Chapter 32
Inter Thread/Process Synchronisation

32.1 Introduction

In this chapter we will look at several facilities supported by both the `threading` and `multiprocessing` libraries that allow for synchronisation and cooperation between Threads or Processes.

In the remainder of this chapter we will look at some of the ways in which Python supports synchronisation between multiple Threads and Processes. Note that most of the libraries are mirrored between threading and multiprocessing so that the same basic ideas hold for both approaches with, in the main, very similar APIs. However, you should *not* mix and match threads and processes. If you are using Threads then you should only use facilities from the `threading` library. In turn if you are using Processes than you should only use facilities in the `multiprocessing` library. The examples given in this chapter will use one or other of the technologies but are relevant for both approaches.

32.2 Using a Barrier

Using a `threading.Barrier` (or `multiprocessing.Barrier`) is one of the simplest ways in which the execution of a set of Threads (or Processes) can be synchronised.

The threads or processes involved in the barrier are known as the *parties* that are taking part in the barrier.

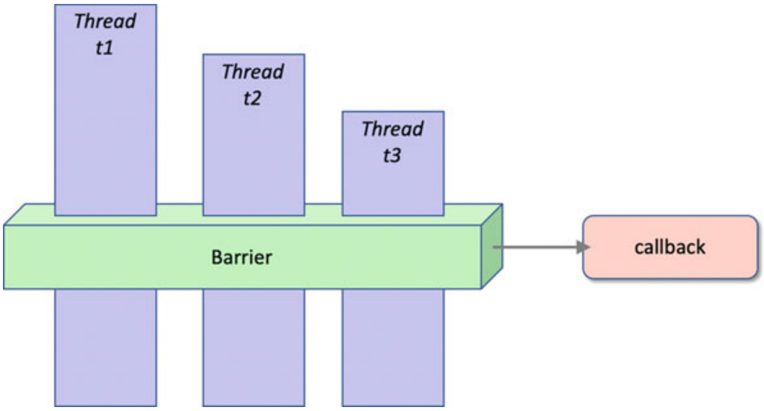
Each of the *parties* in the barrier can work independently until it reaches the barrier point in the code.

The barrier represents an end point that all parties must reach before any further behaviour can be triggered. At the point that all the parties reach the barrier it is possible to optionally trigger a *post-phase* action (also known as the barrier call-back). This post-phase action represents some behaviour that should be run when

© Springer Nature Switzerland AG 2019
J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_32

all parties reach the barrier but before allowing those parties to continue. The post-phase action (the callback) executes in a single thread (or process). Once it is completed then all the parties are unblocked and may continue.

This is illustrated in the following diagram. Threads t1, t2 and t3 are all involved in the barrier. When thread t1 reaches the barrier it must wait until it is released by the barrier. Similarly when t2 reaches the barrier it must wait. When t3 finally reaches the barrier the callback is invoked. Once the callback has completed the barrier releases all three threads which are then able to continue.



An example of using a Barrier object is given below. Note that the function being invoked in each `Thread` must also cooperate in using the barrier as the code will run up to the `barrier.wait()` method and then wait until all other threads have also reached this point before being allowed to continue.

The `Barrier` is a class that can be used to create a barrier object. When the `Barrier` class is instantiated, it can be provided with three parameters:
where

- `parties` the number of individual parties that will participate in the Barrier.
- `action` is a callable object (such as a function) which, when supplied, will be called after all the parties have entered the barrier and just prior to releasing them all.
- `timeout` If a 'timeout' is provided, it is used as the default for all subsequent `wait()` calls on the barrier.

Thus, in the following code

```
b = Barrier(3, action=callback)
```

Indicates that there will be three parties involved in the `Barrier` and that the `callback` function will be invoked when all three reach the barrier (however the timeout is left as the default value `None`).

The `Barrier` object is created outside of the Threads (or Processes) but must be made available to the function being executed by the `Thread` (or `Process`). The easiest way to handle this is to pass the barrier into the function as one of the parameters; this means that the function can be used with different barrier objects depending upon the context.

