Synchronization with Python: Lock, Semaphore

CPEN333 - 2021 W2

University of British Columbia

Introduction

- The Python standard library's threading module provides a number of classes that can be used for thread synchronization.

- Similarly, the Python standard library's multiprocessing module provides a number of classes that can be used for process synchronization.
  - Generally synchronization primitives are not as necessary in a multi-process program as they are in a multithreaded program, though still there are many cases that they are valuable and indispensable.

- In this set of slides, we focus on two of those:
  - Lock
  - Semaphore

Shared data within Threads

- Recall that threads of a process share its data section.
- So, in the simple examples below, the child thread has access to var, but the child process does not have access.

```
import threading
def worker():
    print(f"Has access to var: {var}")

if __name__ == "__main__":
    var = "hello"
    threading.Thread(target=worker).start()
```

output:

Has access to var: hello

```
import multiprocessing
def worker():
    try:
        print(f"Trying to access var: {var}")
    except NameError:
        print("Nope, doesn't have access")

if __name__ == "__main__":
    var = "hello"
    multiprocessing.Process(target=worker).start()
```

output:

Nope, doesn't have access

Problematic example 1

- Let's look at an example, where two threads incrementing a shared data without synchronization causes data consistency problem.

output:

problem

The shared data's final value: 1

```
import threading, time

class SharedData:
    def __init__(self):
        self.data : int = 0
    def increment(self):
        x = self.data
        x += 1
        time.sleep(0.2)  #to emulate possible behaviour
        self.data = x


if __name__ == "__main__":
    y = SharedData()
    t1 = threading.Thread(target=y.increment)
    t2 = threading.Thread(target=y.increment)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(f"The shared data's final value: {y.data}")
```

Note that, although simplified and exaggerated, this is not a silly or rare example. As we explained before, it could happen when x = x + 1 is calculated (e.g. due to an interrupt).

Problematic example 2

- Another example that our intended critical section is not executed atomically.

```
import threading, time

def worker(greetings):
    for _ in range(3):
        print("Say ", end="")
        time.sleep(0.2)
        print(greetings)

if __name__ == "__main__":
    threading.Thread(target=worker, args=("aloha",)).start()
    threading.Thread(target=worker, args=("ciao",)).start()
```

Nothing out of ordinary and no data sharing, but what if we want these three lines to be executed as a whole together indivisibly, so the print is not messed up.

output:

```
Say Say ciao
aloha
Say Say ciao
Say aloha
Say ciao
aloha
```

Synchronization in Python

- There are a number of provisions in Python to allow synchronization.

- We start by considering the primitive `lock` from the `threading` module.

- `class threading.Lock`
  - The class implement a primitive lock.
  - A primitive lock in Python is the lowest level synchronization primitive that is not owned by a particular thread when locked.

threading.Lock

- https://docs.python.org/3/library/threading.html#lock-objects

class `threading.` **Lock** ¶

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Note that `Lock` is actually a factory function which returns an instance of the most efficient version of the concrete Lock class that is supported by the platform.

- Syntax:

```
import threading

lock = threading.Lock() #instantiate a lock object
...
lock.acquire() #acquire the lock
#critical section here
lock.release() #release the lock
```

Lock

- A lock can have either of two states: locked or unlocked

- It defines three methods:
  - All methods are executed atomically.
  - `acquire(blocking=True, timeout=-1)`: acquire a lock.
    - blocking: block until the lock is unlocked, or not to block
    - timeout: -1 means unbounded wait. If a positive value is used instead, it blocks for at most the number of seconds specified by timeout (floating point).
  - `release`(): releases a lock.
  - `locked`(): return true if lock is acquired.

acquire()

- https://docs.python.org/3/library/threading.html#threading.Lock.acquire

acquire(*blocking=True*, *timeout=- 1*)

Acquire a lock, blocking or non-blocking.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is false.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()

- https://docs.python.org/3/library/threading.html#threading.Lock.release

release()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

There is no return value.

Note

- It is important that we correctly and in the correct order use the `acquire` and `release` method pair.

- Failure to do so will cause a deadlock, a topic that we will discuss in a future lecture.

Example

- Let's rewrite the problem example 1 with locks to fix:

```
import threading, time

class SharedData:
    def __init__(self):
        self.data : int = 0
    def increment(self):
        lock.acquire() #acquire the lock
        x = self.data
        x += 1
        time.sleep(0.2)
        self.data = x
        lock.release() #release the lock

if __name__ == "__main__":
    y = SharedData()
    lock = threading.Lock() #instantiate a lock object (shared)
    t1 = threading.Thread(target=y.increment)
    t2 = threading.Thread(target=y.increment)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(f"The shared data's final value: {y.data}")
```

as expected

output:

```
The shared data's final value: 2
```

The `with` statement

- A Lock object can be used as a context manager for a `with` statement.
  - The lock object has `acquire`() and `release`() methods.
  - We use `try/finally` to ensure the release of the lock.
- The following code:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

- is equivalent to:

```
with some_lock:
    # do something...
```

  - The `acquire`() method is called when the with block is entered and `release`() is called when the block is exited.

