25 Concurrency and Parallelism

"Efficient is better..."

Contents

KanNotes

Concurrency and Parallelism

- A task is an activity that we carry out. For example, driving a car, watering a plant, cooking food, etc. are all tasks.
- When we perform multiple tasks in *overlapping* times we are doing them *concurrently*. When we perform tasks *simultaneously* we are doing them *parallelly*.
- Thus though the words concurrency and parallelism indicate happening of two or more tasks at the same time, they are not the same thing.
- Example 1 of concurrency: We watch TV, read a news-paper, sip coffee in overlapping times. At any given moment you are doing only one task.
- Example 2 of concurrency: In a 4 x 100 meter relay race, each runner in a given lane has to run, but unless the first runner hands over the baton, second doesn't start and unless second hands over the baton the third doesn't start. So at any given moment only one runner in running.
- Example 1 of parallelism: Example of parallelism: While driving a car we carry out several activities in parallel—we listen to music, we drive the car and we talk to the co-passengers.
- Example 2 of parallelism: In a 100 meter race each runner is running in his own lane. At a given moment all runners are running.

What are Threads?

- A program may have several units (parts). Each unit of execution is called a thread.
- Example 1 of multiple threads: One unit of execution may carry out copying of files, whereas another unit may display a progress bar.
- Example 2 of multiple threads: One unit of execution may download images, whereas another unit may display text.

- Example 3 of multiple threads: One unit may let you edit a document, second unit may check spellings, third unit may check grammar and fourth unit may do printing.
- Example 4 of multiple threads: One unit may scan disk for viruses, second unit may scan memory for viruses and third unit may let you interact with the program user-interface to stop/pause the scanning of viruses by first two units.

Concurrency and Parallelism in Programming

- **Concurrency** is when multiple threads of a program start, run, and complete in *overlapping* time periods.
- Once the program execution begins one thread may run for some time, then it may stop and the second thread may start running. After some time, second thread may stop and the third may start running.
- Threads may get executed in a round-robin fashion or based on priority of each thread. At any given instance only one thread is running.
- **Parallelism** is when multiple threads of a program literally run *at the same time*. So at any given instance multiple threads are running.
- In concurrency multiple units of a program can run on a single-core processor, whereas, in parallelism multiple units can run on multiple cores of a multi-core processor.
- Figure 25.1 shows working how threads t1, t2 and t3 in a program may run concurrently or in parallel over a period of time.

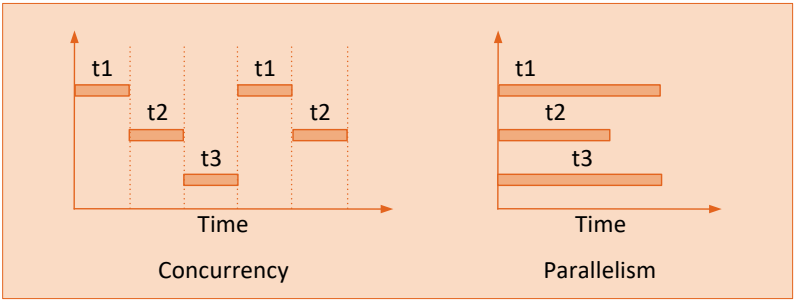

Figure 25.1

- Advantages of Concurrency:
  - Improves application's speed, by making CPU do other things instead of waiting for slow I/O operations to finish
  - Simplifies program design. For example, the logic that copies files and logic that displays the progress bar can be kept separate.
- Advantage of Parallelism:
  - Capability of multi-core processors can be exploited by running different processes in each processor simultaneously.

CPU-bound and I/O-bound Programs

- A program typically performs two types of operations:
  - Operations involving CPU for calculations, comparisons, etc.
  - Operations that perform input or output
- Usually CPU operations run several times faster than I/O operations.
- A program that predominantly performs CPU operations is called CPU-bound program. A program that predominantly performs I/O operations is called I/O-bound program.
- Example of CPU-bound program: A program that perform multiplication of matrices, or a program that finds sum of first 200 prime numbers.
- Example of I/O-bound program: A program that processes files on the disk, or a program that does database queries or sends a lot of data over a network.

