Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Number	COE892
Course Title	Distributed Cloud Computing
Semester/Year	Winter 2025
Instructor	Khalid Abdel Hafeez
TA name	Amirhossein Razavi

Lab No.	1

Section No.	05
Submission Date	February 10th, 2025
Due Date	February 11th, 2025

Student Name	**Student ID**	**Signature***
Maria Labeeba	500960386	M.L

Student Name Student ID Signature*

*By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct

1. Introduction

The objective of this lab was to compare and contrast the execution times when sequentially running a program versus using the threading module. This lab was implemented using PyCharm.

For the first part of the lab, we were required to write a python program to take a sequence of movements done by a rover from an API to navigate through a given man. The map contains a 4 by 3 array of zeros and ones where one represents the location of a mine and a zero represents a spot on the map. Based on the rover's commands, if it moves on top of a mine and its next command is to dig, then the mine has been disarmed, otherwise the mine would have exploded and the rover's path stops at that location. There are a total of 10 rovers and each of them have their own sequence that can be accessed from their API and the output of the program should be text files of each of their paths through the maze.

Similarly, for part of this lab we are required to write a program to have the rover once again using the map to navigate through the maze, however this time if it is on top of a mine, it should disarm it by concatenating a selected PIN with the serial number of the mine to create a Temporary Mine Key. This Temporary Mine Key is then hashed using the sha256 function. If the hash has at least six leading zeros, the mine is considered disarmed, thereby avoiding an explosion and allowing the rover to continue its path safely.

2. Part 1

The first step in this lab was to create a function that will read the map.txt file provided and return the dimensions of the map as well as the actual map data. This implementation can be seen in ***Figure 1*** below.

```python
import ...

# Function: Reads map.txt
# --
def read_map(file_path):
    with open(file_path, 'r') as file:
        dimensions = file.readline().strip().split()
        rows, cols = int(dimensions[0]), int(dimensions[1])
        map_data = [file.readline().strip().split() for _ in range(rows)]
    return rows, cols, map_data
```

Fig. 1: *read_map function*

The next step was to create a function that returns the location of the mines from map.txt, into a 2D list which can be seen in ***Figure 2*** below. This streamlines the process for the rover to recognize whether it's on a mine by comparing its location on the map to the list of mine locations. This process will be seen further into this section.

```python
# Function: Returns location of mines as (x, y) as an array
# --
def get_mines(map_data):  1 usage  Maria *
    mine_locations = []                                   # Stores mine locations into an array
    for r_index, row in enumerate(map_data):              # Parse through map.txt
        for c_index, value in enumerate(row):
            if value == '1':                              # If value of map.txt = 1
                mine_locations.append((c_index, r_index)) # Store mine value as (x, y)
    return mine_locations
```

Fig. 2: *get_mines Function*

Next a function was created to get each rover's commands from its respective API. The API url is ‘`https://coe892.reev.dev/lab1/rover/{rover_num}`’, where ‘`{rover_num}`’ is an integer value from 1 through 10. An example of the contents of rover 2 can be seen in ***Figure 3***.

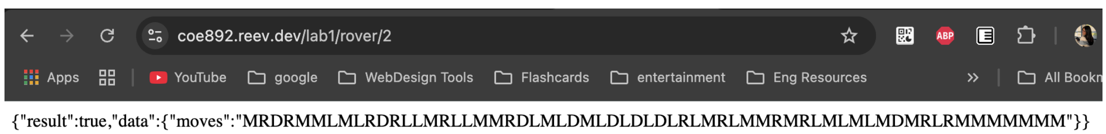

Fig. 3: *API call for rover 2*

The get_rover_commands function retrieves command sequences from a designated API for each rover and converts the response into JSON format. This JSON data is then assigned to the variable data, as seen in ***Figure 4*** below.

```python
# Function: Get rover commands from API
# --
def get_rover_commands(rover_num):  1 usage  Maria *
    url = f"https://coe892.reev.dev/lab1/rover/{rover_num}"
    response = requests.get(url)
    if response.status_code == 200:
        data = response.json()
        return data['data']['moves']
    elif response.status_code == 404:
        return None
```

