Thread synchronization with a condition

A condition identifies a change of state in the application. This is a synchronization mechanism where a thread waits for a specific condition and another thread notifies that this condition has taken place. Once the condition takes place, the thread acquires the lock to get exclusive access to the shared resource.

Getting ready

A good way to illustrate this mechanism is by looking again at a producer/consumer problem. The class producer writes to a buffer as long as it is not full, and the class consumer takes the data from the buffer (eliminating them from the latter), as long as the buffer is full. The class producer will notify the consumer that the buffer is not empty, while the consumer will report to the producer that the buffer is not full.

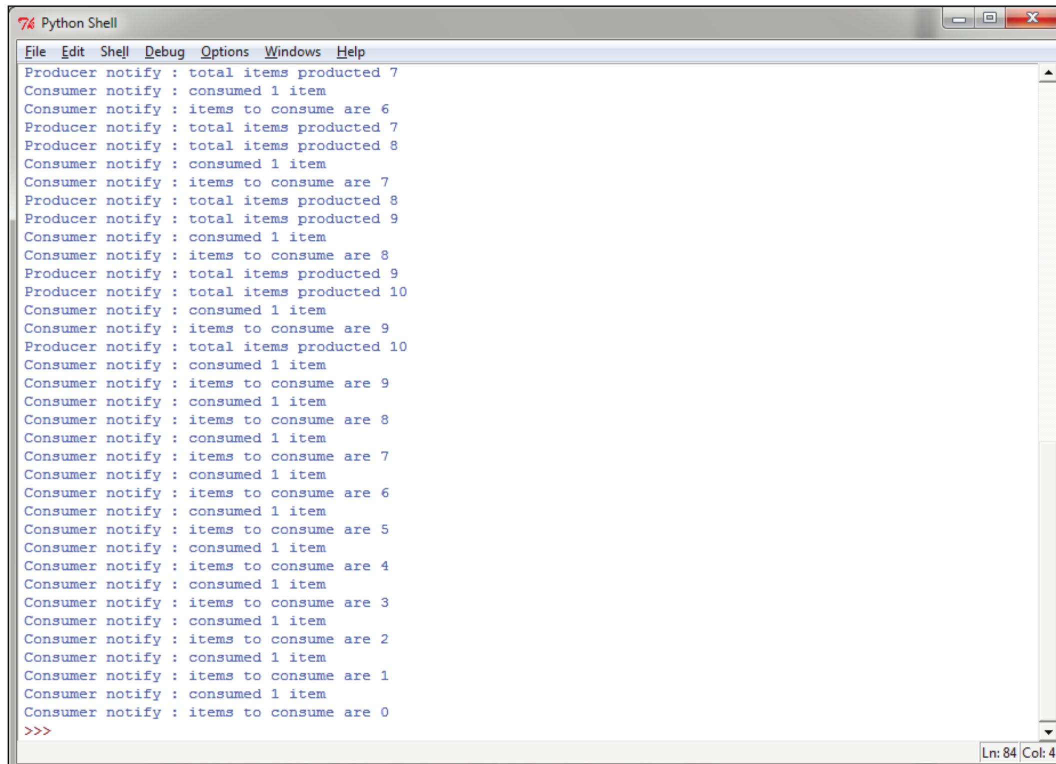
How to do it...

