HILOS EN PYTHON

COMENZANDO CON HILOS

La biblioteca estándar de Python proporciona ***threading***, que contiene la mayoría de las primitivas con las que trabajaremos. Thread, en este módulo, encapsula muy bien los hilos, proporcionando una interfaz limpia para trabajar con ellos.

import threading

COMENZANDO CON HILOS

Para crear un nuevo hilo, creamos un objeto de clase Thread:

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

Donde *target* es la función a ser ejecutada por el hilo y *args* son los argumentos a ser pasados a la función.

Para iniciar el hilo debemos usar el método start():

x.start()

COMENZANDO CON HILOS

EJEMPLO: “01_Starting_a_Thread.py”

DAEMON THREADS

Cómo sabemos, un demonio es un proceso que se ejecuta en segundo plano. El módulo *threading* tiene un significado más específico para *daemon*. Un *thread daemon* se cerrará inmediatamente cuando finalice el programa. Una forma de pensar en estas definiciones es considerar el thread daemon como un hilo que se ejecuta en segundo plano sin preocuparse por apagarlo.
Si un programa ejecuta hilos que no son demonios, el programa esperará a que se completen esos hilos antes de finalizar. Los hilos que son demonios, sin embargo, simplemente se eliminan dondequiera que estén cuando el programa se está cerrando.

DAEMON THREADS

Miremos un poco más de cerca la salida de su programa anterior. Las dos últimas líneas son la parte interesante. Cuando ejecute el programa, notará que hay una pausa (de aproximadamente 2 segundos) después de que __*main*__ haya impreso su mensaje de finalización y antes de que finalice el hilo.
Esta pausa es Python esperando que se complete el hilo no demoníaco. Cuando finaliza su programa de Python, parte del proceso de apagado es limpiar la rutina de subprocesamiento.

DAEMON THREADS

Si observa el código fuente de *threading*, verá que *threading._shutdown()* recorre todos los hilos en ejecución y llama a *.join()* en todos los que no tienen el indicador de demonio establecido.
Entonces su programa espera para salir porque el hilo en sí está esperando en un estado de suspensión. Tan pronto como haya completado e impreso el mensaje, *.join()* regresará y el programa podrá finalizar.

»»»» threading.py

DAEMON THREADS

Con frecuencia, este comportamiento es lo que se desea, pero hay otras opciones disponibles. Primero repitamos el programa con un thread daemon. Lo haces cambiando la forma en que construyes el hilo, agregando el indicador *daemon=True*:

x = threading.Thread(target=thread_function, args=(1,), daemon=True)

DAEMON THREADS

La diferencia aquí es que falta la línea final de la salida, *thread_function()* no tuvo la oportunidad de completarse. Era un thread daemon, así que cuando *__main__* llegó al final de su código y el programa quería terminar, el daemon fue eliminado.

EJEMPLO: “02_Daemon_Threads.py”

JOIN()

Los threads daemon son útiles, pero ¿qué pasa cuando quieres esperar a que se detenga un hilo? ¿Qué pasa cuando quieres hacer eso y no salir de tu programa?
Para decirle a un hilo que espere a que finalice otro hilo, llame a *.join()*. En el siguiente ejemplo, el hilo principal se detendrá y esperará a que el hilo x termine de ejecutarse.

EJEMPLO: “03_Join_Thread.py”

TRABAJAR CON MUCHOS THREADS

El código de ejemplo hasta ahora solo ha estado funcionando con dos hilos: el hilo principal y uno que comenzaste con el objeto *threading.Thread*.
Con frecuencia, querrás iniciar una serie de hilos y hacer que hagan un trabajo interesante. Comencemos mirando la forma más difícil de hacerlo, y luego pasará a un método más fácil.
La forma más difícil de iniciar varios hilos es la que ya conoce. El ejemplo usa el mismo mecanismo que los ejemplos anteriores para iniciar un hilo, crear un objeto *Thread* y luego llamar a *.start()*. El programa mantiene una lista de objetos *Thread* para poder esperarlos más tarde usando *.join()*.

TRABAJAR CON MUCHOS THREADS

Ejecutar este código varias veces probablemente producirá algunos resultados interesantes. El sistema operativo determina el orden en que se ejecutan los hilos y puede ser bastante difícil de predecir. Puede (y es probable que lo haga) variar de una ejecución a otra, por lo que debe tenerlo en cuenta cuando diseñe algoritmos que utilicen hilos.

