11. Concurrent execution

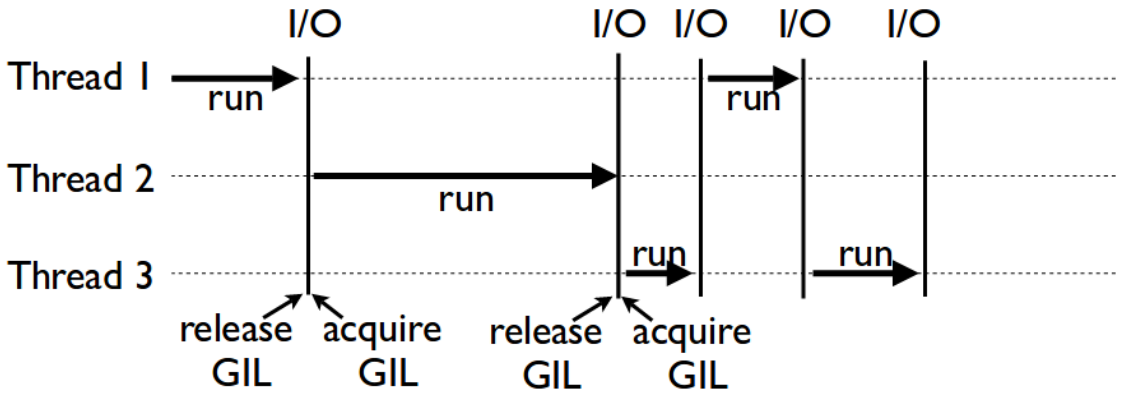
October 16, 2025

1 Concurrent execution

The two main modules in Python Standard Library for parallelizing code are `threading` and `multiprocessing`. When deciding between the two, you should ask yourself: what kind of program are you trying to paralellize? * I/O bound program -> `threading` * CPU bound program -> `multiprocessing`

1.1 `threading`

A thread is a separate flow of execution. This means that your program will have multiple things happening at once. But for most Python implementations the different threads do not actually execute at the same time: they merely appear to. In CPython, multi-threading is supported by introducing the Global Interpreter Lock (GIL) which to prevent multiple threads from accessing the same Python object simultaneously. Only one thread can hold the GIL at a time, one thread must wait for another thread to release the GIL before running.



1.1.1 Using `threading` module

`threading.Thread` class

- Represents an activity that is run in a separate thread of control
- Activity can be specified by passing a callable to constructor or by overriding run() method in a subclass
- Activity should be started by calling start() method of Thread instance
- Threads can be named by setting `name` attribute
- `start()` – Starts thread activity. Can be called only one time for a thread. It launches threads's `run()` method in a separate thread of control

- `run()` – Method representind threads's activity. Defaults to callable passed in constructor as `target` parameter
- `join([timeout])` – Blocking call that waits for specific thread to terminate
- `is_alive()` – Queries thread status. Returns `True` unitl `run()` method terminates

[1]:
```
import threading
import time


def func(nr):
    time.sleep(3)
    print(f"Finished execution from thread {nr}")


time_start = time.time()
threads = []

for i in range(1, 6):
    t = threading.Thread(target=func, args=(i, ))
    threads.append(t)
    t.start()  # target(*args)

    print(f'Thread {t.name} alive status: {t.is_alive()}')

    count = threading.active_count()
    print(f"Total no of threads: {count}")

for thread in threads:
    thread.join()

time_end = time.time()
total_time = time_end - time_start
print(f'Total time: {total_time:.10f}')
```

```
Thread Thread-5 (func) alive status: True
Total no of threads: 9
Thread Thread-6 (func) alive status: True
Total no of threads: 10
Thread Thread-7 (func) alive status: True
Total no of threads: 11
Thread Thread-8 (func) alive status: True
Total no of threads: 12
Thread Thread-9 (func) alive status: True
Total no of threads: 13
Finished execution from thread 1Finished execution from thread 3
Finished execution from thread 5
Finished execution from thread 2
```

```
Finished execution from thread 4
Total time: 3.0063438416
```

1.2 multiprocessing

Multiprocessing allows you to create programs that can run concurrently (bypassing the GIL) and use the entirety of your CPU core. Though it is fundamentally different from the threading library, the syntax is quite similar. The multiprocessing library gives each process its own Python interpreter and each their own GIL.

Because of this, the usual problems associated with threading (such as data corruption and deadlocks) are no longer an issue. Since the processes don't share memory, they can't modify the same memory concurrently.

If when using `threads` *how not to share* is a problem (the memory space is common to all running threads of a certain process), when using `process` *how to share* becomes a problem to solve (all process have their own copy of the initial parent process memory; what one changes, the other would not be able to see)

[1]:
```python
from multiprocessing import Process, current_process
import time
import random


def doubler(nr):
    """
    A doubling function that can be used by a process
    """
    # Mock time-consuming processing:
    x = 0
    for i in range(1000000):
        x += random.randint(1, 10)

    result = nr * 2
    proc_name = current_process().name
    print('{} doubled to {} by: {}'.format(nr, result, proc_name))


if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []

    time0 = time.time()
    for number in numbers:
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    for proc in procs:
```

```
        proc.join()

    time1 = time.time()
    print(f'Total execution time (parallel): {time1-time0:.10f}')

    time0 = time.time()

    for number in numbers:
        doubler(number)

    time1 = time.time()
    print(f'Total execution time (serial): {time1 - time0:.10f}')
```

```
10 doubled to 20 by: Process-2
5 doubled to 10 by: Process-1
25 doubled to 50 by: Process-5
20 doubled to 40 by: Process-4
15 doubled to 30 by: Process-3
Total execution time (parallel): 0.3718290329
5 doubled to 10 by: MainProcess
10 doubled to 20 by: MainProcess
15 doubled to 30 by: MainProcess
20 doubled to 40 by: MainProcess
25 doubled to 50 by: MainProcess
Total execution time (serial): 0.9248859882
```

