APPENDIX

A Multi-threading

A.1 INTRODUCTION

A *thread*, also known as a thread of execution, is defined as the smallest unit that can be scheduled in an operating system. They are usually contained in processes. A process can have more than one thread that can run concurrently. These threads share the memory and the state of the process. That is, they share the instructions and the values of the variables defined in that program. Thus, a single set of code or program can perform two or more tasks simultaneously i.e., it is used by several processors at different stages of execution. This technique is known as *multi-threading*.

A thread is a lightweight process which do not require much memory. Multi-threading or running several threads is similar to running several different programs simultaneously. The main benefit of multi-threading is that multiple threads within a process share the same data space with the main thread. This eases sharing of information between different processes.

A thread can be pre-empted (interrupted), suspended, resumed, and blocked before it finally terminates. There are two different kind of threads—kernel threads and user-space threads (or user threads). The *kernel threads* are a part of the operating system and the user cannot manipulate them from their code. However, through programming, users can definitely control the user threads. A *user-space thread* is similar to a function call. Every process has at least one thread, i.e. the process itself. It can even start multiple threads which are executed by the operating system executes as parallel processes. On a single processor machine, parallelism is achieved by thread scheduling or time-slicing.

There are two modules which help in multithreading in Python 3.x, namely, _thread and threading. We will discuss them later in this appendix.

Advantages of Multi-threading

- Multi-threaded programs can execute faster on computers with multiple CPUs. Each thread can be executed by a separate CPU.
- The program can remain responsive to input.
- Besides having local variables, threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads.
- Handling of threads is simpler than the handling of processes for an operating system. That is why they are sometimes called lightweight process (LWP).

A.2 STARTING A NEW THREAD USING THE _thread MODULE

To begin a new thread, you need to call the `start_new_thread()` method of the `_thread` module. The syntax of `start_new_thread()` can be given as,

```
_thread.start_new_thread(function_name, args[, kwargs])
```

Here, args is a tuple of arguments. kwargs is an optional dictionary of keyword arguments. The start_new_thread() method returns immediately. It starts the child thread and calls function with the passed list of args. When function returns, the thread terminates.

Note In the start_new_thread() method, use an empty tuple to call function without passing any arguments.

Example A.1 Program to implement multi-threading. The threads print the current time.

```
import _thread
import time
# Define a function for the thread
def display_time(threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print("%s: %s" % (threadName, time.ctime(time.time())))
# Create two threads as follows
try:
   _thread.start_new_thread(display_time, ("ONE", 1, ))
   _thread.start_new_thread(display_time, ("TWO", 2, ))
except:
   print("Error: unable to start thread")
```

OUTPUT

```
ONE: Sat Feb 04 21:43:48 2017
TWO: Sat Feb 04 21:43:49 2017
ONE: Sat Feb 04 21:43:49 2017
ONE: Sat Feb 04 21:43:50 2017
TWO: Sat Feb 04 21:43:51 2017
ONE: Sat Feb 04 21:43:51 2017
ONE: Sat Feb 04 21:43:52 2017
TWO: Sat Feb 04 21:43:53 2017
TWO: Sat Feb 04 21:43:55 2017
TWO: Sat Feb 04 21:43:57 2017
```

Although it is very effective for low-level threading, but the _thread module is very limited compared to the newer threading module.

A.3 THE THREADING MODULE

The *threading module,* which was first introduced in Python 2.4, provides much more powerful, high-level support for threads than the thread module. The module defines some additional methods which are as follows.

- threading.activeCount(): Returns the number of active thread objects
- threading.currentThread(): Returns the count of thread objects in the caller's thread control.
- threading.enumerate(): Returns a list of all active thread objects

Besides these methods, the `threading` module has the `Thread` class that implements threading. The methods provided by the `Thread` class include:

- `run()`: This marks the entry point for a thread.
- `start()`: The method starts the execution of a thread by calling the run method.
- `join([time])`: The `join()` method waits for threads to terminate.
- `isAlive()`: The `isAlive()` method checks whether a thread is still executing.
- `getName()`: As the name suggests, it returns the name of a thread.
- `setName()`: This method is used to set the name of a thread.

Creating a Thread using Threading Module

Follow the steps given below to implement a new thread using the `Threading` module.

- Define a new subclass of the `Thread` class.
- Override the `__init__()` method.
- Override the `run()` method. In the `run()` method specify the instructions that the thread should perform when started.
- Create a new `Thread` subclass and then use its object to start the thread by calling its `start()` method which in turn calls the `run()` method. For example, consider the following code which uses the threading module to create threads.