EJEMPLO: “04_Working_With_Many_Threads.py”

THREADPOOLEXECUTOR

Hay una manera más fácil de iniciar un grupo de threads, se llama *ThreadPoolExecutor* y es parte de la biblioteca estándar en *concurrent.futures*.
La forma más fácil de crearlo es como administrador de contexto, utilizando la instrucción *with* para administrar la creación y destrucción del grupo.

EJEMPLO “05_ThreadPoolExecutor.py”

El código crea un *ThreadPoolExecutor* como administrador de contexto, diciéndole cuántos hilos de trabajo quiere en el grupo. Luego usa *.map()* para pasar a través de una iteración de cosas, en su caso *range(3)*, pasando cada una a un hilo en el grupo.
El final del bloque *with* hace que *ThreadPoolExecutor* realice un *.join()* en cada uno de los hilos del grupo.

CONDICIONES DE CARRERA

Las condiciones de carrera pueden ocurrir cuando dos o más hilos acceden a un dato o recurso compartido.
En este ejemplo, vamos a describir una clase que actualice una base de datos. La *BaseDatos* tendrá los métodos *.__init__()* y *.update()*:

EJEMPLO “06_Race_Conditions.py”

CONDICIONES DE CARRERA

BaseDatos realiza un seguimiento de un solo número: *.value*. Estos serán los datos compartidos en los que verá la condición de carrera.

- *.__init__()* simplemente inicializa *.value* a cero.
- *.update()* está simulando la lectura de un valor de una base de datos, haciendo algunos cálculos en él y luego escribiendo 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()* por un rato. Finalmente, vuelve a escribir el valor copiando el valor local nuevamente a *.value*.

CONDICIONES DE CARRERA

El programa crea un *ThreadPoolExecutor* con dos hilos y luego llama a *.submit()* en cada uno de ellos, diciéndoles que ejecuten *database.update()*.

.submit() permite pasar argumentos tanto posicionales como con nombre a la función que se ejecuta en el hilo:

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

En el uso anterior, el índice se pasa como el primer y único argumento posicional a *database.update()*.