Since processes are still suitable for I/O - let's see an example of I/O operations performed: - sequentially - concurrently using threads - concurrently using processes

[2]:
```
from multiprocessing.dummy import Pool as ThreadPool
from multiprocessing import Pool as ProcessPool
from urllib.request import urlopen
import time


def get_url_resp(url):
    resp = urlopen(url)
    return len(resp.read())


urls = [
    'http://www.python.org',
    'http://www.python.org/about/',
    'http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html',
    'http://www.python.org/doc/',
    'http://www.python.org/download/',
    'http://www.python.org/getit/',
    'http://www.python.org/community/',
```

```
]

if __name__ == '__main__':
    d1 = time.time()
    syncres = [get_url_resp(url) for url in urls]
    # list(map(get_url_resp, urls))
    d2 = time.time()
    print(f'Serial result: {syncres}\nExecution time: {d2 - d1:.10f}\n')

    d1 = time.time()
    # make the Pool of thread workers
    pool = ThreadPool(7)

    # open the urls in their own threads
    # and return the results
    asyncres = pool.map(get_url_resp, urls)

    # close the pool and wait for the workers to finish
    pool.close()
    pool.join()

    d2 = time.time()

    print(f'Concurrent result (threads): {asyncres}\nExecution time: {d2 - d1:.
 ↪10f}\n')

    d1 = time.time()
    # make the Pool of process workers
    pool = ProcessPool(7)

    # open the urls in their own processes
    # and return the results
    asyncres = pool.map(get_url_resp, urls)

    # close the pool and wait for the work to finish
    pool.close()
    pool.join()

    d2 = time.time()

    print(f'Concurrent result (processes): {asyncres}\nExecution time: {d2 - d1:
 ↪.10f}\n')
```

```
Serial result: [50156, 9489, 78375, 8855, 19542, 19518, 8625]
Execution time: 5.8237507343

Concurrent result (threads): [50156, 9489, 78375, 8855, 19542, 19518, 8625]
Execution time: 1.3206427097
```

```
Concurrent result (processes): [50156, 9489, 78375, 8855, 19542, 19518, 8625]
Execution time: 1.2490136623
```

1.2.1 `concurrent.futures` — Launching parallel tasks

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

1.2.2 ThreadPoolExecutor

`ThreadPoolExecutor` is used for managing a pool of threads. It is suitable for I/O-bound tasks, as threads can run concurrently, and efficiently handle multiple tasks, such as network requests or file I/O.

Example: Using ThreadPoolExecutor

```
import concurrent.futures
import time

def io_bound_task(seconds):
    print(f"Sleeping for {seconds} second(s)...")
    time.sleep(seconds)
    return f"Slept for {seconds} second(s)"

# Create a ThreadPoolExecutor with a pool of 3 threads
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    # List of tasks to be run concurrently
    tasks = [1, 2, 3, 4, 5]

    # Submit tasks to the executor
    futures = [executor.submit(io_bound_task, t) for t in tasks]

    # Process results as they complete
    for future in concurrent.futures.as_completed(futures):
        print(future.result())
```

1.2.3 ProcessPoolExecutor

`ProcessPoolExecutor` is used for managing a pool of processes. It is suitable for CPU-bound tasks, as processes can run on multiple cores, thus parallelizing the computation.

Example: Using ProcessPoolExecutor

```
import concurrent.futures
import time

def cpu_bound_task(number):
```

```
    print(f"Computing factorial of {number}...")
    result = 1
    for i in range(2, number + 1):
        result *= i
    return result

# Create a ProcessPoolExecutor with a pool of 3 processes
with concurrent.futures.ProcessPoolExecutor(max_workers=3) as executor:
    # List of tasks to be run concurrently
    tasks = [10, 20, 30, 40, 50]

    # Submit tasks to the executor
    futures = [executor.submit(cpu_bound_task, t) for t in tasks]

    # Process results as they complete
    for future in concurrent.futures.as_completed(futures):
        print(f"Factorial computed: {future.result()}")
```

1.2.4 Key Points

1. **ThreadPoolExecutor** is ideal for I/O-bound tasks due to its use of threads.
2. **ProcessPoolExecutor** is ideal for CPU-bound tasks due to its use of processes.
3. Both executors manage a pool of workers, allowing you to submit multiple tasks which are then distributed among the available workers.
4. Use `executor.submit(function, *args, **kwargs)` to submit tasks to the executor.
5. Use `concurrent.futures.as_completed(futures)` to process the results as they complete.

1.2.5 Error Handling

Both executors provide mechanisms to handle errors gracefully. You can wrap your task functions in try-except blocks or use the `future.result()` method to raise exceptions if they occurred during execution.

Example: Error Handling

```
import concurrent.futures

def task_with_error(x):
    if x == 3:
        raise ValueError("An error occurred with input 3")
    return x * 2

with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    tasks = [1, 2, 3, 4, 5]
    futures = [executor.submit(task_with_error, t) for t in tasks]

    for future in concurrent.futures.as_completed(futures):
        try:
            result = future.result()
```

```
        print(f"Result: {result}")
    except Exception as e:
        print(f"Error: {e}")
```

1.2.6 Exercises

1. Try out different strategies presented above to speed up execution for process_files.py. Compare results.

[ ]: