DSAI 3202 – Parallel and distributed computing
Lab – 4: Temperature Monitoring System

1. Objectives:

- Develop a Python program that simulates temperature readings from multiple sensors, calculates average temperatures, and displays the information in real-time in the console.
- ***The objective of this lab is not to display in place. That's just an added bonus for your programming portfolio.***

2. Implement Sensor Simulation.

2.a. Question

- Write a function called **`simulate_sensor`** that simulates temperature readings from a sensor.
- Use **`random.randint(15, 40)`** to generate random temperatures.
- Make **`simulate_sensor`** update a global dictionary **`latest_temperatures`** with its readings every second.

2.b. Solution to *Implement Sensor Simulation*.

i) The function.

```python
def simulate_sensor(sensor_id: int = 0):
    """
    Generate random temperature readings and put them in a dictionary.
    Parameters:
        - sensor_id (int): Default 0. The id of the sensor.
    """
    while True:
        temperature_reading = random.randint(15, 30)
        latest_temperatures[sensor_id] = temperature_reading
        time.sleep(1)
```

3. Implement Data Processing

3.a. Question

- Write a function called **`process_temperatures`** that continuously calculates the average temperature from readings placed in a queue.
- Make **`process_temperatures`** update a global dictionary **`temperature_averages`** with the calculated averages.

3.b. Solution to implement data processing.

i) How are we going to compute the average per sensor?

- The **`simulate_sensor`** gives only the last temperature.
- The average temperature (μ_T) for n samples, is given by:

$$\mu_T(n) = \frac{1}{n}\sum_{i=1}^{n} T_i$$

- Similarly, the average temperature for $n + 1$ samples, is given by:

$$\mu_T(n+1) = \frac{1}{n+1}\sum_{i=1}^{n+1} T_i$$

- We also know that:

$$\sum_{i=1}^{n+1} T_i = \underbrace{\{T_1 + T_2 + \cdots + T_n\}}_{\sum_{i=1}^{n} T_i} + T_{n+1} = \sum_{i=1}^{n} T_i + T_{n+1}$$

- We also can deduce that:

$$\mu_T(n) = \frac{1}{n}\sum_{i=1}^{n} T_i \Leftrightarrow \sum_{i=1}^{n} T_i = n \cdot \mu_T(n)$$

- We replace in the $\mu_T(n+1)$ equation:

$$\mu_T(n+1) = \frac{1}{n+1}\sum_{i=1}^{n+1} T_i$$

$$\mu_T(n+1) = \frac{1}{n+1}\left(\sum_{i=1}^{n} T_i + T_{n+1}\right)$$

$$\mu_T(n+1) = \frac{1}{n+1}(n \cdot \mu_T(n) + T_{n+1})$$

$$\mu_T(n+1) = \frac{n \cdot \mu_T(n) + T_{n+1}}{n+1}$$

In simple terms, to update the average, we take the previous average and multiply it by the previous number of samples. Then, we add the current measurement and divide the whole by whole the current number of samples.

ii) Updating the `simulate_sensor` function.

- In order for the **`process_temperatures`** to compute the average, we need the **`simulate_sensor`** function to make the values available for the **`process_temperatures`**. This is known as the ***Producer-Consumer*** pattern. One part of the program makes available the data and the other part uses them.
- For this program we are going to use a **`Queue`** which is FIFO list. The **`simulate_sensor`** pushes the last temperature into the **`Queue`** and the **`process_temperatures`** process them as they arrive.
- We also introduce a **`counters`** list to help compute the averages per sensor.
- The program becomes:

```python
import random
import time

from queue import Queue
from threading import Thread

# create a global dictionaries for the latest and average temperatures
latest_temperatures = {}
average_temperatures = {}
counters = 0

def simulate_sensor(sensor_id: int = 0,
                    queue: Queue = Queue()):
    """
    Generate random temperature readings and put them in a dictionary.
    Parameters:
        - sensor_id (int): Default 0. The id of the sensor.
        - queue (Queue): Default Queue(). The queue where to push the temperature values.
    """
    while True:
        global counter
        counter += 1
        temperature_reading = random.randint(15, 40)
        queue.put((sensor_id, temperature_reading))
        latest_temperatures[sensor_id] = temperature_reading
        time.sleep(1)