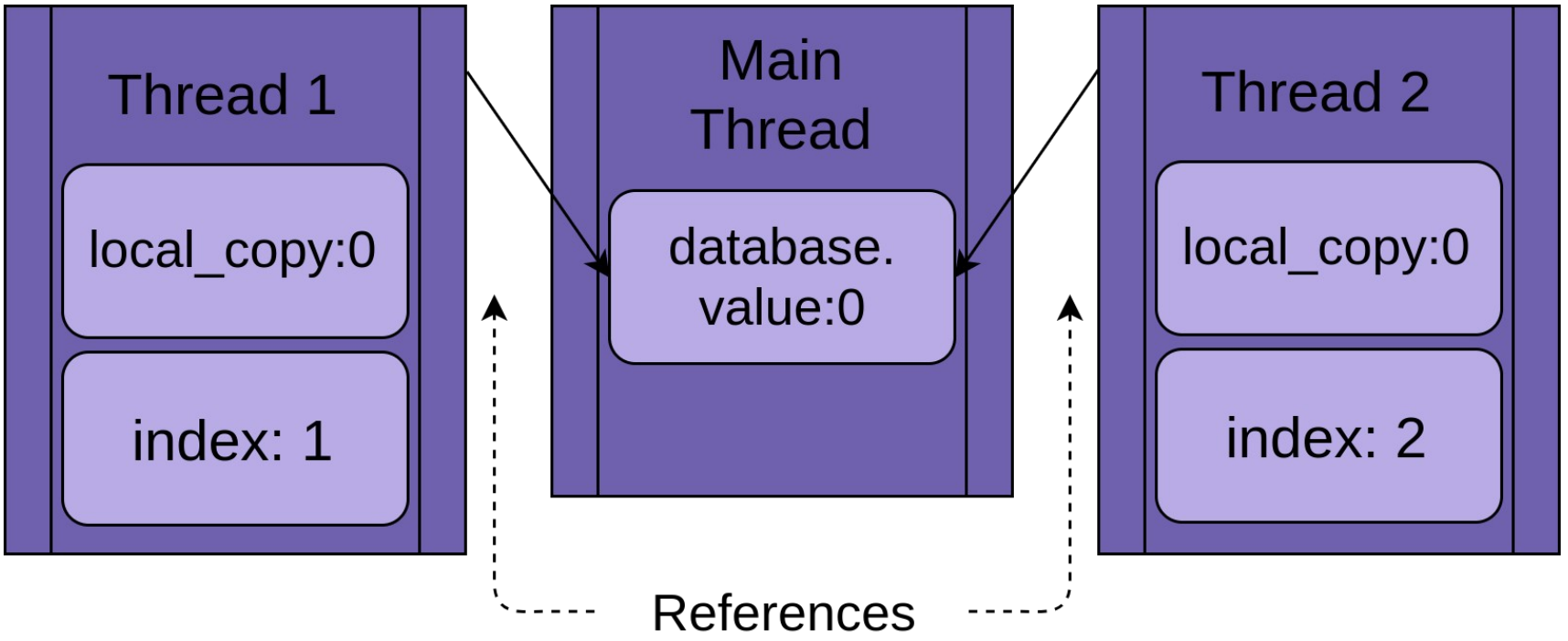
CONDICIONES DE CARRERA

Dado que cada hilo ejecuta *.update()*, y *.update()* agrega uno a *.value*, puede esperar que *database.value* sea 2 cuando se imprima al final.

FUNCIONAMIENTO CON UN HILO

Cuando le dice a su *ThreadPoolExecutor* que ejecute cada hilo, le dice qué función ejecutar y qué parámetros pasarle: *executor.submit(database.update, index)*.
El resultado de esto es que cada uno de los hilos del grupo llamará a *database.update(index)*. Tenga en cuenta que la base de datos es una referencia al objeto *BaseDatos* creado en *__main__*. Llamar a *.update()* en ese objeto llama a un método de instancia en ese objeto.
Cada hilo tendrá una referencia al mismo objeto *BaseDatos*, *database*. Cada hilo también tendrá un valor único, *index*, para que las declaraciones de registro sean un poco más fáciles de leer.

FUNCIONAMIENTO CON UN HILO



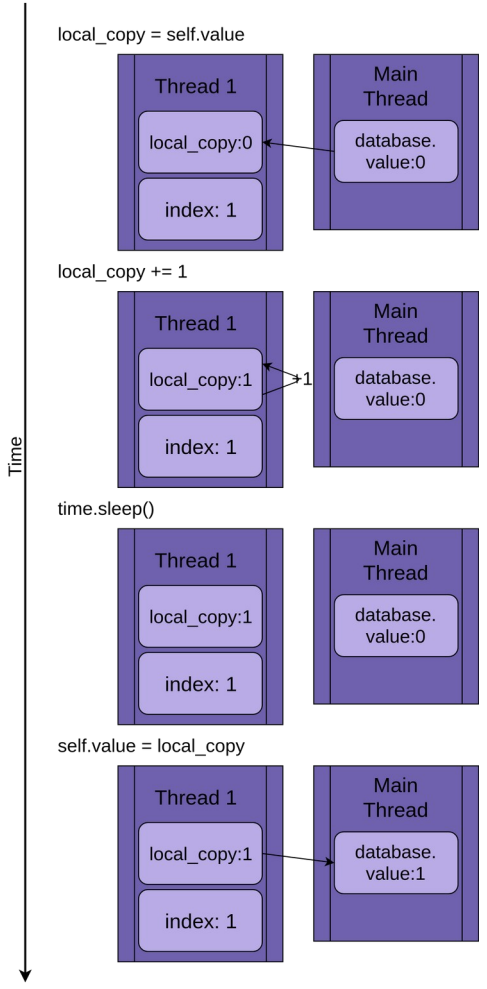
FUNCIONAMIENTO CON UN HILO

Cuando el hilo comienza a ejecutar *.update()*, tiene su propia versión de todos los datos locales de la función. En el caso de *.update()*, esto es *local_copy*. Esto es definitivamente algo bueno, de lo contrario, dos hilos que ejecutan la misma función siempre se confundirían entre sí. Significa que todas las variables que están en el ámbito (o locales) de una función son seguras para los hilos.

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

La siguiente imagen muestra la ejecución de *.update()* si solo se ejecuta un único hilo. La declaración se muestra a la izquierda seguida de un diagrama que muestra los valores en *local_copy* del hilo y el valor de la base de datos compartida.

