11

Threading the Code

Self-Review Questions

Self-review 11.1 What is a 'thread' and what is a 'process'? What is the difference between the two?

Self-review 11.2 What does the 'scheduler' in an operating system take responsibility for?

Self-review 11.3 What does the following line of code do?

```
time.wait ( 5 )
```

Self-review 11.4 Threading is used extensively in interactive programming in applications such as games, GUIs, Web browsers, and so on. Why is threading so important for programs that need to interact with humans?

Self-review 11.5 What is happening in the following code?

```
import threading , time

class Foobar ( threading.Thread ) :
    def __init__ ( self ) :
        threading.Thread.__init__ ( self )
    def run ( self ) :
        time.sleep ( 1 )
        print self.getName ( )

if __name__ == '__main__' :
    pool = [ Foobar ( ) for i in range ( 10 ) ]
    map ( lambda t : t.start ( ) , pool )
```

Self-review 11.6 On one occasion when we ran the code above, we got the following output:

```
Thread-1
Thread-2
Thread-7
Thread-3
```

```
Thread-9
Thread-8
Thread-6
Thread-10
Thread-4
Thread-5
```

Why don't the numbers appear sequentially?

Self-review 11.7 To the nearest second, how long will the code in Self-review 11.5 take to execute?

Self-review 11.8 The second threaded guessing game program can take up to 5 s to terminate after the user has entered Ctlr+d. Why is this?

When the user presses Ctrl+d, the main thread sets the keepGoing variable in the generator thread to **False** and then terminates. The generator thread could just have tested the value and entered a sleep of up to 5 s. Only then will it see the changed value, terminate the loop, terminate the run method and hence terminate.

Self-review 11.9 'Livelock' and 'deadlock' are two problems that are commonly associated with multi-threaded programs. Define 'deadlock' and 'livelock'.

Self-review 11.10 What are 'critical sections' in programs?

Self-review 11.11 What does it mean for a piece of code to be 'atomic'?

Self-review 11.12 What does the threading.Thread.start method do?

Self-review 11.13 Why must the run method be overridden in subclasses of threading.Thread?

Self-review 11.14 What does the following code do?

```
print threading.currentThread ( )
```

What output would it give if used outside a thread object, like this:

```
if __name__ == '__main__':
    print threading.currentThread()
```

Self-review 11.15 Imagine you are writing a multi-player game. Your game will have some interesting graphics that need to be generated quickly to provide a smooth animation. There will be a sound track to the game-play and several players will be interacting at once. Inside the game there will be both player and non-player characters. What parts of this program would you want to separate out into their own threads?

Programming Exercises

Exercise 11.1 Write a program that creates two threads, each of which should sleep for a random number of seconds, then print out a message which includes its thread ID.

Exercise 11.2 Starting with your code from the previous exercise, add a counter to your program. Each thread should include the value of the counter in the message it prints to the screen and should increment the counter every time it wakes up. So, the counter will hold the total number of messages printed to the screen by all the running threads.

You will need to use some form of concurrency control to ensure that the value of the counter is always correct. Several of these are available in Python. One simple solution is to use locks. The idea here is that if a variable is locked, it cannot be changed by code in another thread until it has been unlocked. In this way we can use locks to create critical sections in our code.

Here's a code fragment to get you started:

```
import threading

foobar = 10 # Shared data!
foobar_l = threading.Lock ( )
. . .
# Start critical section
foobar_l.acquire ( ) # Acquire lock on foobar
foobar = . . .
foobar_l.release ( ) # Let other threads alter foobar
# End critical section
. . .
```

Locking isn't everyone's favorite method of concurrency control. Acquiring and releasing locks is relatively very slow – remember concurrency is often used in time-critical applications such as games.

Exercise 11.3 Python provides a special data structure in a class called Queue.Queue that is specifically for use in multi-threaded programs. You have already learned about queue types in Sections ?? (page ??) and ?? (page ??). In Python's queues, data can be added to a queue using a method called put and removed from it using the method get. Here's an example:

```
>>> import Queue
>>> q = Queue.Queue ( )
>>> q.put ( 5 )
>>> q.put ( 4 )
>>> q.get ( )
5
>>> q.get ( )
4
>>> q.get ( )
```

Here we have added the objects 5 and 4 to a queue we have created, then retrieved those two values. Note that we have called `q.get` a third time, even though the queue is now empty. You might have expected Python to raise an exception here, because we tried to access a value that doesn't exist. In fact, what is happening here is that Python is waiting (blocking) on the queue to provide a value. Because queues are expected to be used in concurrent programs, the authors of the `Queue.Queue` class knew that other threads might be placing data on the queue even when one thread is waiting to fetch data from it.

For this exercise, write a small program in which several threads access a single queue at the same time. Each thread should randomly either place some random data on the queue, or take data off it. Make sure you print out lots of useful information about what each thread is doing to the queue so you can watch how the program proceeds!

Challenges

Challenge 11.1 Tkinter provides a number of methods to control concurrency and scheduling in GUIs, including:

- widget.after
- widget.update
- widget.wait_visibility

Read the Tkinter documentation to find out what these methods do.

Start out by creating a simple GUI that implements an alarm. To help with this, Tkinter buttons have methods called bell and flash that 'beep' and make the button flash, respectively. Use the after method to schedule an alarm to sound after a given number of milliseconds. Make sure the user has a way of turning the alarm off!

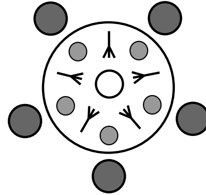
When you have a basic alarm program working, implement the following improvements:

- Allow the user to schedule the alarm to go off at a particular time.
- Allow the user to schedule the alarm to go off at given intervals (for example, every half hour).
- Allow the user to schedule several alarms to go off at various times during the day.

Challenge 11.2 The classic problem in the area of concurrent programming, was proposed by Edsgar Dijkstra in 1971, and recast by Tony Hoare as the The Dining Philosophers Problem.

Five philosophers are sitting at a round table. Philosophers are quite eccentric, all they do all day is think and eat, but they can't

do both at once. Between each pair of philosophers is a fork and each philosopher needs two forks to eat. What would happen if all the philosophers organize themselves to pick up their left fork and then their right fork whenever they stop thinking? Can you think of a better way for philosophers to eat?



Research this problem and write a program that uses threads to represent each philosopher to explore how to avoid deadlock.