```

iii) Writing the process_temperatures function.

```
def process_temperature(queue: Queue = Queue()):
    """
    Calculates the average temperature from the sensor readings.

    Parameters:
        - queue (Queue): default Queue(). A queue of tuples with sensors ans temperature readings.
    """
    while True:
        sensor_id, temperature = queue.get()
        average_temperatures[sensor_id] = \
            ((average_temperatures.get(sensor_id, 0)*(counter-1) + temperature)) // counter
        queue.task_done()

```

4. Integrate Threading

4.a. Question

- Create threads for each call **`simulate_sensor`** and the **`process_temperatures`** function.
- Understand how to use daemon=True to manage thread lifecycle with the main program.

4.b. Solution

i) Daemons:

Daemon threads in Python are used for executing tasks in the ***background.*** These tasks will not prevent the main program from exiting. To make a thread runs as a daemon, the attribute **`daemon`** is set to **`True`**.

Daemon threads run alongside the main thread and are terminated automatically when the main program ends. This makes them ideal for tasks such as monitoring, background computations, or any other operations that need to run in the background without blocking the program's termination.

ii) The whole program.

- The whole program starts with the imports.
- Then, it creates the global dictionary.
- The **`simulate_sensor`** is then added.
- The **`process_temperature`** is then added.
- The main program starts by defining a thread for each sensor. In this case, three sensors. Then starting the processing threads.

```
import random
import threading
import time

latest_temperatures = {}  # create a global dictionary

def simulate_sensor(sensor_id: int = 0):
    """
    Generate random temperature readings and put them in a dictionary.
    Parameters:
```

```python
        - sensor_id (int): Default 0. The id of the sensor.
    """
    while True:
        temperature_reading = random.randint(15, 30)
        latest_temperatures[sensor_id] = temperature_reading
        time.sleep(1)


if __name__ == "__main__":

    sensors = []
    for i in range(3):
        sensors.append(threading.Thread(target=simulate_sensor, args=(i,)))

    for sensor in sensors:
        sensor.daemon = True
        sensor.start()

    while True:
        print(latest_temperatures)
        time.sleep(1)