FUNCIONAMIENTO CON UN HILO



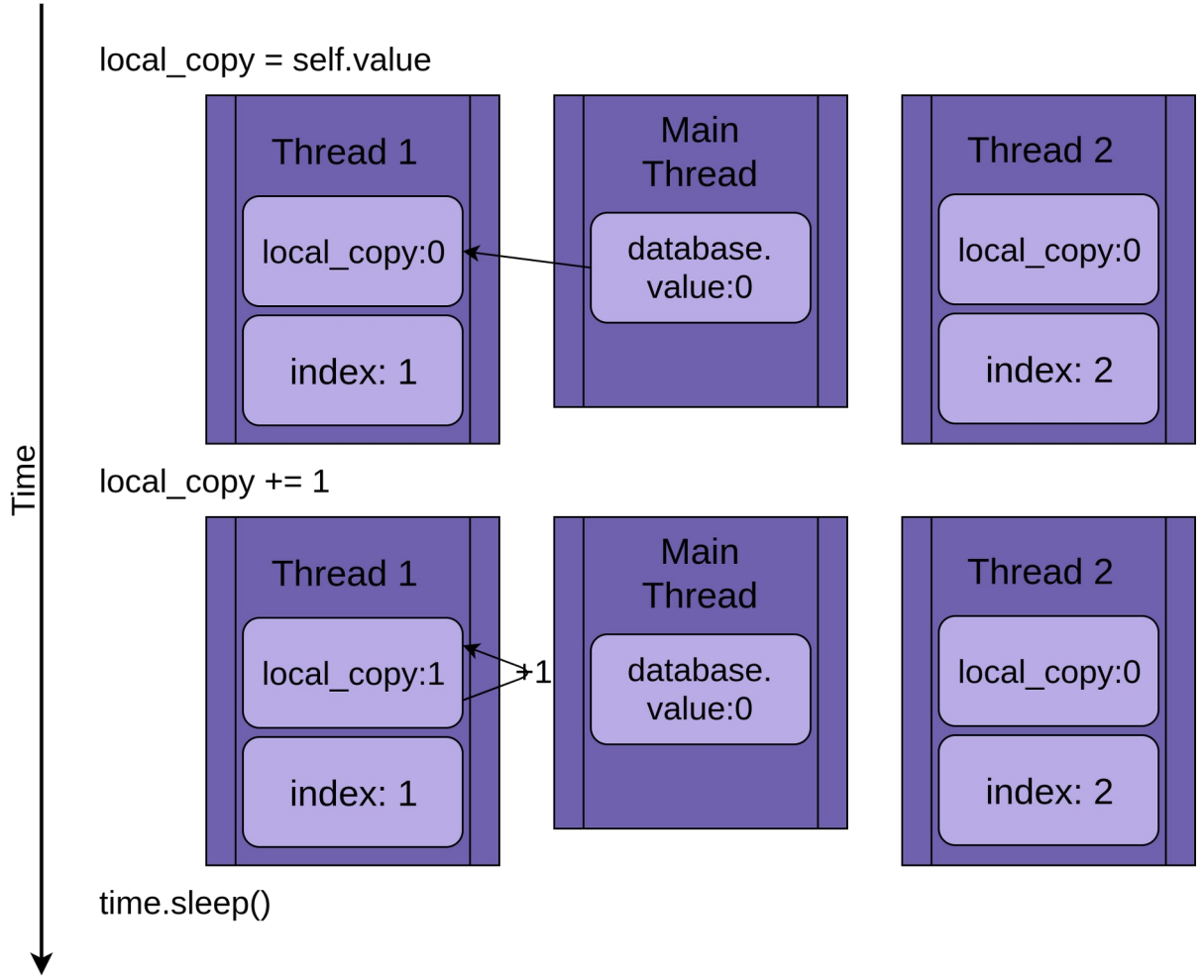
FUNCIONAMIENTO CON UN HILO

Cuando se inicia el Thread 1, *BaseDatos.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 instrucción *local_copy += 1*. Puede ver que *.value* en Thread 1 se establece en uno.
Se llama a *time.sleep()*, lo que hace que el hilo actual se detenga y permite que se ejecuten otros hilos. Como solo hay un hilo en este ejemplo, esto no tiene ningún efecto.
Cuando el Thread 1 se activa y continúa, copia el nuevo valor de *local_copy* a *BaseDatos.value* y luego se completa el hilo. Puede ver que base de *database.value* está establecido en uno.

