Interfaz Visual Intuitiva**: Todos los elementos copiados se muestran en una lista clara y fácil de navegar. Cada entrada muestra una vista previa del contenido copiado, permitiéndole identificar rápidamente el elemento que necesita.

- **Acceso Rápido**: Con un simple movimiento del mouse, puede acceder a cualquier elemento de su historial de copiado. No más copiar y pegar repetitivo o búsqueda en documentos para encontrar información que ya había copiado.

- **Pegado Automático**: Al mantener el cursor sobre un elemento del historial durante un segundo, el contenido se copia automáticamente a su portapapeles actual y se pega en la aplicación activa. Esto agiliza enormemente el proceso de recuperar y utilizar información previamente copiada.

- **Monitoreo Continuo**: La aplicación funciona silenciosamente en segundo plano, capturando cada elemento que copia sin interrumpir su flujo de trabajo.

Beneficios para el Usuario

- **Ahorro de Tiempo**: Recupere rápidamente información que copió anteriormente sin tener que buscar en documentos o navegar por ventanas.

- **Mejora de la Productividad**: Cambie fácilmente entre múltiples fragmentos de información copiada, ideal para tareas que requieren compilar datos de varias fuentes.

- **Reducción de Errores**: Evite errores comunes como sobrescribir accidentalmente información importante en el portapapeles o olvidar contenido crítico que fue copiado anteriormente.

- **Flujo de Trabajo Ininterrumpido**: La naturaleza no intrusiva de la aplicación significa que puede beneficiarse de sus características sin alterar sus hábitos de trabajo establecidos.

- **Versatilidad**: Útil para una amplia gama de tareas, desde redacción y programación hasta investigación y gestión de datos.

Cómo Utilizar la Aplicación

1. **Inicio**: Al abrir la aplicación, verá una ventana que muestra su historial de portapapeles.

2. **Copiado Normal**: Continúe copiando información como lo hace habitualmente (Ctrl+C o clic derecho y copiar). La aplicación registrará automáticamente cada elemento copiado.

3. **Visualización del Historial**: En cualquier momento, puede abrir la ventana de la aplicación para ver una lista de sus elementos copiados recientemente.

4. **Recuperación de Elementos**: Para reutilizar un elemento copiado anteriormente, simplemente mueva el cursor sobre él en la lista y manténgalo allí por un segundo. El elemento se copiará automáticamente y se pegará en su aplicación activa.

5. **Personalización**: Puede ajustar el tamaño de la fuente para una mejor visibilidad y borrar el historial cuando lo desee.

El Visor del Historial del Portapapeles es una herramienta útil para cualquier persona que trabaje intensivamente con procedimientos rutinarios de copiar y pegar información, tales como el llenado de un formulario entre distintas aplicaciones. Al proporcionar un acceso rápido y fácil a su historial de copiado, esta aplicación no solo ahorra tiempo, sino que también mejora la precisión y eficiencia en una amplia gama de tareas informáticas. Ya sea que esté redactando un informe, programando, investigando o simplemente gestionando información diaria, esta herramienta se integrará perfectamente en su flujo de trabajo, mejorando su productividad sin esfuerzo adicional.

15. Análisis del Script pp.py

El script `pp.py` proporciona una implementación práctica de varios conceptos de «threading» y sincronización que se han discutido anteriormente. A continuación, se analizan las partes clave del script.

Importación de Módulos y Configuración Inicial

```
import tkinter as tk
from tkinter import ttk
import win32clipboard
import win32con
import pyautogui
import threading
import time
from collections import deque
from pynput import keyboard
```

Esta sección importa los módulos necesarios, incluyendo «threading», que es necesaria para la implementación de concurrencia en Python, como se discutió en la sección «Implementación Básica de “Threading” en Python».

Definición de la Clase Principal

```
class ClipboardHistoryViewer:
    def __init__(self, root):
        self.root = root
        self.root.title("Visor del Historial del Portapapeles")
        self.clipboard_history = deque(maxlen=20)
        self.font_size = 10
        self.hover_item = None
        self.hover_time = 0
        self.last_copied_content = None
        self.clipboard_lock = threading.Lock()
        self.setup_ui()
        self.start_monitoring()
        self.start_keyboard_listener()
```

La clase `ClipboardHistoryViewer` utiliza un `threading.Lock()` (línea 20) para sincronización, lo cual se relaciona con nuestra discusión sobre «Locks (cerrojos)» en la sección de herramientas de sincronización (Sección 11).