```

iii) Synchronizing threads

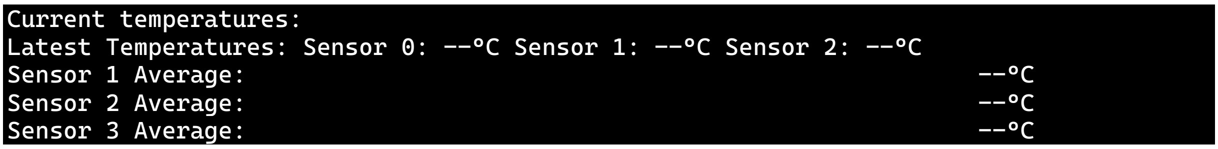
- In the current program, the **`latest_temperatures`** dictionary is accessed by three threads asynchronously. This might cause data corruption problem.
- The solution is to use a lock so only one thread can access the dictionary at a time. Since the threads will need to access the threads multiple times, the best choice is a recurrent lock.
- The program becomes:

```python
import random
import time

from queue import Queue
from threading import Thread
from threading import RLock

# create a global dictionaries for the latest and average temperatures
latest_temperatures = {}
average_temperatures = {}
counter = 0
latest_temperatures_lock = RLock()

def simulate_sensor(sensor_id: int = 0,
                    queue: Queue = Queue()):
    """
    Generate random temperature readings and put them in a dictionary.
    Parameters:
        - sensor_id (int): Default 0. The id of the sensor.
        - queue (Queue): Default Queue(). The queue where to push the temperature values.
    """
    while True:
        global counter
        counter += 1
        temperature_reading = random.randint(15, 40)
        queue.put((sensor_id, temperature_reading))
        with latest_temperatures_lock:
            latest_temperatures[sensor_id] = temperature_reading
        time.sleep(1)

def process_temperature(queue: Queue = Queue()):
    """
    Calculates the average temperature from the sensor readings.

    Parameters:
        - queue (Queue): default Queue(). A queue of tuples with sensors ans temperature readings.
    """
    while True:
        sensor_id, temperature = queue.get()
        average_temperatures[sensor_id] = \
            ((average_temperatures.get(sensor_id, 0)*(counter-1) + temperature)) // counter
        queue.task_done()


if __name__ == "__main__":
    queue = Queue()  # Instantiate the queue

    # Create the sensor threads
    sensors = [Thread(target=simulate_sensor, args=(i, queue)) for i in range(3)]

    # Starting the threads
    for sensor in sensors:
        sensor.daemon = True
        sensor.start()

    # Starting the process thread
    processing_thread = Thread(target=process_temperature,
                               args=(queue,),
                               daemon=True)
    processing_thread.start()

    while True:
        print(f"The latest temperatures {latest_temperatures}")
        print(f"The average temperatures {average_temperatures}")
        time.sleep(1)
```

5. Implement Display Logic

5.a. Question 1

- Write a function **`initialize_display`** to print the initial layout for displaying temperatures. The print should look like this.



```
Current temperatures:
Latest Temperatures: Sensor 0: --°C Sensor 1: --°C Sensor 2: --°C
Sensor 1 Average:                                                    --°C
Sensor 2 Average:                                                    --°C
Sensor 3 Average:                                                    --°C
```

5.b. Solution

This is an easy question 🙄

```python
def initialize_display():
    """
    Initialize the display with fixed labels.
    """
    print("Current temperatures:")
    print("Latest Temperatures:", end='')
    for i in range(3):  # Assuming 3 sensors
        print(f" Sensor {i}: --°C", end='')
    print()  # Move to the next line
    for i in range(1, 4):
        print(f"Sensor {i} Average: ", end='')
        print(" " * 50, end='')  # Placeholder for bars
        print(" --°C")  # Placeholder for average temperature
```

5.c. Question 2

- Develop **`update_display`** to refresh the latest temperatures and averages in place on the console without erasing the console.

5.d. Solution

i) Function to update the display.

- This function updates the display sign ANSI escape characters. *It is not something you are required to know for this course, but you have to admit it pretty cool* 😮.

```python
def update_display():
    """
    Update the display for latest temperatures and the averages.
    """
    while True:
        print("\033[2;0H", end='')  # Move cursor to the start of the latest temperatures
        print("Latest Temperatures:", end='')
```

```
8.          for i in range(3):
9.              temp = latest_temperatures.get(i, '--')
10.             print(f" Sensor {i}: {temp}°C", end='')
11.
12.         for i in range(1, 4):
13.             avg_temp = averages.get(i-1, '--')
14.             bars = '|' * int(avg_temp) if avg_temp != '--' else ''
15.             print(f"\033[{4+i};0H", end='')  # Move cursor to start of each average line
16.             print(f"Sensor {i} Average: {bars:<50} {avg_temp}°C")
17.
```