An example using the `Barrier` class with a set of Threads is given below:

```
from threading import Barrier, Thread
from time import sleep
from random import randint

def print_it(msg, barrier):
    print('print_it for:', msg)
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)
    sleep(randint(1, 6))
    print('Wait for barrier with:', msg)
    barrier.wait()
    print('Returning from print_it:', msg)

def callback():
    print('Callback Executing')

print('Main - Starting')

b = Barrier(3, callback)
t1 = Thread(target=print_it, args=('A', b))
t2 = Thread(target=print_it, args=('B', b))
t3 = Thread(target=print_it, args=('C', b))
t1.start()
t2.start()
t3.start()

print('Main - Done')
```

The output from this is:

```
Main - Starting
print_it for: A
print_it for: B
print_it for: C
ABC
Main - Done
ABCACBACBABCACBCABACBACBBAC
Wait for barrier with: B
Wait for barrier with: A
Wait for barrier with: C
Callback Executing
Returning from print_it: A
Returning from print_it: B
Returning from print_it: C
```

From this you can see that the `print_it()` function is run three times concurrently; all three invocations reach the `barrier.wait()` statement but in a different order to that in which they were started. Once the three have reached this point the `callback` function is executed before the `print_it()` function invocations can proceed.

The `Barrier` class itself provides several methods used to manage or find out information about the barrier:

Method	Description
`wait(timeout=None)`	Wait until all threads have notified the barrier (unless timeout is reached)—returns the number of threads that passed the barrier
`reset()`	Return barrier to default state
`abort()`	Put the barrier into a broken state
`parties`	Return the number of threads required to pass the barrier
`n_waiting`	Number of threads currently waiting

A `Barrier` object can be reused any number of times for the same number of Threads.

The above example could easily be changed to run using `Process` by altering the `import` statement and creating a set of Processes instead of Threads:

```
from multiprocessing import Barrier, Process
...
print('Main - Starting')
b = Barrier(3, callback)
t1 = Process(target=print_it, args=('A', b))
```

Note that you should only use threads with a `threading.Barrier`. In turn you should only use Processes with a `multiprocessing.Barrier`.

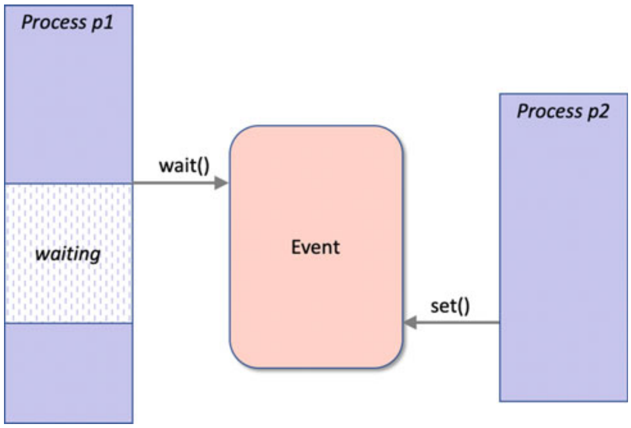
32.3 Event Signalling

Although the point of using multiple Threads or Processes is to execute separate operations concurrently, there are times when it is important to be able to allow two or more Threads or Processes to cooperate on the timing of their behaviour. The `Barrier` object presented above is a relatively high-level way to do this; however, in some cases finer grained control is required. The `threading.Event` or `multiprocessing.Event` classes can be used for this purpose.

An `Event` manages an internal flag that callers can either `set()` or `clear()`. Other threads can `wait()` for the flag to be `set()`, effectively blocking their own progress until allowed to continue by the `Event`. The internal flag is initially set to `False` which ensures that if a task gets to the `Event` before it is *set* then it must *wait*.

You can infact invoke wait with an optional timeout. If you do not include the optional timeout then `wait()` will wait forever while `wait(timeout)` will wait up to the timeout given in seconds. If the time out is reached, then the `wait` method returns `False`; otherwise `wait` returns `True`.

As an example, the following diagram illustrates two processes sharing an event object. The first process runs a function that waits for the event to be set. In turn the second process runs a function that will set the event and thus release the waiting process.



The following program implements the above scenario:

```
from multiprocessing import Process, Event
from time import sleep

def wait_for_event(event):
    print('wait_for_event - Entered and waiting')
    event_is_set = event.wait()
    print('wait_for_event - Event is set: ', event_is_set)

def set_event(event):
    print('set_event - Entered but about to sleep')
    sleep(5)
    print('set_event - Waking up and setting event')
    event.set()
    print('set_event - Event set')

print('Starting')

# Create the event object
event = Event()

# Start a Process to wait for the event notification
p1 = Process(target=wait_for_event, args=[event])
p1.start()

# Set up a process to set the event
p2 = Process(target=set_event, args=[event])
p2.start()

# Wait for the first process to complete
p1.join()

print('Done')
```