Monitoreo del Portapapeles

```
74 def start_monitoring(self):
75     self.monitoring_thread = threading.Thread(target=self.monitor_clipboard,
       ↪ daemon=True)
76     self.monitoring_thread.start()
77 
78 def monitor_clipboard(self):
79     last_content = ""
80     while True:
81         try:
82             with self.clipboard_lock:
83                 content = self.get_clipboard_content()
84             if content != last_content:
85                 self.root.after(0, self.add_to_history, content)
86                 last_content = content
87         except Exception as e:
88             print(f"Error al monitorear el portapapeles: {e}")
89         time.sleep(0.5)
```

Esta parte del script implementa un thread separado para monitorear el portapapeles (líneas 75-76), lo cual es un ejemplo práctico de «Creación y Gestión de Threads Básicos en Python» dicutidas en la Sección 8. El uso de `with self.clipboard_lock:` (línea 82) demuestra la aplicación práctica de locks para proteger recursos compartidos, como se explicó en «Ejemplos Prácticos de Sincronización» (Sección 12).

Manejo de Eventos de Teclado

```
136     def start_keyboard_listener(self):
137         def on_press(key):
138             if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
139                 self.ctrl_pressed = True
140             elif hasattr(key, 'char') and key.char == 'c' and self.ctrl_pressed
                ↪ :
141                 self.root.after(100, self.update_on_ctrl_c)
142 
143         def on_release(key):
144             if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
145                 self.ctrl_pressed = False
146 
147         self.ctrl_pressed = False
148         listener = keyboard.Listener(on_press=on_press, on_release=on_release)
149         listener.start()
```

Esta sección implementa un listener de teclado en un thread separado (línea 148), lo cual es otro ejemplo de la aplicación de «threading» para mejorar la capacidad de respuesta de la aplicación, como se discutió en «Optimización de Programas con “Threading” mejora el rendimiento» (Sección 13).

Actualización de la Interfaz de Usuario

```
151 def update_on_ctrl_c(self):
152     with self.clipboard_lock:
```

```
153         content = self.get_clipboard_content()
154         self.add_to_history(content)
```

Este método utiliza un lock para asegurar el acceso seguro al portapapeles (línea 152), demostrando nuevamente el uso práctico de locks para la sincronización, como se explicó en la sección de herramientas de sincronización (Sección 11).

Manejo de Eventos de la Interfaz de Usuario

```
167     def on_mouse_over(self, event):
168     item = self.tree.identify_row(event.y)
169     if item != self.hover_item:
170         self.hover_item = item
171         self.hover_time = time.time()
172     elif item and time.time() - self.hover_time >= 1:
173         content = self.tree.item(item, "values")[1]
174         if content.endswith("..."):
175             content = list(self.clipboard_history)[self.tree.index(item)]
176 
177         if content != self.last_copied_content:
178             with self.clipboard_lock:
179                 self.set_clipboard_content(content)
180             self.root.update()
181             pyautogui.hotkey('ctrl', 'v')
182             self.last_copied_content = content
183             self.status_var.set(f"Copiado: {content[:30]}...")
184             self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + ('
                    ↪ green_circle',))
185             self.root.after(2000, lambda: self.reset_status_and_color(item))
186         else:
187             self.status_var.set("Contenido ya copiado.")
188             self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + ('
                    ↪ green_circle',))
189             self.root.after(2000, lambda: self.reset_status_and_color(item))
```

Este método maneja los eventos del mouse en la interfaz de usuario. El uso de `with self.clipboard_lock:` (línea 178) para proteger el acceso al portapapeles es otro ejemplo de la aplicación de locks para evitar condiciones de carrera, como se discutió en «Sincronización y Comunicación entre "Threads"» (Sección 10).

