Part 82
-
Multi-Tasking
-
Threads

Version: 2021-01-14

Threads

Threads are one of the ways used to create multitasking applications. They are like sub-processes, but the main difference between a thread and process is that the threads created by an application share the same memory space. The system does not have to initialise a new system virtual memory space and a new environment with a thread. In general, threads are more effective on multi-processor or multi-core systems because all threads within a process share the same address space, communication and thread synchronisation is easier to manage and at the same time, the overhead is reduced.
Threads have unique Thread IDDs, a unique set of registers and stack pointer, local variables, and priorities. Threads in a process share the following:

- Most of the data
- Open files
- Signals
- Working directory
- Process instructions
- Working directory

Because threads share the same memory space, sharing data between them is easy and very fast. This can cause race problems if multiple threads attempt to work on the same data. The programmer has to make sure that the shared data is accessed properly and safely. There is no process level context switching in multi-threaded applications and as a result of this, threads give higher speeds. Threads are fast to start and terminate.

Threads run inside the same main process, in parallel with the main process. Usually, we create functions inside a main program and then activate these functions to run as independent parallel tasks, all as part of the main program. Threads are easy to program since they can simply be functions in a program and the programmer does not have to worry about creating child processes using fork or exec functions.
Although threads seem to be easy to program and run, they have a major disadvantage in that they are not completely independent processes. There is only one standard input (sys.stdin) and only one standard output (sys.stdout) and usage of input or output functions within the threads could easily cause conflicts. Also, threads have full access to shared process resources such as global memory and this requires careful planning to avoid any read or write conflicts.

Using threads

The function

```
thread.start_new(name, arguments)
```

is used to start a new thread. The child thread starts immediately and calls the function with the passed list of arguments. Here, name is usually the name of a function inside the main program which will start the thread. The argument field is optional and provides the means of passing data to the thread. This field should be a tuple. The newly created thread exists when the function returns (i.e exits). Also, the entire program exits when the main thread (i.e. the main program) exits.

An example program is given below. Two functions named `Thread1` and `Thread2` are declared and these functions are created as threads by calling to function `thread.start_new()`. There are no arguments passed to these threads in this example. `Thread1` displays the message `I am Thread1…` twice. Similarly, `Thread2` displays the message `I am Thread2…` 4 times. Both threads wait for one second before they continue. The main program waits forever so that the threads complete their tasks, since terminating the main program will also terminate all the threads.

```
nano thread1.py

import _thread
import time

# Thread 1
def Thread1():
  # Initialize k
  k = 0
  # Do twice
  while k < 2:
    # Display message
    print("I am Thread1…")
    # Wait 1 second
    time.sleep(1)
    k = k + 1 # Increment k

# Thread 2
def Thread2():
  # Initialize j
  j = 0
  # Do 4 time
  while j < 4:
   # Display message
   print("I am Thread2...")
   # Wait 1 second
   time.sleep(1)
   # Increment j
   j = j + 1

# Start Thread1
_thread.start_new_thread(Thread1, ())
# Start Thread2
_thread.start_new_thread(Thread2, ())

while True:
  pass
```

Output when you run (you need to quite by hitting Crtl+C)

```
python3 thread1.py
I am Thread1...
I am Thread2...
I am Thread1...
I am Thread2...
I am Thread2...
I am Thread2...
```

Next example program which shows how data can be passed to a thread through its arguments. In this example, `Thread1` displays the message `This is Thread1` twice, and `Thread2` displays the message `This is Thread2` 4 times.

```
import _thread
import time

# Thread 1
def Thread1(msg, cnt):
  # Initialize k
  k = 0
  # Do cnt times
  while k < cnt:
    # Display message
    print(msg)
    # Wait 1 second
    time.sleep(1)
    # Increment k
    k = k + 1

# Thread 2
def Thread2(msg, cnt):
  # Initialize j
  j = 0
  # Do cnt times
  while j < cnt:
    # Display message
    print(msg)
    # Wait 1 second
    time.sleep(1)
    # Increment j
    j = j + 1

_thread.start_new_thread(Thread1, ("This is Thread1", 2,))
_thread.start_new_thread(Thread2, ("This is Thread2", 4,))

while True:
  pass
```

Output when you run (you need to quite by hitting Crtl+C)

```
python3 thread2.py
This is Thread2
This is Thread1
This is Thread2
This is Thread1
This is Thread2
This is Thread2
```

In the above examples, notice the main parent code waits forever in a while loop and we have to type `Ctrl+C` keys to stop the program. We can modify the code and wait until both threads complete their tasks and exit normally and cleanly.

```
import _thread
import time
ThreadCount = 0

# Thread 1
def Thread1(msg, cnt):
  global ThreadCount
  # Initialize k
  k = 0
  # Do cnt times
  while k < cnt:
    # Display message
    print(msg)
    # Wait 1 second
    time.sleep(1)
    # Increment k
    k = k + 1
    # Increment ThreadCount
  ThreadCount = ThreadCount + 1

# Thread 2
def Thread2(msg, cnt):
  global ThreadCount
  # Initialize j
  j = 0
  # Do cnt (3) times
  while j < cnt:
    # Display message
    print(msg)
    # Wait 1 second
    time.sleep(1)
    # Increment j
    j = j + 1
    # Increment ThreadCount
  ThreadCount = ThreadCount + 1


_thread.start_new_thread(Thread1, ("This is Thread1", 2,))
_thread.start_new_thread(Thread2, ("This is Thread2", 4,))

while ThreadCount != 2: # Wait until both finish
  pass
print("End of program")
```

Output when you run

```
python3 thread3.py
This is Thread2
This is Thread1
This is Thread2
This is Thread1
This is Thread2
This is Thread2
End of program
```