To show you the condition mechanism, we will again use the consumer producer model:

```
from threading import Thread, Condition
import time

items = []
condition = Condition()

class consumer(Thread):
  def __init__(self):
    Thread.__init__(self)

  def consume(self):
    global condition
    global items

    condition.acquire()
    if len(items) == 0:
      condition.wait()
      print("Consumer notify : no item to consume")
    items.pop()
    print("Consumer notify : consumed 1 item")
    print("Consumer notify : items to consume are "\
                  + str(len(items)))
```

```
    condition.notify()
    condition.release()

  def run(self):
    for i in range(0,20):
      time.sleep(10)
      self.consume()


class producer(Thread):
  def __init__(self):
    Thread.__init__(self)

  def produce(self):
    global condition
    global items

    condition.acquire()
    if len(items) == 10:
      condition.wait()
      print("Producer notify : items producted are "\
                        + str(len(items)))
      print("Producer notify : stop the production!!")
    items.append(1)
    print("Producer notify : total items producted "\
                  + str(len(items)))
    condition.notify()
    condition.release()

  def run(self):
    for i in range(0,20):
      time.sleep(5)
      self.produce()

if __name__ == "__main__":
        producer = producer()
        consumer = consumer()
        producer.start()
        consumer.start()
        producer.join()
        consumer.join()
```

This is the result that we get after a single run:



How it works...

The class consumer acquires the shared resource that is modeled through the list `items[]`:

```
condition.acquire()
```

If the length of the list is equal to `0`, the consumer is placed in a waiting state:

```
if len(items) == 0:
   condition.wait()
```

Otherwise, it makes a `pop` operation from the items list:

```
items.pop()
```

So, the consumer's state is notified to the producer and the shared resource is released:

```
condition.notify()
condition.release()
```

The class producer acquires the shared resource and then it verifies that the list is completely full (in our example, we place the maximum number of items, 10, that can be contained in the items list). If the list is full, then the producer is placed in the wait state until the list is consumed:

```
condition.acquire()
if len(items) == 10:
   condition.wait()
```

If the list is not full, a single item is added. The state is notified and the resource is released:

```
condition.notify()
condition.release()
```

There's more...

It's interesting to see the Python internals for the condition synchronizations mechanism. The internal class `_Condition` creates a `RLock()` object if no existing lock is passed to the class's constructor. Also, the lock will be managed when `acquire()` and `released()` are called:

```
class _Condition(_Verbose):
    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self.__lock = lock
```

Thread synchronization with an event

Events are objects that are used for communication between threads. A thread waits for a signal while another thread outputs it. Basically, an event object manages an internal flag that can be set to `true` with the `set()` method and reset to `false` with the `clear()` method. The `wait()` method blocks until the flag is `true`.

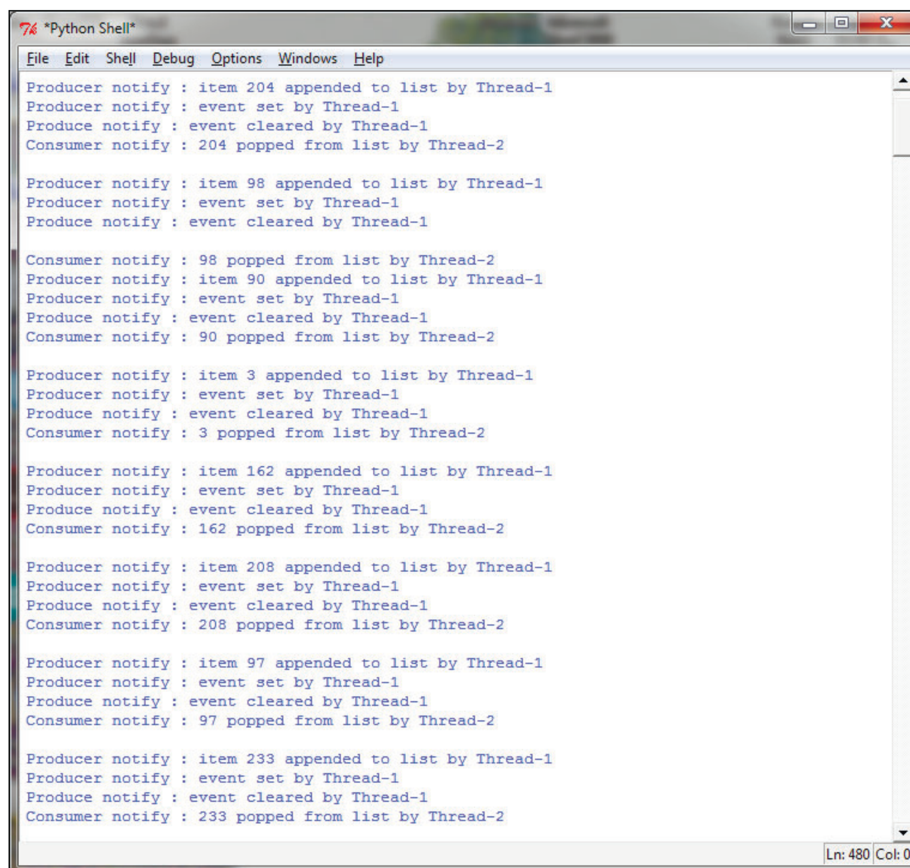
How to do it...

