Introducción a los subprocesos (threads) en Python

Tabla de contenido

- ¿Qué es un hilo (thread)?
- Comenzando un hilo
    - Hilos de demonio
    - unir () un hilo
- Trabajar con muchos subprocesos
- Usando un ThreadPoolExecutor
- Condiciones de carrera
    - Un hilo
    - Dos hilos
    - Por qué este no es un ejemplo tonto
- Sincronización básica mediante bloqueo
- Punto muerto
- Roscado productor-consumidor
    - Productor-consumidor usando bloqueo
    - Productor-consumidor usando cola
- Subprocesos de objetos
    - Semáforo
    - Temporizador
    - Barrera
- Conclusión: Threading en Python

El subproceso de Python nos permite ejecutar diferentes partes de nuestro programa al mismo tiempo y puede simplificar el diseño. Si tienes algo de experiencia en Python y quieres acelerar tu programa usando subprocesos, ¡este tutorial es para ti!

En este artículo, aprenderemos:

- Que son los hilos
- Cómo crear hilos y esperar a que terminen
- Cómo usar un `ThreadPoolExecutor`

- Cómo evitar las condiciones de carrera
- Cómo utilizar las herramientas comunes que `threading` proporciona Python

Este artículo asume que tienes los conceptos básicos de Python al día y que estás usando al menos la versión 3.6 para ejecutar los ejemplos.

¿Qué es un hilo?

Un hilo es un flujo de ejecución independiente. Esto significa que un programa tendrá dos cosas sucediendo a la vez. Pero para la mayoría de las implementaciones de Python 3, los diferentes subprocesos en realidad no se ejecutan al mismo tiempo: simplemente parecen hacerlo.

Es tentador pensar que el subproceso tiene dos (o más) procesadores diferentes ejecutándose en su programa, cada uno haciendo una tarea independiente al mismo tiempo. Eso es casi correcto. Es posible que los subprocesos se estén ejecutando en diferentes procesadores, pero solo se ejecutarán de uno en uno.

Hacer que varias tareas se ejecuten simultáneamente requiere una implementación no estándar de Python, escribir parte de su código en un idioma diferente o usarlo con una sobrecarga adicional, `multiprocessing`.

Debido a la forma en que funciona la implementación de Python en CPython, es posible que el subproceso no acelere todas las tareas. Esto se debe a las interacciones con GIL que esencialmente limitan la ejecución de un hilo de Python a la vez.

Las tareas que pasan gran parte de su tiempo esperando eventos externos son generalmente buenas candidatas para el enhebrado. Es posible que los problemas que requieran un gran cálculo de la CPU y que pasen poco tiempo esperando eventos externos no se ejecuten más rápido.

Esto es cierto para el código escrito en Python y que se ejecuta en la implementación estándar de CPython. Si sus subprocesos están escritos en C, tienen la capacidad de liberar el GIL y ejecutarse al mismo tiempo. Si está ejecutando una implementación de Python diferente, consulte la documentación para ver cómo maneja los subprocesos.

Si estamos ejecutando una implementación estándar de Python, escribiendo solo en Python y tenemos un problema vinculado a la CPU, deberíamos consultar el módulo `multiprocessing` en su lugar. Lo veremos en un próximo tutorial.

La arquitectura de un programa para usar subprocesos también puede proporcionar ganancias en la claridad del diseño. La mayoría de los ejemplos que aprenderemos en este tutorial no necesariamente se ejecutarán más rápido porque usan subprocesos. El uso de hilos en ellos ayuda a que el diseño sea más limpio y fácil de razonar.

Entonces, ¡dejemos de hablar de enhebrar y comencemos a usarlo!

Comenzando un hilo

Ahora que tienes una idea de lo que es un hilo, aprendamos a hacer uno. La biblioteca estándar de Python proporciona `threading`, que contiene la mayoría de las primitivas que veremos en este artículo. `Thread`, en este módulo, encapsula muy bien los hilos, proporcionando una interfaz limpia para trabajar con ellos.

Para iniciar un hilo por separado, creamos una instancia `Thread` y luego le decimos que empiece con `.start()`:

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,))
```

```
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

Si observamos las declaraciones de logging , se puede ver que la sección `main` está creando e iniciando el hilo:

```
x = threading.Thread(target=thread_function, args=(1,))
x.start()
```

Cuando creamos un `Thread`, se le pasa una función y una lista, tupla o cualquier objeto iterable que contiene los argumentos de esa función. En este caso, le está diciendo al `Thread` que se ejecute `thread_function()` y que le pase 1 como un argumento. Le pasamos una lista que contiene un solo valor.

Para este artículo, usaremos números enteros secuenciales como nombres para los subprocesos. Hay `threading.get_ident()`, que devuelve un nombre único para cada hilo, pero estos no suelen ser cortos ni fáciles de leer.

`thread_function()` en sí mismo no hace mucho. Simplemente registra algunos mensajes con un `time.sleep()` entre ellos.

Cuando se ejecuta este programa tal como está (con la línea veinte comentada), la salida se verá así:

```
$ ./simpleThread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
Thread 1: finishing
```

Notarás que `Thread` terminó después de la sección `Main` de nuestro código. Volveremos a explicar por qué es así y se hablará sobre la misteriosa línea veinte en la siguiente sección.

Hilos de demonio

En informática, a `daemon` es un proceso que se ejecuta en segundo plano.

Python `threading` tiene un significado más específico para `daemon`. Un hilo `daemon` se cerrará inmediatamente cuando se cierre el programa. Una forma de pensar en estas definiciones es considerar el hilo `daemon` como un hilo que se ejecuta en segundo plano sin preocuparse por cerrarlo.

Si se está ejecutando un programa `Threads` que no son `daemons`, el programa esperará a que se completen esos subprocesos antes de terminar. `Threads` que son daemons, sin embargo, simplemente se matan donde sea que estén cuando el programa está saliendo.

Veamos un poco más de cerca la salida de nuestro programa anterior. Las dos últimas líneas son la parte interesante. Cuando se ejecute el programa, notaremos que hay una pausa (de aproximadamente 2 segundos) después de que se `__main__` ha impreso su mensaje de `all done` y antes de que finalice el hilo.

Esta pausa es Python esperando a que se complete el hilo no demoníaco. Cuando finaliza el programa Python, parte del proceso de cierre es limpiar la rutina de subprocesos.

Si observamos la documentación en GitHub de Python `threading`, veremos que `threading._shutdown()` recorre todos los subprocesos en ejecución y llama `.join()` a todos los que no tienen la bandera `daemon` establecida.

Entonces, nuestro programa espera salir porque el hilo en sí mismo está esperando en suspensión. Tan pronto como haya completado e impreso el mensaje, `.join()` volverá y el programa podrá salir.

Con frecuencia, este comportamiento es el que desea, pero hay otras opciones disponibles para nosotros. Primero repitamos el programa con un hilo `daemon`. Lo haces cambiando la forma en que construyes el `Thread`, agregando la bandera `daemon=True`:

```
x = threading.Thread(target=thread_function, args=(1,), daemon=True)
```
Cuando ejecute el programa ahora, debería ver este resultado:

```
$ ./daemon_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
```

La diferencia aquí es que falta la línea final de la salida. `thread_function()` no tuvo la oportunidad de completar su código. Era un hilo `daemon`, así que cuando `__main__`llegó al final de su código y el programa quiso terminar, el demonio fue asesinado.

Join() un Hilo

Los subprocesos de demonio son útiles, pero ¿qué pasa cuando quieres esperar a que se detenga un subproceso? ¿Qué pasa cuando quieres hacer eso y no salir de tu programa? Ahora volvamos a su programa original y veamos la línea veinte comentada:

```
# x.join()
```

Para decirle a un hilo que espere a que termine otro hilo, llama `.join()`. Si quita el comentario de esa línea, el hilo principal se detendrá y esperará a `x` a que termine de ejecutarse.

¿Probaste esto en el código con el hilo del demonio o el hilo normal? Resulta que no importa. Si tiene `.join()` un hilo, esa declaración esperará hasta que termine cualquier tipo de hilo.

