

Threads

Antonio Espín Herranz

Threads vs Procesos

- **Procesos:**

- Un proceso es la ejecución de un programa.
- Con una sola CPU se puede simular la ejecución de varios programas simultáneamente.
 - El sistema operativo va asignando pequeñas parcelas de tiempo de CPU para su ejecución.
- El cambio de un proceso a otro se llama “cambio de contexto” y costoso en cuanto a recursos y tiempo.

- **Threads:**

- También llamados procesos ligeros.
- Se ejecutan dentro del mismo proceso.
- Una parte del código de mi programa se ejecuta por varios hilos.
- No requieren cambios de contexto.
- Comparten los recursos entre sí.

El GIL

- **GIL:** Global Interpreter Lock.
 - Permite la ejecución de hilos de forma que sólo un thread puede ejecutarse a la vez, independientemente del número de procesadores con el que cuente la máquina.
 - Cada cierto número de instrucciones de bytecode la máquina virtual para la ejecución del thread y elige otro de entre los que estaban esperando.
 - Por defecto el cambio de thread se realiza cada 10 instrucciones de bytecode,

Conceptos

- ***Bloqueo mutuo (deadlock):***
 - ***Es el bloqueo irreversible de un conjunto de hilos o procesos.***
 - Un bloqueo mutuo es lo que sucede cuando un programa se te queda “colgado” y te ves en la obligación de “matar un proceso” ya que el conjunto de hilos o procesos bloqueados, no tiene solución.
- ***Condición (o estado) de carrera:***
 - ***Es aquel que se produce cuando varios hilos o procesos intentan modificar de forma simultánea a un mismo recurso.***
 - Si más de un hilo o recurso intenta modificar el estado o valor de un mismo recurso al mismo tiempo, los datos dejan de ser confiables y por consiguiente, es correcto afirmar que **los datos quedan corruptos.**

Threads en Python

- El trabajo con threads se lleva a cabo en Python mediante el **módulo thread**.
- **El módulo threading** que se apoya en el primero para proporcionarnos una API de más **alto nivel**, más completa, y orientada a objetos.

Threads en Python

- El **módulo threading** contiene una **clase Thread** que debemos **extender** para crear nuestros propios hilos de ejecución.
- El **método run** contendrá el código que queremos que ejecute el thread.
- El **constructor** de la clase tiene que llamar a **Thread.__init__(self)** para inicializar el objeto correctamente.

Ejemplo

```
import threading
class MiThread(threading.Thread):
    def __init__(self, num):
        threading.Thread.__init__(self)
        self.num = num

    def run(self):
        print "Soy el hilo", self.num
```

- El código principal:

- **Crea** varios hilos.
- **Nunca** se **llama** directamente al método **run**.
- Al **llamar a start** el hilo empieza a ejecutar el método run.

```
print "Soy el hilo principal"
```

```
for i in range(0, 10):
```

```
    t = MiThread(i)
```

```
    t.start()
```

Ojo, si ejecutamos join, hilo a hilo hace que los hilos se ejecuten uno detrás de otro.

```
    t.join()
```

Si no se lanza join, main termina pero los hilos continúan ejecutándose.

El método **join** se utiliza para que el hilo que ejecuta la llamada se bloquee hasta que finalice el **thread** sobre el que se llama



Threads en Python

- Otra posibilidad (pero es mejor heredar de la clase Thread).

```
import threading

def imprime(num):
    print "Soy el hilo", num

print "Soy el hilo principal"
for i in range(0, 10):
    t = threading.Thread(target=imprime, args=(i, ))
    t.start()
```


Métodos

- **isAlive()**
 - Comprobar si el hilo está en ejecución.
- **threading.enumerate**
 - Obtendremos una lista de los objetos Thread que se están ejecutando, incluyendo el hilo principal.
 - Podemos comparar el objeto Thread con la variable `main_thread` para comprobar si se trata del hilo principal.
- **threading.activeCount**
 - Podemos consultar el **número de threads ejecutándose**. Incluye en la cuenta el thread de **main**.

