Python Threading

processes

Each program that is running counts as a **process** in Unix terminology. Multiple copies of a program count as multiple processes.

The processes “take turns” running, of fixed size, say for concreteness 30 milliseconds. After a process has run for the given time, a hardware timer emits an interrupted which causes the OS to run to suspend the process: the process itself has been pre-empted.

The OS saves the current state of the interrupted process so it can be resumed later, then selects the next process to give a turn to. This is known as a **context switch**: the context in which the CPU is running has switched from one process to another.

This cycle repeats: any given process will keep getting turns, and eventually will finish. A turn is called a **quantum** or **timeslice**.

The OS maintains a process table, listing all current processes. Each process will be shown as currently being in either **Run** (ready) state or **Sleep** (waiting) state.

Python Threading

threads

A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.

The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

From Wikipedia

Threads are sometimes called “**lightweight**” processes, because they occupy much less memory and take less time to create, than do processes

In particular, a major difference between ordinary processes and threads is that although each thread has its own local variables, just as is the case for a process, the global variables of the parent program in a threaded environment are shared by all threads, and serve as the main method of **communication** between the threads.

Python threads are accessible via two modules, **thread.py** and **threading.py**. The former is more primitive, the second gives highest level functions.

Python Threading

threads module

The Python's **thread** module provides low-level primitives for working with multiple threads. For synchronization, simple locks (also called binary semaphores) are provided.

It defines some constant and functions, in particular:

thread.start_new_thread(function, args[, kwargs])

this function start a new thread and return its identifier. The thread executes the function ***function*** with the argument list ***args*** (which must be a tuple, eventually use an empty tuple to call function without passing any parameter). The optional ***kwargs*** argument specifies a dictionary of keyword arguments.

When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

Python Threading

threads module

Here is an example of thread module usage:

```
import thread
import time

#function to be invoked as a thread
def print_time(threadname,delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count +=1
        print "%s: %s" % (threadname, time.ctime(time.time()))


#main program: generates two threads from print_time
try:
    thread.start_new_thread(print_time,('thread-1',2))
    thread.start_new_thread(print_time,('thread-2',4))
except:
    print 'unable to start thread'

while 1:
    pass
```

program's output

```
thread-1: Tue Oct 25 23:38:36 2016
thread-2: Tue Oct 25 23:38:38 2016
thread-1: Tue Oct 25 23:38:38 2016
thread-1: Tue Oct 25 23:38:40 2016
thread-2: Tue Oct 25 23:38:42 2016
thread-1: Tue Oct 25 23:38:42 2016
thread-1: Tue Oct 25 23:38:44 2016
thread-2: Tue Oct 25 23:38:46 2016
thread-2: Tue Oct 25 23:38:50 2016
thread-2: Tue Oct 25 23:38:54 2016
^CTraceback (most recent call last):
  File "thread-1.py", line 23, in <module>
    pass
KeyboardInterrupt
```

Python Threading

threading module

The Python's **threading** module constructs higher-level threading interfaces on top of the lower level thread module.

This module defines some functions, in particular:

threading.enumerate()	return a list of all Thread objects currently alive.
threading.activeCount()	return the number of Thread objects currently alive. The returned count is equal to the length of the list returned by enumerate().
threading.currentThread()	return the current Thread object

Python Threading

threading module

The threading also define the **threading.Thread** class that represents a thread object and which can be safely subclassed.

The methods provided by the Thread class are as follows:

run()	the run() method is the entry point for a thread
start()	this methods starts a thread by calling the run() method
join([time])	this method waits for a thread to terminate (optionally, the maximum waiting time can be specified)
isAlive()	this method checks whether a thread is still executing or not, returning a boolean value
getName()	this method returns the name of a thread
setName()	this method can be used to set the name of a thread

Python Threading

threading module

The simplest way to use a **Thread** object is to instantiate it with a target function passing it **arguments** to tell it what work to do and call the **start()** method to let it begin working:

```
import threading

def worker(num):
    print 'thread worker nr. %s' % num
    return

threads = []
for i in range(5):
    t = threading.Thread(target = worker, args = (i, ))
    threads.append(t)
    t.start()
```

program's output

```
thread worker nr. 0
thread worker nr. 1
thread worker nr. 2
thread worker nr. 3
thread worker nr. 4
```

Python Threading

threading module

Each `Thread` instance has a **name** with a default value that can be changed as the thread is created. Naming threads is useful in server processes with multiple service threads handling different operations.

```
import threading
import time

def worker2s():
    print threading.currentThread().getName(), 'starting...'
    time.sleep(2)
    print threading.currentThread().getName(), 'exiting...'

def worker3s():
    print threading.currentThread().getName(), 'starting...'
    time.sleep(3)
    print threading.currentThread().getName(), 'exiting...'


t1 = threading.Thread(target = worker3s, name = 'wait_3')
t2 = threading.Thread(target = worker2s, name = 'wait_2')
t3 = threading.Thread(target = worker2s)

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

program's output

```
wait_3 starting...
wait_2 starting...
Thread-1 starting...
wait_2 exiting...
Thread-1 exiting...
wait_3 exiting...
```

Python Threading

threading module

Instead of printing thread's messages, the **logging** module can be used to create debug records. The logging module supports embedding the thread name in every log message using the formatter code `%(threadName)s, thus making` easy to trace those messages back to their source.

