Threads and Processes

"Well, since you last asked us to stop, this thread has moved from discussing languages suitable for professional programmers via accidental users to computer-phobic users. A few more iterations can make this thread really interesting..."

eff-bot, June 1996

Overview

This chapter describes the thread support modules provided with the standard Python interpreter. Note that thread support is optional, and may not be available in your Python interpreter.

This chapter also covers some modules that allow you to run external processes on Unix and Windows systems.

Threads

When you run a Python program, execution starts at the top of the main module, and proceeds downwards. Loops can be used to repeat portions of the program, and function and method calls transfer control to a different part of the program (but only temporarily).

With threads, your program can do several things at one time. Each thread has its own flow of control. While one thread might be reading data from a file, another thread can keep the screen updated.

To keep two threads from accessing the same internal data structure at the same time, Python uses a *global interpreter lock*. Only one thread can execute Python code at the same time; Python automatically switches to the next thread after a short period of time, or when a thread does something that may take a while (like waiting for the next byte to arrive over a network socket, or reading data from a file).

The global lock isn't enough to avoid problems in your own programs, though. If multiple threads attempt to access the same data object, it may end up in an inconsistent state. Consider a simple cache:

```
def getitem(key):
    item = cache.get(key)
    if item is None:
        # not in cache; create a new one
        item = create_new_item(key)
        cache[key] = item
    return item
```

If two threads call the **getitem** function just after each other with the same missing key, they're likely to end up calling **create_new_item** twice with the same argument. While this may be okay in many cases, it can cause serious problems in others.

To avoid problems like this, you can use *lock objects* to synchronize threads. A lock object can only be owned by one thread at a time, and can thus be used to make sure that only one thread is executing the code in the **getitem** body at any time.

Processes

On most modern operating systems, each program run in its own *process*. You usually start a new program/process by entering a command to the shell, or by selecting it in a menu. Python also allows you to start new programs from inside a Python program.

Most process-related functions are defined by the **os** module. See the *Working with Processes* section for the full story.

The threading module

(Optional). This is a higher-level interface for threading. It's modeled after the Java thread facilities. Like the lower-level **thread** module, it's only available if your interpreter was built with thread support.

To create a new thread, subclass the **Thread** class and define the run method. To run such threads, create one or more instances of that class, and call the **start** method. Each instance's **run** method will execute in its own thread.

Example: Using the threading module

```
# File:threading-example-1.py

import threading
import time, random

class Counter:
    def __init__(self):
        self.lock = threading.Lock()
        self.value = 0

    def increment(self):
        self.lock.acquire() # critical section
        self.value = value = self.value + 1
        self.lock.release()
        return value

counter = Counter()

class Worker(threading.Thread):

    def run(self):
        for i in range(10):
            # pretend we're doing something that takes 10-100 ms
            value = counter.increment() # increment global counter
            time.sleep(random.randint(10, 100) / 1000.0)
            print self.getName(), "-- task", i, "finished", value

#
# try it

for i in range(10):
    Worker().start() # start a worker
```

```
Thread-1 -- task 0 finished 1
Thread-3 -- task 0 finished 3
Thread-7 -- task 0 finished 8
Thread-1 -- task 1 finished 7
Thread-4 -- task 0 Thread-5 -- task 0 finished 4
finished 5
Thread-8 -- task 0 Thread-6 -- task 0 finished 9
finished 6
...
Thread-6 -- task 9 finished 98
Thread-4 -- task 9 finished 99
Thread-9 -- task 9 finished 100
```

This example also uses **Lock** objects to create a critical section inside the global counter object. If you remove the calls to **acquire** and **release**, it's pretty likely that the counter won't reach 100.

The Queue module

This module provides a thread-safe queue implementation. It provides a convenient way of moving Python objects between different threads.

Example: Using the Queue module

```
# File:queue-example-1.py

import threading
import Queue
import time, random

WORKERS = 2

class Worker(threading.Thread):

    def __init__(self, queue):
        self.__queue = queue
        threading.Thread.__init__(self)

    def run(self):
        while 1:
            item = self.__queue.get()
            if item is None:
                break # reached end of queue

            # pretend we're doing something that takes 10-100 ms
            time.sleep(random.randint(10, 100) / 1000.0)

            print "task", item, "finished"

#
# try it

queue = Queue.Queue(0)

for i in range(WORKERS):
    Worker(queue).start() # start a worker

for i in range(10):
    queue.put(i)

for i in range(WORKERS):
    queue.put(None) # add end-of-queue markers
```

```
task 1 finished
task 0 finished
task 3 finished
task 2 finished
task 4 finished
task 5 finished
task 7 finished
task 6 finished
task 9 finished
task 8 finished
```

You can limit the size of the queue. If the producer threads fill the queue, they will block until items are popped off the queue.

