Sharing State

This lesson discusses the facilities available in the multiprocessing module to share state amongst processes.

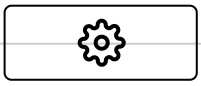
Sharing State

In the previous section we saw how processes can pass objects to eachother using queues or pipes. However, if a process P1 passes an object to process P2, any changes made by P1 to the object after the pass aren't visible to P2 since P2 only receives a copy of the object. For example, consider the below program.

Using queue to pass objects between two processes

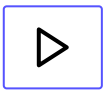
```
from multiprocessing import Process, Queue, Semaphore
import multiprocessing


def child_process(q, sem1, sem2):
    var = q.get()
    print("Child process received var = {0} with id {1} from q
ueue".format(str(var), id(var)))
    sem2.release()
    sem1.acquire()

    print("After changes by parent process var in child proces
s = {0}".format(str(var)), flush=True)


if __name__ == '__main__':
    q = Queue()
    sem1 = Semaphore(0)
    sem2 = Semaphore(0)
    print("This machine has {0} CPUs".format(str(multiprocessi
ng.cpu_count())))

    var = {"key" : "value"}
    print("Parent process puts item on queue with id " + str(i
d(var)))
    q.put(var)

    process = Process(target=child_process, args=(q, sem1, sem
2))
    process.start()

    sem2.acquire()

    # change the var
    var["key"] = "new-value"
    print("Parent process changed the enqueued item to " + str
(var), flush=True)
    sem1.release()
```

```
process.join()
```



In the above program, ignore the semaphore statements which we use to coordinate the two processes. The parent process modifies an object put on the queue but the print statements from the child process show that the change isn't visible to the child process since it has only received a copy of the object.

```
from multiprocessing import Process, Queue, Semaphore
import multiprocessing


def child_process(q, sem1, sem2):
    var = q.get()
    print("Child process received var = {0} with id {1} from queue".format(str(var)
    sem2.release()
    sem1.acquire()

    print("After changes by parent process var in child process = {0}".format(str(v


if __name__ == '__main__':
    q = Queue()
    sem1 = Semaphore(0)
    sem2 = Semaphore(0)
    print("This machine has {0} CPUs".format(str(multiprocessing.cpu_count())))

    var = {"key" : "value"}
    print("Parent process puts item on queue with id " + str(id(var)))
    q.put(var)

    process = Process(target=child_process, args=(q, sem1, sem2))
    process.start()

    sem2.acquire()

    # change the dictionary object
    var["key"] = "new-value"
    print("Parent process changed the enqueued item to " + str(var), flush=True)
    sem1.release()
    process.join()
```




Also note that the ID values of the object being enqueued and dequeued are different, thus proving they are two distinct objects. If we used an integer instead e.g. **var = 1** you'll see the child and parent process display the same id for the object. This is because under the hood Python returns references to existing integer objects it has already created, rather than creating new objects each time the same integer is asked for.

The queue and the semaphore object are truly shared between the two processes and both the parent and the child work on the same object. This is so because we are using the queue and semaphore from the `multiprocessing` module. Had we used the ones from the `threading` module then the child would only receive a copy.

Python allows us to share objects between processes using shared memory. We'll discuss the two ways the multiprocessing module offers to create shared objects.

Value

We can use the `Value` class to create a **ctype** object in shared memory. By default the function `Value()` returns a wrapper over the requested object, thus making the reads and writes to the underlying ctype object process-safe. An example is shown below:

Using Value

```
from multiprocessing import Value

pi = Value('d', 3.1415)
print(pi.value)
```

In the above code, we specify the code for the object we want created. The **'d'** specifies double and the second argument is passed over to the ctype double constructor. The third option not listed above is a boolean which, if set to false, will return an object without synchronized access. On the other hand, if set to true, a recursive lock is automatically created to guard access to the object. We can also specify an external lock and pass that instead of a boolean value. The passed-in lock is then used to synchronize access to the created object.