En conjunto, este script demuestra la aplicación práctica de varios conceptos de «threading» y sincronización que se han discutido a lo largo de este artículo. Utiliza threads para mejorar la capacidad de respuesta de la aplicación, ejemplificando así el empleo de locks para proteger recursos compartidos, el manejo de eventos de usuario, todo lo cual se alinea con las mejores prácticas de programación concurrente en Python.

A continuación se presenta el script completo, que se puede descargar del repositorio github en el que se incluyen las instrucciones para descargar las librerías necesarias para su ejecución: `https://github.com/JavierSalasGarcia/ClipboardPlus`

```
1  import tkinter as tk
2  from tkinter import ttk
3  import win32clipboard
4  import win32con
5  import pyautogui
6  import threading
7  import time
8  from collections import deque
9  from pynput import keyboard
10 
11 class ClipboardHistoryViewer:
```

```
12      def __init__(self, root):
13          self.root = root
14          self.root.title("Visor del Historial del Portapapeles")
15          self.clipboard_history = deque(maxlen=20)
16          self.font_size = 10
17          self.hover_item = None
18          self.hover_time = 0
19          self.last_copied_content = None
20          self.clipboard_lock = threading.Lock()
21          self.setup_ui()
22          self.start_monitoring()
23          self.start_keyboard_listener()
24
25      def setup_ui(self):
26          self.frame = ttk.Frame(self.root, padding="10")
27          self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
28          self.root.columnconfigure(0, weight=1)
29          self.root.rowconfigure(0, weight=1)
30
31          self.tree = ttk.Treeview(self.frame, columns=("Indicador", "Contenido")
                ↪ , show="headings")
32          self.tree.heading("Indicador", text="")
33          self.tree.heading("Contenido", text="Contenido del Portapapeles")
34          self.tree.column("Indicador", width=30, stretch=tk.NO)
35          self.tree.column("Contenido", width=400, stretch=tk.YES)
36          self.tree.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
37          self.frame.columnconfigure(0, weight=1)
38          self.frame.rowconfigure(0, weight=1)
39
40          scrollbar = ttk.Scrollbar(self.frame, orient=tk.VERTICAL, command=self.
                ↪ tree.yview)
41          scrollbar.grid(row=0, column=1, sticky=(tk.N, tk.S))
42          self.tree.configure(yscrollcommand=scrollbar.set)
43
44          self.tree.bind("<Motion>", self.on_mouse_over)
45          self.tree.bind("<Leave>", self.on_mouse_leave)
46
47          button_frame = ttk.Frame(self.frame)
48          button_frame.grid(row=1, column=0, pady=10)
49
50          font_button = ttk.Button(button_frame, text="Aumentar tamaño de fuente"
                ↪ , command=self.increase_font_size)
51          font_button.grid(row=0, column=0, padx=5)
52
53          clear_button = ttk.Button(button_frame, text="Borrar historial",
                ↪ command=self.clear_history)
54          clear_button.grid(row=0, column=1, padx=5)
55
56          self.status_var = tk.StringVar()
57          status_label = ttk.Label(self.frame, textvariable=self.status_var)
58          status_label.grid(row=2, column=0, pady=5)
59
60          self.update_font()
61          self.setup_styles()
62
63      def setup_styles(self):
64          style = ttk.Style()
65          style.configure("Treeview", rowheight=30)
66          style.configure("Treeview", background="white", fieldbackground="white"
                ↪ , foreground="black")
67          style.map("Treeview", background=[('selected', '#a6a6a6')])
68
69          self.tree.tag_configure('odd', background='#F0F0F0')
70          self.tree.tag_configure('even', background='#FFFFFF')
71          self.tree.tag_configure('gray_circle', foreground='gray')
72          self.tree.tag_configure('green_circle', foreground='lime')
73
74      def start_monitoring(self):
```

```
        self.monitoring_thread = threading.Thread(target=self.monitor_clipboard
            ↪ , daemon=True)
        self.monitoring_thread.start()

    def monitor_clipboard(self):
        last_content = ""
        while True:
            try:
                with self.clipboard_lock:
                    content = self.get_clipboard_content()
                if content != last_content:
                    self.root.after(0, self.add_to_history, content)
                    last_content = content
            except Exception as e:
                print(f"Error al monitorear el portapapeles: {e}")
            time.sleep(0.5)

    def get_clipboard_content(self):
        formats = [
            (win32con.CF_UNICODETEXT, self.get_unicode_text),
            (win32con.CF_TEXT, self.get_text),
            (win32con.CF_BITMAP, self.get_image_text),
            (win32con.CF_HDROP, self.get_file_text)
        ]

        try:
            win32clipboard.OpenClipboard()
            for format, getter in formats:
                if win32clipboard.IsClipboardFormatAvailable(format):
                    content = getter()
                    if content:
                        return content
            return "Contenido no soportado"
        except Exception as e:
