Queues & Pipes

This lesson discusses constructs that can be used for inter-process communication.

Queues & Pipes

There are two ways that processes can communicate between themselves:

- Queues
- Pipes

Queues

The multiprocessing module offers three types of queues which are all FIFO structures based on the **queue** module's Queue (queue.Queue) implementation in the standard library. These are:

- Simple Queue
- Queue
- Joinable Queue (a subclass of Queue)

The queues in the multiprocessing module can be shared among multiple processes. Remember the following:

- We can enqueue any element in the queue that is picklable.
- Queues are thread and process safe.
- If multiple processes enqueue objects at the same time in a queue, the receiver may receive them out of order. However, all the object enqueued by a single process are always received in order.
- The official documentation cautions that when an item is placed in an empty queue, there is a "infinitesimal" delay before the `empty()` method returns false.

The last two bullet points if bothersome can be avoided by using a `Queue` created by a `Manager`, that we will discuss in the later sections.

Consider the program below. The main process acts as a producer and fills up a queue with ten messages. The child processes each access the queue and consumes messages from it.

Using queues for interprocess communication

```
def child_process(q):
    count = 0
    while not q.empty():
        print(q.get())
        count += 1

    print("child process {0} processed {1} items from the queu
e".format(current_process().name, count), flush=True)

if __name__ == '__main__':
    multiprocessing.set_start_method("forkserver")
    q = Queue()

    random.seed()
    for _ in range(100):
        q.put(random.randrange(10))

    p1 = Process(target=child_process, args=(q,))
    p2 = Process(target=child_process, args=(q,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

```
from multiprocessing import Process, Queue, current_process
import multiprocessing, sys
import random


def child_process(q):
    count = 0
    while not q.empty():
        print(q.get())
        count += 1

    print("child process {0} processed {1} items from the queue".format(current_pro

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

    random.seed()
    for _ in range(100):
        q.put(random.randrange(10))

    p1 = Process(target=child_process, args=(q,))
    p2 = Process(target=child_process, args=(q,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

The above code snippet has a bug, can you try to spot it before reading further? If you execute the above code multiple times, it is likely that one of the runs will hang and an execution timeout will occur. Consider the queue has one remaining item and the following sequence occurs:

- Thread T1 checks if the queue is empty and finds it non-empty and proceeds to **line-9**

- Thread T2 which is running on a different processor makes the same empty check at the same time or before T1 gets a chance to execute the `get()` call and finds the queue non-empty
- Both T1 and T2 attempt a `get()`.
- As mentioned queue is thread and processor safe so only one process gets an item from the queue.
- `get()` is a blocking call and the other thread gets blocked and the program hangs.

This is a classic synchronization problem encountered both with threads of a single process and with multiple processes trying to access a shared data structure. In later sections, we'll see how to fix this issue.

Pipes

Pipes can be best thought of as a two way connection between two processes. Whatever is written to one end of the pipe can be retrieved from the other end of the pipe. If two threads or processes attempt to write to the same end of the pipe at the same time, the data can potentially become corrupt. One can make a pipe work with more than two processes using synchronization primitives but that isn't the intent of this class. Ideally, it should be used to establish a communication channel between two processes.

The Pipe constructor takes in a boolean value. If passed in False, the pipe acts as a one-way communication where one end can only send messages and the other can only receive messages. For example:

```
recv_conn, send_conn = Pipe(duplex=False)
```

The first argument **`recv_conn`** returned by the constructor can receive messages and the second argument **`send_conn`** can send messages. By default or if the constructor is passed-in True, the connection created is bidirectional.

Consider the program below, the function **`Pipe()`** returns two objects each representing one end of the pipe. The child process writes ten strings to the pipe which the parent prints on the console after retrieving them from the queue.

Using pipes for interprocess communication

```
def child_process(conn):
    for i in range(0, 10):
        conn.send("hello " + str(i + 1))
    conn.close()


if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=child_process, args=(child_conn,))
    p.start()

    for _ in range(0, 10):
        msg = parent_conn.recv()
        print(msg)

    parent_conn.close()
    p.join()
```

```
from multiprocessing import Process, Pipe
import time

def child_process(conn):
    for i in range(0, 10):
        conn.send("hello " + str(i + 1))
    conn.close()


if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=child_process, args=(child_conn,))
    p.start()
    time.sleep(3)

    for _ in range(0, 10):
        msg = parent_conn.recv()
        print(msg)

    parent_conn.close()
    p.join()
```

Non-blocking Method Calls

Note that in this lesson we have shown blocking **`get()`** call in case of queue and blocking **`recv()`** call in case of pipe, however both these classes offer non-blocking versions of respective methods.

As an example, we can fix the blocking bug we came across in the queue section by using a non-blocking version of the get method that accepts a timeout. The runnable code appears below, note that we enclose the get method in a **`try-except block`** because an empty queue raises an empty queue exception when using the non-blocking version of the get call.

```
from multiprocessing import Process, Queue, current_process
import multiprocessing, sys
import random


def child_process(q):
    count = 0
    while not q.empty():
        try:
            print(q.get(block=False, timeout=5))
            count += 1
        except:
            pass

    print("child process {0} processed {1} items from the queue".format(current_pro


if __name__ == '__main__':
    multiprocessing.set_start_method("fork")
    q = Queue()

    print("This machine has {0} CPUs".format(str(multiprocessing.cpu_count())))
    random.seed()
    for _ in range(100):
        q.put(random.randrange(10))

    p1 = Process(target=child_process, args=(q,))
    p2 = Process(target=child_process, args=(q,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

Back

Forkserver

Next

Sharing State

Mark as Completed

Ask a Question

Report an

Report an Issue

(https://discuss.educative.io/tag/queues-pipes__multiprocessing__python-concurrency-for-senior-engineering-interviews)