6. Synchronize Data Access

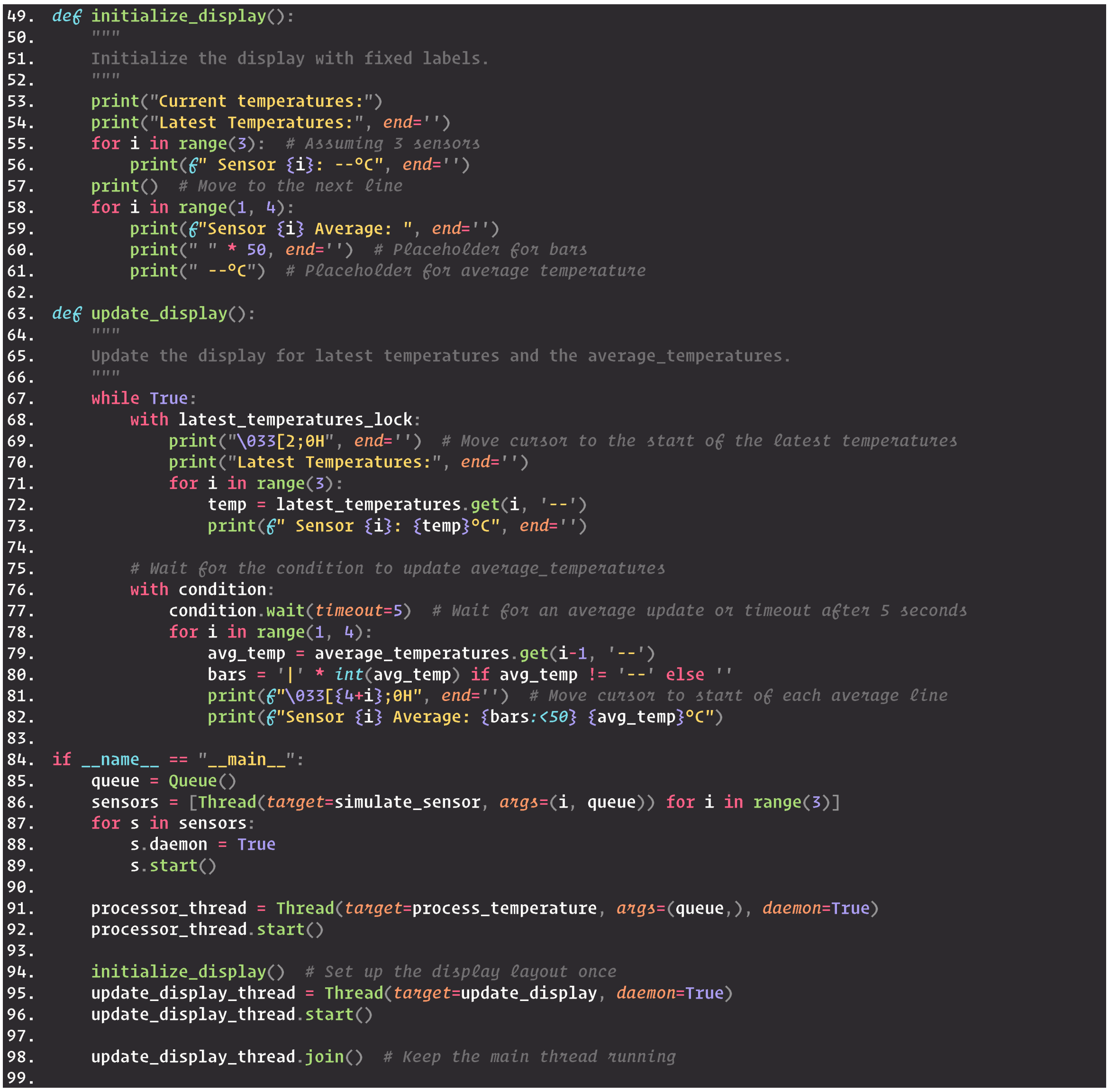
6.a. Question

- Use **`RLock`** and **`Condition`** from the **`threading`** module to synchronize access to shared data structures and control the timing of updates.
  What should you use for which task?