```
import threading
import logging
import time

def worker2s():
    logging.debug('starting...')
    time.sleep(2)
    logging.debug('exiting...')

def worker3s():
    logging.debug('starting...')
    time.sleep(3)
    logging.debug('exiting...')


logging.basicConfig(level = logging.DEBUG, format = '[%(levelname)s] %(threadName)s %(message)s')

t1 = threading.Thread(target = worker3s, name = 'wait_3')
t2 = threading.Thread(target = worker2s, name = 'wait_2')
t3 = threading.Thread(target = worker2s)

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

program's output

```
[DEBUG] wait_3 starting...
[DEBUG] wait_2 starting...
[DEBUG] Thread-1 starting...
[DEBUG] wait_2 exiting...
[DEBUG] Thread-1 exiting...
[DEBUG] wait_3 exiting...
```

Python Threading

threading module

programs can also spawn a thread as a ***daemon*** that runs without blocking the main program from exiting; To mark a thread as a daemon, call its **`setDaemon()`** method passing it a boolean True value as argument (the default is for threads to not be daemons).

```python
import threading
import logging
import time

def daemon():
    logging.debug('starting...')
    time.sleep(2)
    logging.debug('exiting...')

def non_daemon():
    logging.debug('starting...')
    logging.debug('exiting...')


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

d = threading.Thread(target = daemon, name = 'daemon')
d.setDaemon(True)
t = threading.Thread(target = non_daemon, name = 'non_daemon')

d.start()
t.start()
```

program's output

```
daemon starting...
non_daemon starting...
non_daemon exiting...
```

the output does not include the "`Exiting`" message from the daemon thread, since all of the non-daemon threads (including the main thread) exit before the daemon thread wakes up from its two second sleep

Python Threading

threading module

To wait until a daemon thread has completed its work, use the **`join()`** method:

```
import threading
import logging
import time

def daemon():
    logging.debug('starting...')
    time.sleep(2)
    logging.debug('exiting...')

def non_daemon():
    logging.debug('starting...')
    logging.debug('exiting...')


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

d = threading.Thread(target = daemon, name = 'daemon')
d.setDaemon(True)
t = threading.Thread(target = non_daemon, name = 'non_daemon')

d.start()
t.start()

d.join()
t.join()
```

program's output

```
daemon starting...
non_daemon starting...
non_daemon exiting...
daemon exiting...
```

Python Threading

threading module

By default, the `join()` method blocks indefinitely. It is also possible to pass it a timeout argument (a float representing the number of seconds to wait for the thread to become inactive): if the thread does not complete within the timeout period, `join()` returns anyway

```
import threading
import logging
import time

def daemon():
    logging.debug('starting...')
    time.sleep(2)
    logging.debug('exiting...')

def non_daemon():
    logging.debug('starting...')
    logging.debug('exiting...')


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

d = threading.Thread(target = daemon, name = 'daemon')
d.setDaemon(True)
t = threading.Thread(target = non_daemon, name = 'non_daemon')

d.start()
t.start()

d.join(1)
print 'd.isAlive() value: ', d.isAlive()
t.join()
```

program's output

```
daemon starting...
non_daemon starting...
non_daemon exiting...
d.isAlive() value:  True
```

Python Threading

threading module

It is not necessary to retain an explicit handle to all of the daemon threads in order to ensure they have completed before exiting the main process; the **enumerate()** function of the threading module, returns a list of active Thread instances.
Notice that the list includes the current thread, and since joining the current thread is not allowed (it introduces a deadlock situation), it must be skipped.

```
import threading
import logging
import time
import random

def worker():
    pause = random.randint(1,5)
    logging.debug('sleeping %s' % pause)
    time.sleep(pause)
    logging.debug('exiting')
    return

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

for i in range(3):
    t = threading.Thread(target = worker)
    t.setDaemon(True)
    t.start()

maint = threading.currentThread()

for t in threading.enumerate():
    if t is maint:
        continue
    logging.debug('joining %s' % t.getName())
    t.join()
```

program's output

```
Thread-1 sleeping 5
Thread-2 sleeping 4
Thread-3 sleeping 2
MainThread joining Thread-1
Thread-3 exiting
Thread-2 exiting
Thread-1 exiting
MainThread joining Thread-3
MainThread joining Thread-2
```