Fig. 4: *get_mines Function*

The save_path function seen below in ***Figure 5*** saves each rover’s path returned by the get_rover_path function into their respective text files. Details about how the get_rover_path function is outlined in both the sequential and threading subsections.

```
# Function: Save Rover's Path as a text file
# --
def save_path(rover_id, path):  1 usage  Maria *
    directory = "rover_paths"
    os.makedirs(directory, exist_ok=True)
    file_name = os.path.join(directory, f"path_{rover_id}.txt")
    with open(file_name, 'w') as file:
        for row in path:
            file.write(" ".join(row) + "\n")
    print(f"Path for Rover {rover_id} saved to {file_name}")
```

The primary difference between the sequential and threading implementation is how the get_rover_path function and the main function is coded. These differences are highlighted in the subsections below.

a. Sequential

The get_rover_path function determines the path for each rover which can be seen in Figure 5 below. The function starts by initializing the directions, as well as the libraries for the instances when the rover is required to turn left and right. Then within the for loop seen on line 49, there are if and elif statements for each command. The comments in the code below explain what is happening line by line.

```
35  # Function: Navigate through the maop
36  # --
37  def get_rover_path(map_data, commands, mine_locations):  1 usage  Maria *
38      direction = 'S'         # Rover starts facing South ...
39      pos = [0, 0]            # ... @ location (0, 0) on map
40
41      directions = {'N': [0, -1], 'S': [0, 1], 'E': [1, 0], 'W': [-1, 0]}     # Define rovers directions
42      turn_left = {'N': 'W', 'E': 'N', 'W': 'S', 'S': 'E'}                    # Define library for when the rover turns left
43      turn_right = {'N': 'E', 'E': 'S', 'S': 'W', 'W': 'N'}                   # Define library for when the rover turns right
44
45      path = [['0' for _ in range(len(map_data[0]))] for _ in range(len(map_data))]       # Create 2D list initialized with '0's
46      path[pos[1]][pos[0]] = '*'                                                          # Mark position of rover with '*'s
47      mine_hit = False
48
49      for i, command in enumerate(commands):
50
51          # Left Turn
52          if command == 'L':
53              direction = turn_left[direction]
54
55          # Right Turn
56          elif command == 'R':
57              direction = turn_right[direction]
58
59          # Move Forward
60          elif command == 'M':
61
62              # Calculate new (x, y) positions based on current direction
63              new_x = pos[0] + directions[direction][0]
64              new_y = pos[1] + directions[direction][1]
```

```
            # If new position is within the map grid
            if 0 <= new_x < len(map_data[0]) and 0 <= new_y < len(map_data):
                pos[0], pos[1] = new_x, new_y           # Update current position w/ new position
                path[pos[1]][pos[0]] = '*'              # Mark new position with '*'

                # If new position is on a mine
                if (pos[0], pos[1]) in mine_locations:
                    mine_hit = True         # Change flag to true

                    # If the new command is not dig 'D', break
                    if not (i + 1 < len(commands) and commands[i + 1] == 'D'):
                        break
                    '''
                    if i + 1 < len(commands) and commands[i + 1] == 'D':
                        print(f"Mine detected at {pos[0]}, {pos[1]}, waiting for disarm.")
                    else:
                        print(f"Mine detected at {pos[0]}, {pos[1]}, stopping movement.")
                        break
                    '''

        # Dig
        elif command == 'D' and mine_hit and (pos[0], pos[1]) in mine_locations:
            #print(f"Mine disarmed at {pos[0]}, {pos[1]}")
            mine_locations.remove((pos[0], pos[1]))     # Remove disarmed mines location from the list of mine locations
            mine_hit = False                            # Change 'mine_hit' flag to false

    return path
```