The output from this program is:

```
Starting
wait_for_event - Entered and waiting
set_event - Entered but about to sleep
set_event - Waking up and setting event
set_event - Event set
wait_for_event - Event is set:  True
Done
```

To change this to use Threads we would merely need to change the import and to create two Threads:

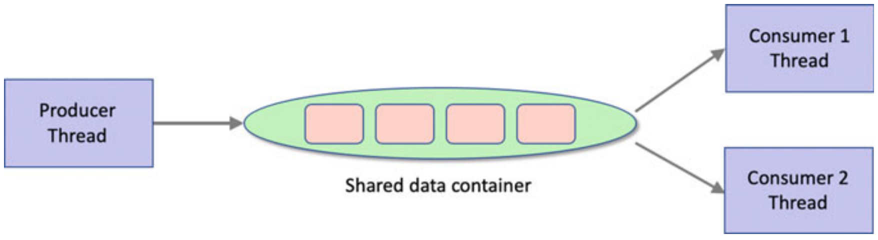
```
from threading import Thread, Event
...
print('Starting')
event = Event()
t1 = Thread(target=wait_for_event, args=[event])
t1.start()
t2 = Thread(target=set_event, args=[event])
t2.start()
t1.join()
print('Done')
```

32.4 Synchronising Concurrent Code

It is not uncommon to need to ensure that critical regions of code are protected from concurrent execution by multiple Threads or Processes. These blocks of code typically involve the modification of, or access to, shared data. It is therefore necessary to ensure that only one Thread or Process is updating a shared object at a time and that consumer threads or processes are blocked while this update is occurring.

This situation is most common where one or more Threads or Processes are the producers of data and one or more other Threads or Processes are the consumers of that data.

This is illustrated in the following diagram.



In this diagram the *Producer* is running in its own Thread (although it could also run in a separate Process) and places data onto some common shared data container. Subsequently a number of independent *Consumers* can consume that data when it is available and when they are free to process the data. However, there is no point in the consumers repeatedly checking the container for data as that would be a waste of resources (for example in terms of executing code on a processor and of context switching between multiple Threads or Processes).

We therefore need some form of *notification* or *synchronisation* between the Producer and the Consumer to manage this situation.

Python provides several classes in the threading (and also in the multi-processing) library that can be used to manage critical code blocks. These classes include Lock, Condition and Semaphore.

32.5 Python Locks

The Lock class defined (both in the threading and the multiprocessing libraries) provides a mechanism for synchronising access to a block of code. The Lock object can be in one of two states *locked* and *unlocked* (with the initial state being *unlocked*). The Lock grants access to a single thread at a time; other threads must wait for the Lock to become free before progressing.

The Lock class provides two basic methods for acquiring the lock (acquire()) and releasing (release()) the lock.

- When the state of the Lock object is unlocked, then acquire() changes the state to locked and returns immediately.
- When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns.
- The release() method should only be called in the *locked* state; it changes the state to *unlocked* and returns immediately. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

An example of using a Lock object is shown below:

```
from threading import Thread, Lock

class SharedData(object):
    def __init__(self):
        self.value = 0
        self.lock = Lock()

    def read_value(self):
        try:
            print('read_value Acquiring Lock')
            self.lock.acquire()
            return self.value
        finally:
            print('read_value releasing Lock')
            self.lock.release()

    def change_value(self):
        print('change_value acquiring lock')
        with self.lock:
            self.value = self.value + 1
        print('change_value lock released')
```

The SharedData class presented above uses locks to control access to critical blocks of code, specifically to the read_value() and the change_value() methods. The Lock object is held internally to the ShareData object and both methods attempt to acquire the lock before performing their behavior but must then release the lock after use.

The read_value() method does this explicitly using try: finally: blocks while the change_value() method uses a with statement (as the Lock type supports the *Context Manager Protocol*). Both approaches achieve the same result but the with statement style is more concise.

The SharedData class is used below with two simple functions. In this case the SharedData object has been defined as a *global* variable but it could also have been passed into the reader() and updater() functions as an argument. Both the reader and updater functions loop, attempting to call the read_value() and change_value() methods on the shared_data object.

As both methods use a lock to control access to the methods, only one thread can gain access to the locked area at a time. This means that the reader() function may start to read data before the updater() function has changed the data (or vice versa).