Which to use when?

- A CPU-bound program will perform better on a faster CPU. For example, using i7 CPU instead of i3 CPU.
- An IO-bound program will perform better on a faster I/O subsystem. For example using a faster disk or faster network.
- The solution to improve performance cannot always be to replace existing CPU with a faster CPU or an existing I/O subsystem with a faster I/O subsystem.

- Instead, we should organize our program to use concurrency or parallelism to improve performance.
- Performance of I/O-bound program can improve if different units of the program are executed in overlapping times.
- Performance of CPU-bound program can improve if different units of the program are executed parallelly on multiple cores of a processor.
- It is quite easy to imagine how performance of a CPU-bound program can improve with parallelism. Performance improvement of an I/O-bound program using concurrency is discussed in the next section.

Concurrency for improving Performance

- Suppose we wish to write a program that finds squares and cubes of first 5000 natural numbers and prints them on the screen.
- We can write this program in two ways:
  - A single-threaded program - calculation of squares, calculation of cubes and printing are done in same thread.
  - A multi-threaded program - calculation of squares is done in one thread, calculation of cubes in second thread and printing in third thread.
- In the single-threaded program the CPU has to frequently wait for printing of square/cube (I/O operation) to get over before it can proceed to calculate square or cube of the next number. So CPU remains under-utilized. This scenario is shown in Figure 25.2.

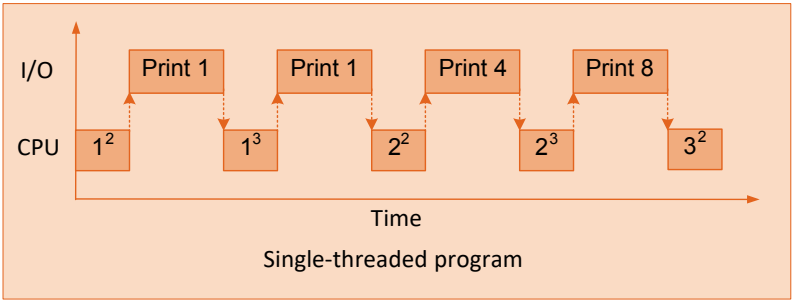

Figure 25.2

- In the multi-threaded program the CPU can proceed with the next calculation (square or cube) and need not wait for the square or cube to get printed on the screen. This scenario is shown in Figure 25.3.

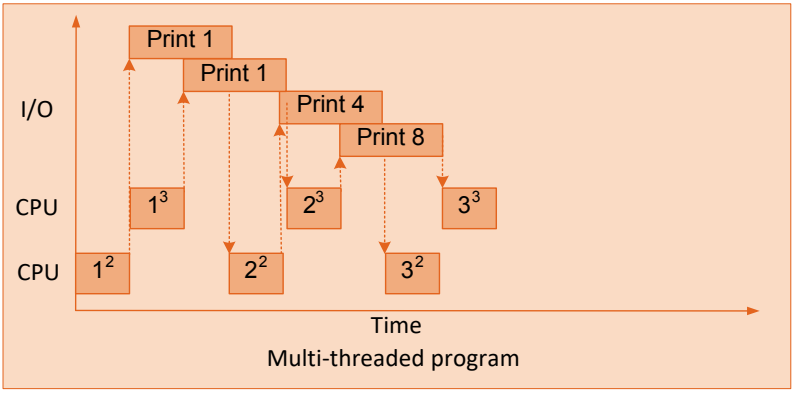

Figure 25.3

Types of Concurrencies

- In a multi-threaded program one thread runs for some time, then it stops and the second thread starts running. After some time, second thread stops and the third thread starts running. This is true even if the program is being executed on a multi-core processor.
- When context would switch from one thread to another depends on the type of concurrency that we use in our program.
- Concurrencies are of two types:
  - Pre-emptive concurrency - The OS decides when to switch from one thread to another.
  - Cooperative concurrency - The thread decides when to give up the control to the next task.
- Python modules available for implementing concurrency and parallelism in our program are as follows:

  Pre-emptive concurrency - **threading**
  Cooperative concurrency - **asyncio**
  Parallelism - **multiprocessing**

This book discusses the technique for pre-emptive concurrency alone.