Fig. 6: *get_rover_path Function*

Finally the main functions run the program sequentially for each rover's commands and on line 125 of ***Figure** 7* we see it print the total execution time of generating the paths for each rover.

```
105  # Main function
106  # --
107  def main():  1 usage  Maria *
108      file_path = 'map1.txt'
109      rows, cols, map_data = read_map(file_path)
110      mine_locations = get_mines(map_data)
111      #print("Mine locations:", mine_locations)
112
113      start_time = time.time()    # Start Executing
114
115      for rover_id in range(1, 11):
116          commands = get_rover_commands(rover_id)
117          if commands:
118              rover_path = get_rover_path(map_data, commands, mine_locations[:])
119              save_path(rover_id, rover_path)
120          else:
121              print(f"No commands received for Rover {rover_id}")
122
123      end_time = time.time()      # Stop Executing
124
125      print(f"Total execution time: {end_time - start_time:.2f} seconds")
126
127  if __name__ == "__main__":
128      main()
```

Fig. 7: *main Function*

b. Threading Module

The main difference between the sequential implementation and the threading implementation is how the dig command in the get_rover_path function and the main function is coded.

Figure 8 showcases the implementation of the 'D' (dig) command in the get_rover_path function. As seen below, when the command in the sequence is D, mine_hit is true (meaning it is on a mine) and the rover's current position corresponds to a location in the mine_locations list, it does the following from lines 90-92. By using 'lock' it removes the mine location from the list and changes the mine_hit flag to false meaning the mine has been disarmed and is no longer a threat. Locking prevents multiple threads from modifying the mine_locations list at the same time. This ultimately prevents data inconsistencies and runtime errors.

```
88          # Dig
89          elif command == 'D' and mine_hit and (pos[0], pos[1]) in mine_locations:
90              with lock:                                  # Use 'lock' to ensure that shared data can be safely modified
91                  mine_locations.remove((pos[0], pos[1]))     # Remove the mine safely using lock
92              mine_hit = False                            # Reset the 'mine_hit' flag
93
94      return path
```

Fig 8. *Dig command implementation within the get_rover_path function using threading*

The threading module is implemented in the main function where a threading and a lock object are first created as seen on lines 126 and 127 in ***Figure 9*** below. Within a for loop that loops, a thread is created for each rover id and is passed to the execute rover function that runs the code. This eliminates a direct call and instead assigns each rover a thread. Line 135 and 136 are very important as they ensure that the main program waits for all threads to finish executing before continuing.

```
117  # Main function
118  # --
119  def main():  1 usage  Maria *
120      file_path = 'map1.txt'
121      rows, cols, map_data = read_map(file_path)
122      mine_locations = get_mines(map_data)
123
124      start_time = time.time()    # Start Executing
125
126      threads = []                    # Threading object created
127      lock = threading.Lock()         # Lock object created
128
129      for rover_id in range(1, 11):
130          # Create a thread and pass the 'execute_rover' function as the target without calling it
131          thread = threading.Thread(target=execute_rover, args=(rover_id, map_data, mine_locations[:], lock))
132          threads.append(thread)
133          thread.start()
134
135      for thread in threads:          # Ensures that the main program waits for all  ...
136          thread.join()               # ... threads to complete before continuing
137
138      end_time = time.time()      # Stop Executing
139
140      execution_time = end_time - start_time
141      print(f"Total execution time: {execution_time:.2f} seconds")
142
143  if __name__ == "__main__":
144      main()
```

