MultiThreading

Introduction:

Multithreading allows the execution of multiple parts of a program at the same time (in parallel). These parts are known as threads and are lightweight processes available within the process (like a nested process but their features not same as process). Multithreading leads to maximum utilization of the CPU by multitasking.

"concurrency is a condition that exists when at least two threads are making progress a more generalised form of parallelism that can include time-slicing as a form of virtual parallelism". -Sun's Multithreaded Programming Guide

"Parallelism: A condition that arises when at least two threads are executing simultaneously."
Sun's Multithreaded Programming Guide

The main Goal of MultiThreading is to to prevent tasks from blocking each other by switching threads back and forth, This is when one task takes long or gets into some waitable i/o task, the other thread will continue to make progress.

Libraries to Implement MultiThreading in Python:

- Threading
- Concurrent

Some definitions in MultiThreading:

1. Thread: *A thread is a path of execution within a process.*
2. CPython: an interpreter and a compiler as it compiles Python code into bytecode before interpreting it.
3. GIL(Global Interpreter Lock): lock that allows only one thread to hold the control of the Python interpreter. It only allow one thread to execute at a time even in a multi-threaded architecture with more than one CPU core. (ref: https://realpython.com/python-gil/)
4. *time.sleep(): suspend the execution of the current thread or process for some span of time.*
5. *Context Switching: A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU from one process or thread to another.*
6. syscall: **procedure that provides the interface between a process and the operating system**. It is the way by which a computer program requests a service from the kernel of the operating system.
7. Demon Thread: Any Background running thread is called Demon Thread
8. Race Condition: *When one resource gets modified by the multiple threads at same time.*
9. Semaphore: a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking OS.

Example 1:

In this Example we are creating a thread using `Timer` class, which call display function in new thread but with the buffer of some seconds.

```
from threading import Timer  # utilized to run a code after a specified time period

def display(msg):
  # https://docs.python.org/3/library/time.html
  print(msg + ' ' + time.strftime('%H:%M:%S'))
  return "YO"

# Basic timer:
def run_once():
  """
  a new thread start a counter of 5 sec, then display called with args
  this is now non-blocking, the context of execution will skip this
  then once the thread is over it, this child thread automatically join with the
  main/parent thread
  """
  ans = display('Run once:')
  print(ans)


t = Timer(2, display, args=['Timeout:'])
t.start()  # starting a new Thread

run_once()
print("GO!")


O/P:
Run once: 12:50:51
YO
GO!
Timeout: 12:50:53
```

Example 2:

Move a functions to multiple thread and will wait for all of them to complete.

```
import time
import logging
from threading import Thread

no_tasks = 7  #

def long_task(name):
```

```
    """This Function will take 14 sec to complete"""
    # no_tasks = random.randrange(start=0, stop=10, step=1)
    logging.info(f"Task: {name} performing {no_tasks} tasks with each 2 sec.")
    for x in range(no_tasks):
        logging.info(f'Task {name}: {x}')
        time.sleep(2)
    logging.info(f'Task: {name}: completed')


# Main function
def main():
  logging.basicConfig(format='%(levelname)s - %(asctime)s: %(message)s', datefmt='%H:%M:%S',
level=logging.DEBUG)
  logging.info('Starting')

  threads: list[Thread] = []
  for x in range(no_tasks):
    t = Thread(target=long_task, args=['thread: ' + str(x)])
    threads.append(t)
    t.start()
```

Example to Demonstrate `Semaphore` :

`Semaphore` grants the **access to the thread to access the shared resource which it tries to acquire a permit**.