            print(f"Error al obtener contenido del portapapeles: {e}")
            return "Error al leer el portapapeles"
        finally:
            win32clipboard.CloseClipboard()

    def get_unicode_text(self):
        return win32clipboard.GetClipboardData(win32con.CF_UNICODETEXT)

    def get_text(self):
        return win32clipboard.GetClipboardData(win32con.CF_TEXT).decode('utf-8'
            ↪ )

    def get_image_text(self):
        return "[Imagen en el portapapeles]"

    def get_file_text(self):
        files = win32clipboard.GetClipboardData(win32con.CF_HDROP)
        return "[Archivos en el portapapeles]: " + ", ".join(files)

    def set_clipboard_content(self, content):
        try:
            win32clipboard.OpenClipboard()
            win32clipboard.EmptyClipboard()
            win32clipboard.SetClipboardText(content, win32con.CF_UNICODETEXT)
        except Exception as e:
            print(f"Error al establecer contenido en el portapapeles: {e}")
        finally:
            win32clipboard.CloseClipboard()

    def start_keyboard_listener(self):
        def on_press(key):
            if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
                self.ctrl_pressed = True
            elif hasattr(key, 'char') and key.char == 'c' and self.ctrl_pressed
```

```
                ↪ :
141                 self.root.after(100, self.update_on_ctrl_c)
142
143         def on_release(key):
144             if key == keyboard.Key.ctrl_l or key == keyboard.Key.ctrl_r:
145                 self.ctrl_pressed = False
146
147         self.ctrl_pressed = False
148         listener = keyboard.Listener(on_press=on_press, on_release=on_release)
149         listener.start()
150
151     def update_on_ctrl_c(self):
152         with self.clipboard_lock:
153             content = self.get_clipboard_content()
154             self.add_to_history(content)
155
156     def add_to_history(self, content):
157         if content and content not in self.clipboard_history:
158             self.clipboard_history.appendleft(content)
159             self.update_treeview()
160
161     def update_treeview(self):
162         self.tree.delete(*self.tree.get_children())
163         for i, content in enumerate(self.clipboard_history):
164             item = self.tree.insert("", "end", values=("*", content[:50] + "...
                ↪ " if len(content) > 50 else content),
165                                     tags=('odd' if i % 2 else 'even', '
                                        ↪ gray_circle'))
166
167     def on_mouse_over(self, event):
168         item = self.tree.identify_row(event.y)
169         if item != self.hover_item:
170             self.hover_item = item
171             self.hover_time = time.time()
172         elif item and time.time() - self.hover_time >= 1:
173             content = self.tree.item(item, "values")[1]
174             if content.endswith("..."):
175                 content = list(self.clipboard_history)[self.tree.index(item)]
176
177             if content != self.last_copied_content:
178                 with self.clipboard_lock:
179                     self.set_clipboard_content(content)
180                 self.root.update()
181                 pyautogui.hotkey('ctrl', 'v')
182                 self.last_copied_content = content
183                 self.status_var.set(f"Copiado: {content[:30]}...")
184                 self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + (
                    ↪ 'green_circle',))
185                 self.root.after(2000, lambda: self.reset_status_and_color(item)
                    ↪ )
186             else:
187                 self.status_var.set("Contenido ya copiado.")
188                 self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + (
                    ↪ 'green_circle',))
189                 self.root.after(2000, lambda: self.reset_status_and_color(item)
                    ↪ )
190
191     def on_mouse_leave(self, event):
192         self.hover_item = None
193         self.hover_time = 0
194         self.status_var.set("")
195
196     def reset_status_and_color(self, item):
197         self.status_var.set("")
198         self.tree.item(item, tags=self.tree.item(item, "tags")[:-1] + ('
            ↪ gray_circle',))
199
200     def increase_font_size(self):
```

```
        self.font_size += 2
        self.update_font()

    def update_font(self):
        style = ttk.Style()
        style.configure("Treeview", font=("TkDefaultFont", self.font_size))
        style.configure("Treeview.Heading", font=("TkDefaultFont", self.
            ↪ font_size, "bold"))

    def clear_history(self):
        self.clipboard_history.clear()
        self.update_treeview()
        self.status_var.set("Historial borrado")
        self.root.after(100, self.clear_clipboard)

    def clear_clipboard(self):
        with self.clipboard_lock:
            try:
                win32clipboard.OpenClipboard()
                win32clipboard.EmptyClipboard()
                win32clipboard.CloseClipboard()
            except Exception as e:
                print(f"Error al limpiar el portapapeles: {e}")

if __name__ == "__main__":
    root = tk.Tk()
    app = ClipboardHistoryViewer(root)
    root.mainloop()
```