Otra forma

- Para crear hilos podemos declarar una función.
 - Representará la función a ejecutar por el hilo.
 - `def funcion_thread(param1,param2,...):`
 - `# código a ejecutar por el hilo.`
- Llamar a la función:
 - **`start_new_thread(funcion_thread,(param1,param2))`**
 - La función **`start_new_thread`** recibe **dos parámetros** la **función a ejecutar** por el hilo **y una tupla** con los parámetros que le pasamos.
 - **Ojo**, como es una **tupla** si solo necesitamos un parámetro lo indicaremos así: **`(param1,)`**
 - **La coma es obligatoria para indicar que es una tupla.**

Sincronización

- ¿Qué ocurre cuando tenemos varios hilos y comparten cierta información?
- Por ejemplo, en un esquema **productor – consumidor** con un buffer común.
- La zona compartida se llama **región crítica** y si hay varios hilos que pueden acceder a ella a la vez se puede dar problemas con la sincronización.

Sincronización

- Necesitamos mecanismos de sincronización que nos garanticen el acceso exclusivo a una región crítica.
- **Mecanismos de sincronización:**
 - Locks.
 - locks reentrant.
 - Semáforos.
 - Condiciones .
 - Eventos.

Locks

- Los **locks**, también llamados **mutex** (de mutual exclusion), cierres de exclusión mutua, cierres o candados, son objetos con **dos estados** posibles: **adquirido** o **libre**.
- Cuando un thread adquiere el candado, los demás threads que lleguen a ese punto posteriormente y pidan adquirirlo se bloquearán hasta que el thread que lo ha adquirido libere el candado, momento en el cuál podrá entrar otro thread.

Locks

- El candado se representa mediante la **clase Lock**.
- Para adquirir el candado se utiliza el **método acquire** del objeto, al que se le puede pasar un booleano para indicar si queremos esperar a que se libere (True) o no (False).
- Si indicamos que no queremos esperar, el método devolverá True o False dependiendo de si se adquirió o no el candado, respectivamente.
- **Por defecto, si no se indica nada, el hilo se bloquea indefinidamente.**

Ejemplo

```
lista = []  
lock = threading.Lock()
```

```
def anyadir(obj):  
    lock.acquire()  
    lista.append(obj)  
    lock.release()
```

```
def obtener():  
    lock.acquire()  
    obj = lista.pop()  
    lock.release()  
    return obj
```

RLock

- La clase **RLock** funciona de forma similar a Lock, pero en este caso **el candado puede ser adquirido por el mismo thread varias veces**, y no quedará liberado hasta que el thread llame a release tantas veces como llamó a acquire.
- Como en Lock, es posible indicar a acquire si queremos que se bloquee o no.

Locks vs RLocks

- Ambos tienen los métodos `acquire()` y `release()`.
- **RLock** hay que utilizarlo o es útil cuando queremos tener un **ACCESO SEGURO** desde fuera de una clase y se utilizan los mismos métodos dentro de la clase.

VER EJEMPLO_RLOCK

Semaphore

- Los semáforos son otra clase de candados.
- La clase correspondiente, **Semaphore**, también cuenta con métodos **acquire** y **release**.
- El **constructor de Semaphore** puede tomar como **parámetro opcional un entero value** indicando el **número máximo de threads que pueden acceder a la vez a la sección de código crítico**.
 - Si no se indica nada permite el acceso a un solo thread.

Semaphore

- La operación **acquire** → **decrementa en 1** el valor del contador.
- Cuando el contador es cero si un hilo intenta entrar se bloqueará.
- La operación **release** → **incrementa en 1** el valor del contador.

