KAPLAN BUSINESS SCHOOL AUSTRALIA

ASSESSMENT 2: INTERNAL CODING REPORT

TECH6200: Advanced Programming

Brief description: Case Study
John's routine: sequential, asynchronous, multithreading.

Javier Tejeda Castro
1846855@kaplanstudent.edu.au

1. Scenario: sequential routine

The provided scenario was very direct: John performs several tasks with fixed durations, one after the other one. In this case, asynchronous programming and multithreading are not helpful because the flow is strictly sequential, and can be implemented just with the use of "time.sleep()".

Code Structure:
-Duration values are assigned to global variables that can be easily spotted.
-Single generic function "routine_activity" with "activity name and duration parameters is created to reduce repetition and improve reusability.
-Looping to call "routine_activity" in main function.

Additional Features:
-Variable "SPEED" is added with testing purposes, to scale down the waiting time.
-Transition time (5 seconds): an additional delay between activities is added to simulate the time spent when moving from one task to the next one.

Insights:
It is important to understand that efficiency is usually related to simplicity, especially with the vast range of python resources, and scenario analysis is key to provide the right solution and not overcomplicate the code. In this case, because a concept can be applied (code could have been done importing asyncio library), it does not mean that is the best solution (simple sequential code).
I also understood how "time.sleep()" blocks the execution and prevents overlapping.

2. Scenario: asynchronous routine

In this scenario John uses waiting time (check notifications while tea cools down), which implies that the tasks overlap and run concurrently making it an asynchronous programming case. Using "asyncio" library, tasks can be written as coroutines that can be started and paused without blocking the program.

Code Structure:
-Global constants regarding task duration, task transition and speed(for testing).
-Generic asynchronous function definition with general task behaviour (using asyncio.sleep()).
-Asynchronous function definition per each task performed by John.
-Task orchestration happens in "main()", where it is decided which activities sequentially block and which ones overlap.

Additional Features:
Modified routine: new tasks and different ways of overlapping to dig more into asynchronous programming:
-1st: John "eats sandwich" without overlap (could be written as sequential task (1st scenario), but for consistency and potential scalability, I used "async def").
-2nd: John "enjoys tea", and check "work Notifications" while tea gradually cools down.
-3rd: John "eats fruit". While he chews, he finishes checking work notifications and after checking them all, he starts "scrolling down through social media".
-4th: After finishing the fruit, he keeps scrolling down social media.
-5th: imagine social media is taking some time to load, and John gets distracted by a "kookaburra laughing" (which finish at the same time than social media scrolling).

Insights:
The program made me understand the difference between blocking(time.sleep) and not-blocking(await asyncio.sleep) waits. I experimented with extra overlapping tasks and how to manage them using “await” (code flow pauses until task is complete), “asyncio.create_task” (launchs task on the background) and “asyncio.gather” (wait until several tasks finish together). The decision of keeping the first task in consistency with the rest of the code made me think about the topic of clarity vs efficiency, and, like in this case, efficiency is not always above clarity and it is important to find a balance in your code.

3. Scenario: multithreading routine

In this scenario, John performs tasks within cycles. All the activities progress together in each cycle tehrefore it is considered a multithreading case. ThreadPoolExecutor has been chosen instead of manual threads (Threading library) because it is more efficient and clearer when running little and repeated tasks.

Code Structure:
-Global variable dictionary that stores different tasks as keys and their progress as values.
-Generic “routine activity” worker function that update the dictionary values under a lock.
-ThreadPool creation and cycle iteration happen in “main()”. Each cycle ends when the last activity finishes, thanks to “task.result()” (although “wait()” is equally effective).

Additional Features:
- Introduction of task progress with a lock:
    - Tasks start with 0% and finish when reach 100% of progression.
    - Addition of 10-20% of progression per cycle.
    - Progression is recorded in a global dictionary variable that needs to be updated from all the threads. The use of a Lock while reading/updating the progression of the task is needed.
    - Logic is defined in “routine_activity” function and it is called from “main()”.
- Added meaningful waiting time with “time.sleep()” to make overlapping visible.
- Speed variable to modify running time when testing.
- For every cycle, the general progress of the routine is displayed.

Insights:
When testing, I realized that the code run faster using a single thread than four threads. After searching for information, I realised of several things that were “counterproductive” and changed my code to improve it:

- Threading is really useful when the tasks have enough waiting time, not with quick activities.
- Keeping most of the task activity under the lock means that threads spend more time waiting for the lock and less time running in “parallel”. Efficient lock placement and taking “snapshots” of shared variables to use them out of the lock resulted very useful.
- “Print()” function has an internal lock and slows down the code when used frequently.

References:

CodersLegacy (n.d.) *Python lock in with statement.* Available at: https://coderslegacy.com/python/lock-in-with-statement/
(Accessed: 26 August 2025).

GeeksforGeeks (2023) *Asyncio in Python*. Available at: https://www.geeksforgeeks.org/python/asyncio-in-python/
(Accessed: 26 August 2025).

Kaplan Business School (n.d.) *TECH6200 Advanced Programming — Lesson 05: Concurrency: Threads, Coroutines and Asynchronous Programming* (Workshop slides). Kaplan Business School.

Kaplan Business School (n.d.) *TECH6200 Advanced Programming — Lesson 06: Parallelism: Multi-processing* (Workshop slides). Kaplan Business School.

Python Software Foundation (2024a) *concurrent.futures — Launching parallel tasks.* Python 3 documentation. Available at: https://docs.python.org/3/library/concurrent.futures.html
(Accessed: 26 August 2025).

Python Software Foundation (2024b) *threading — Thread-based parallelism*. Python 3 documentation. Available at: https://docs.python.org/3/library/threading.html
(Accessed: 26 August 2025).