Trabajar con muchos subprocesos

El código de ejemplo hasta ahora solo ha estado funcionando con dos subprocesos: el subproceso principal y uno que comenzó con el objeto `threading.Thread`.

Con frecuencia, querremos iniciar una serie de subprocesos y hacer que realicen un trabajo interesante. Comencemos por ver la forma más difícil de hacer eso, y luego pasaremos a un método más fácil.

La forma más difícil de iniciar varios subprocesos es la que ya conoce:

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    threads = list()
    for index in range(3):
        logging.info("Main    : create and start thread %d.", index)
        x = threading.Thread(target=thread_function, args=(index,))
        threads.append(x)
        x.start()

    for index, thread in enumerate(threads):
        logging.info("Main    : before joining thread %d.", index)
        thread.join()
        logging.info("Main    : thread %d done", index)
```

Este código utiliza el mismo mecanismo que se vio anteriormente para iniciar un hilo, crear un objeto `Thread` y luego llamar al método `.start()`. El programa mantiene una lista de objetos `Thread` para que luego pueda esperarlos más tarde con el método `.join()`.

Ejecutar este código varias veces probablemente producirá algunos resultados interesantes. Aquí hay un ejemplo de salida de mi máquina:

```
$ ./subprocesosThreads.py
Main    : create and start thread 0.
Thread 0: starting
Main    : create and start thread 1.
Thread 1: starting
Main    : create and start thread 2.
Thread 2: starting
Main    : before joining thread 0.
Thread 2: finishing
Thread 1: finishing
Thread 0: finishing
Main    : thread 0 done
Main    : before joining thread 1.
Main    : thread 1 done
Main    : before joining thread 2.
Main    : thread 2 done
```

Si recorremos la salida con cuidado, veremos que los tres subprocesos comienzan en el orden que se podría esperar, ¡pero en este caso terminan en el orden opuesto! Varias ejecuciones producirán diferentes resultados. Busque el mensaje `Thread x: finishing` que nos dice cuándo termina cada hilo.

El orden en el que se ejecutan los subprocesos lo determina el sistema operativo y puede ser bastante difícil de predecir. Puede (y probablemente variará) de una ejecución a otra, por lo que debemos tenerlo en cuenta cuando se diseñen algoritmos que utilicen subprocesos.

Afortunadamente, Python nos brinda varias primitivas que veremos más adelante para ayudar a coordinar los hilos y hacer que se ejecuten juntos. Antes de eso, veamos cómo hacer que la administración de un grupo de subprocesos sea un poco más fácil.

Usando un ThreadPoolExecutor

Hay una forma más fácil de iniciar un grupo de hilos que la que se vio arriba. Se llama a `ThreadPoolExecutor`, y es parte de la biblioteca estándar en `concurrent.futures`(a partir de Python 3.2).

La forma más sencilla de crearlo es con un administrador de contexto, utilizando la declaración `with` para gestionar la creación y destrucción del grupo.

Aquí está el `__main__`del último ejemplo reescrito para usar un `ThreadPoolExecutor`:

```
import concurrent.futures

# [rest of code]

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as
executor:
        executor.map(thread_function, range(3))
```

El código crea un administrador de contexto `ThreadPoolExecutor` indicándole cuántos subprocesos de trabajo se quieren en el grupo. Luego, usa `.map()` para recorrer una serie de cosas, en su caso `range(3)`, pasando cada una a un hilo en el grupo.

El final del bloque `with` hace que `ThreadPoolExecutor` haga un `.join()` en cada uno de los subprocesos del grupo. Se recomienda encarecidamente que se use `ThreadPoolExecutor` como administrador de contexto cuando podamos para que nunca se nos olvide el `.join()` de los hilos.

Nota: El uso de a `ThreadPoolExecutor` puede causar algunos errores confusos.

Por ejemplo, si se llama a una función que no toma parámetros, pero le pasa parámetros `.map()`, el hilo generará una excepción.

Desafortunadamente, `ThreadPoolExecutor` ocultará esa excepción y (en el caso anterior) el programa termina sin salida. Esto puede resultar bastante confuso de depurar al principio.

Ejecutar nuestro código de ejemplo corregido producirá una salida que se ve así:

```
$ ./poolExecutorThreads.py
Thread 0: starting
Thread 1: starting
Thread 2: starting
Thread 1: finishing
Thread 0: finishing
Thread 2: finishing
```

Nuevamente, observamos cómo `Thread 1` terminó antes que el `Thread 0`. La programación de subprocesos la realiza el sistema operativo y no sigue un plan que sea fácil de entender. Además dicho campo queda fuera de este tutorial y curso.

Condiciones de carrera

Antes de pasar a algunas de las otras características escondidas en Python `threading`, hablemos un poco sobre uno de los problemas más difíciles con los que nos encontraremos al escribir programas con subprocesos: las condiciones de carrera .

Una vez que haya visto qué es una condición de carrera y haya visto cómo ocurre una, pasará a algunas de las primitivas proporcionadas por la biblioteca estándar para evitar que ocurran las condiciones de carrera.

Las condiciones de carrera pueden ocurrir cuando dos o más subprocesos acceden a un dato o recurso compartido. En este ejemplo, va a crear una condición de carrera grande que ocurre todo el tiempo, pero tengamos en cuenta que la mayoría de las condiciones de carrera no son tan obvias. Con

frecuencia, solo ocurren en raras ocasiones y pueden producir resultados confusos. Como se puede imaginar, esto los hace bastante difíciles de depurar.

Afortunadamente, esta condición de carrera sucederá siempre, y la analizaremos en detalle para explicar lo que está sucediendo.

Para este ejemplo, vamos a escribir una clase que actualice una base de datos. De acuerdo, en realidad no vamos a tener una base de datos: simplemente la falsificaremos, porque ese no es el objetivo de este tutorial.

Tu `FakeDatabase` tendrá los métodos `.__init__()` y `.update()`:

```
class FakeDatabase:
    def __init__(self):
        self.value = 0

    def update(self, name):
        logging.info("Thread %s: starting update", name)
        local_copy = self.value
        local_copy += 1
        time.sleep(0.1)
        self.value = local_copy
        logging.info("Thread %s: finishing update", name)
```

`FakeDatabase` es hacer el seguimiento de un solo número: `.value`. Estos serán los datos compartidos en los que se vera la condición de carrera.

`.__init__()` simplemente se inicializa con `.value` a cero. Hasta aquí todo bien.

`.update()` parece un poco extraño. Simula la lectura de un valor de una base de datos, realiza algunos cálculos y luego escribe un nuevo valor en la base de datos.

En este caso, leer de la base de datos solo significa copiar `.value` a una variable local. El cálculo es solo agregar uno al valor y luego `.sleep()` un poco. Finalmente, vuelve a escribir el valor copiando el valor local en `.value`.

Así es como usará esto `FakeDatabase`:

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    database = FakeDatabase()
    logging.info("Testing update. Starting value is %d.",
database.value)
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as
executor:
        for index in range(2):
            executor.submit(database.update, index)
    logging.info("Testing update. Ending value is %d.",
database.value)
```

El programa crea un `ThreadPoolExecutor` con dos subprocesos y luego llama a `.submit()` con cada uno de ellos, diciéndoles que se ejecute `database.update()`.

`.submit()` tiene una firma que permite pasar argumentos posicionales y con nombre a la función que se ejecuta en el hilo:

```
.submit(function, *args, **kwargs)
```

En el uso anterior, `index` se pasa como el primer y único argumento posicional a `database.update()`. Veremos más adelante en este tutorial donde podemos pasar múltiples argumentos de manera similar.

Dado que cada hilo se ejecuta `.update()` y `.update()` agrega uno `.value`, es de esperar que `database.value` esté en 2cuando se imprima al final. Pero no estamos mirando este ejemplo si ese fuera el caso. Si ejecutamos el código anterior, la salida se ve así:

```
$ ./raceCond.py
Testing unlocked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
```

```
Testing unlocked update. Ending value is 1.
```

Es posible que hubiéramos esperado que eso sucediera, pero veamos los detalles de lo que realmente está sucediendo aquí, ya que eso hará que la solución a este problema sea más fácil de entender.

