Thread Synchronization Mechanisms in Python

Synchronizing Access to Shared Resources

One important issue when using threads is to avoid conflicts when more than one thread needs to access a single variable or other resource. If you're not careful, overlapping accesses or modifications from multiple threads may cause all kinds of problems, and what's worse, those problems have a tendency of appearing only under heavy load, or on your production servers, or on some faster hardware that's only used by one of your customers.

For example, consider a program that does some kind of processing, and keeps track of how many items it has processed:

```
counter = 0

def process_item(item):
    global counter
    ... do something with item ...
    counter += 1
```

If you call this function from more than one thread, you'll find that the counter isn't necessarily accurate. It works in most cases, but sometimes misses one or more items. The reason for this is that the increment operation is actually executed in three steps; first, the interpreter fetches the current value of the counter, then it calculates the new value, and finally, it writes the new value back to the variable.

If another thread gets control after the current thread has fetched the variable, it may fetch the variable, increment it, and write it back, *before* the current thread does the same thing. And since they're both seeing the same original value, only one item will be accounted for.

Another common problem is access to incomplete or inconsistent state, which can happen if one thread is initializing or updating some non-trivial data structure, and another thread attempts to read the structure while it's being updated.

Atomic Operations

The simplest way to synchronize access to shared variables or other resources is to rely on atomic operations in the interpreter. An atomic operation is an operation that is carried out in a single execution step, without any chance that another thread gets control.

In general, this approach only works if the shared resource consists of a single instance of a core data type, such as a string variable, a number, or a list or dictionary. Here are some thread-safe operations:

- reading or replacing a single instance attribute
- reading or replacing a single global variable
- fetching an item from a list
- modifying a list in place (e.g. adding an item using **append**)
- fetching an item from a dictionary
- modifying a dictionary in place (e.g. adding an item, or calling the **clear** method)

Note that as mentioned earlier, operations that read a variable or attribute, modifies it, and then writes it back are not thread-safe. Another thread may update the variable after it's been read by the current thread, but before it's been updated.

Also note that Python code may be executed when objects are destroyed, so even seemingly simple operations may cause other threads to run, and may thus cause conflicts. When in doubt, use explicit locks.

Locks

Locks are the most fundamental synchronization mechanism provided by the **threading** module. At any time, a lock can be held by a single thread, or by no thread at all. If a thread attempts to hold a lock that's already held by some other thread, execution of the first thread is halted until the lock is released.

Locks are typically used to synchronize access to a shared resource. For each shared resource, create a **Lock** object. When you need to access the resource, call **acquire** to hold the lock (this will wait for the lock to be released, if necessary), and call **release** to release it:

```
lock = Lock()

lock.acquire() # will block if lock is already held
... access shared resource
lock.release()
```

For proper operation, it's important to release the lock even if something goes wrong when accessing the resource. You can use **try-finally** for this purpose:

```
lock.acquire()
try:
    ... access shared resource
finally:
    lock.release() # release lock, no matter what
```

In Python 2.5 and later, you can also use the **with** statement. When used with a lock, this statement automatically acquires the lock before entering the block, and releases it when leaving the block:

```
from __future__ import with_statement # 2.5 only

with lock:
    ... access shared resource
```

The **acquire** method takes an optional wait flag, which can be used to avoid blocking if the lock is held by someone else. If you pass in False, the method never blocks, but returns False if the lock was already held:

```
if not lock.acquire(False):
    ... failed to lock the resource
else:
    try:
        ... access shared resource
    finally:
        lock.release()
```

You can use the **locked** method to check if the lock is held. Note that you cannot use this method to determine if a call to **acquire** would block or not; some other thread may have acquired the lock between the method call and the next statement.

```
if not lock.locked():
    # some other thread may run before we get
    # to the next line
    lock.acquire() # may block anyway
```

Problems with Simple Locking

The standard lock object doesn't care which thread is currently holding the lock; if the lock is held, any thread that attempts to acquire the lock will block, even if the same thread is already holding the lock. Consider the following example:

```
lock = threading.Lock()

def get_first_part():
    lock.acquire()
    try:
        ... fetch data for first part from shared object
    finally:
        lock.release()
    return data

def get_second_part():
    lock.acquire()
    try:
        ... fetch data for second part from shared object
    finally:
        lock.release()
```

```
    return data
```

Here, we have a shared resource, and two access functions that fetch different parts from the resource. The access functions both use locking to make sure that no other thread can modify the resource while we're accessing it.