- **Ejemplo**

```
semaforo = threading.Semaphore(4)
```

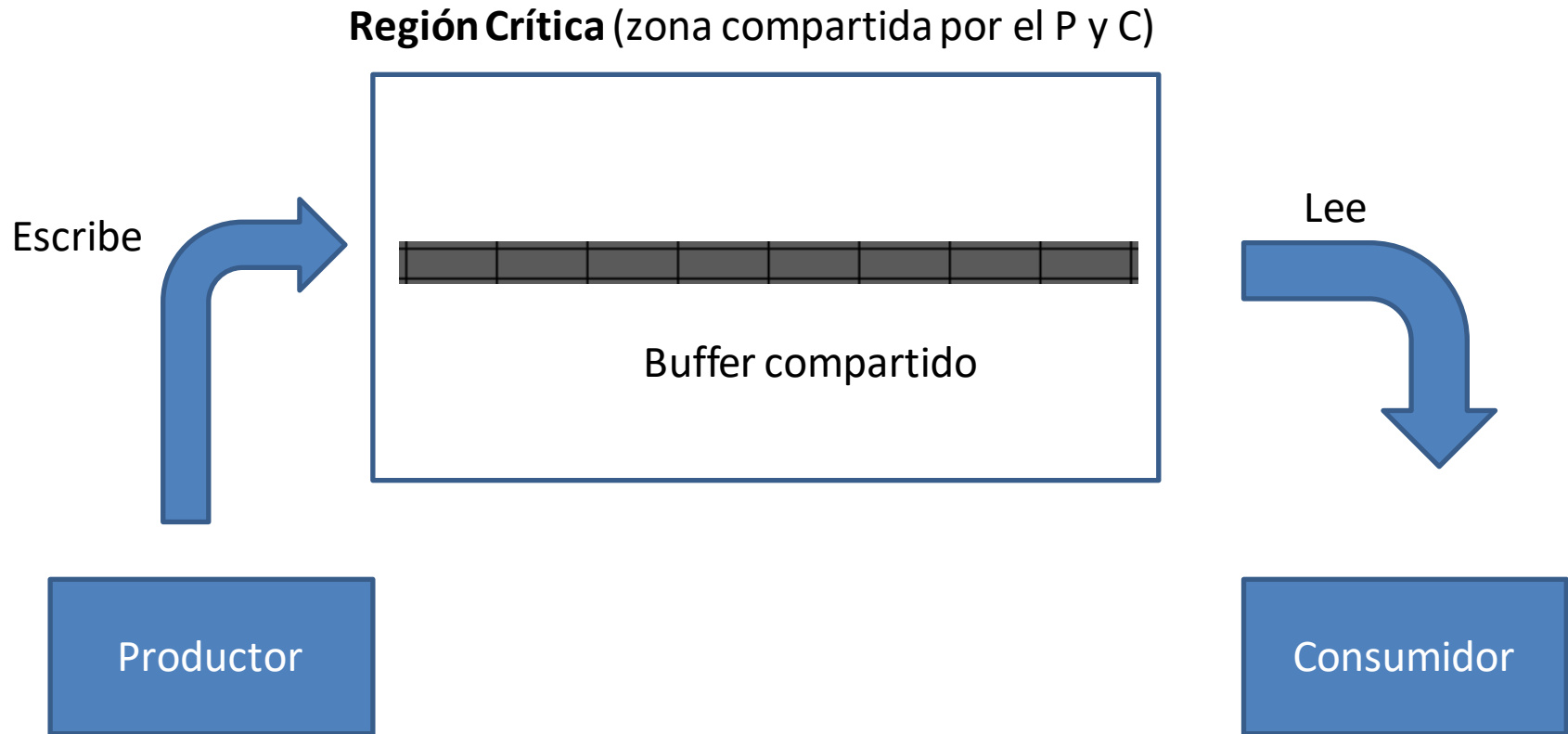
```
def descargar(url):
```

```
    semaforo.acquire()
```

```
    urllib.urlretrieve(url)
```

```
    semaforo.release()
```