This is indicated by the output where the *reader* thread accesses the value ‘0’ twice before the *updater* records the value ‘1’. However, the updater() function runs a second time before the reader gains access to locked block of code which is why the value 2 is missed. Depending upon the application this may or may not be an issue.

```
shared_data = SharedData()

def reader():
    while True:
        print(shared_data.read_value())

def updater():
    while True:
        shared_data.change_value()

print('Starting')

t1 = Thread(target=reader)
t2 = Thread(target=updater)

t1.start()
t2.start()

print('Done')
```

The output from this is:

```
Starting
read_value Acquiring Lock
read_value releasing Lock
0
read_value Acquiring Lock
read_value releasing Lock
0
Done
change_value acquiring lock
change_value lock released
1
change_value acquiring lock
change_value lock released
change_value acquiring lock
change_value lock released
3
change_value acquiring lock
change_value lock released
4
```

Lock objects can only be acquired once; if a thread attempts to acquire a lock on the same Lock object more than once then a RuntimeError is thrown.

If it is necessary to re-acquire a lock on a Lock object then the threading.RLock class should be used. This is a *Re-entrant* Lock and allows the same Thread (or Process) to acquire a lock multiple times. The code must however release the lock as many times as it has acquired it.

32.6 Python Conditions

Conditions can be used to synchronise the interaction between two or more Threads or Processes. Conditions objects support the concept of a notification model; ideal for a shared data resource being accessed by multiple consumers and producers.

A `Condition` can be used to notify one or all of the waiting Threads or Processes that they can proceed (for example to read data from a shared resource). The methods available that support this are:

- `notify()` notifies one waiting thread which can then continue
- `notify_all()` notifies all waiting threads that they can continue
- `wait()` causes a thread to wait until it has been notified that it can continue

A `Condition` is always associated with an *internal* lock which must be acquired and released before the `wait()` and `notify()` methods can be called. The `Condition` supports the *Context Manager Protocol* and can therefore be used via a `with` statement (which is the most typical way to use a `Condition`) to obtain this lock. For example, to obtain the condition lock and call the wait method we might write:

```
with condition:
    condition.wait()
    print('Now we can proceed')
```

The condition object is used in the following example to illustrate how a producer thread and two consumer threads can cooperate. A `DataResource` class has been defined which will hold an item of data that will be shared between a consumer and a set of producers. It also (internally) defines a `Condition` attribute. Note that this means that the `Condition` is completely internalised to the `DataResource` class; external code does not need to know, or be concerned with, the `Condition` and its use. Instead external code can merely call the `consumer()` and `producer()` functions in separate Threads as required.

The `consumer()` method uses a `with` statement to obtain the (internal) lock on the `Condition` object before waiting to be notified that the data is available. In turn the `producer()` method also uses a `with` statement to obtain a lock on the condition object before generating the data attribute value and then notifying anything *waiting* on the condition that they can proceed. Note that although the consumer method obtains a lock on the condition object; if it has to wait it will release the lock and re obtain the lock once it is notified that it can continue. This is a subtly that is often missed.

```
from threading import Thread, Condition, currentThread
from time import sleep
from random import randint

class DataResource:
    def __init__(self):
        print('DataResource - Initialising the empty data')
        self.data = None
        print('DataResource - Setting up the Condition object')
        self.condition = Condition()

    def consumer(self):
        """wait for the condition and use the resource"""
        print('DataResource - Starting consumer method in',
              currentThread().name)
        with self.condition:
            self.condition.wait()
            print('DataResource - Resource is available to',
                  currentThread().name)
            print('DataResource - Data read in',
                  currentThread().name, ':', self.data)

    def producer(self):
        """set up the resource to be used by the consumer"""
        print('DataResource - Starting producer method')
        with self.condition:
            print('DataResource - Producer setting data')
            self.data = randint(1, 100)
            print('DataResource - Producer notifying all
waiting threads')
            self.condition.notifyAll()

print('Main - Starting')
print('Main - Creating the DataResource object')
resource = DataResource()

print('Main - Create the Consumer Threads')
c1 = Thread(target=resource.consumer)
c1.name = 'Consumer1'
c2 = Thread(target=resource.consumer)
c2.name = 'Consumer2'
print('Main - Create the Producer Thread')
p = Thread(target=resource.producer)

print('Main - Starting consumer threads')
c1.start()
c2.start()
sleep(1)

print('Main - Starting producer thread')
p.start()

print('Main - Done')
```