FUNCIONAMIENTO CON DOS HILOS

Volviendo a la condición de carrera, los dos hilos se ejecutarán simultáneamente pero no al mismo tiempo. Cada uno tendrá su propia versión de *local_copy* y cada uno apuntará a la misma base de datos. Es este objeto de base de datos compartido el que causará los problemas.
El programa comienza con Thread 1 ejecutando *.update(),* y cuando el Thread 1 llama a *time.sleep()*, permite que el otro hilo comience a ejecutarse.

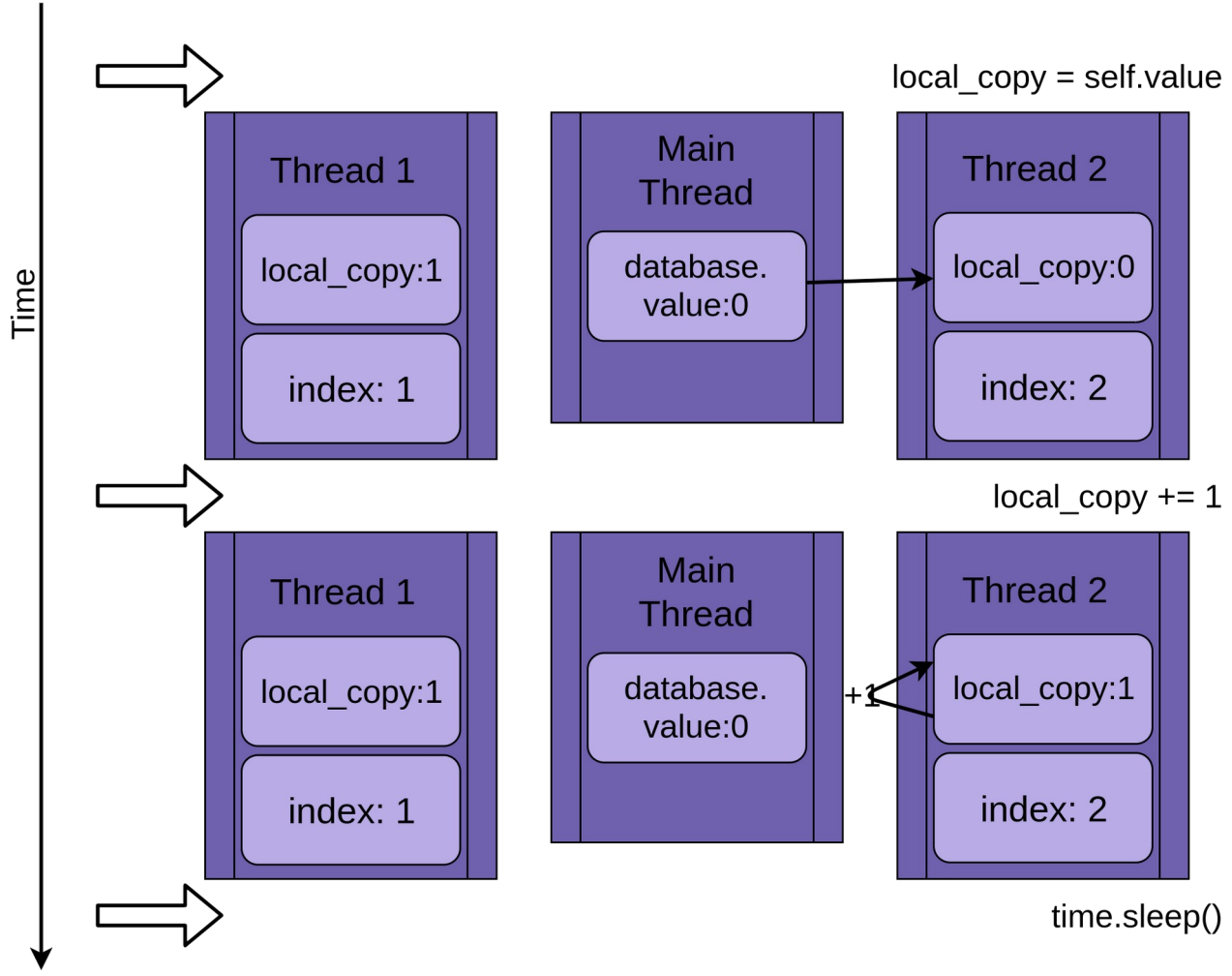
FUNCIONAMIENTO CON DOS HILOS



FUNCIONAMIENTO CON DOS HILOS

El Thread 2 se inicia y realiza las mismas operaciones. También está copiando *database.value* en su *local_copy* privada, y esta *database.value* compartida aún no se ha actualizado.