Esquema Productor / Consumidor



Esquema: 1 PROD. → 1 CON.

semaphore fillCount = 0; *// items produced*

semaphore emptyCount = BUFFER_SIZE; *// remaining space*

procedure producer() {

while (true) {

 item = produceItem();

 down(emptyCount);

 putItemIntoBuffer(item);

 up(fillCount);

 }

}

procedure consumer() {

while (true) {

 down(fillCount);

 item = removeItemFromBuffer();

 up(emptyCount);

 consumeItem(item);

 }

}

Esquema: N PROD. \rightarrow N CON.

```
semaphore mutex = 1;  
semaphore fillCount = 0;  
semaphore emptyCount = BUFFER_SIZE;
```

```
procedure producer() {  
    while (true) {  
        item = produceItem();  
        down(emptyCount);  
        down(mutex);  
        putItemIntoBuffer(item); # Solo puede escribir uno en el buffer  
        up(mutex);  
        up(fillCount);  
    }  
}
```

```
procedure consumer() {  
    while (true) {  
        down(fillCount);  
        down(mutex);  
        item = removeItemFromBuffer(); # Solo puede leer uno del buffer  
        up(mutex);  
        up(emptyCount);  
        consumeItem(item);  
    }  
}
```

Instrucción with

- Es posible utilizar los semáforos, locks, etc. De la siguiente manera:

```
semaforo = threading.Semaphore(4)
def descargar(url):
    with semaforo
        urllib.urlretrieve(url)
```

- La instrucción **with** se encarga de hacer a la entrada “**acquire**” y “**release**” a la salida.

Condition

- Una condición **identifica un cambio de estado en una aplicación**. Es un mecanismo de sincronización donde **un hilo espera** por una condición específica y **otro hilo notifica** que la condición se ha producido.
- Una vez que se da la condición, el hilo adquiere el Lock y tiene acceso exclusivo al recurso compartido.
- Para esto utilizan un Lock pasado como parámetro, o crean un objeto RLock automáticamente si no se pasa ningún parámetro al constructor.

Condition

- Son especialmente adecuadas para el clásico problema de **productor-consumidor**.
- La clase cuenta con métodos **acquire** y **release**, que llamarán a los métodos correspondientes del candado asociado.
- También tenemos métodos **wait**, **notify** y **notifyAll**.

Condition

- El método **wait** debe llamarse después de haber adquirido el **candado con acquire**.
- Este método **libera el candado y bloquea** al **thread** hasta que una llamada a **notify** o **notifyAll** en otro thread le indican que se ha cumplido la condición por la que esperaba.
- El **thread** que informa a los demás de que se ha producido la **condición**, también debe llamar a **acquire** antes de llamar a **notify** o **notifyAll**.
- Al llamar a **notify**, se informa del evento a un solo thread, y por tanto se despierta un solo thread.
- Al llamar a **notifyAll** se **despiertan todos** los threads que esperaban a la condición.

Ejemplo

```
lista = []
```

```
cond = threading.Condition()
```

```
def consumir():  
    cond.acquire()  
    cond.wait()  
    obj = lista.pop()  
    cond.release()  
    return obj
```

```
def producir(obj):  
    cond.acquire()  
    lista.append(obj)  
    cond.notify()  
    cond.release()
```

Otro ejemplo

```
import logging
import threading
import time
```

```
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(threadName)-2s %(message)s', )
```

```
def consumer(cond):
```

```
    """wait for the condition and use the resource"""
```

```
    logging.debug('Starting consumer thread')
```

```
with cond:
```

```
    cond.wait()
```

```
    logging.debug('Resource is available to consumer')
```

```
def producer(cond):
```

```
    """set up the resource to be used by the consumer"""
```

```
    logging.debug('Starting producer thread')
```

```
with cond:
```

```
    logging.debug('Making resource available')
```

```
    cond.notifyAll()
```

```
condition = threading.Condition()
```

```
c1 = threading.Thread(name='c1', target=consumer, args=(condition,))
```

```
c2 = threading.Thread(name='c2', target=consumer, args=(condition,))
```

```
p = threading.Thread(name='p', target=producer, args=(condition,))
```

```
c1.start()
```

```
time.sleep(2)
```

```
c2.start()
```

```
time.sleep(2)
```

```
p.start()
```

