CSI4109 Assignment #1
Due: Mar 15th 12:59pm

Secure Your House

Your goal is to write, using the C/Rust programming language, a program which implements the following security policy for your house. Your house will be *simulated*, and the name of your house simulator is called `secure_house`.

Policy.

- Only users with an `authorized key` can enter the house. To enter the house, the user must first insert their key in the lock, then turn the lock, then enter the house, only if the key is valid.
- It is **not** guaranteed that it is the same user who inserts the lock, turns the lock, and enters the house. That is, if an authorized key has been inserted, *anyone* can successfully turn the lock, and even enter the house.
- Firefighters can enter with the secret key (a string literal) `FIREFIGHTER_SECRET_KEY`.
- A house can be rekeyed with new keys only by the owner, and only if the owner is inside the house. Rekey could happen at any point in time, and resets the lock state.
- Inserting a key replaces an already-inserted key, if any, and resets the lock state.
- Turning the lock without inserting any key leads to a failure.
- Entering the house without inserting a key or turning the lock is denied.
- Commands other than (i) insert, (ii) turn, and (iii) enter can be issued in between them; however, they do not affect the lock state, *except for* successful rekey operations.
- Any key, regardless of whether they are valid or not, can be inserted to the lock. Whether an inserted key is valid is checked when the lock is turned.
- Once the house is unlocked, only a signel person (this could be anybody) can enter the house. After entering the house, the lock is automatically locked.

Interface. You must implement the following command-line interface for your server:

```
./secure_house <owner_name> <key_1> <key_2> ... <key_n>
```

where `<owner_name>` is the name of the owner, and `<key_1>` through `<key_n>` are the authorized keys for the house.

All inputs to the program (keys and names) will be `[a-zA-Z0-9_\-]` (i.e., alphanumeric, underscore, and dash). All matching is case-sensitive.

The input to your program (on standard input) will be a series of events separated by newlines. Your program must track these events and respond appropriately, while enforcing the security policy.

Every input event will end with a newline, and every response must end with a newline.

- `INSERT KEY <user_name> <key>`

  `<user_name>` inserts key `<key>` into the door. Response should be: `KEY <key> INSERTED BY <user_name>`

- `TURN KEY <user_name>`

  `<user_name>` turns the key in the door. Possible responses are:

  - `SUCCESS <user_name> TURNS KEY <key>`
  - `FAILURE <user_name> HAD INVALID KEY <key> INSERTED`
  - `FAILURE <user_name> HAD NO KEY INSERTED`

- `ENTER HOUSE <user_name>`

  `<user_name>` enters the house. Possible responses are: `ACCESS DENIED` or `ACCESS ALLOWED`.

- `WHO'S INSIDE?`

  Who is currently inside the house? Response must be a comma-separated list of user names, ordered by access time (earlier access first): `<user_name_1>, <user_name_2>, <user_name_3>, ...` or `NOBODY HOME` if there are no users in the house.

- `CHANGE LOCKS <user_name> <key_1> <key_2> ... <key_n>`

  `<user_name>` wishes to rekey the house with new given keys `<key_1>`, `<key_2>`, ..., `<key_n>`. Possible responses are: `LOCK CHANGE DENIED` or `LOCK CHANGED`

- `LEAVE HOUSE <user_name>`

  `<user_name>` leaves the house. Possible responses are: `<user_name> LEFT` or `<user_name> NOT HERE`

- An empty newline should terminate the program.
- If any events are received that are not according to this specification, the response must be: `ERROR`.

Example. Running the program as follows:

```
./secure_house selina foobar
```

- Given the input:

```
INSERT KEY david key
TURN KEY david
ENTER HOUSE david
INSERT KEY pat foobar
TURN KEY pat
ENTER HOUSE pat
WHO'S INSIDE?
```

- The program will produce the following output:

```
KEY key INSERTED BY david
FAILURE david HAD INVALID KEY key INSERTED
ACCESS DENIED
KEY foobar INSERTED BY pat
SUCCESS pat TURNS KEY foobar
ACCESS ALLOWED
pat
```

Implementation.

- **Your program must work on Ubuntu 22.04 64-bit with the default packages installed.** In addition to the default packages, the following packages for languages are also installed:
  - C (`gcc`)
  - Rust (`rustc`)

  You're probably better off if you set up a virtual machine to work on the course assignments early on. You can use VirtualBox, a free and open-source VM monitor software. Or, if you are using MS Windows, you may want to use WSL (WSL version 2 is recommended.) (Ubuntu 22.04 on Microsoft Store).

- We've created a test script called `test.sh` to help you test your program before submitting. Download `test.sh` to the directory where your code lives (including `README` and `Makefile`). Ensure that `test.sh` is executable:

```
chmod +x test.sh
./test.sh
```

- There is also a `test_debug.sh` that gives you the output of your program. This can help you with debugging when the program appears to work from the command line, but not in the `test.sh` script (it's happened before). Your program must be able to accept arbitrarily large input (and this will be tested by the autograder).

Submission Instructions. Submit on LearnUs (ys.learnus.org) your source code, along with a `Makefile` and `README`. The `Makefile` must create your executable, called `secure_house`, when the command `make` is run. Note that we may invoke `make` multiple times, and it needs to work every single time. Your `README` file must be **plain text** and should contain your name, student ID, and a description of how your program works.

Grading Rubric

- All required files exist, and `make` successfully creates `secure_house`. (1 pt)
- Passes basic functional tests. (2 pt)
- Handles rekey operations. (0.5 pt)
- Handles firefighter accesses. (0.5 pt)
- Handles very large inputs. (0.5 pt)
- Handles `ERROR` cases. (0.5 pt)

Note: It is your responsibility to comply with the specification as well as our grading environment. You can request regrading for minor problems (e.g., issues caused by using different environments), but each fix will result in a deduction of 1 or more points.

Late policy: 1 pt deduction for every 3 hours that is late. This means that late submission up to 3 hours get one point deduction, 6 hours two point deduction, and so on.