Fig 9. *Main function of the threading implementation*

c. *Analysis*

Figure 10 and ***Figure 11*** display the results of both the sequential and threading approach. The total execution time for the sequential approach was 0.55 seconds, whereas for the threading approach, it was 0.08 seconds. This means that the threading implementation was 0.47 seconds faster which is a significant difference. This is understandable since the threading module allows all the rover's path to be determined at the same time since each rover is running the program on their own thread. Whereas in the sequential approach, rover 2's path can not be determined until rover 1's path is completely processed and determined.

```
Run   sequential_P1
/usr/local/bin/python3.12 /Users/labeeba/Documents/GitHub/COE892/COE892-Lab1/part_1/sequential/sequential_P1.py
Path for Rover 1 saved to rover_paths/path_1.txt
Path for Rover 2 saved to rover_paths/path_2.txt
Path for Rover 3 saved to rover_paths/path_3.txt
Path for Rover 4 saved to rover_paths/path_4.txt
Path for Rover 5 saved to rover_paths/path_5.txt
Path for Rover 6 saved to rover_paths/path_6.txt
Path for Rover 7 saved to rover_paths/path_7.txt
Path for Rover 8 saved to rover_paths/path_8.txt
Path for Rover 9 saved to rover_paths/path_9.txt
Path for Rover 10 saved to rover_paths/path_10.txt
Total execution time: 0.55 seconds

Process finished with exit code 0
```

Fig 10. *Output of Sequential Implementation*

```
Run   threads_P1
/usr/local/bin/python3.12 /Users/labeeba/Documents/GitHub/COE892/COE892-Lab1/part_1/threads/threads_P1.py
Total execution time: 0.08 seconds

Process finished with exit code 0
```

Fig 11. *Output of Threading Implementation*

3. Part 2

The code for this part of the lab is identical to part 1 with exceptions to the move forward and dig command in the get_rover_path function as well as the main function. In depth explanation of these functions can be seen in the subsections below.

a. *Sequential*

In the move forward command of the get_rover_path function seen in ***Figure 12*** below, if the rover is on a mine, then the program assigns an available serial number from the mines.txt file provided to the current mine the rover is on. Using that serial number it calls the valid_pin_finder function to find a valid pin for the mine. Once a valid pin is found it is then used to disarm the mine and its position on the map is now set to 0 indicating that there is no mine. The serial number is then incremented so that the next mine found does not use the same serial number.

```python
        # Move Forward
        elif command == 'M':

            # Calculate new (x, y) positions based on current direction
            new_x = pos[0] + directions[direction][0]
            new_y = pos[1] + directions[direction][1]

            # If new position is within the map grid
            if 0 <= new_x < len(map_data[0]) and 0 <= new_y < len(map_data):
                pos[0], pos[1] = new_x, new_y  # Update current position w/ new position
                #print(f"Move to: {pos}")

                # If new position is on a mine
                if map_data[pos[1]][pos[0]] == '1':

                    # If there are available serial numbers
                    if serial_index < len(serial_numbers):
                        serial_number = serial_numbers[serial_index]        # Assign serial number to current mine
                        print(f"Warning: Rover is now on a mine at {pos}. Serial number: {serial_number}")
                        valid_pin = valid_pin_finder(serial_number)         # Find a valid pin to disarm mine
                        map_data[pos[1]][pos[0]] = '0'                      # Disarm the mine and clear the spot
                        serial_index += 1                                   # Increment serial number
                    else:
                        print(f"Warning: Rover is on a mine at {pos}, but no more serial numbers are available.")
```

Fig 12. *Move Forward Command within get_rover_path for the sequential implementation*

The Dig command performs similarly to the Move Forward command where if the current position of the rover is on a mine, it will function the same as the move forward command. Using that serial number it calls the valid_pin_finder function to find a valid pin for the mine. Once a valid pin is found it is then used to disarm the mine and its position on the map is now set to 0 indicating that there is no mine. The serial number is then incremented so that the next mine found does not use the same serial number. This can be seen in ***Figure 13*** below.

```python
        # Dig
        elif command == 'D':

            # If current position of rover is on a mine
            if map_data[pos[1]][pos[0]] == '1':
                serial_number = serial_numbers[serial_index]    # Assign a serial number for the mine
                print(f"Dig at {pos}: Mine found. Serial number: {serial_number}")
                valid_pin = valid_pin_finder(serial_number)     # Find a valid pin to disarm mine
                map_data[pos[1]][pos[0]] = '0'                  # Disarm the mine and clear the spot
                serial_index += 1                               # Increment Serial Number
```

Fig 13. *Dig Command within get_rover_path for the sequential implementation*

In the main function below in ***Figure 14***, we see that it sequentially disarms the mines .

```python
def main():  1 usage  Maria *
    start_time = time.time()  # Start timing

    map_file_path = 'map1.txt'  # Ensure the file path is correct and accessible
    serial_file_path = 'mines.txt'  # Ensure the file path is correct and accessible
    rows, cols, map_data = read_map(map_file_path)
    serial_numbers = read_mine_serials(serial_file_path)

    # Fetch commands only for Rover 1
    commands = get_rover_commands(1)
    if commands:
        simulate_rover_movement(map_data, commands, serial_numbers)
    else:
        print("No commands received for Rover 1")

    end_time = time.time()  # End timing

    print(f"Total execution time: {end_time - start_time:.2f} seconds")

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

Fig 14. *Main function for the sequential implementation*

b. Threading Module

For the implementation with threading, modifications to the Move Forward and dig command were made within the get_rover_path function as well as to the main function.

The key difference with the move forward command for the threading implementation is the use of the Locking mechanism which can be seen on line 81 in ***Figure 15*** below. This allows the safe use of shared resources between threads. The rest is the same as the sequential implementation.

```python
# Move Forward
elif command == 'M':

    # Calculate new position based on the current direction
    new_x = pos[0] + directions[direction][0]
    new_y = pos[1] + directions[direction][1]

    # Check if new position is within the bounds of the map
    if 0 <= new_x < len(map_data[0]) and 0 <= new_y < len(map_data):
        pos[0], pos[1] = new_x, new_y  # Update rover position

        # Locking mechanism to ensure safe access to shared resources
        with lock:
            # Check if the rover has landed on a mine
            if map_data[pos[1]][pos[0]] == '1' and serial_index < len(serial_numbers):
                serial_number = serial_numbers[serial_index]
                valid_pin = valid_pin_finder(serial_number)  # Attempt to disarm the mine
                map_data[pos[1]][pos[0]] = '0'               # Disarm the mine and clear the spot
                serial_index += 1                            # Move to the next serial number
```

Fig 15. *Move Forward command for the threading implementation*

Similarly, the Dig function uses the locking mechanism seen on line 92 in ***Figure 16*** and the rest of the code remains the same as the sequential implementation.

```python
90  # Dig
91  elif command == 'D':
92      with lock:
93          # Check if the rover's current position has a mine
94          if map_data[pos[1]][pos[0]] == '1' and serial_index < len(serial_numbers):
95              serial_number = serial_numbers[serial_index]
96              valid_pin = valid_pin_finder(serial_number)      # Disarm the mine
97              map_data[pos[1]][pos[0]] = '0'                   # Clear the spot
98              serial_index += 1                                # Increment serial number index
```

Fig 16. *Dig command for the threading implementation*

The main function for the threading implementation is similar to part 1 of the threading implementation. A new thread object is created which is then used in a for loop to create threads for each rover. However, this is unnecessary since threads should be used for concurrent processes and this however is not a concurrent process. More details regarding this will be discussed in the analysis subsection.

```python
# Function: Main function
# --
def main():  1 usage  Maria *
    start_time = time.time()  # Start Executing
    lock = threading.Lock()

    map_file_path = 'map1.txt'
    serial_file_path = 'mines.txt'
    rows, cols, map_data = read_map(map_file_path)
    serial_numbers = read_mine_serials(serial_file_path)

    threads = []        # New threads object
    rover_ids = [1]     # Rover id set to 1

    for rover_id in rover_ids:
        commands = get_rover_commands(rover_id)
        if commands:
            # Create a thread and pass the 'execute_rover' function as the target without calling it
            thread = threading.Thread(target=get_rover_path, args=(map_data, commands, serial_numbers, lock, rover_id))
            threads.append(thread)
            thread.start()

    for thread in threads:  # Ensures that the main program waits for all  ...
        thread.join()  # ... threads to complete before continuing

    end_time = time.time()  # Stop Executing
    print(f"Total execution time: {end_time - start_time:.2f} seconds")

