CIS*2750
Assignment 4
Deadline: Friday, December 1, 9:00pm
Weight: 10%

1. Introduction

For this assignment you will expand your A3 in the following ways:
- Add a menu item for Database activities (see Section 2)
- Add an additional window for interacting with the database (see Section 2)
- Create and maintain tables in an SQL database (see Sections 2 and 3)
- Add connectivity to a MySQL database (see Section 4)

2. User Interface Additions

2.1 Database menu

To interact with the calendar database, add a **Database** menu with the following menu items. Each menu item is only active when it is logically meaningful. After every command (except **Display DB Status** and **Execute Query**), print in the log panel a status line based on a count of each table's rows: "Database has N organizers and N events"

- **Store All Events**: This command is used to insert all the events presently in the File View Panel into your database. This is active only if the File View Panel contains at least one event, and must be disabled if no calendar is loaded, or the loaded calendar does not have any events (e.g. you've created a calendar, but haven't added an event to it yet). For every event, go through the following steps:
  - Obtain all necessary data that should be stored. You will need a few short C helper functions that you'll call from Python. However, all the data is easy to pull out of an Event's property list, or directly from the Event struct. This is why we've added the start date/time to the Event struct in A3. The only property that will require some (minimal) additional parsing is ORGANIZER property, as mentioned above.
  - Check if the event's organizer name is already in the `ORGANIZER` table. If not, insert it along with its `contact` field. Keep in mind that an iCalendar file might be inconsistent: the same organizer name might appear in multiple events of the same file with differing contact values. In this case, the first "mailto" occurrence would become the "correct" one. However, we will simply exclude such instances from the testing data to avoid overcomplicating things.
  - For an event, check if it is already in the `EVENT` table; both `summary` and `start_time` must match, otherwise it's considered a different event. If not, insert it, including a reference to its organizer, if applicable, and `NULL` for any missing fields (`0` if no alarms) .
- **Store Current Event**: Same as **Store All Events**, except it works only on the event currently selected in the File View Panel - i.e. the one for which you can display the Alarms and optional properties. This is only active if a event is selected.
- **Clear All Data**: Delete all rows from all the tables. This may have to be done in a certain order due to the foreign keys. This is only active if any table is not zero length. Do not drop the tables themselves.
- **Display DB Status**: This displays in the log panel the status line described above and is always active.
- **Execute Query**: this menu item, which is always active, opens a secondary window (not modal) that is used to query the database. Only a single query window can be opened, and it can be closed by clicking on its [X] title bar button. It displays the following two panels:
  - **Query**: These controls are used to make ad hoc queries on the database. There user should be able to select one of several standard queries, which they user can fill in and submit. There is also a text area initialized with the word "SELECT" where the user can finish typing an arbitrary SELECT statement, or even backspace and type any

SQL statement, if they want to risk it. There is a Submit button to send the chosen standard query or SQL statement to the database. There is also a Help button which pops up a small panel showing the output of "DESCRIBE ORGANIZER; DESCRIBE EVENT;" giving precise table and column names--information the user will need to correctly format a query.

- **Results**: The results (or errors) from the submitted query are displayed here in a scrolling text area. It functions similar to the log panel in the main window: the new results are added to the bottom, and it has a Clear button. Output a line of dashes between each set of results.

2.2 Standard queries

The **Execute Query** menu item displays five standard queries, with some "variables" (textboxes) that the user may fill in. There should be some way (your choice) to select one of the queries to be performed when Submit is pressed. The queries must be displayed in simple English, but your program will generate the underlying SQL statements incorporating any filled-in variables. The intended user of this functionality is someone who wants to access their personal calendar, so think about what they might want to know, and how to make it easier for them to provide the information necessary to execute the query.

Two required queries are given below. You should come up with three additional different ones that are not overly simplistic. For our purposes, "overly simplistic" means anything that doesn't have conditions, a join, a nested query, and/or aggregate functions. The first required query **is** overly simplistic, and is given to you as a warm-up. You must supply an iCalendar demo file `A4demo.ics` that contains all the necessary calendar information that can be used to demonstrate your queries. In other words, if we load `A2demo.ics`, we can execute any of the five queries, and get meaningful (and non-empty) results.

Queries:

1. Display all events sorted by start date. **(required)**

2. Display the events from a specific organizer. This query fetches the start dates and summaries of all events. How you sort the events is up to you. **(required)**

3. Something related to a location

4. Something that allows you to contact the organizers

5. Something related to alarms

3. Database Design

3.1 Naming conventions

You **must not** change the names of menus, menu commands, buttons, tables, columns, and the like that are specified below. Note that user-specified names in SQL (tables, columns) are case sensitive, but SQL keywords are not. This requirement is intended simplify and speed up marking and allow, for example, for tables to be prepared with test data in advance. If you change specified names, you **will lose marks**.

Please note that there is no universally accepted consensus on whether tables should have singular or plural names (e.g. EVENT or EVENTS). However, singular names are more common. On the theoretical side, a table is meant to represent a relation, and a singular name makes more sense. On the practical size, a singular name is typically more readable, avoids ambiguities (e.g. PEOPLE vs PERSONS), and is easier for non-native speakers. Therefore we're sticking with singular names.

For multi-word column names, we will use the typical C/Python "lower_case_with_underscores" convention, instead of the "camelCase" we've been using all semester. This practice avoids ambiguity when working with platforms that are not case sensitive, and helps comply with some aspects of the SQL standard.