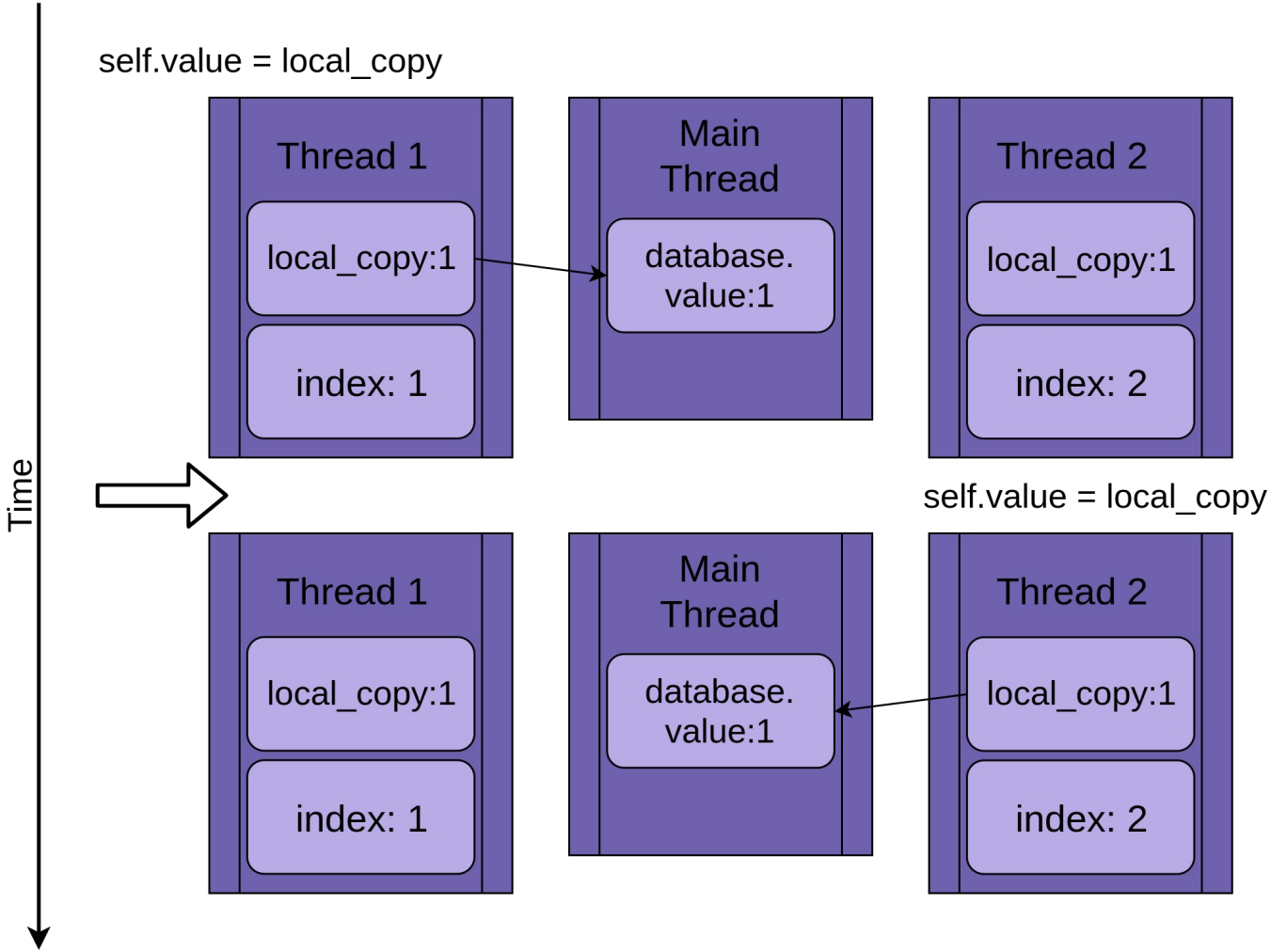
FUNCIONAMIENTO CON DOS HILOS



FUNCIONAMIENTO CON DOS HILOS

Cuando el Thread 2 finalmente entra en suspensión, *database.value* aún no se modifica en cero, y ambas versiones privadas de *local_copy* tienen el valor uno.
El Thread 1 ahora se activa y guarda su versión de *local_copy* y luego finaliza, lo que le da al Thread 2 una última oportunidad de ejecutarse. El Thread 2 no tiene idea de que el Thread 1 se ejecutó y actualizó *database.value* mientras estaba inactivo. Almacena su versión de *local_copy* en *database.value*, y también lo establece en uno.

FUNCIONAMIENTO CON DOS HILOS



FUNCIONAMIENTO CON DOS HILOS

Los dos hilos tienen acceso intercalado a un solo objeto compartido, sobrescribiendo los resultados del otro. Pueden surgir condiciones de carrera similares cuando un hilo libera memoria o cierra un identificador de archivo antes de que el otro hilo termine de acceder a él.

LOCK

Hay varias formas de evitar o resolver las condiciones de carrera. Para resolver la condición de carrera anterior, se 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 otros lenguajes, esta misma idea se llama **mutex (MUTual EXclusion)**.
Un lock (o cerradura) es un objeto que actúa como un control de acceso. Solo un hilo a la vez puede tener el bloqueo. Cualquier otro hilo que desee el bloqueo debe esperar hasta que el propietario del bloqueo lo abandone.

LOCK

Las funciones básicas para hacer esto son *.acquire()* y *.release()*. Un hilo llamará a *my_lock.acquire()* para obtener el bloqueo. Si el bloqueo ya está retenido, el hilo de llamada esperará hasta que se libere. Hay un punto importante aquí: si un hilo obtiene el bloqueo pero nunca lo devuelve, su programa se bloqueará.

LOCK

Afortunadamente, los Locks de Python funcionan como un administrador de contexto, por lo que puede usarlo en una declaración *with*, y se libera automáticamente cuando el bloque *with* sale por cualquier motivo.

EJEMPLO “07_Basic_Synchronization_Using_Lock.py”

