Simple example

Sequential version

The `sleeper` function below sleeps for `n` seconds. It is described by the variable `identifier`. Recall that during sleep time, the CPU is not used.

In [1]:
```
from datetime import datetime
from time import sleep

def gt():                                          # get_time
    return datetime.now().strftime('%H:%M:%S')

def sleeper(n, identifier):
    print(f"[{gt()}] I am '{identifier:^8}', and I will sleep for {n} seconds")
    sleep(n)
    print(f"[{gt()}] I am '{identifier:^8}', and my sleeping time was so good!")
```

Then the `f_sequential` calls `sleeper` using differents arguments.

No surprise here, everything runs sequentially:

In [2]:
```
args = [(3, 'First'), (2, 'Second'), (4, 'Third')]

def f_sequential():
    for arg in args:
        sleeper(*arg)

f_sequential()
```

```
[19:24:31] I am ' First  ', and I will sleep for 3 seconds
[19:24:34] I am ' First  ', and my sleeping time was so good!
[19:24:34] I am ' Second ', and I will sleep for 2 seconds
[19:24:36] I am ' Second ', and my sleeping time was so good!
[19:24:36] I am ' Third  ', and I will sleep for 4 seconds
[19:24:40] I am ' Third  ', and my sleeping time was so good!
```

Threaded version

Code

Let's define a function `f_threading` that separate each call to `sleeper` in a dedicated thread.

The use of threads in Python is made simple by the **`ThreadPoolExecutor` class** of the `concurrent` package (rather than the `threading` package).

In [2]:
```
from concurrent.futures import ThreadPoolExecutor

def f_threading(max_workers=2):
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        for arg in args:
            executor.submit(sleeper, *arg) # unpacking is needed here
                                           # because `arg` is a tuple
```

Notice `max_workers=2`. This means that at every time no more than 2 threads can be running (no matter they use the CPU or not).

The executor submits the call to `sleeper` for each argument tuple.

Here is the result:

In [4]:
```
f_threading()
```

```
[19:24:40] I am ' First  ', and I will sleep for 3 seconds[19:24:40] I am '
Second ', and I will sleep for 2 seconds

[19:24:42] I am ' Second ', and my sleeping time was so good!
[19:24:42] I am ' Third  ', and I will sleep for 4 seconds
[19:24:43] I am ' First  ', and my sleeping time was so good!
[19:24:46] I am ' Third  ', and my sleeping time was so good!
```

Explanation

- The first call to `sleeper` is with `(3, 'First')`. `sleeper` performs the first `print` and then go to sleep for 3 seconds.
- While the first call is sleeping, it does not need the CPU. Thus the second call (with `(2, 'Second')`) starts: first `print` statement and then go to sleep too.
- Neither the `sleep` of the first call nor the one of the second call came to an end. Yet, executor is not allowed to start a 3rd thread due to `max_worker=2`. Thus, the executor wait for the second call to end (the shortest one) t perform the third call.
- The first call terminates.
- The third call terminates too.

Eventually the execution of `f_threading` took only 6 seconds, which is lower than the 9 seconds of the sequential version.

What if `max_workers` is set to 3?

In [5]:
```
f_threading(max_workers=3)
```

```
[19:24:46] I am ' First  ', and I will sleep for 3 seconds
[19:24:46] I am ' Second ', and I will sleep for 2 seconds
[19:24:46] I am ' Third  ', and I will sleep for 4 seconds
[19:24:48] I am ' Second ', and my sleeping time was so good!
[19:24:49] I am ' First  ', and my sleeping time was so good!
[19:24:50] I am ' Third  ', and my sleeping time was so good!
```

Notes

In a real code, this is not a call to the `sleep` function that monopolize the CPU. It may be:

- the writing of a big file on disk
- the time waiting for some internet server response

Sharing some variables

Sequential version

Code

Let's define a function, similar to `sleeper`, that modifies a **mutable** variable passed as an argument. This variable is a one-integer list, and the modification consists in incrementing this integer.

In [3]:
```
def sleeper(n, identifier, var):
    actual_value = var[0]
    print(f"[{gt()}] I am '{identifier:^8}', and I will sleep for {n} seconds")
    sleep(n)
    var[0] = actual_value + 1
    print(f"[{gt()}] I am '{identifier:^8}', and my sleeping time was so good!")
    print(f"[{gt()}] I am '{identifier:^8}', and I modified var[0]: it was {actual_va
```

Let's assume that `var` is shared among all calls. Thus it is defined once before `args` and a reference is passed to `args` (this is not a copy).