Un hilo

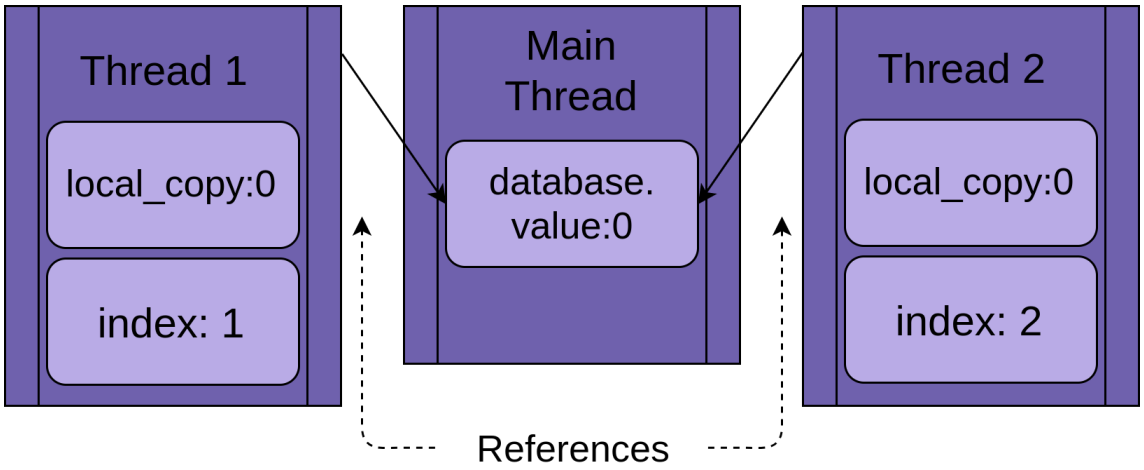
Antes de sumergirse en este problema con dos hilos, retrocedamos y hablemos un poco sobre algunos detalles de cómo funcionan los hilos.

No profundizaremos en todos los detalles aquí, ya que eso no es importante en este nivel. También simplificaremos algunas cosas de una manera que no será técnicamente precisa, pero nos dará la idea correcta de lo que está sucediendo.

Cuando le dice a su `ThreadPoolExecutor` que ejecute cada hilo, le pasamos la función a ejecutar y los parámetros que se pasan a la misma: `executor.submit(database.update, index)`.

El resultado de esto es que cada uno de los subprocesos del grupo llamará a `database.update(index)`. Tengamos en cuenta que `database` es una referencia al único objeto `FakeDatabase` creado en `__main__`. Llamar a ese objeto llama a un método `.update()` de instancia en ese objeto.

Cada hilo va a tener una referencia al mismo objeto `FakeDatabase`, `database`. Cada hilo también tendrá un valor único `index`, para que las declaraciones de registro sean un poco más fáciles de leer:

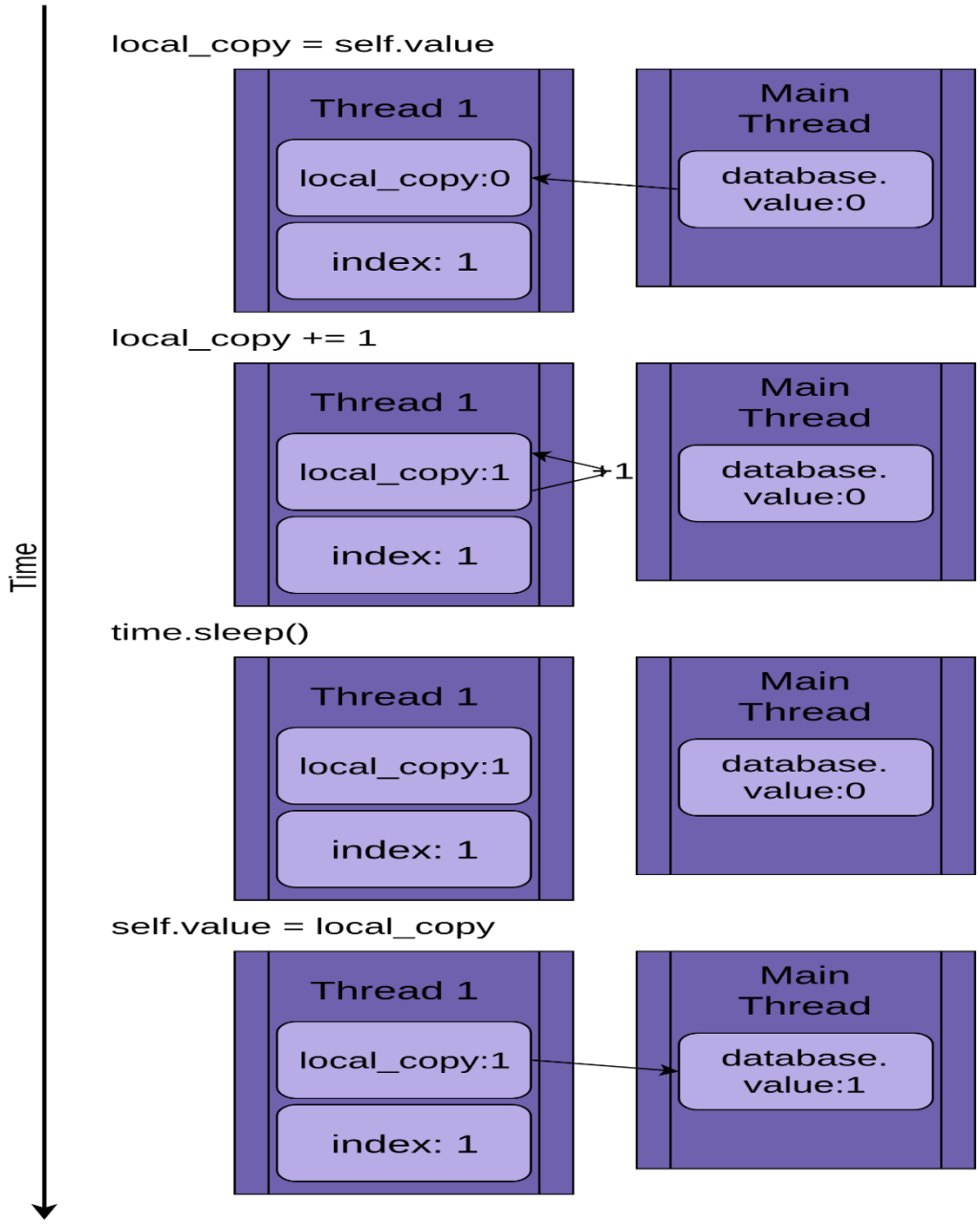


Cuando el hilo comienza a ejecutarse `.update()`, tiene su propia versión de todos los datos **locales** de la función. En el caso de `.update()`, esto

es `local_copy`. Definitivamente esto es algo bueno. De lo contrario, dos subprocesos que ejecutan la misma función siempre se confundirían entre sí. Significa que todas las variables que están dentro del ámbito (o locales) de una función son **seguras para subprocesos** .

Ahora se puede comenzar a analizar lo que sucede si ejecuta el programa anterior con un solo hilo y una sola llamada a `.update()`.

La imagen a continuación muestra un paso a través de la ejecución de `.update()`si solo se ejecuta un hilo. La declaración se muestra a la izquierda seguida de un diagrama que muestra los valores en el hilo `local_copy` y el compartido `database.value`:



El diagrama está diseñado para que el tiempo aumente a medida que avanza de arriba hacia abajo. Comienza cuando `Thread 1`se crea y finaliza cuando se termina.

Cuando `Thread 1` comienza, `FakeDatabase.value` es cero. La primera línea de código del método `local_copy = self.value`, copia el valor cero en la variable local. A continuación, incrementa el valor de `local_copy` con la `local_copy += 1` declaración. Puede ver `.value` en la `Thread 1`puesta a uno.