Example: Using the Queue module with a maximum size

```
# File:queue-example-2.py

import threading
import Queue

import time, random

WORKERS = 2

class Worker(threading.Thread):

    def __init__(self, queue):
        self.__queue = queue
        threading.Thread.__init__(self)

    def run(self):
        while 1:
            item = self.__queue.get()
            if item is None:
                break # reached end of queue

            # pretend we're doing something that takes 10-100 ms
            time.sleep(random.randint(10, 100) / 1000.0)

            print "task", item, "finished"
```

```
#
# run with limited queue

queue = Queue.Queue(3)

for i in range(WORKERS):
    Worker(queue).start() # start a worker

for item in range(10):
    print "push", item
    queue.put(item)

for i in range(WORKERS):
    queue.put(None) # add end-of-queue markers
```

```
push 0
push 1
push 2
push 3
push 4
push 5
task 0 finished
push 6
task 1 finished
push 7
task 2 finished
push 8
task 3 finished
push 9
task 4 finished
task 6 finished
task 5 finished
task 7 finished
task 9 finished
task 8 finished
```

You can modify the behavior through subclassing. The following class provides a simple priority queue. It expects all items added to the queue to be tuples, where the first member contains the priority (lower value means higher priority):

Example: Using the Queue module to implement a priority queue

```
# File:queue-example-3.py

import Queue
import bisect

Empty = Queue.Empty

class PriorityQueue(Queue.Queue):
    "Thread-safe priority queue"

    def _put(self, item):
        # insert in order
        bisect.insort(self.queue, item)

#
# try it

queue = PriorityQueue(0)

# add items out of order
queue.put((20, "second"))
queue.put((10, "first"))
queue.put((30, "third"))

# print queue contents
try:
    while 1:
        print queue.get_nowait()
except Empty:
    pass
```

```
third
second
first
```

And here's a simple stack implementation (last-in first-out, instead of first-in, first-out):

Example: Using the Queue module to implement a stack

```
# File:queue-example-4.py

import Queue

Empty = Queue.Empty

class Stack(Queue.Queue):
    "Thread-safe stack"

    def _put(self, item):
        # insert at the beginning of queue, not at the end
        self.queue.insert(0, item)

    # method aliases
    push = Queue.Queue.put
    pop = Queue.Queue.get
    pop_nowait = Queue.Queue.get_nowait

#
# try it

stack = Stack(0)

# push items on stack
stack.push("first")
stack.push("second")
stack.push("third")

# print stack contents
try:
    while 1:
        print stack.pop_nowait()
except Empty:
    pass

third
second
first
```

The thread module

(Optional). This module provides a low-level interface for threading. It's only available if your interpreter is built with thread support. New code should use the higher-level interface in the **threading** module instead.

Example: Using the thread module

```
# File:thread-example-1.py

import thread
import time, random

def worker():
    for i in range(50):
        # pretend we're doing something that takes 10-100 ms
        time.sleep(random.randint(10, 100) / 1000.0)
        print thread.get_ident(), "-- task", i, "finished"

#
# try it out!

for i in range(2):
    thread.start_new_thread(worker, ())

time.sleep(1)

print "goodbye!"
```

```
311 -- task 0 finished
265 -- task 0 finished
265 -- task 1 finished
311 -- task 1 finished
...
265 -- task 17 finished
311 -- task 13 finished
265 -- task 18 finished
goodbye!
```

Note that when the main program exits, all threads are killed. The **threading** module doesn't have that problem.

The commands module

(Unix only). This function contains a few convenience functions, designed to make it easier to execute external commands under Unix.

Example: Using the commands module

```
# File:commands-example-1.py

import commands

stat, output = commands.getstatusoutput("ls -lR")

print "status", "=>", stat
print "output", "=>", len(output), "bytes"

status => 0
output => 171046 bytes
```

The pipes module

(Unix only). This module contains support functions to create "conversion pipelines". You can create a pipeline consisting of a number of external utilities, and use it on one or more files.

Example: Using the pipes module

```
# File:pipes-example-1.py

import pipes

t = pipes.Template()

# create a pipeline
t.append("sort", "--")
t.append("uniq", "--")

# filter some text
t.copy("samples/sample.txt", "")
```

```
Alan Jones (sensible party)
Kevin Phillips-Bong (slightly silly)
Tarquin Fin-tim-lin-bin-whin-bim-lin-bus-stop-F'tang-F'tang-Olé-Biscuitbarrel
```

The popen2 module

This module allows you to run an external command and access stdin and stdout (and possibly also stderr) as individual streams.

In Python 1.5.2 and earlier, this module is only supported on Unix. In 2.0, the functions are also implemented on Windows.

Example: Using the popen2 module to sort strings

```
# File:popen2-example-1.py

import popen2, string

fin, fout = popen2.popen2("sort")

fout.write("foo\n")
fout.write("bar\n")
fout.close()

print fin.readline(),
print fin.readline(),
fin.close()
```

```
bar
foo
```

The following example shows how you can use this module to control an existing application.

Example: Using the popen2 module to control gnuchess

```
# File:popen2-example-2.py

import popen2
import string

class Chess:
    "Interface class for chesstool-compatible programs"

    def __init__(self, engine = "gnuchessc"):
        self.fin, self.fout = popen2.popen2(engine)
        s = self.fin.readline()
        if s != "Chess\n":
            raise IOError, "incompatible chess program"

    def move(self, move):
        self.fout.write(move + "\n")
        self.fout.flush()
        my = self.fin.readline()
        if my == "Illegal move":
            raise ValueError, "illegal move"
        his = self.fin.readline()
        return string.split(his)[2]

    def quit(self):
        self.fout.write("quit\n")
        self.fout.flush()

#
# play a few moves

g = Chess()

print g.move("a2a4")
print g.move("b2b3")

g.quit()
```

```
b8c6
e7e5
```

The signal module

This module is used to install your own signal handlers. When the interpreter sees a signal, the signal handler is executed as soon as possible.

Example: Using the signal module

```
# File:signal-example-1.py

import signal
import time

def handler(signo, frame):
    print "got signal", signo

signal.signal(signal.SIGALRM, handler)

# wake me up in two seconds
signal.alarm(2)

now = time.time()

time.sleep(200)

print "slept for", time.time() - now, "seconds"
```

```
got signal 14
slept for 1.99262607098 seconds
```