Example (using `with`)

- Now the same code using `with`:

```
class SharedData:
    def __init__(self):
        self.data : int = 0
    def increment(self):
        with lock: #lock is acquired here
            x = self.data
            x += 1
            time.sleep(0.2)
            self.data = x
        #if locked, lock is released here automatically and surely

if __name__ == "__main__":
    y = SharedData()
    lock = threading.Lock() #instantiate a lock object (shared)
    t1 = threading.Thread(target=y.increment)
    t2 = threading.Thread(target=y.increment)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(f"The shared data's final value: {y.data}")
```

output:

```
The shared data's final value: 2
```

Lock object for multiprocessing

- class multiprocessing.Lock
  - https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Lock

class `multiprocessing.` **Lock**

A non-recursive lock object: a close analog of `threading.Lock`. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of `threading.Lock` as it applies to threads are replicated here in `multiprocessing.Lock` as it applies to either processes or threads, except as noted.

Note that `Lock` is actually a factory function which returns an instance of `multiprocessing.synchronize.Lock` initialized with a default context.

`Lock` supports the context manager protocol and thus may be used in `with` statements.

Example

- Example from: https://docs.python.org/3/library/multiprocessing.html#synchronization-between-processes
  - To ensure only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def foo(lock, i):
    lock.acquire()
    try:
        print('hello world', i)
    finally:
        lock.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=foo, args=(lock, num)).start()

    #note: this doesn't work on macOS => must use join() to work
```

threading.Semaphore

- https://docs.python.org/3/library/threading.html#semaphore-objects

class `threading.` **Semaphore**(*value=1*)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, `ValueError` is raised.

Changed in version 3.3: changed from a factory function to a class.

Semaphore

- class `threading.Semaphore(value=1)`
  - implements semaphore objects. One of the oldest synchronization primitives, a semaphore manages an atomic counter representing the number of release() calls minus the number of acquire() calls, plus an initial value.
- Similar to Locks, Semaphore objects can be used as a context manager for a `with` statement.
- It defines two methods:
  - `acquire(blocking=True, timeout=None)`: acquire a semaphore.
  - `release(n=1)`: release a semaphore, incrementing the internal counter by n.

release(*n=1*)

Release a semaphore, incrementing the internal counter by *n*. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up *n* of those threads.

Changed in version 3.9: Added the *n* parameter to release multiple waiting threads at once.

Origin

- Semaphore (Greek): signal bearer (https://en.wikipedia.org/wiki/Semaphore)
- Comes from railway terminology: Railway semaphore signal is one of the earliest forms of fixed railway signals.
- Dijkstra (1965) adopted the idea for synchronizing concurrent processes from trains sharing the railways.

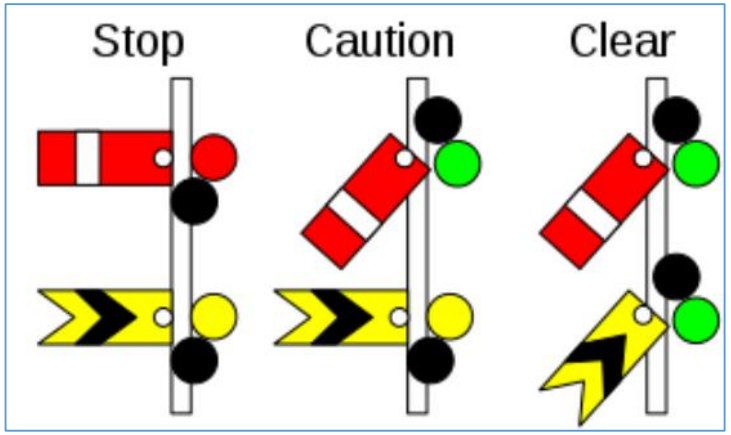
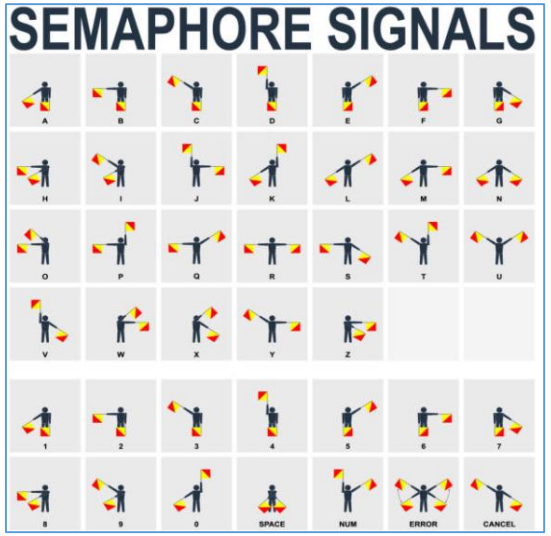


image: Wikipedia



Example

- Simplified, we can consider a semaphore as a variable that reflect the number of currently available resource.
  - Printer as the resource: if we have 3 printers, at any given time three threads can print.

```
import threading, time

def worker(printers):
    with printers:
        print(f"Thread {threading.current_thread().name} can print")
        time.sleep(1.5)

if __name__ == "__main__":
    printers = threading.Semaphore(3)   #3 printers
    for i in range(10):
        threading.Thread(target=worker, args=(printers,)).start()
```

Three printers, so three threads can print concurrently

Semaphore object for multiprocessing

- `class multiprocessing.Semaphore([value])`
  - https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Semaphore

class `multiprocessing.` **Semaphore**([*value*])

A semaphore object: a close analog of `threading.Semaphore`.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with `Lock.acquire()`.

References

- Python documentation:
  - Lock: https://docs.python.org/3/library/threading.html#lock-objects
  - Semaphore: https://docs.python.org/3/library/threading.html#semaphore-objects

- *The Python Standard Library*, D. Hellmann

- *Python programming a practical approach*, V.K. Sharma et al.