EPAM Python Software Engineer Training

Lesson 2: Python Threads and Processes

Contents

1 Course **1**

1.1 GIL 1

1.2 Threads 1

1.3 Processes 2

2 Tasks **2**

2.1 Counter 2

2.2 Fork 2

2.3 Producer-Consumers 2

1 Course

1.1 GIL

In Python there is a GIL which is evil and the most evil it is when multi-threading paradigm is used. Hence, incorrect usage of multi-threading is evil in Python. In general it is OK to run many threads in Python, but if an application has both I/O bound and CPU bound threads - they will run into a conflict because of GIL - CPU-bound threads will always win GIL conflict, but an I/O bound thread do always run with higher priority by a system kernel. This is called a GIL contention. Please, see http://www.dabeaz.com/GIL/ for more in-deep research.

Hence, having even one CPU-bound thread in multi-threaded Python application is evil. Although, having multiple I/O bound threads shall be OK on a single-core hardware. However, most hardware is now multi-core. **Hence, each Python application shall be single-threaded.** Threads are only allowed to do some rare (regular) tasks like heartbit, ping etc.

1.2 Threads

Due to GIL threads are not recommended in Python, but sometimes they give a benefit. In most cases you don't have to spawn a thread in Python manually as most libraries do this already. If you still need to - this is as easy as creating either a *Thread* or a *Timer* class instance and passing a function into it.

A *threading* module also defines a set of useful synchronization primitives wrapped into convenient classes.

All signals in Python are handled in a main thread, but due to GIL it sometimes happens that a main thread does not get a chance to run in time causing a signal to be handled long time after it was triggered (sometimes it takes several dozens minutes). The worst thing is that during that time performance degrades majorly (due to GIL). Hence, think twice before you choose to rely onto signals in a multi-threaded application.

A thread in Python can be demonized. If there are only daemon threads left running in a process - it will exit.

1.3 Processes

Processes are a *golden hord* for Python multi-tasking. In most cases, if you don't need to pass large data between two tasks, spawning several processes shall be fine.

The easiest way to spawn a process in Python is to do *fork*. If some complex task is required like establishing a reliable data transfer channel between a parent and a child processes, a *multiprocessing* module can be of help.

2 Tasks

2.1 Counter

Spawn several *counter* threads each counting from `1` to a specified number and printing those numbers to the standard output. Each *counter* thread shall sleep for a specified number of seconds (float) between prints. There shall be a *status* thread which prints how many threads are still alive in a specified time interval. An application shall exit when all *counter* threads finish their work.

2.2 Fork

Write an application which recursively forks itself until a recursion depth reaches a specified value (passed as a command-line argument). Each child application shall print its process ID on start and exit; each parent application shall print a child process ID it has forked and wait for its completion. Below is an example log of an application stack:

```
Task 1 pid 1234 started
Task 1 pid 1234 spawned child 1238
Task 2 pid 1238 started
Task 3 pid 1251 started
Task 2 pid 1238 spawned child 1251
Task 3 pid 1251 complete
Task 2 pid 1238 complete
Task 1 pid 1234 complete
```

Note that it is OK if a log messages from a parent and a child go out of sync.

2.3 Producer-Consumers

Write an application which spawns several children threads or processes (based on a command line argument). A parent shall read from a file and put all lines into a queue. Children shall take those lines and append them into another file if and only if those lines start with a capital letter. An order of lines in a resulting file is not important, however, all lines shall be put intact. All threads (processes) shall exit gracefully after an input file ends and all necessary lines are put to output an file.

Compare a performance difference between the two solutions for `alice.txt`.