The output from an example run of this program is:

```
Main - Starting
Main - Creating the DataResource object
DataResource - Initialising the empty data
DataResource - Setting up the Condition object
Main - Create the Consumer Threads
Main - Create the Producer Thread
Main - Starting consumer threads
DataResource - Starting consumer method in Consumer1
DataResource - Starting consumer method in Consumer2
Main - Starting producer thread
DataResource - Starting producer method
DataResource - Producer setting data
Main - Done
DataResource - Producer notifying all waiting threads
DataResource - Resource is available to Consumer1
DataResource - Data read in Consumer1 : 36
DataResource - Resource is available to Consumer2
DataResource - Data read in Consumer2 : 36
```

32.7 Python Semaphores

The Python `Semaphore` class implements Dijkstra's counting semaphore model.

In general, a semaphore is like an integer variable, its value is intended to represent a number of available resources of some kind. There are typically two operations available on a semaphore; these operations are `acquire()` and `release()` (although in some libraries Dijkstra's original names of `p()` and `v()` are used, these operation names are based on the original Dutch phrases).

- The `acquire()` operation subtracts one from the value of the semaphore, unless the value is 0, in which case it blocks the calling thread until the semaphore's value increases above 0 again.
- The `signal()` operation adds one to the value, indicating a new instance of the resource has been added to the pool.

Both the `threading.Semaphore` and the `multiprocessing.Semaphore` classes also supports the *Context Management Protocol.*

An optional parameter used with the Semaphore constructor gives the initial value for the internal counter; it defaults to 1. If the value given is less than 0, `ValueError` is raised.

The following example illustrates 5 different Threads all running the same `worker()` function. The `worker()` function attempts to acquire a semaphore; if it does then it continues into the `with` statement block; if it doesn't, it waits until it can acquire it. As the semaphore is initialised to 2 there can only be two threads that can acquire the `Semaphore` at a time.

The sample program however, starts up five threads, this therefore means that the first 2 running Threads will acquire the semaphore and the remaining thee will have to wait to acquire the semaphore. Once the first two release the semaphore a further two can acquire it and so on.

```
from threading import Thread, Semaphore, currentThread
from time import sleep

def worker(semaphore):
    with semaphore:
        print(currentThread().getName() + " - entered")
        sleep(0.5)
        print(currentThread().getName() + " - exiting")

print('MainThread - Starting')

semaphore = Semaphore(2)
for i in range(0, 5):
    thread = Thread(name='T' + str(i),
                    target=worker,
                    args=[semaphore])
    thread.start()

print('MainThread - Done')
```

The output from a run of this program is given below:

```
MainThread - Starting
T0 - entered
T1 - entered
MainThread - Done
T0 - exiting
T2 - entered
T1 - exiting
T3 - entered
T2 - exiting
T4 - entered
T3 - exiting
T4 - exiting
```

32.8 The Concurrent Queue Class

As might be expected the model where a producer `Thread` or `Process` generates data to be processed by one or more Consumer Threads or Processes is so common that a higher level abstraction is provided in Python than the use of Locks, Conditions or Semaphores; this is the blocking queue model implemented by the `threading.Queue` or `multiprocessing.Queue` classes.

Both these `Queue` classes are Thread and Process safe. That is they work appropriately (using internal locks) to manage data access from concurrent Threads or Processes.

An example of using a `Queue` to exchange data between a worker process and the main process is shown below.