Example A.2 Program to create a thread using the threading module

```
import threading
import time
class myThread(threading.Thread):   # create a sub class
    def __init__(self, name, count):
        threading.Thread.__init__(self)
        self.name = name
        self.count = count
    def run(self):
        print("\n Starting " + self.name)
        i=0
        while i<self.count:
            display(self.name, i)
            time.sleep(1)
            i+=1
        print("\n Exiting " + self.name)
def display(threadName, i):
    print("\n", threadName, i)
# Create new threads
thread1 = myThread("ONE", 5)
thread2 = myThread("TWO", 5)
# Start new Threads
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

```
print("\n Exiting Main Thread")
```

OUTPUT

```
Starting ONE Starting TWO
ONE 0
TWO 0
ONE 1
TWO 1
ONE 2
TWO 2
ONE 3
TWO 3
ONE 4
TWO 4
Exiting ONE
Exiting TWO
Exiting Main Thread
```

Synchronizing Threads

The `Threading` module also includes a locking mechanism to synchronize threads. For this, the module supports the following methods.

- `Lock()` method which when invoked returns the new lock.
- `acquire([blocking])` method of the new lock object forces threads to run synchronously. The optional *blocking* parameter is used to control whether the thread waits to acquire the lock.
  - If the value of *blocking* is 0, the thread returns 0 if the lock cannot be acquired and 1 if the lock was acquired.
  - If *blocking* is set to 1, the thread blocks and waits for the lock to be released.
- `release()` method of the new lock object releases the lock when it is no longer required.

Note The output of the following programs may vary on your PC subject to the processor's speed and number of applications running currently.

Example A.3 Program to synchronize threads by locking mechanism

```
import threading
import time
class myThread(threading.Thread):   # create a sub class
   stopFlag = 0
   def __init__(self, name, msg):
      threading.Thread.__init__(self)
      self.name = name
      self.msg = msg
   def run(self):
      print("\n Starting " + self.name)
```

```
        self.display()
        time.sleep(1)
        print("\n Exiting " + self.name)
    def display(self):
        while self.stopFlag!=3:
            Lock.acquire()
            print("[",self.msg,"]")
            Lock.release()
            self.stopFlag += 1
Lock = threading.Lock()
# Create new threads
thread1 = myThread("ONE","HELLO")
thread2 = myThread("TWO","WORLD")
# Start new Threads
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("\n Exiting Main Thread")
```

OUTPUT

```
Starting ONE
Starting TWO
[ HELLO ]
[ WORLD ]
[ HELLO ]
[ WORLD ]
[ HELLO ]
[ WORLD ]
Exiting ONE
Exiting TWO
Exiting Main Thread
```

Example A.4 Program to create two threads to keep a count of number of even numbers entered by the user.

```
import threading
import time

numEvens = 0
class myThread(threading.Thread):   # create a sub class
    stopFlag = 0
    def __init__(self, name):
        threading.Thread.__init__(self)
        self.name = name
    def run(self):
```

```
    print("\n Starting " + self.name)
    self.display()
    print("\n Exiting " + self.name)
def display(self):
    global numEvens
    while self.stopFlag!=1:
        num = int(input("Enter a number :"))
        Lock.acquire()
        if num%2 == 0:
            numEvens +=  1
        print("\n",self.name, numEvens)
        Lock.release()
        time.sleep(1)
    def stop(self):
        self.stopFlag = 0
Lock = threading.Lock()
# Create new threads
thread1 = myThread("ONE")
thread2 = myThread("TWO")
# Start new Threads
thread1.start()
thread2.start()
time.sleep(20)
thread1.stop()
thread2.stop()
print("\n Exiting Main Thread")
```

OUTPUT

```
Starting ONE
Starting TWO
Enter a number : 1
Enter a number : 2
ONE 0
TWO 1
Enter a number :4
Enter a number :5
ONE 2
TWO 2
Enter a number :7
Enter a number : 8
ONE 2
TWO 3
Enter a number :10
Enter a number : 22
ONE 4
TWO 5
Exiting Main Thread
```

Exercises

1. A thread is a _____ process.
2. The user cannot manipulate the _____ thread.
3. Which of the following cannot be done with a thread?
   (a) pre-empted (b) suspended
   (c) resumed (d) None of these
4. Which method returns the count of thread objects in the caller's thread control?
   (a) `threading.activeCount()`
   (b) `threading.enumerate()`
   (c) `threading.currentThread()`
   (d) None of these
5. A process can have only two threads that can run concurrently. (True/ False)
6. To begin a new thread, you need to call the `run()` method. (True/ False)
7. What is a thread?
8. Define the term multi-threading. Give its advantages.
9. Explain the methods that are used to synchronize threads.
10. Write a program that creates thread using `threading` module.
11. Write a program that creates threads to print `Have a Good Day` using the `__thread` module.

Answers

1. lightweight 2. kernel 3. (d) 4. (c) 5. False 6. False