Python concurrency

All Versions

Examples

The multiprocessing module

```
from __future__ import print_function
import multiprocessing


def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()

    p1.join()
    p2.join()
```

Here, each function is executed in a new process. Since a new instance of Python VM is running the code, there is no `GIL` and you get parallelism running on multiple cores.

The `Process.start` method launches this new process and run the function passed in the `target` argument with the arguments `args` . The `Process.join` method waits for the end of the execution of processes `p1` and `p2` .

The new processes are launched differently depending on the version of python and the plateform on which the code is running *e.g.* :

- Windows uses `spawn` to create the new process.
- With unix systems and version earlier than 3.3, the processes are created using a `fork` .
  Note that this method does not respect the POSIX usage of fork and thus leads to unexpected behaviors, especially when interacting with other multiprocessing libraries.
- With unix system and version 3.4+, you can choose to start the new processes with either `fork` , `forkserver` or `spawn` using `multiprocessing.set_start_method` at the beginning of your program. `forkserver` and `spawn` methods are slower than forking but avoid some unexpected behaviors.

POSIX fork usage :

> After a fork in a multithreaded program, the child can safely call only async-signal-safe functions until such time as it calls execve.
> (see)

Using fork, a new process will be launched with the exact same state for all the current mutex but only the `MainThread` will be launched. This is unsafe as it could lead to race conditions *e.g.* :

- If you use a `Lock` in `MainThread` and pass it to an other thread which is suppose to lock it at some point. If the `fork` occures simultaneously, the new process will start with a locked lock which will never be released as the second thread does not exist in this new process.

Actually, this kind of behavior should not occured in pure python as `multiprocessing` handles it properly but if you are interacting with other library, this kind of behavior can occures, leading to crash of your system (for instance with numpy/accelerated on macOS).

The threading module

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
t2 = threading.Thread(target=countdown,args=(20,))
t2.start()
```

In certain implementations of Python such as CPython, true parallelism is not achieved using threads because of using what is known as the GIL, or **G** lobal **I** nterpreter **L** ock.

Here is an excellent overview of Python concurrency:

Python concurrency by David Beazley (YouTube)

Passing data between multiprocessing processes

Because data is sensitive when dealt with between two threads (think concurrent read and concurrent write can conflict with one another, causing race conditions), a set of unique objects were made in order to facilitate the passing of data back and forth between threads. Any truly atomic operation can be used between threads, but it is always safe to stick with Queue.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Creates a queue with an undefined maximum size
#this can be dangerous as the queue becomes increasingly large
#it will take a long time to copy data to/from each read/write thread
```

Most people will suggest that when using queue, to always place the queue data in a try: except: block instead of using empty. However, for applications where it does not matter if you skip a scan cycle (data can be placed in the queue while it is flipping states from `queue.Empty==True` to `queue.Empty==False`) it is usually better to place read and write access in what I call an Iftry block, because an 'if' statement is technically more performant than catching the exception.

```
import multiprocessing
import queue
'''Import necessary Python standard libraries, multiprocessing for classes and queue for the
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''This global method for the Iftry block is provided for it's reuse and
standard functionality, the if also saves on performance as opposed to catching
 the exception, which is expencive.
        It also allows the user to specify a function for the outgoing data to use,
 and a default value to return if the function cannot return the value from the queue'''
        if get_queue.empty():
            if use_default:
                return default
        else:
            try:
                value = get_queue.get_nowait()
            except queue.Empty:
                if use_default:
                    return default
            else:
                if use_func:
                    return func(value)
                else:
                    return value
    def Queue_Iftry_Put(put_queue, value):
        '''This global method for the Iftry block is provided because of its reuse
and
standard functionality, the If also saves on performance as opposed to catching
 the exception, which is expensive.
        Return True if placing value in the queue was successful. Otherwise, false'''
        if put_queue.full():
            return False
        else:
            try:
                put_queue.put_nowait(value)
            except queue.Full:
                return False
```

Syntax

Parameters

Remarks

The Python developers made sure that the API between `threading` and `multiprocessing` is similar so that switching between the two variants is easier for programmers.