Now, if we want to add a third function that fetches both parts, we quickly get into trouble. The naive approach is to simply call the two functions, and return the combined result:

```
def get_both_parts():
    first = get_first_part()
    second = get_second_part()
    return first, second
```

The problem here is that if some other thread modifies the resource between the two calls, we may end up with inconsistent data. The obvious solution to this is to grab the lock in this function as well:

```
def get_both_parts():
    lock.acquire()
    try:
        first = get_first_part()
        second = get_second_part()
    finally:
        lock.release()
    return first, second
```

However, this won't work; the individual access functions will get stuck, because the outer function already holds the lock. To work around this, you can add flags to the access functions that enables the outer function to disable locking, but this is error-prone, and can quickly get out of hand. Fortunately, the **threading** module contains a more practical lock implementation; re-entrant locks.

Re-Entrant Locks (RLock)

The **RLock** class is a version of simple locking that only blocks if the lock is held by *another* thread. While simple locks will block if the same thread attempts to acquire the same lock twice, a re-entrant lock only blocks if another thread currently holds the lock. If the current thread is trying to acquire a lock that it's already holding, execution continues as usual.

```
lock = threading.Lock()
lock.acquire()
lock.acquire() # this will block

lock = threading.RLock()
lock.acquire()
lock.acquire() # this won't block
```

The main use for this is nested access to shared resources, as illustrated by the example in the previous section. To fix the access methods in that example, just replace the simple lock with a re-entrant lock, and the nested calls will work just fine.

```
lock = threading.RLock()

def get_first_part():
    ... see above

def get_second_part():
    ... see above

def get_both_parts():
    ... see above
```

With this in place, you can fetch either the individual parts, or both parts at once, without getting stuck or getting inconsistent data.

Note that this lock keeps track of the recursion level, so you still need to call **release** once for each call to **acquire**.

Semaphores

A semaphore is a more advanced lock mechanism. A semaphore has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore. Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously.

```
semaphore = threading.BoundedSemaphore()
semaphore.acquire() # decrements the counter
... access the shared resource
semaphore.release() # increments the counter
```

The counter is decremented when the semaphore is acquired, and incremented when the semaphore is released. If the counter reaches zero when acquired, the acquiring thread will block. When the semaphore is incremented again, one of the blocking threads (if any) will run.

Semaphores are typically used to limit access to resource with limited capacity, such as a network connection or a database server. Just initialize the counter to the maximum number, and the semaphore implementation will take care of the rest.

```
max_connections = 10

semaphore = threading.BoundedSemaphore(max_connections)
```

If you don’t pass in a value, the counter is initialized to 1.

Python's **threading** module provides two semaphore implementations; the **Semaphore** class provides an unlimited semaphore which allows you to call **release** any number of times to increment the counter. To avoid simple programming errors, it's usually better to use the **BoundedSemaphore** class, which considers it to be an error to call **release** more often than you've called **acquire**.

Synchronization Between Threads

Locks can also be used for synchronization between threads. The **threading** module contains several classes designed for this purpose.

Events

An event is a simple synchronization object; the event represents an internal flag, and threads can wait for the flag to be set, or set or clear the flag themselves.

```
event = threading.Event()

# a client thread can wait for the flag to be set
event.wait()

# a server thread can set or reset it
event.set()
event.clear()
```

If the flag is set, the **wait** method doesn't do anything. If the flag is cleared, **wait** will block until it becomes set again. Any number of threads may wait for the same event.

Conditions

A condition is a more advanced version of the event object. A condition represents some kind of state change in the application, and a thread can wait for a given condition, or signal that the condition has happened. Here's a simple consumer/producer example. First, you need a condition object:

```
# represents the addition of an item to a resource
condition = threading.Condition()
```

The producing thread needs to acquire the condition before it can notify the consumers that a new item is available:

```
# producer thread
... generate item
condition.acquire()
... add item to resource
condition.notify() # signal that a new item is available
condition.release()
```

The consumers must acquire the condition (and thus the related lock), and can then attempt to fetch items from the resource:

```
# consumer thread
condition.acquire()
while True:
    ... get item from resource
    if item:
        break
    condition.wait() # sleep until item becomes available
condition.release()
... process item
```

The **wait** method releases the lock, blocks the current thread until another thread calls **notify** or **notifyAll** on the same condition, and then reacquires the lock. If multiple threads are waiting, the **notify** method only wakes up one of the threads, while **notifyAll** always wakes them all up.

To avoid blocking in **wait**, you can pass in a timeout value, as a floating-point value in seconds. If given, the method will return after the given time, even if **notify** hasn't been called. If you use a timeout, you must inspect the resource to see if something actually happened.

Note that the condition object is associated with a lock, and that lock must be held before you can access the condition. Likewise, the condition lock must be released when you're done accessing the condition. In production code, you should use **try-finally** or **with**, as shown earlier.

To associate the condition with an existing lock, pass the lock to the **Condition** constructor. This is also useful if you want to use several conditions for a single resource:

```
lock = threading.RLock()
condition_1 = threading.Condition(lock)
condition_2 = threading.Condition(lock)
```