Hilos en Python

Introducción

Un hilo (también conocido como *thread*) es una ejecución de peso ligero que no tiene destinado su propio espacio en memoria; en otras palabras, es la secuencia más pequeña de instrucciones programadas que puede ser manejada independientemente por el planificador del sistema operativo.

La programación concurrente lidia justamente con el manejo de estos hilos. Como el orden en el que estos se ejecutan no es determinista, por lo que es importante que existan los mecanismos adecuados para administrarlos. Sin embargo, esto no es una tarea sencilla, ya que el hecho de lidiar con procesos en paralelo introduce un sinnúmero de posibilidades de error, entre las cuales se incluyen condiciones de carrera, desincronizaciones, y el llamado *deadlocking*, o bloqueo mutuo. A pesar de todo, lidiar con todos estos problemas ciertamente vale la pena, ya que existen un sinfín de tareas que se benefician de la concurrencia.

Python es uno de los lenguajes de programación más populares del mundo; por lo tanto, el aprender cómo escribir programas que puedan aprovechar los beneficios de la programación concurrente es vital. En este trabajo cubriremos este tema, con un especial enfoque en las distintas implementaciones y propiedades de los hilos.

Implementaciones básicas en Python

La implementación de hilos en Python recae en los módulos **`Thread`**, **`Multiprocessing`** y **`Threading`**. **`Thread`** es un módulo que ofrece primitivas de bajo nivel para trabajar con múltiples hilos, también contiene candados simples (mutexes y semáforos) que facilitan la sincronización entre los hilos. Este módulo es un poco complicado de usar a priori ya que está pensado para programar en un bajo nivel, mientras que **`Threading`** es un módulo pensado para programar a alto nivel hecho con el módulo **`Thread`**.

El módulo **`Threading`** está basado en el funcionamiento de hilos de Java. Sin embargo cuenta con algunas diferencias, ya que Python realiza la separación de objetos en los que Java realiza locks y variables condicionales. En la clase **`Thread`** que proporciona **`Threading`** no hay prioridades ni grupos de hilos, ni se pueden alterar el estado de los hilos (destruidos, detenidos, suspendidos, retomados o interrumpidos).

La clase **`Thread`** representa una tarea que se está ejecutando en un hilo separado del resto. Para especificar la tarea que deseamos hacer, se realiza cualesquiera de estas dos formas:

1. Pasando un objeto invocable (callable) al constructor
2. Hacer una clase que herede de Thread y sobrescribir el método run

Cuando un **`Thread`** se crea se debe activar llamar al método start para que el hilo llame al método run y comience a hacer la actividad por la cual fue activado. Una vez que el hilo llama al método run se le considerara “vivo” , y seguirá en ese estado hasta que termine el método run o se encuentre con una excepción, se puede consultar si el hilo está vivo mediante el método **`is_alive`**. Los hilos poseen un método llamado **`join`** que al ser llamado bloquea al hilo que lo invocó hasta que el hilo al que llamaron termine dicho método.

Un hilo puede ser marcado como un «hilo demonio». El significado de esta marca es que la totalidad del programa de Python finalizará cuando solo queden hilos demonio. El valor inicial es heredado del hilo creador. Existe un hilo principal que corresponde al hilo controlador del programa de Python, este hilo no puede ser demonio

Ejemplo

Este código crea 5 hilos que ejecutan la función **`Trabajador`** e imprimen ‘Soy el trabajador’. Esto se logra creando un objeto **`Thread`** en **`t`**, y posteriormente iniciándolo con **`t.start`**.

```
import threading
def Trabajador():
   """Hilo de la función Trabajador"""
   print('Soy el trabajador')
   return
hilos = []
for i in range(5):
   t = threading.Thread(target=Trabajador)
   hilos.append(t)
   t.start()
```

Restricciones

El mayor obstáculo a la hora de implementar programas concurrentes en Python es el llamado *Global Interpreter Lock* (GIL). Esta es una restricción que otorga todo el control del interpretador de Python a un único hilo, lo cual significa que solo un hilo puede estar ejecutándose en cualquier punto del tiempo.

El GIL existe debido a que Python administra la memoria mediante *conteo de referencias.* Esto significa que cada objeto en Python lleva un conteo interno de cuántas referencias apuntan hacia él; cuando este número llega a cero, Python sabe que es seguro limpiar la memoria ocupada por el objeto.

El problema con este método surge cuando hay más de un hilo ejecutando el mismo código. En este caso, ambos hilos pueden intentar operar sobre un objeto al mismo tiempo, lo cual puede dar lugar a *condiciones de carrera*. Si esto ocurre, puede originarse una fuga de memoria, o peor aún, puede borrarse el objeto aún cuando haya referencias hacia él.

En principio, esto puede solucionarse poniéndole un *lock* (candado) a cada uno de los objetos que son compartidos entre hilos; empero, esto ocasiona un nuevo problema: *deadlocks* (bloqueo mutuo). Si hay múltiples *locks* presentes, el lock A puede estar esperando a que el lock B se libere, y el lock B puede estar esperando a que el lock A lo haga, suspendiendo la ejecución indefinidamente. Adicionalmente, el poner y quitar *locks* varias veces para cada objeto es computacionalmente costoso, y añade al tiempo de ejecución.

La solución a todo esto fue el GIL: un único *lock* que impide que cualquier variable sea accesada por más de un hilo a la vez. Por un lado, esta es una excelente solución, ya que remueve todos los dolores de cabeza que conlleva administrar múltiples hilos, y permite que librerías enteras de C puedan ser importadas a Python sin tener que reescribirlas.

Sin embargo, el GIL también introdujo una gigantesca restricción a los programas de Python: si su tiempo de ejecución depende de la capacidad del CPU, no se verán beneficiados si los ejecutamos de manera concurrente. En algunos casos, el tiempo de ejecución concurrente *es mayor* que el tiempo con un solo hilo.

Es por esto que el GIL es un tema tan controversial en la comunidad de Python. Y si bien a estas alturas es esencialmente imposible removerlo de manera global sin causar un millar de problemas de compatibilidad, hay muchas implementaciones que lo desactivan (tales como Gilectomy), o librerías, algunas de las cuales fueron mencionadas anteriormente.

Conclusión

Python no es el mejor lenguaje para realizar cómputo concurrente. Aún con librerías que permitan ignorar el GIL, debido a su alto nivel y énfasis en la compatibilidad y extensibilidad, las implementaciones suelen ser más lentas que sus equivalentes en C. Aun así, como Python es utilizado en prácticamente todas las áreas de la ciencia e investigación, es necesario aprender tipo de métodos.

Referencias

threading — Paralelismo basado en hilos

_thread — API de bajo nivel para manejo de hilos

Understanding the Linux 2.6.8.1 CPU Scheduler

Introducción a concurrencia