To understand the thread synchronization through the event object, let's take a look again at the producer/consumer problem:

```
import time
from threading import Thread, Event
import random

items = []
event = Event()

class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        while True:
            time.sleep(2)
            self.event.wait()
            item = self.items.pop()
            print ('Consumer notify : %d popped from list by %s'\
                   %(item, self.name))


class producer(Thread):
    def __init__(self, integers, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        global item
        for i in range(100):
            time.sleep(2)
            item = random.randint(0, 256)
            self.items.append(item)
            print ('Producer notify : item N° %d appended \
                   to list by %s'\
                   % (item, self.name))
            print ('Producer notify : event set by %s'\
                   % self.name)
```

```
                self.event.set()
                print ('Produce notify : event cleared by %s \n'\
                       % self.name)
                self.event.clear()


if __name__ == '__main__':
    t1 = producer(items, event)
    t2 = consumer(items, event)
    t1.start()
    t2.start()
    t1.join()
        t2.join()
```

This is the output that we get when we run the program. The `t1` thread appends a value to the list and then sets the event to notify the consumer. The consumer's call to `wait()` stops blocking and the integer is retrieved from the list.



How it works...

The `producer` class is initialized with the list of items and the `Event()` function. Unlike the example with condition objects, the item list is not global, but it is passed as a parameter:

```
class consumer(Thread):
    def __init__(self, items, event):
        Thread._init__(self)
        self.items = items
        self.event = event
```

In the run method for each item that is created, the `producer` class appends it to the list of items and then notifies the event. There are two steps that you need to take for this and the first step is as follows:

```
self.event.set()
```

The second step is:

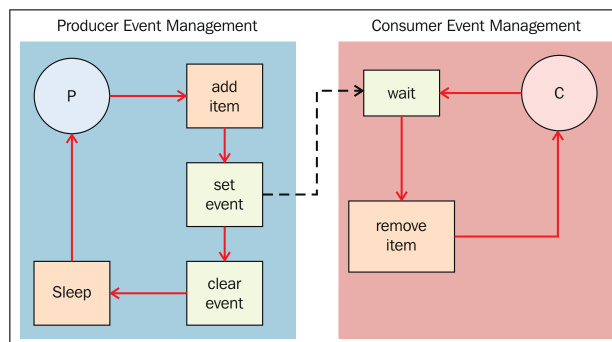
```
self.event.clear()
```

The `consumer` class is initialized with the list of items and the `Event()` function.

In the run method, the consumer waits for a new item to consume. When the item arrives, it is popped from the item list:

```
def run(self):
    while True:
       time.sleep(2)
       self.event.wait()
       item = self.items.pop()
       print ('Consumer notify : %d popped from list by %s' %
              (item, self.name))
```

All the operations between the `producer` and the `consumer` classes can be easily resumed with the help of the following schema:



Thread synchronization with event objects

Using the with statement

Python's `with` statement was introduced in Python 2.5. It's useful when you have two related operations that must be executed as a pair with a block of code in between. Also, with the `with` statement, you can allocate and release some resource exactly where you need it; for this reason, the `with` statement is called a context manager. In the threading module, all the objects provided by the `acquire()` and `release()` methods may be used in a `with` statement block.