Thread Properties

- Every running thread has a name a number called thread identifier associated with it.
- The name of all running threads need not be unique, whereas the identifier must be unique.
- The identifier could be reused for other threads, if the current thread ends.

```
import threading
t = threading.current_thread( )   # returns current Thread object
print("Current thread:", t)   # prints thread name, identifier & status
print("Thread name:", t.name)
print("Thread identifier:", t.ident)
print("Is thread alive:", t.is_alive( ))
t.name = 'MyThread'
print("After name change:", t.name)
```

Here, **current_thread( )** is a function defined in **threading** module and **name** and **ident** are attributes of **Thread** object.

Launching Threads

- There are two ways to launch a new thread:
  - By passing the name of the function that should run as a separate thread, to the constructor of the **Thread** class.
  - By overriding **__init__( )** and **run( )** methods in a subclass of **Thread** class.
- Method 1 - thread creation

```
th1 = threading.Thread(name = 'My first thread', target = func1)
th2 = threading.Thread(target = func2)      # use default name
th1.start( )
th2.start( )
```

- Method 2 - thread creation

```
class SquareGeneratorThread(threading.Thread) :
    def __init__(self) :
        threading.Thread.__init__(self)

    def run(self) :
        print('Launching...')

th = SquareGeneratorThread( )
th.start( )
```

- Once a thread object is created, its activity must be started by calling the thread's **start( )** method. This method in turn invokes the **run( )** method.
- **start( )** method will raise an exception **RuntimeError** if called more than once on the same thread object.

Passing parameters to a Thread

- Sometimes we may wish to pass some parameters to the target function of a thread object.

  ```
  th1 = threading.Thread(target = squares, args = (a, b))
  th2 = threading.Thread(target = cubes, args = (a,))
  ```

  Arguments being passed to the constructor of **Thread** class will ultimately be passed to the target function. Arguments must be in the form of a tuple.

- Once thread have been launched we have no control over the order in which they are executed. It is controlled by the thread scheduler of the Python runtime environment.
- Sometimes we may wish to pass some parameters to the **run( )** method in the **thread** class. For this pass the parameters to the constructor while creating the thread object. The constructor should store them in object's variables. Once stored, **run( )** will be able to access them.

```
th = SquareGeneratorThread(a, b, c)
```

P</> Programs

Problem 25.1

Write a program that launches three threads, assigns new names to two of them. Suspend each thread for 1 second after it has been launched.

Program

```
import threading
import time

def fun1( ):
    t = threading.current_thread( )
    print('Starting', t.name)
    time.sleep(1)
    print('Exiting', t.name)

def fun2( ):
    t = threading.current_thread( )
    print('Starting', t.name)
    time.sleep(1)
    print('Exiting', t.name)

def fun3( ):
    t = threading.current_thread( )
    print('Starting', t.name)
    time.sleep(1)
    print('Exiting', t.name)

t1 = threading.Thread(target=fun1)   # use default name
t2 = threading.Thread(name = 'My second thread', target = fun2)
t3 = threading.Thread(name = 'My third thread', target = fun3)
t1.start( )
t2.start( )
t3.start( )
```

Output

```
Starting Thread-1
Starting My second thread
```

```
Starting My third thread
Exiting Thread-1
Exiting My third thread
Exiting My second thread
```

Tips

- **sleep( )** function of **time** module suspends execution of the calling thread for the number of seconds passed to it.

Problem 25.2

Write a program that calculates the squares and cubes of first 6 odd numbers through functions that are executed sequentially. Incorporate a delay of 0.5 seconds after calculation of each square/cube value. Report the time required for execution of the program.