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

Fig 17. *Main function for threading implementation*

c. Analysis

Figure 18 and ***Figure 19*** display the console outputs for both the sequential and threading implementation. The total execution time for the sequential implementation was 17.73 seconds and for the threading implementation it was 18.35 seconds. This means that the sequential implementation was 0.63 seconds faster. This is because the threading module has an unnecessary use since we are only disarming mines for the first rover. Threading is optimal for concurrent processes and if we were to use threading for this part of the lab, it would be more beneficial if we are disarming mines for all the rovers. The threading implementation may have had a larger executing time in comparison to the sequential because it may have added more complexity to the program.

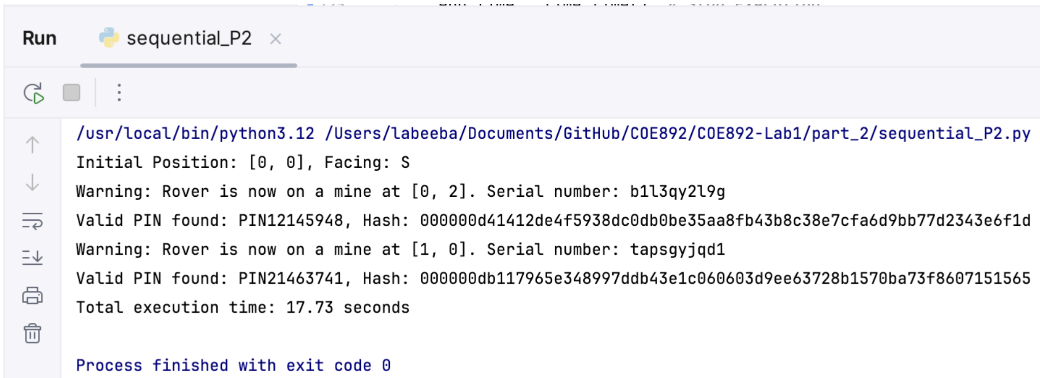

Fig 18. *Sequential implementation console output*

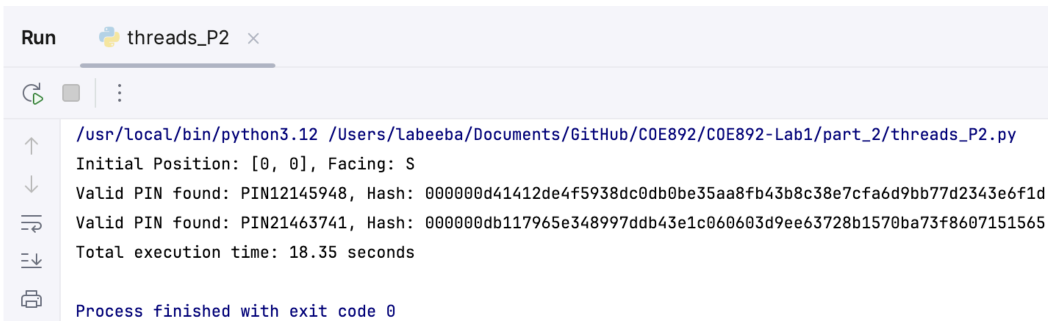

Fig 19. *Threading implementation console output*

4. Conclusion

In conclusion, this lab allowed us to compare and contrast the effectiveness of sequential and threaded implementations. For concurrent processes, threading is optimal and results in faster execution times; however , in non-concurrent processes, threading is unnecessary and can be more damaging as it results in slower execution times due to its complexity. By applying these concepts to a real life scenario, we are able to truly understand how crucial it is to select the right approach based on the specific requirements and nature of the task at hand.