Python Threading
threading module

Using the threading module, it's possible to implement a new thread creating a *sub-class* of the main Thread class. In this case:

- override the ***__init__*** constructor, to add additional attributes to the new thread;
- override the ***run()*** method to implement what the thread has to do when started.

```
import threading
import logging

class myThread(threading.Thread):
    def __init__(self, args = (), kwargs = None ):
        threading.Thread.__init__(self)
        self.args = args
        self.kwargs = kwargs

    def run(self):
        logging.debug('running with args: %s and kwargs: %s' % (self.args, self.kwargs))


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

for i in range(3):
    t = myThread((i,),{'a' : 1, 'b' : 2})
    t.start()
```

program's output

```
Thread-1 running with args: (0,) and kwargs: {'a': 1, 'b': 2}
Thread-2 running with args: (1,) and kwargs: {'a': 1, 'b': 2}
Thread-3 running with args: (2,) and kwargs: {'a': 1, 'b': 2}
```

Python Threading

threading module

One more example of *sub-classing* the threading.Thread class:

```
import logging
import threading
import time

class myThread(threading.Thread):
    def __init__(self, name, delay = 1, counter = 2):
        threading.Thread.__init__(self)
        self.name = name
        self.delay = delay
        self.counter = counter

    def run(self):
        logging.debug('starting')
        worker(self.delay, self.counter)
        logging.debug('exiting')

def worker(delay, counter):
    for i in range(counter):
        time.sleep(delay)
        logging.debug('iteration: %s' % i)

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

t1 = myThread('thd-1', 2, 3)
t2 = myThread('thd-2', 4, 3)

t1.start()
t2.start()
```

program's output

```
thd-1 2016-11-02 18:21:44,252 starting
thd-2 2016-11-02 18:21:44,253 starting
thd-1 2016-11-02 18:21:46,255 iteration: 0
thd-2 2016-11-02 18:21:48,257 iteration: 0
thd-1 2016-11-02 18:21:48,258 iteration: 1
thd-1 2016-11-02 18:21:50,261 iteration: 2
thd-1 2016-11-02 18:21:50,261 exiting
thd-2 2016-11-02 18:21:52,263 iteration: 1
thd-2 2016-11-02 18:21:56,266 iteration: 2
thd-2 2016-11-02 18:21:56,267 exiting
```

Python Threading

access to shared resources

One important issue when using threads is to avoid conflicts when more than one thread needs to access a single variable or other resource.

For example, consider a program that does some kind of processing, and keeps track of how many items it has processed:

```
counter = 0
def process_item(item):
    global counter
    ... do something with item ...
    counter += 1
```

If you call this function from more than one thread, you'll find that the counter isn't necessarily accurate. It works in most cases, but <u>sometimes</u> it misses one or more items.

The reason for this is that the increment operation is actually executed in three steps: the interpreter fetches the current value of the counter, then it calculates the new value, and finally, it writes the new value back. If another thread gets control after the current thread has fetched the variable, it may modify its value before the current thread does the same thing: since they're both seeing the same original value, only one item will be accounted for. And the inconsistency can be even more critical with non-trivial data structures.

So, operations that read a variable or attribute, modifies it, and then writes it back **are not thread-safe**.

Python Threading

access to shared resources

Locks are the most fundamental mechanism provided by the threading module to synchronize access to a shared resource.

At any time, a lock can be held by a single thread, or by no thread at all. If a thread attempts to hold a lock that's already held by some other thread, execution of the first thread is halted until the lock is released.

For each shared resource, create a Lock object. When you need to access that particular resource, call acquire() to hold the corresponding lock (this will wait for the lock to be released, if another thread already held it), then use the resource and finally call release() to release it:

```
lock = Lock()
lock.acquire()
... access shared resource
lock.release()
```

Python Threading

access to shared resources

For proper operation, it’s important to release the lock even if something goes wrong when accessing the resource. You can use **try-finally** or **with** statements for this purpose:

```
lock.acquire()
try:
    ... access shared resource
finally:
    lock.release() # release lock, no matter what
```

or:

```
with lock:
    ... access shared resource
```

The acquire method takes an optional wait flag, which can be used to avoid blocking if the lock is held by someone else. The method returns False if the lock was already held:

```
if not lock.acquire(False):
    ... failed to lock the resource
else:
try:
    ... access shared resource
finally:
    lock.release()
```

Python Threading

access to shared resources

An example about locks usage:

```python
import threading
import logging
import random
import time

class counter:
    def __init__(self, initial_value = 0):
        self.lock = threading.Lock()
        self.value = initial_value

    def increment(self):
        self.lock.acquire()
        self.value += 1
        self.lock.release()

    def __str__(self):
        return str(self.value)

def worker(c):
    logging.debug('Starting...')
    for i in range(2):
        pause = random.random()
        logging.debug('sleeping %0.02f' % pause)
        time.sleep(pause)
        c.increment()
        logging.debug('counter: %s' % c)
    logging.debug('...Done')
```