Eventos

- Los eventos, implementados mediante la clase Event, son un **wrapper por encima de Condition** y sirven principalmente para **coordinar threads mediante señales que indican que se ha producido un evento**.
- Los eventos nos abstraen del hecho de que estemos utilizando un Lock por debajo, por lo que carecen de métodos acquire y release.

Ejemplo

```
import threading, time

class MiThread(threading.Thread):
    def __init__(self, evento):
        threading.Thread.__init__(self)
        self.evento = evento

    def run(self):
        print self.getName(), "esperando al evento"
        self.evento.wait()
        print self.getName(), "termina la espera"

evento = threading.Event()
t1 = MiThread(evento)
t1.start()
t2 = MiThread(evento)
t2.start()
# Esperamos un poco
time.sleep(5)
evento.set()
```

Datos globales independientes

- Los threads comparten las variables globales.
- Podemos utilizar variables **globales** pero que estas variables **se comporten** como si fueran **locales a un solo thread**.
- La clase **threading.local**, que crea un almacén de datos locales.
 - `datos_locales = threading.local()`
 - `datos_locales.mi_var = "hola"`
 - `print datos_locales.mi_var`

Compartir información

- La clase **Queue.Queue**, que implementa una **cola** (una estructura de datos de tipo FIFO) **con soporte multihilo**.
- El constructor de **Queue** toma un parámetro opcional indicando el **tamaño máximo de la cola**.
 - Si no se indica ningún valor no hay límite de tamaño.

Compartir información

- Métodos Queue:

- **put(item)**: Para añadir un elemento a la cola.
- **get()**: para obtener el siguiente elemento
- Ambos métodos tienen un **parámetro booleano opcional block** que indica si queremos que se espere hasta que haya algún elemento en la cola para poder devolverlo o hasta que la cola deje de estar llena para poder introducirlo.
- Otro parámetro opcional **timeout** que indica, en **segundos**, el tiempo máximo a esperar.
 - Si el timeout acaba sin poder haber realizado la operación debido a que la cola estaba llena o vacía, o bien si block era False, se lanzará una excepción de tipo Queue.Full o Queue.Empty, respectivamente.

Ejemplo

```
q = Queue.Queue()
class MiThread(threading.Thread):
    def __init__(self, q):
        self.q = q
        threading.Thread.__init__(self)

    def run(self):
        while True:
            try:
                obj = q.get(False)
            except Queue.Empty:
                print "Fin"
                break
            print obj

for i in range(10):
    q.put(i)

t = MiThread(q)
t.start()
t.join()
```

Apéndice

- Logging en los Threads
- `threading.Timer`
- `threading.Barrier`

Loggin + Threads

- Obtener trazas de los hilos para luego depurar:

```
import threading
```

```
import logging
```

```
import time
```

```
logging.basicConfig( level=logging.DEBUG, format='[%(levelname)s] -  
%(threadName)-10s : %(message)s')
```

```
def hilo():
```

```
    logging.debug('Lanzado')
```

```
    time.sleep(2)
```

```
    logging.debug('Deteniendo')
```

```
d = threading.Thread(target=hilo, name='Hilo')
```

```
d.start()
```

```
d.join()
```

```
[DEBUG] - Hilo           : Lanzado  
[DEBUG] - Hilo           : Deteniendo
```