`time.sleep()`Se llama a Next , lo que hace que el hilo actual se detenga y permita que se ejecuten otros hilos. Dado que solo hay un hilo en este ejemplo, esto no tiene ningún efecto.

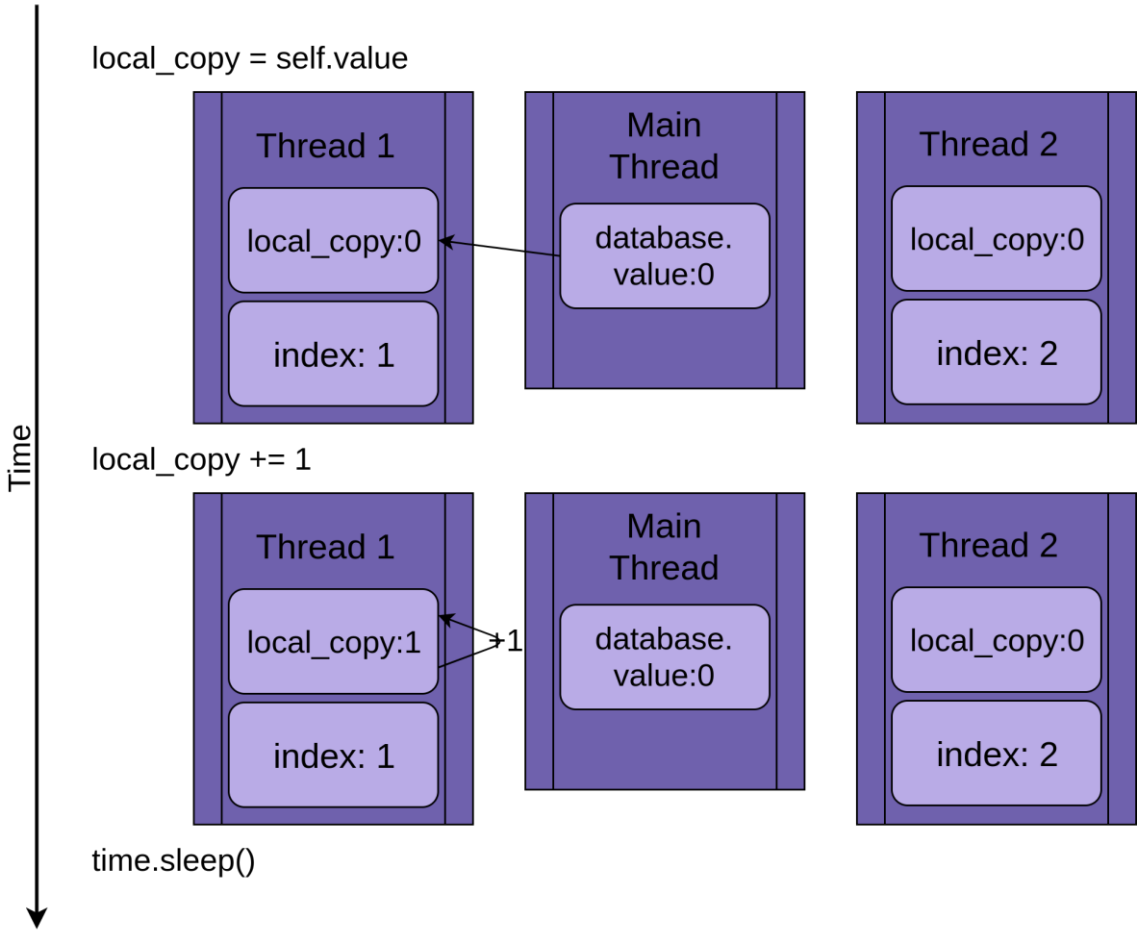
Cuando se `Thread 1`despierta y continúa, copia el nuevo valor de `local_copy`a `FakeDatabase.value`, y luego el hilo está completo. Puede ver que `database.value` está configurado en uno.

Hasta aquí todo bien. Corriste `.update()`una vez y te `FakeDatabase.value` incrementaste a uno.

Dos hilos

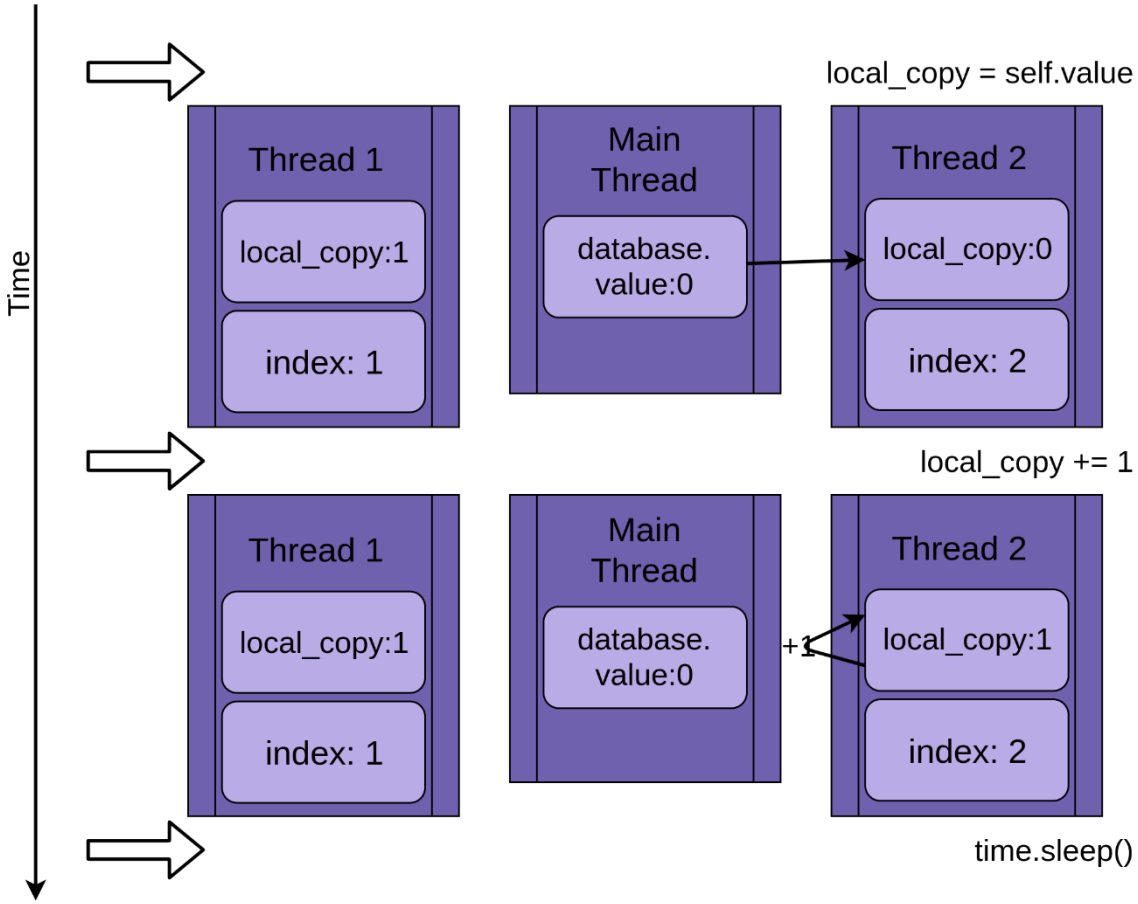
Volviendo a la condición de carrera, los dos subprocesos se ejecutarán simultáneamente, pero no al mismo tiempo. Cada uno tendrá su propia versión `local_copy` y cada uno señalará lo mismo `database`. Es este objeto `database` compartido el que va a causar problemas.