So the following objects can be used as context managers for a `with` statement:

- Lock
- RLock
- Condition
- Semaphore

Getting ready

In this example, we simply test all the objects using the `with` statement.

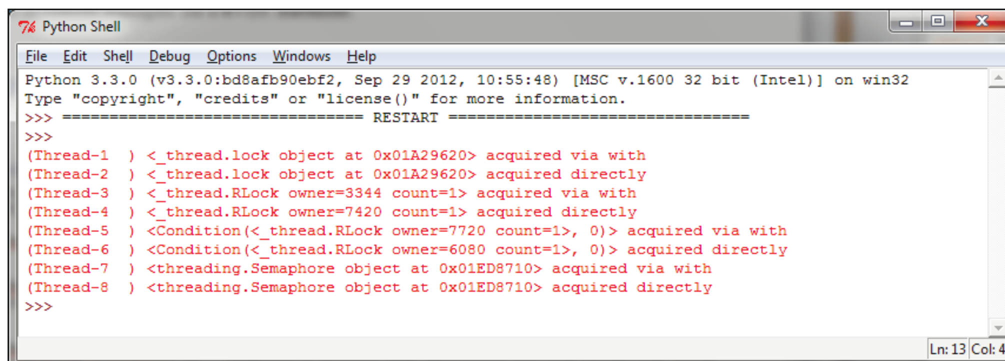
How to do it...

This example shows the basic use of the `with` statement. We have a set with the most important synchronization primitives. So, we test them by calling each one with the `with` statement:

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                   format='(%(threadName)-10s) %(message)s',)

def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with'  %statement)

def threading_not_with(statement):
    statement.acquire()
    try:
        logging.debug('%s acquired directly' %statement )
    finally:
        statement.release()
```

```
if __name__ == '__main__':

#let's create a test battery
    lock = threading.Lock()
    rlock = threading.RLock()
    condition = threading.Condition()
    mutex = threading.Semaphore(1)
    threading_synchronization_list = \
      [lock, rlock, condition, mutex]

#in the for cycle we call the threading_with
# e threading_no_with function
    for statement in threading_synchronization_list :
        t1 = threading.Thread(target=threading_with,
             args=(statement,))
        t2 = threading.Thread(target=threading_not_with,
             args=(statement,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

The output shows the use of the `with` statement for each function and also where it is not used:



How it works...

In the main program, we have defined a list, `threading_synchronization_list`, of thread communication directives that are to be tested:

```
lock = threading.Lock()
rlock = threading.RLock()
condition = threading.Condition()
mutex = threading.Semaphore(1)
threading_synchronization_list = \

  [lock, rlock, condition, mutex]
```

After defining them, we pass each object in the `for` cycle:

```
for statement in threading_synchronization_list :
        t1 = threading.Thread(target=threading_with,
             args=(statement,))
        t2 = threading.Thread(target=threading_not_with,
             args=(statement,))
```

Finally, we have two target functions, in which the `threading_with` tests the `with` statement:

```
def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with'  %statement)
```

There's more...

In the following example we have used the Python support for logging, as we can see:

```
logging.basicConfig(level=logging.DEBUG,
                   format='(%(threadName)-10s) %(message)s',)
```

It embeds the thread name in every log message using the formatter code's `%(threadName)s` statement. The logging module is thread-safe, so the messages from different threads are kept distinct in the output.

Thread communication using a queue

As discussed earlier, threading can be complicated when threads need to share data or resources. As we saw, the Python threading module provides many synchronization primitives, including semaphores, condition variables, events, and locks. While these options exist, it is considered a best practice to instead concentrate on using the module queue. Queues are much easier to deal with and make threaded programming considerably safer, as they effectively funnel all access to a resource of a single thread and allow a cleaner and more readable design pattern.

