Institute for Advancing Intelligence, TCG CREST
(TCG Centres for Research and Education in Science and Technology)

Introduction to Programming and Data Structures
Ph.D. Coursework: First year, First Semester (Session: 2024-25)
Assignment #04

Full Marks: 300
Clarification Deadline: **2024-Sep-26**

Instructor: Dr. Laltu Sardar
Submission Deadline: **2024-Sep-29**

This is not the final problem-set. Only problem AP0401 is the final one.

Instructions

1. Errors must be handled in all possible functions used, whether from libraries or written by yourself
2. Function names and variable names should clearly describe their purpose.
3. Write the program in such a way, that program does not fails.
4. Magic numbers (like 100 in `array[100]`) should not be hard-coded across the programs. Instead define them as macros (E.g. `#define ARRAY_SIZE 100` and later `array[ARRAY_SIZE]`).

1 Problem #AP0401: Indentation

Indentation in programming is the use of spaces or tabs at the beginning of lines to structure code for readability and, in some languages like Python, it is a required syntax to define code blocks.

Indentation Rules in C

1. Indent code inside control structures like `if`, `else`, `for`, `while`, and `switch` by a fixed number of spaces or a tab.
2. Always indent code within functions and blocks of code enclosed by curly braces {}.
3. Maintain consistent indentation throughout the codebase, typically using 2, 4, or 8 spaces or a single tab.
4. Avoid mixing tabs and spaces for indentation to ensure uniformity across editors.
5. Code inside nested blocks should be indented further than outer blocks to reflect hierarchy.
6. Align the closing brace } of a block with the statement that opened the block.
7. Use indentation to visually separate declarations and statements within functions.

Problem statement: Given a list of files on the **command line**, convert them with proper indentation.

Input: Give user option to use 2,4,8 spaces or tabs. and a set of filenames from command line.

Output: The files with same name and with indentation.

[100]

2 Problem #AP0402: Queue Implementation

The objective of this assignment is to design and implement a system that manages the job scheduling for a quantum computer shared by public users. The public users can have either **Free** or **Premium** accounts, and jobs created by **Premium** users are to be given higher priority in execution. The system should enqueue jobs into one or two queues based on the user's account type and execute the jobs according to a specified priority order.

Problem Description

You are tasked with building a program to handle user jobs submitted to a quantum computer. The system must process jobs based on the following rules:

1. **Premium users' jobs** should always be executed before **Free users' jobs**.
2. Free users' jobs are processed in the order of their submission, but only after all premium jobs in the queue are completed.
3. Each job contains the following attributes:
   - **Creator name**: The name of the user who created the job.
   - **Estimated execution time**: A positive integer value representing the time the job will take to execute, in microseconds.
   - **Account type**: Either "free" or "premium".
   - **Timestamp**: The date and time when the job was created, in the format `YYYY-MM-DD HH:MM:SS`.
4. Upon submission of a job, a **job ID** should be generated and returned to the user.
5. The system must display the job queue after each job submission and provide a status report of each job when it is executed.

Input Format

The input to the system is a file containing job data. Each line in the file represents one job in the following format:

```
<timestamp> <creator_name> <execution_time_in_microseconds> <account_type>
```

For example:

```
2024-09-19 10:15:00 Alice 500 premium
2024-09-19 10:16:30 Bob 300 free
2024-09-19 10:17:45 Charlie 200 premium
2024-09-19 10:19:00 Diana 400 free
```

Output Format

1. After every job is enqueued, display the current state of the job queues, showing the jobs in both the premium and free queues.
2. As jobs are executed, display a status report in the terminal. The report should include the **job ID**, **creator's name**, **estimated execution time**, **timestamp**, and **execution status**.

Example Scenario

Consider the following input from the file `job_data.txt`:

```
2024-09-19 10:15:00 Alice 500 premium
2024-09-19 10:16:30 Bob 300 free
2024-09-19 10:17:45 Charlie 200 premium
2024-09-19 10:19:00 Diana 400 free
```

Job Submission

- **Alice** submits a job at `2024-09-19 10:15:00` with an execution time of 500 microseconds (premium account).
- **Bob** submits a job at `2024-09-19 10:16:30` with an execution time of 300 microseconds (free account).
- **Charlie** submits a job at `2024-09-19 10:17:45` with an execution time of 200 microseconds (premium account).
- **Diana** submits a job at `2024-09-19 10:19:00` with an execution time of 400 microseconds (free account).

Queue Status after Enqueuing

After the jobs are enqueued, the state of the queues should be:

```
Premium Queue: [job_1 (Alice, 500ms, 2024-09-19 10:15:00),
job_3 (Charlie, 200ms, 2024-09-19 10:17:45)]
Free Queue:    [job_2 (Bob, 300ms, 2024-09-19 10:16:30),
job_4 (Diana, 400ms, 2024-09-19 10:19:00)]
```

Job Execution

The system will execute the jobs in the following order:

1. Execute **Alice's job** (`job_1`): 500 microseconds.
2. Execute **Charlie's job** (`job_3`): 200 microseconds.
3. Execute **Bob's job** (`job_2`): 300 microseconds.
4. Execute **Diana's job** (`job_4`): 400 microseconds.

For each execution, the system will print a status report like:

```
Executing job_1: Creator - Alice, Estimated Time - 500ms,
Submitted - 2024-09-19 10:15:00, Status - Completed
Executing job_3: Creator - Charlie, Estimated Time - 200ms,
Submitted - 2024-09-19 10:17:45, Status - Completed
Executing job_2: Creator - Bob, Estimated Time - 300ms,
Submitted - 2024-09-19 10:16:30, Status - Completed
Executing job_4: Creator - Diana, Estimated Time - 400ms,
Submitted - 2024-09-19 10:19:00, Status - Completed
```

Requirements

1. The system must handle an arbitrary number of job submissions.
2. Jobs should be executed in the correct order based on their priority: premium jobs first, followed by free jobs.
3. The queue should be displayed after each job is submitted.
4. During execution, print the job status to the terminal.

Hints for Implementation

- **Job Structure**: Each job should have a unique ID, a creator name, estimated execution time, timestamp, and account type.
- **Queue Management**: Use two separate queues (e.g., lists or deques) to manage premium and free jobs.
- **Job IDs**: Use a counter to generate unique job IDs.
- **Input Handling**: Read the input file line by line and process each job accordingly.
- **Execution Simulation**: Use `time.sleep()` to simulate the job execution time (convert microseconds to seconds for this purpose).

[200]