threading.Timer

- `import threading`
- `import time`
- `import logging`
- `logging.basicConfig(level=logging.DEBUG,`
- `format='%(threadName)-10s) %(message)s',`
- `)`
- `def delayed():`
- `logging.debug('worker running')`
- `return`
- `t1 = threading.Timer(3, delayed)`
- `t1.setName('t1')`
- `t2 = threading.Timer(3, delayed)`
- `t2.setName('t2')`
- `logging.debug('starting timers')`
- `t1.start()`
- `t2.start()`
- `logging.debug('waiting before canceling %s', t2.getName())`
- `time.sleep(2)`
- `logging.debug('canceling %s', t2.getName())`
- `t2.cancel()`
- `logging.debug('done')`

Timer(sg, function)

Se arrancar con **start()**

Y se pueden cancelar con **cancel()**

Esta clase representa una acción que Debe ejecutarse sólo después de que Haya pasado una cierta cantidad de Tiempo.

threading.Barrier

- Las barreras son otro mecanismo de sincronización de hilos.
- Una Barrier establece un punto de control y todos los hilos participantes bloquean hasta que todas las «partes» participantes hayan alcanzado ese punto.
- Permite que los hilos se inicien por separado y luego se pause hasta que todos están listos para continuar.

Métodos / Propiedades

- **Barrier**(parties, action=None, timeout=None)
 - Crea la barrera para el valor que indique parties. Action una función que ejecutará el primero hilo que pase la barrera.
- **wait**(timeout=None)
 - Para pasar la barrera, cuando llegan todos los hilos a este punto continúan luego todos simultáneamente.
- **abort**()
 - Aborta la barrera.
- **reset**()
 - Retorna la barrera al estado vacío
- **parties**
 - Número de hilos necesarios para pasar la barrera.
- **n_waiting**
 - Número de hilos actual que esperan en la barrera.
- **broken**: boolean
 - Indica si la barrera esta rota
- <https://docs.python.org/3/library/threading.html>

```
import threading
import time
```

Ejemplo

```
def worker(barrier):
    print(threading.current_thread().name,
          'waiting for barrier with {} others'.format(barrier.n_waiting))
    worker_id = barrier.wait()
    print(threading.current_thread().name, 'after barrier', worker_id)

def mensajePrimero():
    print(threading.current_thread().name + " soy el primero")
```

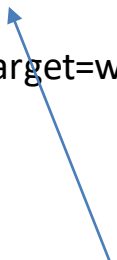
```
NUM_THREADS = 3
```

```
barrier = threading.Barrier(NUM_THREADS, mensajePrimero)
```

```
threads = [ threading.Thread( name='worker-%s' % i, target=worker, args=(barrier,), )
            for i in range(NUM_THREADS)]
```

```
for t in threads:
    print(t.name, 'starting')
    t.start()
    time.sleep(0.1)
```

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



La acción la realiza el primero que pasa
La barrera.

Demonios

- Existen **dos modos diferentes de finalizar un programa basado en hilos**.
 - En el primer modo el hilo principal del programa espera a que todos los hilos creados con **Thread** terminan su trabajo.
 - En el segundo modo, el hilo principal del programa puede finalizar aunque uno o más hilos hijos no hayan terminado su tarea; teniendo en cuenta que cuando finalice el hilo principal también lo harán estos hilos especiales llamados **demonios**.
 - Si existen hilos no-demonios el hilo principal esperará a que estos concluyan su trabajo. Los **demonios** son útiles para programas que realizan operaciones de monitorización o de chequeo de recursos, servicios, aplicaciones, etc.

Para declarar un hilo como **demonio** se asigna **True** al argumento **daemon** al crear el objeto **Thread**, o bien, se establece dicho valor con posterioridad con el método **set_daemon()**.

Ejemplo

```
hilo1 = threading.Thread(name='chequear',  
                           target=chequear,  
                           args=(nombre,),  
                           daemon=True)
```

Enlaces

- <http://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/>