Program

```
import time
import threading

def squares(nos) :
    print('Calculating squares...')
    for n in nos :
        time.sleep(0.5)
        print('n = ', n, ' square =', n * n)

def cubes(nos) :
    print('Calculating cubes...')
    for n in nos :
        time.sleep(0.5)
        print('n = ', n, ' cube =', n * n * n)

arr = [1, 3, 5, 7, 9, 11]
startTime = time.time( )
squares(arr)
cubes(arr)
endTime = time.time( )
print('Time required = ', endTime - startTime, 'sec')
```

Output

```
Calculating squares...
n =  1  square = 1
n =  3  square = 9
n =  5  square = 25
n =  7  square = 49
n =  9  square = 81
n =  11  square = 121
Calculating cubes...
n =  1  cube = 1
n =  3  cube = 27
n =  5  cube = 125
n =  7  cube = 343
n =  9  cube = 729
n =  11  cube = 1331
Time required =  6.000343322753906 sec
```

Tips

- The functions **squares( )** and **cubes( )** are running in the same thread.
- **time( )** function returns the time in seconds since the epoch (Jan 1, 1970, 00:00:00) as a floating point number.

Problem 25.3

Write a program that calculates squares and cubes of first 6 odd numbers through functions that are executed in two independent threads. Incorporate a delay of 0.5 seconds after calculation of each square/cube value. Report the time required for execution of the program.

Program

```
import time
import threading

def squares(nos) :
```

```
    print('Calculating squares...')
    for n in nos :
        time.sleep(0.5)
        print('n = ', n, ' square =', n * n)

def cubes(nos) :
    print('Calculating cubes...')
    for n in nos :
        time.sleep(0.5)
        print('n = ', n, ' cube =', n * n * n)

arr = [1, 3, 5, 7, 9, 11]
startTime = time.time( )

th1 = threading.Thread(target = squares, args = (arr,))
th2 = threading.Thread(target = cubes, args = (arr,))
th1.start( )
th2.start( )
th1.join( )
th2.join( )
endTime = time.time( )
print('Time required = ', endTime - startTime, 'sec')
```

Output

```
Calculating squares...
Calculating cubes...
n =  1  square = 1
n =  1  cube = 1
n =  3  square = 9
n =  3  cube = 27
n =  5  square = 25
n =  5  cube = 125
n =  7  square = 49
n =  7  cube = 343
n =  9  square = 81
n =  9  cube = 729
n =  11  square = 121
n =  11  cube = 1331
Time required =  3.001171588897705 sec
```

Tips

- **squares( )** and **cubes( )** are being launched in separate threads.
- Since **squares( )** and **cubes( )** need **arr**, it is passed to the constructor while launching the threads.
- Arguments meant for target functions must be passed as a tuple.
- **join( )** waits until the thread on which it is called terminates.
- If this program is executed on a single processor machine it will still work faster than the one in Problem 25.2. This is because when one thread is performing I/O, i.e. printing value of square/cube, the other thread can proceed with the calculation of cube/square.
- The output shows values of squares and cubes mixed. How to take care of it has been shown in Chapter 26.

Problem 25.4

Write a program that reads the contents of 3 files a.txt, b.txt and c.txt sequentially and reports the number of lines present in it as well as the total reading time. These files should be added to the project and filled with some text. The program should receive the file names as command-line arguments. Suspend the program for 0.5 seconds after reading a line from any file.

Program

```
import time, sys

startTime = time.time( )
lst = sys.argv
lst = lst[1:]

for file in lst:
    f = open(file, 'r')
    count = 0
    while True :
        data = f.readline( )
        time.sleep(0.5)
        if data == '' :
```

```
                break
            count = count + 1

        print('File:', file, 'Lines:', count)

endTime = time.time( )
print('Time required =', endTime - startTime, 'sec')
```

Output

```
File: a.txt Lines: 5
File: b.txt Lines: 24
File: c.txt Lines: 6
Time required = 19.009087324142456 sec
```

Tips

- If you are using IDLE then create three files a.txt, b.txt and c.txt these files in the same folder as the source file.
- If you are using NetBeans add files a.txt, b.txt and c.txt to the project as 'Empty' files by right-clicking the project in Project window in NetBeans. Once created, add some lines to each of these files.
- If you are using IDLE then provide command-line arguments as follows:

  c:\>idle -r SingleThreading.py a.txt b.txt c.txt

  Ensure that the path of idle batch file given below is added to PATH environment variable through Control Panel:

  C:\Users\Kanetkar\AppData\Local\Programs\Python\Python36-32\Lib\idlelib

- If you are using NetBeans, to provide a.txt, b.txt and c.txt as command-line arguments, right-click the project in Project window in NetBeans and select 'Properties' followed by 'Run'. Add 'a.txt b.txt c.txt' as 'Application Arguments'.
- Application arguments become available through **sys.argv** as a list. This list also includes application name as the 0^{th} element in the list. So we have sliced the list to eliminate it.