LOCK

Además de agregar un montón de registros de depuración para que pueda ver el bloqueo más claramente, el gran cambio aquí es agregar un miembro llamado *._lock*, que es un objeto *threading.Lock()*. Este *._lock* se inicializa en el estado desbloqueado y se bloquea y libera mediante la instrucción *with*.
Vale la pena señalar aquí que el hilo que ejecuta esta función mantendrá ese bloqueo hasta que termine por completo de actualizar la base de datos. En este caso, eso significa que mantendrá el bloqueo mientras copia, actualiza, duerme y luego vuelve a escribir el valor en la base de datos.

LOCK

Puede ver que Thread 0 adquiere el bloqueo y aún lo mantiene cuando se va a dormir. El Thread 1 luego comienza e intenta adquirir el mismo bloqueo. Debido a que el Thread 0 aún lo está reteniendo, el Thread 1 tiene que esperar. Esta es la exclusión mutua que proporciona un bloqueo.

DEADLOCK

Antes de continuar, debe observar un problema común al usar bloqueos. Como vio, si ya se adquirió el bloqueo, una segunda llamada a *.acquire()* esperará hasta que el hilo que contiene el bloqueo llame a *.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")
```

DEADLOCK

Cuando el programa llama a *l.acquire()* por segunda vez, se cuelga esperando que se libere el bloqueo. En este ejemplo, puede solucionar el deadlock (interbloqueo) eliminando la segunda llamada, pero los deadlocks generalmente ocurren por una de dos cosas sutiles:

1) Un error de implementación en el que un bloqueo no se libera correctamente.
2) Un problema de diseño en el que una función de utilidad debe ser llamada por funciones que pueden o no tener el bloqueo.

La primera situación ocurre a veces, pero usar un bloqueo como administrador de contexto reduce en gran medida la frecuencia. Se recomienda escribir código siempre que sea posible para utilizar administradores de contexto, ya que ayudan a evitar situaciones en las que una excepción pasa por alto la llamada *.release()*.