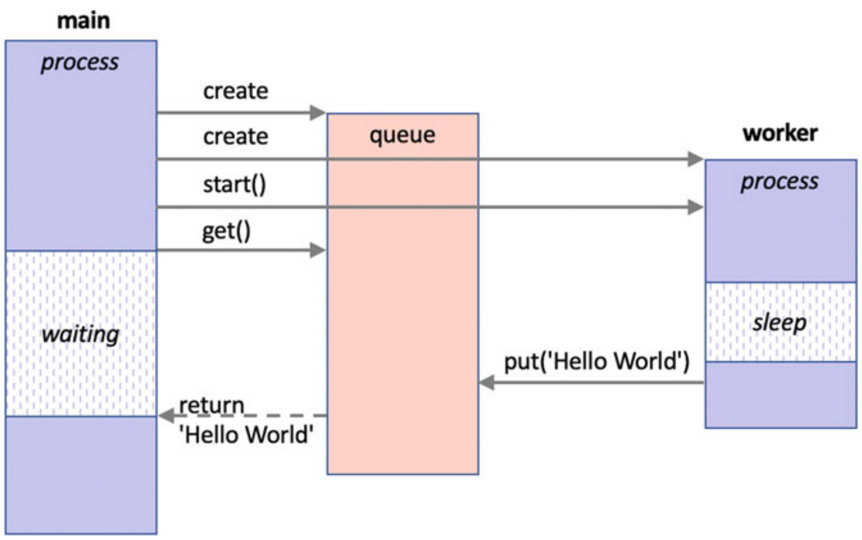
The *worker* process executes the `worker()` function sleeping, for 2 s before putting a string 'Hello World' on the queue. The main application function sets up the queue and creates the process. The queue is passed into the process as one of its arguments. The process is then started. The main process then waits until data is available on the queue via the (blocking) `get()` methods. Once the data is available it is retrieved and printed out before the main process terminates.

```
from multiprocessing import Process, Queue
from time import sleep

def worker(queue):
    print('Worker - going to sleep')
    sleep(2)
    print('Worker - woken up and putting data on queue')
    queue.put('Hello World')

def main():
    print('Main - Starting')
    queue = Queue()
    p = Process(target=worker, args=[queue])
    print('Main - Starting the process')
    p.start()
    print('Main - waiting for data')
    print(queue.get())
    print('Main - Done')

if __name__ == '__main__':
    main()
```

The output from this is shown below:

```
Main - Starting
Main - Starting the process
Main - wait for data
Worker - going to sleep
Worker - woken up and putting data on queue
Hello World
Main – Done
```

However, this does not make it that clear how the execution of the two processes interweaves. The following diagram illustrates this graphically:



In the above diagram the main process waits for a result to be returned from the queue following the call to the `get()` method; as it is waiting it is not using any system resources. In turn the worker process sleeps for two seconds before putting some data onto the queue (via `put('Hello World')`). After this value is sent to the `Queue` the value is returned to the main process which is woken up (moved out of the waiting state) and can continue to process the rest of the main function.

32.9 Online Resources

See the following online resources for information discussed in this chapter:

- https://docs.python.org/3/library/threading.html for information on Thread based barriers, locks, conditions, semaphores and events.
- https://docs.python.org/3/library/multiprocessing.html for information on Process based barriers, locks, conditions, semaphores and events.
- https://en.wikipedia.org/wiki/Semaphore_programming Semaphore programming model.

32.10 Exercises

The aim of this exercise is to implement a concurrent version of a `Stack` based container/collection.

It should be possible to safely add data to the stack and *pop* data off the stack using multiple Threads.

It should follow a similar pattern to the `Queue` class described above but support the First In Last Out (FILO) behaviour of a Stack and be usable with any number of producer and consumer threads (you can ignore processes for this exercise).

The key to implementing the `Stack` is to remember that no data can be read from the stack until there is some data to access; it is therefore necessary to wait for data to become available and then to read it. However, it is a producer thread that will provide that data and then inform any waiting threads that there is not data available. You can implement this in any way you wish; however a common solution is to use a `Condition`.

To illustrate this idea, the following test program can be used to verify the behaviour of your `Stack`:

```
from stack.Stack import Stack
from time import sleep
from threading import Thread

def producer(stack):
    for i in range(0,6):
        data = 'Task' + str(i)
        print('Producer pushing:', data)
        stack.push(data)
        sleep(2)

def consumer(label, stack):
    while True:
        print(label, 'stack.pop():', stack.pop())

print('Create shared stack')
stack = Stack()
print('Stack:', stack)

print('Creating and starting consumer threads')
consumer1 = Thread(target=consumer, args=('Consumer1', stack))
consumer2 = Thread(target=consumer, args=('Consumer2', stack))
consumer3 = Thread(target=consumer, args=('Consumer3', stack))
consumer1.start()
consumer2.start()
consumer3.start()

print('Creating and starting producer thread')
producer = Thread(target=producer, args=[stack])
producer.start()
```

The output generated from this sample program (which includes print statements from the `Stack`) is given below:

```
Create shared stack
Stack: Stack: []
Creating and starting consumer threads
Creating and starting producer thread
Producer pushing: Task0
Consumer1 stack.pop(): Task0
Producer pushing: Task1
Consumer2 stack.pop(): Task1
Producer pushing: Task2
Consumer3 stack.pop(): Task2
Producer pushing: Task3
Consumer1 stack.pop(): Task3
Producer pushing: Task4
Consumer2 stack.pop(): Task4
Producer pushing: Task5
Consumer3 stack.pop(): Task5
```