- File is opened for reading using **open( )** and file is read line by line in a loop using **readline( )**.

Problem 25.5

Write a program that reads the contents of 3 files a.txt, b.txt and c.txt in different threads and reports the number of lines present in it as well as the total reading time. These files should be added to the project and filled with some text. The program should receive the file names as command-line arguments. Suspend the program for 0.5 seconds after reading a line from any file.

Program

```
import time
import sys
import threading

def readFile(inputFile):
    f = open(inputFile, 'r')
    count = 0
    while True :
        data = f.readline( )
        time.sleep(0.5)
        if data == '' :
            break
        count = count + 1

    print('File:', inputFile, 'Lines:', count)

startTime = time.time( )
lst = sys.argv
lst = lst[1:]

tharr = [ ]
for file in lst:
    th = threading.Thread(target = readFile, args = (file,))
    th.start( )
    tharr.append(th)

for th in tharr:
```

```
    th.join( )

endTime = time.time( )
print('Time required = ', endTime - startTime, 'sec')
```

Output

```
File: a.txt Lines: 5
File: c.txt Lines: 6
File: b.txt Lines: 24
Time required = 12.504715204238892 sec
```

Tips

- For details of adding files to the project, making them available to application as command-line arguments and slicing the command-line argument list refer tips in Problem 25.4.
- As each thread is launched, the thread object is added to the thread array through **tharr.append( )**. This is necessary, so that we can later call **join( )** on each thread.
- This program performs better than the one in Problem 25.4 because as one thread is busy printing the file statistics, the other thread can continue reading a file.

[A] State whether the following statements are True or False:

(a) Multi-threading improves the speed of execution of the program.

(b) A running task may have several threads running in it.

(c) Multi-processing is same as multi-threading.

(d) If we create a class that inherits from the **Thread** class, we can still inherit our class from some other class.

(e) It is possible to change the name of the running thread.

(f) To launch a thread we must explicitly call the function that is supposed to run in a separate thread.

(g) To launch a thread we must explicitly call the **run( )** method defined in a class that extends the **Thread** class.

(h) Though we do not explicitly call the function that is supposed to run in a separate thread, it is possible to pass arguments to the function.

(i) We cannot control the priority of multiple threads that we may launch in a program.

[B] Answer the following questions:

(a) What is the difference between multi-processing and multi-threading?

(b) What is the difference between preemptive multi-threading and cooperative multi-threading?

(c) Which are the two methods available for launching threads in a Python program?

(d) If **Ex** class extends the **Thread** class, then can we launch multiple threads for objects of **Ex** class? If yes, how?

(e) What do different elements of the following statement signify?

```
th1 = threading.Thread(target = quads, args = (a, b))
```

(f) Write a multithreaded program that copies contents of one folder into another. The source and target folder paths should be input through keyboard.

(g) Write a program that reads the contents of 3 files a.txt, b.txt and c.txt sequentially and converts their contents into uppercase and writes them into files aa.txt, bb.txt and cc.txt respectively. The program should report the time required in carrying out this conversion. The files a.txt, b.txt and c.txt should be added to the project and filled with some text. The program should receive the file names as command-line arguments. Suspend the program for 0.5 seconds after reading a line from any file.

(h) Write a program that accomplishes the same task mentioned in Exercise [B](g) above by launching the conversion operations in 3 different threads.

[C] Match the following pairs:

a. Multiprocessing	1. use multiprocessing module
b. Pre-emptive multi-threading	2. use multi-threading
c. Cooperative multi-threading	3. use threading module
d. CPU-bound programs	4. use multi-processing
e. I/O-bound programs	5. use asyncio module