6.b. Solution

- We can use the same used **`RLock`** before to manage the access to the **`latest_temperatures (line 67).`**
- We can use the condition to notify the **`update_display`** that the average has been computed **`(lines 45-46, 75)`**.
- This program also updates the counter to be a list counters. Where there is a counter for each sensors, for an accurate count of the average.

```
1.  import random
2.  import time
3.
4.  from queue import Queue
5.  from threading import Thread
6.  from threading import RLock
7.  from threading import Condition
8.
9.  # create a global dictionaries for the latest and average temperatures
10. latest_temperatures = {}
11. average_temperatures = {}
12. counters = 3*[0]
13. latest_temperatures_lock = RLock()
14. condition = Condition()
15.
16. def simulate_sensor(sensor_id: int = 0,
17.                     queue: Queue = Queue()):
18.     """
19.     Generate random temperature readings and put them in a dictionary.
20.     Parameters:
21.         - sensor_id (int): Default 0. The id of the sensor.
22.         - queue (Queue): Default Queue(). The queue where to push the temperature values.
23.     """
24.     while True:
25.         global counters
26.         counters[sensor_id] += 1
27.         temperature_reading = random.randint(15, 40)
28.         queue.put((sensor_id, temperature_reading))
29.         with latest_temperatures_lock:
30.             latest_temperatures[sensor_id] = temperature_reading
31.         time.sleep(1)
32.
33. def process_temperature(queue: Queue = Queue()):
34.     """
35.     Calculates the average temperature from the sensor readings.
36.
37.     Parameters:
38.         - queue (Queue): default Queue(). A queue of tuples with sensors ans temperature readings.
39.     """
40.     while True:
41.         sensor_id, temperature = queue.get()
42.         average_temperatures[sensor_id] = \
43.             ((average_temperatures.get(sensor_id, 0)*(counters[sensor_id]-1) \
44.                 + temperature)) // counters[sensor_id]
45.         queue.task_done()
46.         with condition:
47.             condition.notify()  # Notify every time an average is updated
48.
```



```python
def initialize_display():
    """
    Initialize the display with fixed labels.
    """
    print("Current temperatures:")
    print("Latest Temperatures:", end='')
    for i in range(3):  # Assuming 3 sensors
        print(f" Sensor {i}: --°C", end='')
    print()  # Move to the next line
    for i in range(1, 4):
        print(f"Sensor {i} Average: ", end='')
        print(" " * 50, end='')  # Placeholder for bars
        print(" --°C")  # Placeholder for average temperature

def update_display():
    """
    Update the display for latest temperatures and the average_temperatures.
    """
    while True:
        with latest_temperatures_lock:
            print("\033[2;0H", end='')  # Move cursor to the start of the latest temperatures
            print("Latest Temperatures:", end='')
            for i in range(3):
                temp = latest_temperatures.get(i, '--')
                print(f" Sensor {i}: {temp}°C", end='')

        # Wait for the condition to update average_temperatures
        with condition:
            condition.wait(timeout=5)  # Wait for an average update or timeout after 5 seconds
            for i in range(1, 4):
                avg_temp = average_temperatures.get(i-1, '--')
                bars = '|' * int(avg_temp) if avg_temp != '--' else ''
                print(f"\033[{4+i};0H", end='')  # Move cursor to start of each average line
                print(f"Sensor {i} Average: {bars:<50} {avg_temp}°C")

if __name__ == "__main__":
    queue = Queue()
    sensors = [Thread(target=simulate_sensor, args=(i, queue)) for i in range(3)]
    for s in sensors:
        s.daemon = True
        s.start()

    processor_thread = Thread(target=process_temperature, args=(queue,), daemon=True)
    processor_thread.start()

    initialize_display()  # Set up the display layout once
    update_display_thread = Thread(target=update_display, daemon=True)
    update_display_thread.start()

    update_display_thread.join()  # Keep the main thread running

```

7. Questions:

1) Why did the professor not ask you to compute metrics?

This program is for concurrent execution, not for speeding up tasks.