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

count = counter()
t1 = threading.Thread(target = worker, args = (count, ))
t2 = threading.Thread(target = worker, args = (count, ))

t1.start()
t2.start()
```

program's output

```
Thread-1 Starting...
Thread-1 sleeping 0.78
Thread-2 Starting...
Thread-2 sleeping 0.42
Thread-2 counter: 1
Thread-2 sleeping 0.20
Thread-2 counter: 2
Thread-2 ...Done
Thread-1 counter: 3
Thread-1 sleeping 0.17
Thread-1 counter: 4
Thread-1 ...Done
```

Python Threading

access to shared resources

A **Semaphore** is a more advanced lock mechanism that has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore.

Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously:

```
semaphore = threading.Semaphore()
semaphore.acquire()              # decrements the counter
... access the shared resource
semaphore.release()              # increments the counter
```

The counter is decremented when the semaphore is acquired, and incremented when the semaphore is released. If the counter reaches zero when acquired, the acquiring thread will block. When the semaphore is incremented again, one of the blocking threads (if any) will run.

Python Threading

access to shared resources

A Semaphore allow more than one thread access to a resource at a time, while still limiting the overall number. For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads. Here is an example:

```python
import threading
import logging
import time

class activePool:
    def __init__(self):
        self.active = []
        self.lock = threading.Lock()

    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
            logging.debug('running: %s' % self.active)

    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
            logging.debug('running: %s' % self.active)

def worker(s, pool):
    with s:
        name = threading.currentThread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)
```

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

pool = activePool()
s = threading.Semaphore(2)

for i in range(5):
    t = threading.Thread(target = worker, args = (s, pool))
    t.start()
```

program's output

```
Thread-1 running: ['Thread-1']
Thread-2 running: ['Thread-1', 'Thread-2']
Thread-1 running: ['Thread-2']
Thread-2 running: []
Thread-3 running: ['Thread-3']
Thread-4 running: ['Thread-3', 'Thread-4']
Thread-3 running: ['Thread-4']
Thread-4 running: []
Thread-5 running: ['Thread-5']
Thread-5 running: []
```

Python Threading

synchronization between threads

Although the point of using multiple threads is to spin separate operations off to run concurrently, there are times when it is important to be able to synchronize the operations in two or more threads.

A simple way to communicate between threads is using **Event** objects. 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 progress until allowed to continue:

```
event = threading.Event()

# a client thread can wait for the flag to be set:
event.wait()

# a server thread can set or reset it:
event.set()
event.clear()
```

If the flag is set, the wait method doesn’t do anything. If the flag is cleared, wait will block until it becomes set again. Any number of threads may wait for the same event.

Python Threading

synchronization between threads

Here is an example of using events to synchronize threads. Notice that the wait() method may optionally take an argument representing the number of seconds to wait for the event before timing out and that it returns a boolean indicating whether or not the event is set. The isSet() method can be used separately to test the event status, without fear of blocking

```
def waitEvent(e):
    logging.debug('wait for event')
    es = e.wait()
    logging.debug('event is %s' % es)

def waitEventTimeout(e, t):
    logging.debug('wait for event or timeout')
    while not e.isSet():
        es = e.wait(t)
        if es:
            logging.debug('event is %s' % es)
        else:
            logging.debug('%ss timeout' % t)
```

program's output

```
(block     ) 2016-11-03 09:30:59,984 wait for event
(non-block ) 2016-11-03 09:30:59,984 wait for event or timeout
(MainThread) 2016-11-03 09:30:59,984 wait 5s before setting event
(non-block ) 2016-11-03 09:31:01,985 2s timeout
(non-block ) 2016-11-03 09:31:03,986 2s timeout
(MainThread) 2016-11-03 09:31:04,990 now setting event
(block     ) 2016-11-03 09:31:04,991 event is True
(non-block ) 2016-11-03 09:31:05,012 event is True
```

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

e = threading.Event()

t1 = threading.Thread(name = 'block', target = waitEvent, args = (e,))
t2 = threading.Thread(name = 'non-block', target = waitEventTimeout, args = (e,2))
t1.start()
t2.start()

logging.debug('wait 5s before setting event')
time.sleep(5)
logging.debug('now setting event')
e.set()
```

Bibliography, Sitography

- Libreria di riferimento Python, moduli thread (http://docs.python.it/html/lib/module-thread.html) e threading (http://docs.python.it/html/lib/module-threading.html)
- Python Multithreaded Programming, www.tutorialspoint.com/python/python_multithreading.htm
- Python threading – Manage concurrent threads, pymotw.com/2/threading/
- Thread Synchronization Mechanisms in Python, http://effbot.org/zone/thread-synchronization.htm