Execution Concurrency in Python

Concurrency in Python

- Two kinds of tasks that can be used for concurrency:
  - Processes
    - Python multiprocessing module
    - have their own memory (e.g., variables)
    - A process has one or more threads
  - Threads
    - Python threading module
    - share the memory of the process that they are part of
    - are considered more lightweight in that they can spawn very quickly

Multiprocessing Pros/Cons

- Pros:
  - Separate memory space
  - Code is generally more straightforward
  - Takes better advantage of multiple CPUs/cores
  - Child processes can be interrupted/killed
- Cons:
  - Larger memory footprint
  - Interprocess communications is more complicated, i.e., IPC versus shared memory objects.

Threading Pros/Cons

- Pros:
  - Lightweight, i.e., low memory footprint
  - Shared memory that makes access to state from another thread easier
- Cons:
  - Code is a bit more complicated due to memory sharing.

Python Threading

Python Threads

- Enables multiple execution threads so that different tasks can be performed (almost) concurrently, i.e., in parallel (it is really timesharing the processor between threads).
- Example: a python network server can create a thread whenever a new client connects. This helps avoid some of the socket blocking behaviour that might otherwise occur.

Python Threading Module

- Import the threading module:

  `import threading`

- Create a new thread:

  `t = threading.Thread(target=handler, args=(i,))`

  where handler is the function to be threaded and args is a tuple of arguments passed to the function (Note that the comma may be needed in **`(i,)`** above to ensure that args is a tuple). Other arguments include naming the thread, e.g., **`name=<name>`**.

- Normally one keeps a list of the created threads:

  thread_list = [ ]

Python Threading Module

- Add the new thread to the thread list:

  ```
  thread_list.append(new_thread)
  ```

- We must start the execution of the thread, once it is created.

  ```
  new_thread.start()
  ```

- You can wait until a thread terminates. This is a way of synchronizing your code, e.g.,

  ```
  new_thread.join()
  ```

  This blocks the calling thread until the called thread terminates.

Python Threading Module

- Daemon vs non-daemon threads:

  `new_thread.daemon = True  # or False`

- When true, it means that the thread will be terminated when the calling script exits. Otherwise the script will not exit until any threads have completed execution.
- When true, the calling script doesn't have to worry about terminating threads that it has created.

Python Threading Examples

- See threading_example.py.
- daemon_thread_illustration.py
- See ping_threading_example.py
- See EchoClientServer_Thread.py

Synchronizing Access

- When using threads it is important to avoid conflicts when multiple threads need access to a single variable or resource.
- Overlapping accesses/modifications may cause all kinds of problems that can be load dependent and difficult to debug.
- One way of handling this is using atomic operations. In Python the follow are atomic:
  - reading/replacing a single instance attribute
  - reading/replacing a single global variable
  - getting an item from a list
  - modifying a list in place (e.g. using "append")
  - fetching an item from a dictionary
  - modifying a dictionary in place (e.g. adding an item, or calling the clear method)

Mutual Exclusion via Locking

- Note that operations that read a variable, modifies it, then writes it back are not atomic!
- It is safest to use locking, which is part of the threading module. This works as follows:

```
lock = threading.Lock()
...
lock.acquire() # this operation will block if
               # the lock is already held

... The shared resource is locked. Do whatever
you want with it ...

lock.release() # release the lock so another
               # thread can use the resource.
```

Locking

- In the previous example you need to make sure that the resource is released if something goes wrong. An alternate syntax:

```
lock = threading.Lock()
...
with lock:
    The shared resource is locked. Do whatever
    you want with it. When you exit this block,
    the lock is released.
```

- The advantage of this syntax is that acquire() and release() are automatically called when entering and leaving the "with" block. If something goes wrong, you will not be accidently leaving the resource locked forever.

Locking

- counter_function_variable.py
- counter_function_list.py
- counter_dict_locking_example.py

Multiprocessing

Multiprocessing

- The basic multiprocessing module uses a similar API as the threading module. e.g.,

```
import multiprocessing

new_process = multiprocessing.Process(

    target=self.connection_handler,
    args=(connection,))

new_process.start()
```

Multiprocessing

- In Windows you need to include at the start of your Python 3 script:

```
if sys.platform == 'win32':
    import multiprocessing.reduction
```

- You need to also call:

```
multiprocessing.freeze_support()
```

- See **EchoClientServer_Multiprocessing.py** example.

Multiprocessing

- This module contains a lot if sophisticated functionality, e.g.,
  - Queues and pipes for exchanging data between processes.
  - Contains equivalent of locking in thread module.
  - State can be shared between processes using shared memory