We will simply consider these four queue methods:

- `put()`: This puts an item in the queue
- `get()`: This removes and returns an item from the queue
- `task_done()`: This needs to be called each time an item has been processed
- `join()`: This blocks until all items have been processed

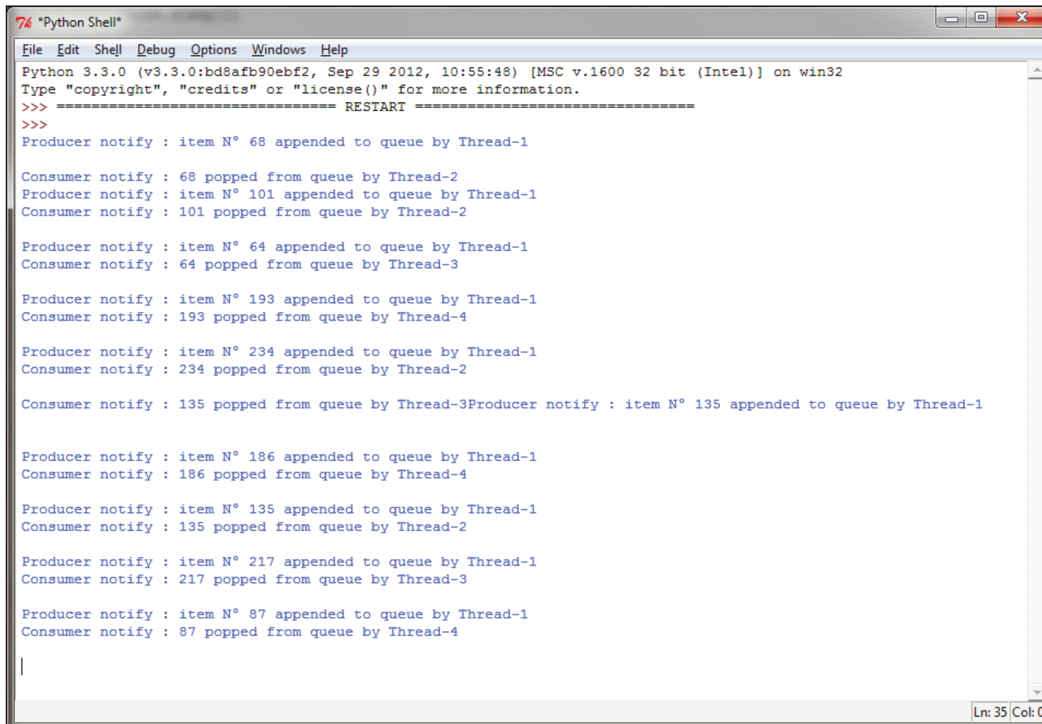
How to do it…

In this example, we will see how to use the threading module with the queue module. Also, we have two entities here that try to share a common resource, a queue. The code is as follows:

```
from threading import Thread, Event
from queue import Queue
import time
import random


class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
```

```
            item = random.randint(0, 256)
            self.queue.put(item)
            print ('Producer notify: item N°%d appended to queue by %s
                   \n'\
                   % (item, self.name))
            time.sleep(1)


class consumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            item = self.queue.get()
            print ('Consumer notify : %d popped from queue by %s'\
                   % (item, self.name))
            self.queue.task_done()


if __name__ == '__main__':
        queue = Queue()
        t1 = producer(queue)
        t2 = consumer(queue)
        t3 = consumer(queue)
        t4 = consumer(queue)
        t1.start()
        t2.start()
        t3.start()
        t4.start()
        t1.join()
        t2.join()
        t3.join()
        t4.join()
```

After running the code, you should have an output similar to this:



How it works...