This link (https://docs.python.org/3/library/array.html#module-array) lists the type codes that can be used to create the ctype objects. For instance, **'h'** is for signed short and **'L'** is for unsigned long.

```
from multiprocessing import Process, Semaphore, Value
import multiprocessing


def child_process(sem1 , sem2, var):
    print("Child process received var = {0} with id {1} from queue".format(str(var.
    sem2.release()
    sem1.acquire()

    print("After changes by parent process var = {0}".format(var.value), flush=True


if __name__ == '__main__':
    # multiprocessing.set_start_method('spawn')
    sem1 = Semaphore(0)
    sem2 = Semaphore(0)
    print("This machine has {0} CPUs".format(str(multiprocessing.cpu_count())))

    var = Value('I', 1)
    print("Parent process puts item on queue with id " + str(id(var)))

    process = Process(target=child_process, args=(sem1, sem2, var))
    process.start()

    sem2.acquire()

    # change the var
    var.value += 2
    print("Parent process changed the enqueued item to " + str(var.value), flush=Tr
    sem1.release()
    process.join()
```

Pay attention to the ID of the **`var`** variable printed in the child and parent process. Both are the same, emphasizing that the object is shared between the two processes. However, if you uncomment the start method **line 14** to change it to "spawn" and rerun the program, the IDs will be different but the output will be the same.

Array

`Array` is very similar to how we use `Value`. Below is a sample script demonstrating how to use Array.

Using Array

```
def child_process(sem1, sem2, arr):
    print("Child process received var = {0} with id {1} from q
ueue".format(str(arr[0]), id(arr)), flush=True)
    sem1.release()
    sem2.acquire()

    print("After changes by parent process, child process see
s var as = {0}".format(arr[0]), flush=True)


if __name__ == '__main__':
    sem1 = Semaphore(0)
    sem2 = Semaphore(0)
    print("This machine has {0} CPUs".format(str(multiprocessi
ng.cpu_count())))

    arr = Array('i', range(5))
    print("Parent process puts item on queue with id " + str(i
d(arr)))

    process = Process(target=child_process, args=(sem1, sem2,
 arr))
    process.start()

    sem1.acquire()

    # change var and verify the change is reflected in the chi
ld process
    arr[0] += 100
    print("Parent process changed the enqueued item to " + str
(arr[0]), flush=True)
    sem2.release()
    process.join()
```

Note we are using semaphores only to coordinate the two processes.

```
from multiprocessing import Process, Semaphore, Array
import multiprocessing

def child_process(sem1, sem2, arr):
    print("Child process received var = {0} with id {1} from queue".format(str(arr[
    sem1.release()
    sem2.acquire()

    print("After changes by parent process, child process sees var as = {0}".format


if __name__ == '__main__':
    sem1 = Semaphore(0)
    sem2 = Semaphore(0)
    print("This machine has {0} CPUs".format(str(multiprocessing.cpu_count())))

    arr = Array('i', range(5))
    print("Parent process puts item on queue with id " + str(id(arr)))

    process = Process(target=child_process, args=(sem1, sem2, arr))
    process.start()

    sem1.acquire()

    # change var and verify the change is reflected in the child process
    arr[0] += 100
    print("Parent process changed the enqueued item to " + str(arr[0]), flush=True)
    sem2.release()
    process.join()
```

Note that these shared state objects can only be **inherited** by the child process. Trying to pass them via a **`Queue`** or a **`Pipe`** will result in an error. An example of the failure is shown below:

```
from multiprocessing import Process, Value, Queue
import multiprocessing


def child_process( var, q):
    print("Child process received var = {0} with id {1} from queue".format(str(var.


if __name__ == '__main__':
    q = Queue()
    print("This machine has {0} CPUs".format(str(multiprocessing.cpu_count())))

    var = Value('I', 1, lock=False)

    # generates error
    q.put(var)

    print("Parent process puts item on queue with id " + str(id(var)))

    process = Process(target=child_process, args=(var, q))
    process.start()
    process.join()
```

Back
Queues & Pipes

Next
Locks & Reentrant Lock

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/sharing-state__multiprocessing__python-concurrency-for-senior-engineering-interviews)