El programa comienza con la `Thread 1`ejecución `.update()`:



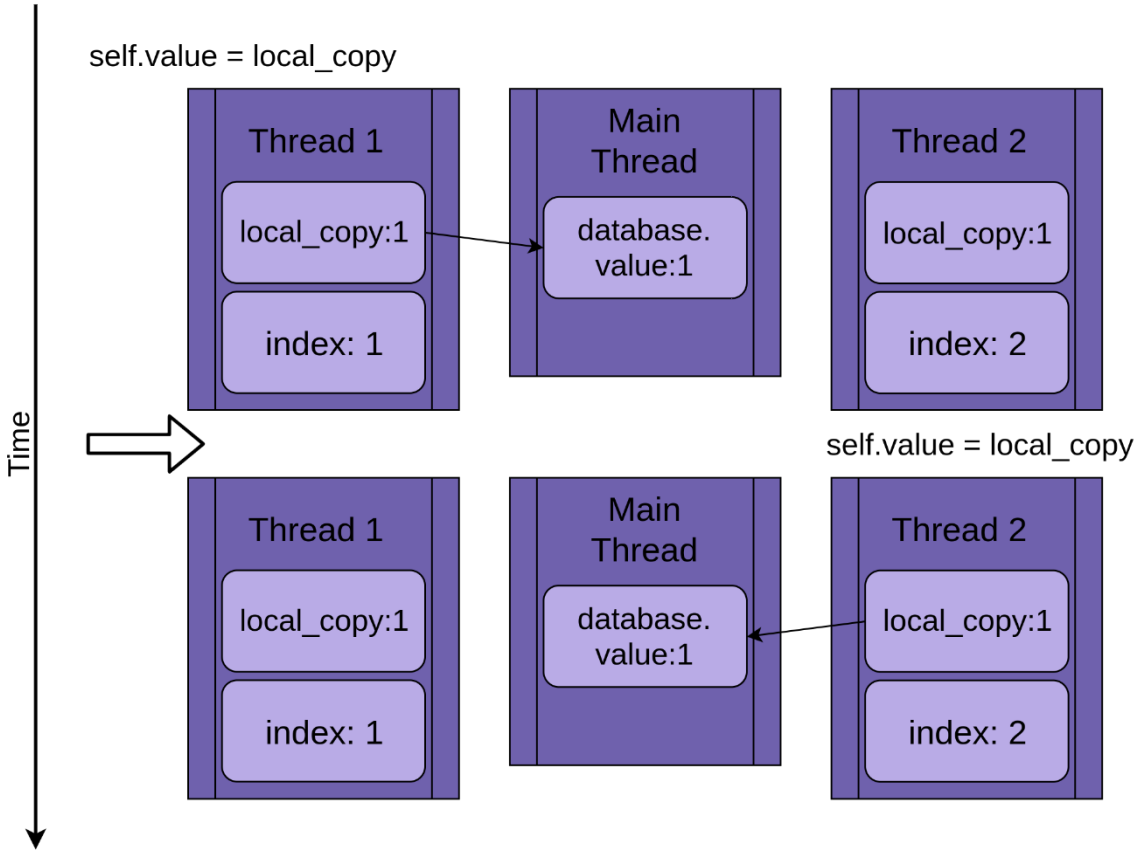
Cuando se `Thread 1`llama `time.sleep()`, permite que el otro hilo comience a ejecutarse. Aquí es donde las cosas se ponen interesantes.

`Thread 2`se inicia y realiza las mismas operaciones. También se está copiando `database.value` en privado `local_copy`, y este recurso compartido `database.value` aún no se ha actualizado:



Cuando `Thread 2` finalmente se duerme, lo compartido `database.value` sigue sin modificarse en cero, y ambas versiones privadas de `local_copy` tienen el valor uno.

`Thread 1` ahora se despierta y guarda su versión de `local_copy` y luego termina, dando `Thread 2` una última oportunidad de ejecutarse. `Thread 2` no tiene idea de que se `Thread 1` ejecutó y actualizó `database.value` mientras estaba durmiendo. Almacena *su* versión de `local_copy` en `database.value`, también configurándola en uno:



Los dos subprocesos tienen acceso entrelazado a un solo objeto compartido, sobrescribiendo los resultados de cada uno. Pueden surgir condiciones de carrera similares cuando un subproceso libera memoria o cierra un identificador de archivo antes de que el otro subproceso termine de acceder a él.

Por qué este no es un ejemplo tonto

El ejemplo anterior está diseñado para garantizar que la condición de carrera ocurra cada vez que ejecute su programa. Debido a que el sistema operativo puede intercambiar un hilo en cualquier momento, es posible interrumpir una declaración como `x = x + 1` después de haber leído el valor de `x` pero antes de haber escrito de nuevo el valor incrementado.

Los detalles de cómo sucede esto son bastante interesantes, pero no son necesarios para el resto de este artículo, así que siéntase libre de omitir esta sección oculta.

¿Cómo funciona esto realmente? Mostrar ocultar

Ahora que ha visto una condición de carrera en acción, ¡descubramos cómo resolverla!

Sincronización básica usando Lock

Hay varias formas de evitar o resolver las condiciones de carrera. No los verá todos aquí, pero hay algunos que se utilizan con frecuencia. Empecemos por `Lock`.

Para resolver su condición de carrera anterior, necesita encontrar una manera de permitir solo un hilo a la vez en la sección de lectura-modificación-escritura de su código. La forma más común de hacer esto se llama `Lock`en Python. En algunos otros idiomas, esta misma idea se llama a `mutex`. Mutex proviene de MUTual EXclusion, que es exactamente lo que `Lock` hace.

`Lock` es un objeto que actúa como un pase de pasillo. Solo un hilo a la vez puede tener la extensión `Lock`. Cualquier otro hilo que quiera el `Lock` debe esperar hasta que el propietario del `Lock` mismo lo abandone.

Las funciones básicas para hacer esto son `.acquire()` y `.release()`. Un hilo llamará `my_lock.acquire()` para obtener el bloqueo. Si el bloqueo ya está retenido, el hilo de llamada esperará hasta que se libere. Aquí hay un punto importante. Si un hilo obtiene el bloqueo pero nunca lo devuelve, su programa se bloqueará. Leerás más sobre esto más adelante.

Afortunadamente, Python `Lock` también funcionará como un administrador de contexto, por lo que puede usarlo en una `with`declaración y se libera automáticamente cuando el bloque `with` sale por cualquier motivo.

Veamos el `FakeDatabase` con un `Lock` agregado. La función de llamada permanece igual:

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()

    def locked_update(self, name):
        logging.info("Thread %s: starting update", name)
```

```
        logging.debug("Thread %s about to lock", name)
        with self._lock:
            logging.debug("Thread %s has lock", name)
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug("Thread %s about to release lock", name)
        logging.debug("Thread %s after release", name)
        logging.info("Thread %s: finishing update", name)
```

Además de agregar un montón de registros de depuración para que pueda ver el bloqueo con mayor claridad, el gran cambio aquí es agregar un miembro llamado `._lock`, que es un `threading.Lock()`objeto. Esto `._lock` se inicializa en el estado desbloqueado y bloqueado y liberado por la declaración `with`.

Vale la pena señalar aquí que el hilo que ejecuta esta función se mantendrá `Lock` hasta que termine por completo de actualizar la base de datos. En este caso, eso significa que retendrá `Lock` mientras copia, actualiza, duerme y luego escribe el valor en la base de datos.

Si ejecuta esta versión con el registro configurado en el nivel de advertencia, verá esto:

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

Mira eso. ¡Tu programa finalmente funciona!

Puede activar el registro completo configurando el nivel en `DEBUG` agregando esta declaración después de configurar la salida de registro en `__main__`:

```
logging.getLogger().setLevel(logging.DEBUG)
```

Ejecutar este programa con el `DEBUG`registro activado se ve así:

```
$ ./fixrace.py
```

```
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 0 about to lock
Thread 0 has lock
Thread 1: starting update
Thread 1 about to lock
Thread 0 about to release lock
Thread 0 after release
Thread 0: finishing update
Thread 1 has lock
Thread 1 about to release lock
Thread 1 after release
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

En esta salida se puede ver que `Thread 0`adquiere el candado y lo sigue sosteniendo cuando se va a dormir. `Thread 1` luego comienza e intenta adquirir el mismo bloqueo. Porque `Thread 0` todavía lo está sosteniendo, `Thread 1` tiene que esperar. Ésta es la exclusión mutua que `Lock` proporciona a.