In [7]:
```
var = [10]
args = [(3, 'First', var), (2, 'Second', var), (4, 'Third', var)]
f_sequential()
```

```
[19:24:50] I am ' First  ', and I will sleep for 3 seconds
[19:24:53] I am ' First  ', and my sleeping time was so good!
[19:24:53] I am ' First  ', and I modified var[0]: it was 10, it is now 11!
[19:24:53] I am ' Second ', and I will sleep for 2 seconds
[19:24:55] I am ' Second ', and my sleeping time was so good!
[19:24:55] I am ' Second ', and I modified var[0]: it was 11, it is now 12!
[19:24:55] I am ' Third  ', and I will sleep for 4 seconds
[19:24:59] I am ' Third  ', and my sleeping time was so good!
[19:24:59] I am ' Third  ', and I modified var[0]: it was 12, it is now 13!
```

Explanation

The sequential version is ok. There are 3 function calls, each call adds 1 to var[0] which is initialized with 10, thus we end up with $13 = 10 + 3$.

Naïve threading

Code

Let's call `f_threading` using these new arguments.

In [8]:
```
var[0] = 10
f_threading()
```

```
[19:24:59] I am ' First  ', and I will sleep for 3 seconds
[19:24:59] I am ' Second ', and I will sleep for 2 seconds
[19:25:01] I am ' Second ', and my sleeping time was so good!
[19:25:01] I am ' Second ', and I modified var[0]: it was 10, it is now 11!
[19:25:01] I am ' Third  ', and I will sleep for 4 seconds
[19:25:02] I am ' First  ', and my sleeping time was so good!
[19:25:02] I am ' First  ', and I modified var[0]: it was 10, it is now 11!
[19:25:05] I am ' Third  ', and my sleeping time was so good!
[19:25:05] I am ' Third  ', and I modified var[0]: it was 11, it is now 12!
```

Explanation

What is going on?

- `'First'` stores the value `var[0]` in `actual_value`.
- While `'First'` is sleeping, `'Second'` modifies `var[0]`. `var[0]` now redirects to another integer, which is different from what `'First'` stored in `actual_value`.
- `First` wakes up and modifies `var[0]` according to the obsolete `actual_value`. Thus the incrementation process is broken.
- etc ...

Protected threading

The `var` variable must be protected, for instance using a `Lock` object (or similarly, a `Semaphore`).

A `Lock` object can be used only by one thread at a time. A thread must 'acquire' the permission and 'release' it.

Let's redefine `sleeper`.

In [9]:

```
from threading import Lock

var[0] = 10

def sleeper(lock, n, identifier, var):
    with lock:
        actual_value = var[0]
        print(f"[{gt()}] I am '{identifier:^8}'",
              f" and I will sleep for {n} seconds")
        sleep(n)
        var[0] = actual_value + 1
        print(f"[{gt()}] I am '{identifier:^8}",
              " and my sleeping time was so good!")
        print(f"[{gt()}] I am '{identifier:^8}', and I modified var[0]:",
              f" it was {actual_value}, it is now {var[0]}!")

def f_threading_lock(max_workers=2):
    lock = Lock()
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        for arg in args:
            executor.submit(sleeper, lock, *arg)

f_threading_lock()
```

```
[19:25:05] I am ' First  '  and I will sleep for 3 seconds
[19:25:08] I am ' First    and my sleeping time was so good!
[19:25:08] I am ' First  ', and I modified var[0]:  it was 10, it is now 11!
[19:25:08] I am ' Third  '  and I will sleep for 4 seconds
[19:25:12] I am ' Third    and my sleeping time was so good!
[19:25:12] I am ' Third  ', and I modified var[0]:  it was 11, it is now 12!
[19:25:12] I am ' Second '  and I will sleep for 2 seconds
[19:25:14] I am ' Second   and my sleeping time was so good!
[19:25:14] I am ' Second ', and I modified var[0]:  it was 12, it is now 13!
```

This is working! By asking the acquisition of `lock`, everything that is inside the `with` bloc of `sleeper` **must be ran in one go**.

Notes

- In this example, the protection using `Lock` is a particular extreme case because all the instructions of `sleeper` are protected. Thus execution is sequential.
- Only mutable variables must be protected.

  Similarly, whenever some data is written to the disk, the writing process must be protected too in order to prevent data corruption.