Práctica 4

Gestión de hilos para poder trabajar concurrentemente

Grado en Ingeniería en Robótica Software
GSyC, Universidad Rey Juan Carlos

1. Introducción

Hasta ahora hemos aprendido que en un programa está el `main`, que es una función especial que se invoca al ejecutar el programa. Y es especial porque se tiene que llamar así, y no es por casualidad, es porque el sistema operativo, cuando busca en el ejecutable ha de empezar por algún sitio, y por facilitarle la vida, se establece la función (etiqueta del ejecutable) `main`, y el SO buscará siempre dicha etiqueta. El mismo motivo se da en programación orientada a objeto con el nombre de un constructor en una clase; no es casualidad que deba llamarse igual que la clase... Pero esto es otro cantar.

En resumen, en programación secuencial solo hay un hilo de ejecución, cuya entrada es el `main`. A partir de aquí, las instrucciones se suceden una tras otra. Ahora vamos a ver que, a partir de la entrada inicial con *main*, podemos —entre otras cosas— lanzar otros hilos (otros *mains*), que se bifurcarán del principal u origen de la creación. A este proceso se le conoce como *fork*, por su parecido gráfico con un tenedor.

2. La gestión de hilos en robótica

Gestionar hilos o *threads* es una labor prácticamente inherente a la robótica. Lanzar varios threads supone tener varios hilos de ejecución simultáneamente, y esto en robótica es crucial, pues hemos de estar *pendientes* de varias *cosas* al mismo tiempo.

Nosotros, los humanos, tenemos los sentidos y los músculos que están funcionando al mismo tiempo; los primeros para percibir el mundo que nos rodea, y los segundos, para actuar en consecuencia según las órdenes de nuestra *CPU* en base a la información dada por los primeros. Y todo ello transcurre al mismo tiempo. Es de entender que, en robótica, resulta crucial perseguir el mismo objetivo para con los sensores y los actuadores.

Es por ello que en esta prática vamos a dar unas pinceladas al tema de la concurrencia mediante la gestión de hilos, si bien es un tema bastante más amplio que suele tener entidad como una asignatura propia bajo el paradigma de la *programación concurrente*.

3. Gestión de hilos en Python

Desde `Python 2.4` la librería que se emplea comúnmente para la gestión de hilos es `threading`. Aunque existe otra que ya se usaba en versiones anteriores, `thread`, que aún hoy se mantiene bajo el nombre de `\_thread` para facilitar la compatibilidad con las versiones anteriores de Python, pero que hoy en día está *deprecated*.

En cualquier caso, los conceptos de hilos son los mismos, se emplee una librería u otra, un lenguaje u otro... A continuación, comenzaremos viendo cómo se crea un thread.

3.1. El holamundo de la gestión de hilos

En este primer ejemplo, cuyo código se ha facilitado en `threads1.py`, vemos cómo se lanza un hilo de ejecución de la forma más sencilla posible. Veamos el código completo:

```
import threading
import time

def tarea():
  time.sleep (1) # simula thread ocupado en gran carga de trabajo
  print ("Tarea del nuevo hilo hecha")

hilo1 = threading.Thread(target=tarea)
hilo1.start()

print ("Hilo principal sigue su curso y termina")
```

Concretamente, fijémonos que, para crear un hilo, basta con la línea:

```
hilo1 = threading.Thread(target=tarea)
```

Con esta línea, decimos que `hilo1` es de la clase `Thread`, vamos, que es un thread. Y, antes de nada, le asignamos la tarea que tendrá que realizar, que es —simplemente— una función a realizar por este hilo.

Y a continuación, lanzamos el hilo (aunque podríamos lanzarlo más adelante cuando deseásemos) con la instrucción:

```
hilo1.start()
```

3.2. Uso del join para esperar a que termine un hilo

En este ejemplo vamos a crear dos threads, y en lugar de que el hilo principal siga su curso sin tener en cuenta la tarea del hilo lanzado, como ocurría con el anterior ejemplo, hacemos un `join()` de cada uno de ellos para así esperar a que terminen su tarea, y a continuación el hilo principal sigue su curso.

El código de este ejemplo se ha facilitado en `threads2.py`, y podemos ver un *snippet* del mismo a continuación:

```
hilo1 = threading.Thread(target=tarea, args=("hilo1",))
hilo1.start()

hilo2 = threading.Thread(target=tarea, args=("hilo2",))
hilo2.start()

hilo1.join() # esperamos a que hilo1 termine
hilo2.join() # idem con hilo2

print ("Hilo principal sigue su curso y termina")
```

En este ejemplo resulta muy interesante modificar el tiempo de *descanso* que se toma cada hilo. ¿Qué comportamiento observas si lo modificas?

3.3. Sincronización

La gestión de hilos se vuelve impredecible; esto es, no sabemos cuándo va a ejecutar qué ni quién. Para solucionar esto, conviene tener control sobre cómo se están entrelazando los hilos en su ejecución. Este hecho se conoce como sincronización de hilos. Y el concepto universal para conseguir este mecanismo de sincronización es mediante un testigo único o cerrojo o bloqueo o lock. La clase de la librería `threading` que nos facilita los recursos para ello se llama `Lock`.

Así, si queremos que una cierta zona del código, o directamente una función entera sea de uso exclusivo de un thread mientras este ejecuta todo ese trozo, y que no entre ningún otro durante la ejecución del primero, el hilo que llega a esa zona de exclusión mutua (o también denominada *mutex*) adquiere el testigo al entrar (*acquire*), y no lo suelta hasta que sale (*release*). Y asunto arreglado.

Veamos un *snippet* del código facilitado en `threads3.py` donde hacemos uso de este recurso de `Lock` para sincronizar los dos threads que tenemos.

```
testigo.acquire() # LOCK - inicio zona de exclusión mutua (mutex)
contador = 0
while contador < 5:
  time.sleep(0.3)
  print (nombre + " ejecutando iteracion n.º " + str(contador))
  contador += 1
testigo.release() # UNLOCK - fin de zona de exclusión mutua (mutex)
```

¿Qué ocurre ahora al ejecutar el programa? Obsérvalo. Además, hemos añadido un vector donde almacenar los threads y así tenerlos mejor controlados. Resulta muy práctico, sobre todo cuando tengamos muchos hilos, para evitar que se escapen de nuestro control.

4. Ejercicio

Con estos pequeños pero útiles conocimientos básicos sobre threads, vuelve a la práctica anterior y resuelve los ejercicios, aplicando convenientemente los disintos recursos según convenga.