Every class in this example has two same functions `double_it` and `half_it` , as name suggest both of them try to access `num` and try to double it and half it using multiple threads.

```
import time
import threading

import logging
logging.basicConfig(format='%(levelname)s - %(asctime)s: %(message)s', datefmt='%H:%M:%S',
level=logging.DEBUG)
logging.info('Starting')


class SimpleMultiThreadingExample:
    """
    The result of this Example is:
    the value of `num` seems never be changed
    because in the `main` we have spawned two threads which parallely try to change the value.
    hence, always print 1
    """
    num = 1

    def double_it(self):
```

```
        # global num
        while 0 < self.num < 20_000:
            self.num *= 2
            time.sleep(1)
            logging.info(self.num)

        logging.info("Exiting double_it!")

    def half_it(self):
        # global num
        while 0 < self.num < 40_000:
            self.num /= 2
            logging.info(self.num)
            time.sleep(1)

        logging.info("Exiting half_it!")


class LockMultiThreadingExample:
    """
    Note: the both the lock objects in both the methods are same and
    if you will create new object in both the methods this will not work.
    """
    num = 1
    lock = threading.Lock()

    def double_it(self):
        lock_obj = self.lock
        lock_obj.acquire()
        while 0 < self.num < 20_000:
            self.num *= 2
            time.sleep(1)
            logging.info(self.num)

        logging.info("Exiting double_it!")
        lock_obj.release()

    def half_it(self):
        lock_obj = self.lock
        lock_obj.acquire()
        while 1 < self.num < 40_000:
            self.num /= 2
            logging.info(self.num)
            time.sleep(1)

        logging.info("Exiting half_it!")
        lock_obj.release()


class SemaphoreThreadingExample:
```

```
    """
    usage of semaphore is very much same as Locking
    Let's limit the access to the variable so that not my unlimited threads try to access
    my single mf resource.
    """
    num = 1
    sem = threading.Semaphore()

    def double_it(self):
        sem_obj = self.sem
        sem_obj.acquire()
        while 0 < self.num < 20_000:
            self.num *= 2
            time.sleep(1)
            logging.info(self.num)

        logging.info("Exiting double_it!")
        sem_obj.release()

    def half_it(self):
        sem_obj = self.sem
        sem_obj.acquire()
        while 1 < self.num < 40_000:
            self.num /= 2
            logging.info(self.num)
            time.sleep(1)

        logging.info("Exiting half_it!")
        sem_obj.release()


class AttackingSemaphoreThreadingExample:
    """
    when multiple Threads try to access the single Resource
    """
    num = 1
    sem = threading.Semaphore()

    def double_it(self):
        logging.info(f"Trying to Access: {threading.get_ident()}")
        sem_obj = self.sem
        sem_obj.acquire()
        while 0 < self.num < 20_000:
            logging.info(f"Updating by: {threading.get_ident()}")
            self.num *= 2
            time.sleep(1)
            logging.info(self.num)

        logging.info("Exiting double_it!")
        sem_obj.release()
```

```
    def half_it(self):
        sem_obj = self.sem
        sem_obj.acquire()
        while 1 < self.num < 40_000:
            self.num /= 2
            logging.info(self.num)
            time.sleep(1)

        logging.info("Exiting half_it!")
        sem_obj.release()


if __name__ == '__main__':
    # 1.
    obj = SimpleMultiThreadingExample()
    t1 = threading.Thread(target=obj.double_it)
    t2 = threading.Thread(target=obj.half_it)

    # t1.start()
    # t2.start()

    # 2.
    obj = LockMultiThreadingExample()
    t3 = threading.Thread(target=obj.double_it)
    t4 = threading.Thread(target=obj.half_it)

    # t3.start()
    # t4.start()

    # 3.
    obj = SemaphoreThreadingExample()
    t5 = threading.Thread(target=obj.double_it)
    t6 = threading.Thread(target=obj.half_it)

    # t5.start()
    # t6.start()

    # 4.
    obj = AttackingSemaphoreThreadingExample()
    for _ in range(5):
        t7 = threading.Thread(target=obj.double_it)
        t7.start()
        time.sleep(0.4)
```

Example to use `ThreadPoolExecutor`:

ThreadPoolExecuter, class defined in Concurrent Package, designed to handle Threads in a more controlled manner. It gives you the control to reuse the idle threads without spawning new Threads.

```
wait_time = 10

def some_task(item):
  """This Function will take 14 sec to complete"""
  # no_tasks = random.randrange(start=0, stop=10, step=1)
  logging.info(f"Task: {item} started!")
  # id of current Thread, is created by OS and id belongs to the worker
  logging.info(f'Thread {item}: id = {get_ident()}'
  logging.info(f'Thread {item}: sleeping for {wait_time}')
  time.sleep(random.randrange(wait_time)
  logging.info(f'Thread {item}: finished')

# Main function
def main():
  logging.basicConfig(
    format='%(levelname)s - %(asctime)s: %(message)s',
    datefmt='%H:%M:%S',
    level=logging.DEBUG)
   logging.info('App Start')

cores = 4  # MacBook Pro cores
workers = 2*cores + 1
items = 20

# No need to Join the Threads
# No need to Monitor or Handle the Threads
# automatically spawn a new worker when there is
# Said objects use significant amount of memory and for last project uses the large memory.
# To reduce this memory management overhead (allocating and deallocating many threads)

with ThreadPoolExecutor(max_workers=workers) as executor:
  executor.map(some_task, range(0, items))
  # some of the ids will gets repeated in the terminal that depicts the reuse of Threads
  logging.info('App Finished')

if __name__ == "__main__":
  main()
```