3.2 Tables

When your program executes, it must create these tables in your database if they do not already exist. Use the `try`/`except` blocks, and look at the error messages. If creation fails with the message indicating that the table already exists, you're good to go. However, if creating a table fails for some other reason (e.g. invalid SQL syntax), you must fix the problem! Also, make sure that your program **does not drop** any tables. Your `A4main.py` can create, edit, and clear tables, but it can't drop them. If you need to drop a table, do so manually through the `mysql` command line tool.

We're interested in storing data about calendar events and their respective organizers. Thus, the schema for your database consists of two tables named `EVENT` and `ORGANIZER`. The idea is that every unique organizer named by various events stored in the `ORGANIZER` table, while the events refer to their organizers by means of foreign keys.

Column names, data types, and constraint keywords are listed below:

Table `ORGANIZER`

1. `org_id: INT, AUTO_INCREMENT, PRIMARY KEY.` The `AUTO_INCREMENT` keyword gives MySQL the job of handing out a unique number for each organizer so your program doesn't have to do it. There is no obvious unique data value that we can use for the organizer primary key, and we don't want to cobble together a composite key out of multiple property values and parameters.

2. `name: VARCHAR(60), NOT NULL`. The value from an ORGANIZER property's CN parameter.

3. `contact: VARCHAR(60), NOT NULL`. The value of the ORGANIZER property (typically "mailto:...").

This means you'll need a couple of short "accessor" C functions to pull the ORGANIZER for a particular event from that event's property list. You would call these accessor functions from Python. What they return is up to you. For example, given the original iCalendar ORGANIZER property:

```
ORGANIZER;CN=John Smith:mailto:jsmith@example.com
```

You can either return the entire "value" of the organizer property, i.e.
`CN=John Smith:mailto:jsmith@example.com`,
to the Python code and parse is there, or parse it on the C side and return 2 separate strings to Python, one containing "`John Smith`", and the other - "`mailto:jsmith@example.com`".

Table `EVENT`

1. `event_id: INT, AUTO_INCREMENT, PRIMARY KEY.`

2. `summary: VARCHAR(60), NOT NULL`. The value from the SUMMARY property.

3. `start_time: DATETIME, NOT NULL`. The value from the DTSTART property, converted into MySQL's datetime format. This is already a member of the Event struct, so accessing it is easy and requires no extra parsing.

4. `location: VARCHAR(60)`. The value from the LOCATION property. NULL if missing.

5. `organizer: INT`. The value of the ORGANIZER property's CN property, represented as a row in the ORGANIZER table. NULL if missing. `FOREIGN KEY REFERENCES` establishes a foreign key to the `org_id` column over in the `ORGANIZER` table, and deleting the latter's row will automatically cascade to delete all its referencing events.

6. `num_alarms: INT`. The number of alarms for that event (`0` if none).

7. Additional constraint: `FOREIGN KEY(organizer) REFERENCES ORGANIZER(org_id) ON DELETE CASCADE`

4. Connecting to the SOCS database

4.1 Connection details

Our official MySQL server has the hostname of `dursley.socs.uoguelph.ca`. Your username is the same as your usual SoCS login ID, and your password is your 7- digit student number. A database has been created for you with the same name as your username. You have permission to create and drop tables within your own named database.

To access the database from home, connect to the school network using a VPN (Cisco AnyConnect), and

- Login from home using the `mysql` command line tool, if assuming you have it installed in your machine or VM
- Login to linux.socs.uoguelph.ca, and use the `mysql` command line tool from there (it is installed)
- Simply execute Python code with SQL queries from your machine and look at the output.

See Lecture 21 on details for how to connect to a MySQL server, create/drop tables, insert/delete data, and run queries. I have also posted a simple Python program that connects to the SoCS MySQL server and runs some queries (`dbTest.py`) in Week 10 notes.

4.2 Implementation details

You will develop your assignment using your own credentials and database, but part of the grading process will involve connecting your code to our database. This means that you cannot simply hardcode your credentials into the Python code. Your Python program will accept an optional command line argument. If it is not provided, your program will connect to your database using your credentials. If it is provided, e.g.

```
>python3 A4main.py someUser
```

your code will try to connect as `someUser`. Your program will then prompt the user to enter the password and database name, and will attempt the connection. These prompts can be done using either a command-line UI, or the GUI - our choice. Is the connection fails, your program will prompt the user to re-enter the username, DB name, and password two more times. If the connection is unsuccessful after 3 attempts (initial and 2 re-connects), the program will display an error message and exit.

5. Grade breakdown

- Correct database tables and database connection, including command line arguments: 10 marks
- Usability of UI additions, including error handling: 15 marks
- Each of the first four elements of the **Database** menu, fully functional: 45 marks
  - **Store All Events**: 20 marks
  - **Store Current Event**: 10 marks
  - **Clear All Data**: 10 marks
  - **Display DB**: 5 marks
- **Execute Query**, fully functional: 30 marks
  - UI, two required queries: 15 marks
  - Three additional queries: 10 marks
  - A4demo.ics 5 marks

Deductions
- Compiler errors: 0 on the assignment
- Any warnings: -15 marks of assignment grade
- Marks will also be deducted for deviating from assignment requirements, incorrect table/column names, missing or incorrect error handling, etc..

6. Submission

Submit all your C and Python code, along with the necessary Makefile. All of it must placed in the directory `Assignment4`. File name must be be `A4FirstnameLastname.zip`. As for A3, we must be able to compile and run your assignment by entering the Assignment4 directory and typing:

```
make
python3 bin/A4main.py
```
(with or without the command line argument)

Late submissions: see course outline for late submission policies.

This assignment is individual work and is subject to the University Academic Misconduct Policy. See course outline for details)