First, the `producer` class. We don't need to pass the integers list because we use the queue to store the integers that are generated:

```
class producer(Thread):
    def _init__(self, queue):
        Thread._init__(self)
        self.queue = queue
```

The thread in the `producer` class generates integers and puts them in the queue in a `for` loop:

```
def run(self) :
        for i in range(100):
            item = random.randint(0, 256)
            self.queue.put(item)
```

The producer uses `Queue.put(item[, block[, timeout]])` to insert data into the queue. It has the logic to acquire the lock before inserting data in a queue.

There are two possibilities:

- If optional args `block` is `true` and `timeout` is `None` (this is the default case that we used in the example), it is necessary for us to block until a free slot is available. If timeout is a positive number, it blocks at most `timeout` seconds and raises the full exception if no free slot is available within that time.
- If the block is `false`, put an item in the queue if a free slot is immediately available; otherwise, raise the full exception (timeout is ignored in this case). Here, `put()` checks whether the queue is full and then calls `wait()` internally and after this, the producer starts waiting.

Next is the `consumer` class. The thread gets the integer from the queue and indicates that it is done working on it using `task_done()`:

```
def run(self):
        while True:
            item = self.queue.get()
            self.queue.task_done()
```

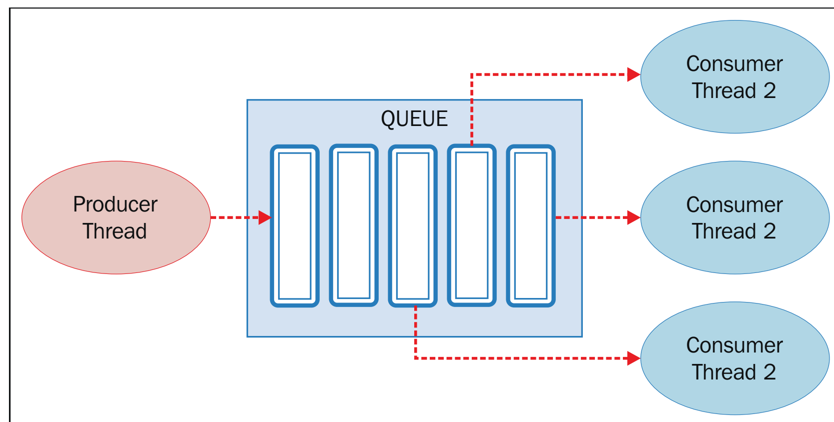
The consumer uses `Queue.get([block[, timeout]])` and acquires the lock before removing data from the queue. If the queue is empty, it puts the consumer in a waiting state.

Finally, in the main, we create the `t` thread for the producer and three threads, `t1`, `t2`, and `t3` for the `consumer` class:

```
if __name__ == '__main__':
        queue = Queue()
        t = producer(queue)
        t1 = consumer(queue)
        t2 = consumer(queue)
        t3 = consumer(queue)
```

```
t.start()
t1.start()
t2.start()
t3.start()

t.join()
t1.join()
t2.join()
t3.join()
```

All the operations between the `producer` class and the `consumer` class can easily be resumed with the following schema:



Thread synchronization with the queue module

Evaluating the performance of multithread applications

In this recipe, we will verify the impact of the GIL, evaluating the performance of a multithread application. The GIL, as described in the previous chapter, is the lock introduced by the CPython interpreter. The GIL prevents parallel execution of multiple threads in the interpreter. Before being executed each thread must wait for the GIL to release the thread that is running. In fact, the interpreter forces the executing thread to acquire the GIL before it accesses anything on the interpreter itself as the stack and instances of Python objects. This is precisely the purpose of GIL—it prevents concurrent access to Python objects from different threads. The GIL then protects the memory of the interpreter and makes the garbage work in the right manner. The fact is that the GIL prevents the programmer from improving the performance by executing threads in parallel. If we remove the GIL from the CPython interpreter, the threads would be executed in parallel. The GIL does not prevent a process from running on a different processor, it simply allows only one thread at a time to turn inside the interpreter.