Muchos de los ejemplos en el resto de este artículo tendrán `WARNING` y `DEBUG` el registro de nivel. Por lo general, solo mostraremos la `WARNING` salida del nivel, ya que los `DEBUG` registros pueden ser bastante largos. Pruebe los programas con el registro activado y vea lo que hacen.

Punto muerto

Antes de continuar, debe considerar un problema común al usar `Locks`. Como vio, si `Lock` ya se ha adquirido, una segunda llamada a `.acquire()`esperará hasta que el hilo que contiene las llamadas `Lock .release()`. ¿Qué crees que sucede cuando ejecutas este código?

```
import threading

l = threading.Lock()
print("before first acquire")
l.acquire()
print("before second acquire")
l.acquire()
print("acquired lock twice")
```

Cuando el programa llama por `l.acquire()` segunda vez, se cuelga esperando `Lock` que se libere. En este ejemplo, puede arreglar el interbloqueo eliminando la segunda llamada, pero los interbloqueos generalmente ocurren por una de dos cosas sutiles:

1. Un error de implementación en el que a `Lock`no se publica correctamente
2. Un problema de diseño en el que una función de utilidad necesita ser llamada por funciones que podrían tener o no `Lock`

La primera situación ocurre a veces, pero usar a `Lock` como administrador de contexto reduce en gran medida la frecuencia. Se recomienda escribir código siempre que sea posible para hacer uso de administradores de contexto, ya que ayudan a evitar situaciones en las que una excepción lo salte la llamada `.release()`.

El problema del diseño puede ser un poco más complicado en algunos idiomas. Afortunadamente, el subproceso de Python tiene un segundo objeto, llamado `RLock`, que está diseñado solo para esta situación. Permite que un hilo se `.acquire()` repita `RLock` varias veces antes de llamar `.release()`. Ese hilo todavía debe llamar `.release()` la misma cantidad de veces que llamó `.acquire()`, pero debería hacerlo de todos modos.

`Lock`y `RLock`son dos de las herramientas básicas que se utilizan en la programación de subprocesos para evitar condiciones de carrera. Hay algunos otros que funcionan de diferentes maneras. Antes de mirarlos, pasemos a un dominio de problemas ligeramente diferente.

Roscado productor-consumidor

El problema productor-consumidor es un problema informático estándar que se utiliza para analizar problemas de sincronización de procesos o subprocesos. Verá una variante para obtener algunas ideas de las primitivas que `threading` proporciona el módulo de Python .

Para este ejemplo, imaginará un programa que necesita leer mensajes de una red y escribirlos en el disco. El programa no solicita un mensaje cuando quiere. Debe estar escuchando y aceptar los mensajes a medida que llegan. Los mensajes no llegarán a un ritmo regular, sino que llegarán en ráfagas. Esta parte del programa se llama productor.

Por otro lado, una vez que tenga un mensaje, debe escribirlo en una base de datos. El acceso a la base de datos es lento, pero lo suficientemente rápido como para mantener el ritmo promedio de mensajes. No es *lo* suficientemente rápido para mantenerse al día cuando llega una ráfaga de mensajes. Esta parte es el consumidor.

Entre el productor y el consumidor, creará una `Pipeline` que será la parte que cambiará a medida que aprenda sobre los diferentes objetos de sincronización.

Ese es el diseño básico. Veamos una solución usando `Lock`. No funciona a la perfección, pero utiliza herramientas que ya conoce, por lo que es un buen lugar para comenzar.

Productor-consumidor que usa `Lock`

Dado que este es un artículo sobre Python `threading`, y dado que acaba de leer sobre la primitiva `Lock`, intentemos resolver este problema con dos subprocesos usando uno `Lock`o dos.

El diseño general es que hay un hilo `producer` que lee desde la red falsa y coloca el mensaje en un `Pipeline`:

```
import random

SENTINEL = object()

def producer(pipeline):
    """Pretend we're getting a message from the network."""
    for index in range(10):
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        pipeline.set_message(message, "Producer")

    # Send a sentinel message to tell consumer we're done
    pipeline.set_message(SENTINEL, "Producer")
```

Para generar un mensaje falso, `producer` obtiene un número aleatorio entre uno y cien. Pide `.set_message()` al `pipeline` para enviarlo al `consumer`.

El `producer` también utiliza un valor `SENTINEL` para indicar que el consumidor parado después de que se ha enviado diez valores. Esto es un

poco incómodo, pero no se preocupe, verá formas de deshacerse de este valor `SENTINEL` después de trabajar con este ejemplo.

En el otro lado está `pipeline` el consumidor:

```
def consumer(pipeline):
    """Pretend we're saving a number in the database."""
    message = 0
    while message is not SENTINEL:
        message = pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info("Consumer storing message: %s", message)
```

La `consumer` lee un mensaje del `pipeline` y lo escribe en una base de datos falsos, que en este caso solo se está imprimiendo a la pantalla. Si obtiene el `SENTINEL`valor, regresa de la función, que terminará el hilo.

Antes de mirar la parte realmente interesante `Pipeline`, aquí está la `__main__`sección, que genera estos hilos:

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    # logging.getLogger().setLevel(logging.DEBUG)

    pipeline = Pipeline()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline)
        executor.submit(consumer, pipeline)
```

Esto debería parecer bastante familiar ya que está cerca del `__main__`código de los ejemplos anteriores.

Recuerde que puede activar el registro `DEBUG` para ver todos los mensajes de registro descomentando esta línea:

```
# logging.getLogger().setLevel(logging.DEBUG)
```

Puede valer la pena revisar los mensajes `DEBUG` de registro para ver exactamente dónde adquiere y libera cada hilo los bloqueos.

Ahora echemos un vistazo al `Pipeline` que pasa mensajes del `producer` al `consumer`:

```
class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer.
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self, name):
        logging.debug("%s:about to acquire getlock", name)
        self.consumer_lock.acquire()
        logging.debug("%s:have getlock", name)
        message = self.message
        logging.debug("%s:about to release setlock", name)
        self.producer_lock.release()
        logging.debug("%s:setlock released", name)
        return message

    def set_message(self, message, name):
        logging.debug("%s:about to acquire setlock", name)
        self.producer_lock.acquire()
        logging.debug("%s:have setlock", name)
        self.message = message
        logging.debug("%s:about to release getlock", name)
        self.consumer_lock.release()
        logging.debug("%s:getlock released", name)
```

¡Woah! Eso es mucho código. Un porcentaje bastante alto de eso es solo declaraciones de registro para que sea más fácil ver lo que sucede cuando lo ejecuta. Aquí está el mismo código con todas las declaraciones de registro eliminadas:

```
class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer.
    """
    def __init__(self):
        self.message = 0
```

```
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self, name):
        self.consumer_lock.acquire()
        message = self.message
        self.producer_lock.release()
        return message

    def set_message(self, message, name):
        self.producer_lock.acquire()
        self.message = message
        self.consumer_lock.release()
```

Eso parece un poco más manejable. El `Pipeline` en esta versión de su código tiene tres miembros:

1. **`.message`** almacena el mensaje para pasar.
2. **`.producer_lock`** es un objeto `threading.Lock` que restringe el acceso al mensaje por parte del `producer` hilo.
3. **`.consumer_lock`** también es una `threading.Lock` que restringe el acceso al mensaje por parte del `consumer` hilo.

`__init__()`inicializa estos tres miembros y luego llama `.acquire()`al `.consumer_lock`. Este es el estado en el que desea comenzar. `Producer` le permite agregar un nuevo mensaje, pero `consumer` debe esperar hasta que haya un mensaje.

`.get_message()` y `.set_messages()`son casi opuestos. `.get_message()` llama `.acquire()` al `consumer_lock`. Esta es la llamada que hará la `consumer` espera hasta que un mensaje esté listo.

Una vez que `consumer` ha adquirido el `.consumer_lock`, copia el valor `.message` y luego llama `.release()` al `.producer_lock`. Liberar este bloqueo es lo que permite `producer` insertar el siguiente mensaje en el archivo `pipeline`.

Antes de continuar `.set_message()`, hay algo sutil `.get_message()`que es bastante fácil de pasar por alto. Puede parecer tentador deshacerse de él `message` y hacer que la función termine con `return self.message`. Vea si puede averiguar por qué no quiere hacer eso antes de continuar.

Esta es la respuesta. Tan pronto como las llamadas `consumer.producer_lock.release()`, se puede intercambiar y `producer` puede comenzar a funcionar. ¡Eso podría suceder antes de las devoluciones `.release()`! Esto significa que existe una pequeña posibilidad de que cuando la función regrese `self.message`, ese podría ser el *siguiente* mensaje generado, por lo que podría perder el primer mensaje. Este es otro ejemplo de condición de carrera.

Pasando a `.set_message()`, puede ver el lado opuesto de la transacción. Lo llamará `producer` con un mensaje. Adquirirá el `.producer_lock`, establecerá el `.message` y la llamada `.release()` en ese momento `consumer_lock`, lo que le permitirá `consumer` leer ese valor.

Ejecutemos el código que tiene el registro configurado `WARNING` y veamos cómo se ve:

```
$ ./prodcom_lock.py
Producer got data 43
Producer got data 45
Consumer storing data: 43
Producer got data 86
Consumer storing data: 45
Producer got data 40
Consumer storing data: 86
Producer got data 62
Consumer storing data: 40
Producer got data 15
Consumer storing data: 62
Producer got data 16
Consumer storing data: 15
Producer got data 61
Consumer storing data: 16
Producer got data 73
Consumer storing data: 61
Producer got data 22
Consumer storing data: 73
Consumer storing data: 22
```

Al principio, puede resultarle extraño que el productor reciba dos mensajes antes de que el consumidor incluso se ejecute. Si mira hacia atrás en `producer` y `.set_message()`, notará que el único lugar donde

esperará `Lock` es cuando intente poner el mensaje en la canalización. Esto se hace después de que `producer` recibe el mensaje y registra que lo tiene.

Cuando `producer`intente enviar este segundo mensaje, llamará por `.set_message()`segunda vez y se bloqueará.

El sistema operativo puede intercambiar subprocesos en cualquier momento, pero generalmente permite que cada subproceso tenga una cantidad de tiempo razonable para ejecutarse antes de cambiarlo. Es por eso que `producer`normalmente se ejecuta hasta que se bloquea en la segunda llamada a `.set_message()`.

Sin embargo, una vez que se bloquea un hilo, el sistema operativo siempre lo cambiará y buscará un hilo diferente para ejecutar. En este caso, el único otro hilo con algo que hacer es el `consumer`.

Las llamadas `consumer .get_message()`, que lee el mensaje y llama `.release()` al `.producer_lock`, lo que permite que `producer` se ejecute nuevamente la próxima vez que se intercambien subprocesos.

Observe que el primer mensaje fue `43`, y eso es exactamente lo que `consumer`leyó, aunque `producer`ya había generado el `45`mensaje.

Si bien funciona para esta prueba limitada, no es una gran solución para el problema del productor-consumidor en general porque solo permite un valor único en la tubería a la vez. Cuando el `producer` reciba una ráfaga de mensajes, no tendrá dónde colocarlos.

Pasemos a una mejor manera de resolver este problema, usando a `Queue`.

Productor-consumidor que usa `Queue`

Si desea poder manejar más de un valor en la canalización a la vez, necesitará una estructura de datos para la canalización que permita que el número aumente y se reduzca a medida que los datos se respaldan desde `producer`.

La biblioteca estándar de Python tiene un `queue` módulo que, a su vez, tiene una `Queue` clase. Cambiemos `Pipeline` para usar a `Queue` en lugar de solo una variable protegida por a `Lock`. También usará una forma diferente

de detener los subprocesos de trabajo mediante el uso de una primitiva diferente de Python `threading`, una `Event`.

Comencemos con el `Event`. El objeto `threading.Event` permite que un hilo emita una señal `evento` mientras que muchos otros hilos pueden estar esperando que eso suceda `event`. El uso clave en este código es que los subprocesos que están esperando el evento no necesariamente necesitan detener lo que están haciendo, solo pueden verificar el estado `Event` de vez en cuando.

El desencadenamiento del evento puede ser muchas cosas. En este ejemplo, el hilo principal simplemente dormirá por un tiempo y luego `.set()`:

```
1if __name__ == "__main__":
2    format = "%(asctime)s: %(message)s"
3    logging.basicConfig(format=format, level=logging.INFO,
4                        datefmt="%H:%M:%S")
5    # logging.getLogger().setLevel(logging.DEBUG)
6
7    pipeline = Pipeline()
8    event = threading.Event()
9    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
10        executor.submit(producer, pipeline, event)
11        executor.submit(consumer, pipeline, event)
12
13        time.sleep(0.1)
14        logging.info("Main: about to set event")
15        event.set()
```

Los únicos cambios aquí son la creación del objeto `event` en la línea 6, pasando `evento` como parámetro en las líneas 8 y 9, y la sección final en las líneas 11 a 13, que duermen por un segundo, registran un mensaje y luego llaman `.set()`al evento.

El `producer` tampoco tuvo que cambiar demasiado:

```
1def producer(pipeline, event):
2    """Pretend we're getting a number from the network."""
3    while not event.is_set():
4        message = random.randint(1, 101)
```

```
5         logging.info("Producer got message: %s", message)
6         pipeline.set_message(message, "Producer")
7
8     logging.info("Producer received EXIT event. Exiting")
```

Ahora se repetirá hasta que vea que el evento se estableció en la línea 3. Además, ya no coloca el `SENTINEL` valor en `pipeline`.

`consumer` Tuvo que cambiar un poco más:

```
1def consumer(pipeline, event):
2    """Pretend we're saving a number in the database."""
3    while not event.is_set() or not pipeline.empty():
4        message = pipeline.get_message("Consumer")
5        logging.info(
6            "Consumer storing message: %s  (queue size=%s)",
7            message,
8            pipeline.qsize(),
9        )
10
11    logging.info("Consumer received EXIT event. Exiting")
```

Si bien pudo eliminar el código relacionado con el `SENTINEL` valor, tuvo que hacer una condición `while` un poco más complicada . No solo se repite hasta que `evento` se establece, sino que también debe seguir repitiéndose hasta que `pipeline` se haya vaciado.

Asegurarse de que la cola esté vacía antes de que el consumidor termine evita otro problema divertido. Si `consumer` sale mientras `pipeline` tiene mensajes, hay dos cosas malas que pueden suceder. El primero es que pierde esos mensajes finales, pero el más serio es que `producer` pueden quedar atrapados intentando agregar un mensaje a una cola completa y nunca regresar.

Esto sucede si `evento` se activa después de que `producer` haya verificado la `.is_set()`condición pero antes de que llame `pipeline.set_message()`.

Si eso sucede, es posible que el productor se despierte y salga con la cola aún completamente llena. A continuación `producer`, llamará `.set_message()` y esperará hasta que haya espacio en la cola para el nuevo mensaje.
El `consumer` ya ha salido, por lo que esto no sucederá y `producer` no saldrá.

El resto `consumer` debería resultar familiar.

La `Pipeline` ha cambiado drásticamente, sin embargo:

```
class Pipeline(queue.Queue):
    def __init__(self):
        super().__init__(maxsize=10)

    def get_message(self, name):
        logging.debug("%s:about to get from queue", name)
        value = self.get()
        logging.debug("%s:got %d from queue", name, value)
        return value

    def set_message(self, value, name):
        logging.debug("%s:about to add %d to queue", name, value)
        self.put(value)
        logging.debug("%s:added %d to queue", name, value)
```

Puede ver que `Pipeline`es una subclase de `queue.Queue`. `Queue` tiene un parámetro opcional al inicializar para especificar un tamaño máximo de la cola.

Si da un número positivo para `maxsize`, limitará la cola a ese número de elementos, lo `.put()`que provocará que se bloqueen hasta que haya menos de `maxsize` elementos. Si no lo especifica `maxsize`, la cola crecerá hasta los límites de la memoria de su computadora.

`.get_message()` y se `.set_message()` hizo mucho más pequeño. Básicamente envuelven `.get()` y `.put()`en el `Queue`. Es posible que se pregunte dónde fue todo el código de bloqueo que evita que los subprocesos causen condiciones de carrera.

Los desarrolladores principales que escribieron la biblioteca estándar sabían que a `Queue` se usa con frecuencia en entornos de subprocesos múltiples e incorporaron todo ese código de bloqueo dentro de `Queue` sí mismo. `Queue` es seguro para subprocesos.

La ejecución de este programa se parece a lo siguiente:

```
$ ./prodcom_queue.py
Producer got message: 32
Producer got message: 51
Producer got message: 25
```

```
Producer got message: 94
Producer got message: 29
Consumer storing message: 32 (queue size=3)
Producer got message: 96
Consumer storing message: 51 (queue size=3)
Producer got message: 6
Consumer storing message: 25 (queue size=3)
Producer got message: 31

[many lines deleted]

Producer got message: 80
Consumer storing message: 94 (queue size=6)
Producer got message: 33
Consumer storing message: 20 (queue size=6)
Producer got message: 48
Consumer storing message: 31 (queue size=6)
Producer got message: 52
Consumer storing message: 98 (queue size=6)
Main: about to set event
Producer got message: 13
Consumer storing message: 59 (queue size=6)
Producer received EXIT event. Exiting
Consumer storing message: 75 (queue size=6)
Consumer storing message: 97 (queue size=5)
Consumer storing message: 80 (queue size=4)
Consumer storing message: 33 (queue size=3)
Consumer storing message: 48 (queue size=2)
Consumer storing message: 52 (queue size=1)
Consumer storing message: 13 (queue size=0)
Consumer received EXIT event. Exiting
```

Si lee el resultado de mi ejemplo, puede ver que suceden algunas cosas interesantes. Justo en la parte superior, puede ver la `producer` opción de crear cinco mensajes y colocar cuatro de ellos en la cola. El sistema operativo lo cambió antes de que pudiera colocar el quinto.

La `consumer` RAN a continuación y se quitó el primer mensaje. Imprimió ese mensaje, así como la profundidad de la cola en ese punto:

```
Consumer storing message: 32 (queue size=3)
```

Así es como sabrá que el quinto mensaje aún no ha llegado al `pipeline`. La cola se reduce al tamaño de tres después de que se eliminó un solo mensaje. También sabe que `queue` puede contener diez mensajes, por lo que el `producer` hilo no fue bloqueado por `queue`. Fue reemplazado por el sistema operativo.

Nota: su salida será diferente. Su salida cambiará de una ejecución a otra. ¡Esa es la parte divertida de trabajar con hilos!

A medida que el programa comienza a concluir, ¿puede ver el hilo principal generando el `evento` que hace que se cierre el `producer` inmediatamente? El `consumer` todavía tiene un montón de trabajo a hacer, por lo que sigue funcionando hasta que se haya limpiado el `pipeline`.

Intente jugar con diferentes tamaños de cola y llamadas a `time.sleep()` en `producer` o `consumer` para simular tiempos más largos de acceso a la red o al disco, respectivamente. Incluso pequeños cambios en estos elementos del programa harán grandes diferencias en sus resultados.

Esta es una solución mucho mejor para el problema productor-consumidor, pero puede simplificarlo aún más. La realidad `Pipeline` no es necesario para este problema. Una vez que quita el registro, simplemente se convierte en un `queue.Queue`.

Así es como se ve el código final si se usa `queue.Queue` directamente:

```
import concurrent.futures
import logging
import queue
import random
import threading
import time

def producer(queue, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        queue.put(message)

    logging.info("Producer received event. Exiting")
```

```
def consumer(queue, event):
    """Pretend we're saving a number in the database."""
    while not event.is_set() or not queue.empty():
        message = queue.get()
        logging.info(
            "Consumer storing message: %s (size=%d)", message, queue.qsize()
        )

    logging.info("Consumer received event. Exiting")

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    pipeline = queue.Queue(maxsize=10)
    event = threading.Event()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline, event)
        executor.submit(consumer, pipeline, event)

        time.sleep(0.1)
        logging.info("Main: about to set event")
        event.set()
```

Eso es más fácil de leer y muestra cómo el uso de las primitivas integradas de Python puede simplificar un problema complejo.

`Lock`y `Queue` son clases útiles para resolver problemas de concurrencia, pero hay otras proporcionadas por la biblioteca estándar. Antes de concluir este tutorial, hagamos un breve estudio de algunos de ellos.

Subprocesos de objetos

Hay algunas primitivas más ofrecidas por el módulo `threading` de Python . Si bien no los necesitaba para los ejemplos anteriores, pueden

ser útiles en diferentes casos de uso, por lo que es bueno estar familiarizado con ellos.

Semáforo

El primer objeto `threading`de Python a mirar es `threading.Semaphore`. A `Semaphore` es un mostrador con algunas propiedades especiales. El primero es que el conteo es atómico. Esto significa que hay una garantía de que el sistema operativo no intercambiará el hilo en medio del incremento o decremento del contador.

El contador interno aumenta cuando llama `.release()` y disminuye cuando llama `.acquire()`.

La siguiente propiedad especial es que si un hilo llama `.acquire()` cuando el contador es cero, ese hilo se bloqueará hasta que otro hilo llame `.release()` e incremente el contador a uno.

Los semáforos se utilizan con frecuencia para proteger un recurso que tiene una capacidad limitada. Un ejemplo sería si tiene un grupo de conexiones y desea limitar el tamaño de ese grupo a un número específico.

Temporizador

A `threading.Timer` es una forma de programar una función para que se llame después de que haya pasado una cierta cantidad de tiempo. Creas un `Timer` pasando un número de segundos para esperar y una función para llamar:

```
t = threading.Timer(30.0, my_function)
```

Empiece llamando `Timer` `.start()`. La función se llamará en un nuevo hilo en algún momento después del tiempo especificado, pero tenga en cuenta que no hay ninguna promesa de que se llamará exactamente en el momento que desee.

Si desea detener un programa `Timer` que ya ha comenzado, puede cancelarlo llamando `.cancel()`. Llamar `.cancel()`después de que se `Timer` haya disparado no hace nada y no produce una excepción.

A `Timer` se puede utilizar para pedirle a un usuario que actúe después de un período de tiempo específico. Si el usuario realiza la acción antes de que expire `Timer`, `.cancel()`se puede llamar.

Barrera

Se puede utilizar `threading.Barrier` para mantener sincronizados un número fijo de subprocesos. Al crear un `Barrier`, la persona que llama debe especificar cuántos subprocesos se sincronizarán en él. Cada hilo llama `.wait()` al `Barrier`. Todos permanecerán bloqueados hasta que el número especificado de subprocesos esté esperando, y luego se liberarán todos al mismo tiempo.

Recuerde que los subprocesos están programados por el sistema operativo, por lo que, aunque todos los subprocesos se liberan simultáneamente, se programarán para que se ejecuten de uno en uno.

Un uso de a `Barrier` es permitir que un grupo de subprocesos se inicialicen. Hacer que los subprocesos esperen después de que se inicialicen `Barrier` garantizará que ninguno de los subprocesos comience a ejecutarse antes de que todos los subprocesos terminen con su inicialización.

Conclusión: Threading en Python

Ahora ha visto mucho de lo que Python `threading` tiene para ofrecer y algunos ejemplos de cómo construir programas con subprocesos y los problemas que resuelven. También ha visto algunos casos de problemas que surgen al escribir y depurar programas con subprocesos.

Si desea explorar otras opciones de simultaneidad en Python, consulte Acelere su programa Python con simultaneidad .

Si está interesado en profundizar en el módulo `asyncio`, lea Async IO en Python: un tutorial completo .

Hagas lo que hagas, ¡ahora tienes la información y la confianza que necesitas para escribir programas usando Python Threading!