Mecanismos de sincronización en *Threading*

Computación Concurrente
Dante Bermúdez Marbán

15 de octubre del 2019

1. Lock

Puede estar en dos estados: bloqueado y desbloqueado (estado por default cuando se crea).

El funcionamiento es sencillo: `acquire` bloquea la ejecución hasta que se llame `release`, ya sea en el mismo procedimiento o en alguno otro.

Adicionalmente, el método acquire puede recibir dos parámetros: el primero es *blocking* y es un booleano el cual representa si se debería bloquear la ejeución (valor verdadero por defecto), y el segundo es *timeout*, el cual es un número que representa cuantos segundos puede intentar conseguir el bloqueo, por defecto es -1, es decir, no hay tiempo. Retorna verdadero si lo logra y falso si el tiempo expira

2. Rlock

A diferencia de `Lock`, `RLock` (de *Re-entrant lock*) es un candado que tiene sentido de pertenencia cuando está bloqueado (se dice que algún hilo posee el bloqueo). Puede ser adquirido múltiples veces.

Por consiguiente, solamente el hilo que adquirió el bloqueo puede liberarlo. También puede adquirirse varias veces (muy útil para la recursión).

Utiliza los mismos métodos de `Lock`: `acquire` y `release`.

3. Condition

Las variables de tipo condición siempren tienen asociado algún tipo de bloqueo (se le puede pasar o se crea uno). Pasarle uno es útil cuando varias varaibles de condiciones deben compartir el candado.

La operación `wait` espera hasta que se le notifique que puede reanudar su actividad (con `notify`) ó hasta que determinado tiempo se acabase. Por lo anterior se puede inferir que puede recibir un parámetro opcional *timeout*, el cual es un número.

De la misma manera *wait_for* es un método parecido al descrito anterior con la diferencia de que la espera se hace hasta que se cumpla una condición (o un predicado, para ser más precisos). Recibe de parámetros *predicate* y *timeout*. Es equivalente a tener

```
while not predicate():
    c.wait()
```

Por otro lado `notify`, despierta **un** hilo que estuviese esperando la condición (si es que hay alguno). Si se le especifica algún número, se despiertan ese número de hilos.

Existe también `notify_all`, el cual es muy parecido a `notify` pero éste despierta a todos los hilos que están esperando la condición.

4. Semaphore

Utiliza los métodos de `acquire` y `release` que hemos estado usando. Adicional a esto, gestiona un contador atómico, el cual debemos indicarle su valor inicial al crearlo, que se decrementa cuando se llama `acquire` e incrementa cuando se llama `release`.

Al utilizar `acquire`, podemos distinguir dos casos

- Si el contador es mayor que cero, se decrementa en uno y se retorna verdadero (de que si se consiguió el semáforo) inmediatamente.
- Cuando el contador es cero, se bloquea el semáforo hasta que se haga algún llamado a `release`. En ese caso, se incrementará el contador en uno por dicha llamada y se volverá a decrementar a cero para el hilo que estuviese esperando.

Existe una clase muy parecida llamada `BoundedSemaphore`, el cual no permite que el valor del contador sea más grande que el valor inicial.

5. Event

Un hilo espera por un evento y otro hace la señal.

Gestiona una bandera interna booleana la cual puede establecerse como verdadera con el método `set` y restablecer a false con el método `clear`.

Podemos consultar el valor de la bandera con el método `is_set`.

El método `wait` realiza el bloqueo hasta que la bandera sea verdadera.

Sentencia *with*

Las primitivas que hagan uso de `acquire` y `release` pueden usarse con el manejador de contexto, el cual se usa con *with*

```
1 with candado:
2     # Lo que se tenga que hacer con el bloqueo
```

Lo que sucede internamente con estos manejadores, es que se hace la llamada a `acquire` al entrar al bloque y se hace la llamada a `release` en cuando se termina el bloque.

Referencias

[1] Python Software Foundation. (2019). *threading — Thread-based parallelism Recuperado el 14 de octubre del 2019 de* `https://docs.python.org/3/library/threading.html`