Bibliografía

[1] Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

[2] Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: A quantitative approach*. Elsevier.

[3] Rauber, T., & Rünger, G. (2013). *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media.

[4] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483-485).

[5] Gorelick, M., & Ozsvald, I. (2014). *High performance Python: Practical performant programming for humans*. O'Reilly Media, Inc.

[6] Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

[7] Liu, X., & Chen, T. (2011). Survey of real-time processing systems for big data. *Procedia Engineering*, 15, 3750-3754.

[8] Goetz, B., & Peierls, T. (2006). *Java concurrency in practice*. Pearson Education.

[9] Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, 31(5), 532-533.

[10] Li, D., de Supinski, B. R., Schulz, M., Nikolopoulos, D. S., & Cameron, K. W. (2013). Thread-level energy efficiency optimization for multi-threaded applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)* (pp. 178-187). IEEE.

[11] Herlihy, M., & Shavit, N. (2011). *The art of multiprocessor programming*. Morgan Kaufmann.

[12] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 202-210.

[13] Tanenbaum, A. S., & Van Steen, M. (2016). *Distributed systems: Principles and paradigms*. Prentice-Hall.

[14] Drepper, U. (2007). What every programmer should know about memory. *Red Hat, Inc.*, 11.

[15] Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems*. Pearson.

[16] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts*. Wiley.

[17] Stallings, W. (2018). *Operating systems: Internals and design principles*. Pearson.

[18] Love, R. (2013). *Linux kernel development*. Addison-Wesley Professional.

[19] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating systems: Three easy pieces*. Arpaci-Dusseau Books.

[20] Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux Kernel: From I/O ports to process management*. O'Reilly Media, Inc.

[21] Beazley, D. M. (2010). *Python essential reference*. Addison-Wesley Professional.

[22] Ramalho, L. (2015). *Fluent Python: Clear, concise, and effective programming*. O'Reilly Media, Inc.

[23] Python Software Foundation. (2022). multiprocessing — Process-based parallelism. `https://docs.python.org/3/library/multiprocessing.html`

[24] Python Software Foundation. (2022). asyncio — Asynchronous I/O. `https://docs.python.org/3/library/asyncio.html`

[25] Smith, S., & Wouters, E. V. (2022). PEP 703 – Making the Global Interpreter Lock Optional in CPython. `https://peps.python.org/pep-0703/`

[26] Hellmann, D. (2011). *The Python standard library by example*. Addison-Wesley Professional.

[27] Python Software Foundation. (2023). threading — Thread-based parallelism. `https://docs.python.org/3/library/threading.html`

[28] Dijkstra, E. W. (1971). Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2), 115-138.

[29] Dijkstra, E. W. (1968). Cooperating sequential processes. In *The origin of concurrent programming* (pp. 65-138). Springer.

Direcciones de correo de los autores

Salas-García, Javier: jsalasg@uaemex.mx

Portillo-Rodríguez, Otniel: oportillor@uaemex.mx

Valle-Cruz, David: dvallec@uaemex.mx

Moran-Gonzalez, Miguel Armando: miguel@poilower.com