Assignment 2

Assignment 2 is due 2/9/22 on or before 11:59:59pm MST.

Part 1 — Make this (25 points)

All of the coding assignments in this course (including Parts 2 and 3 of this assignment) allow you to write your assignment in any programming language. To allow this, you will need to write a Makefile that creates an executable file based on your source code.

In this assignment, you'll practice writing a Makefile for two different types of programs, one a C program that must be compiled, and the other a Python program.

Here are some Makefile resources:

- https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
- https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_make.html
- https://www.gnu.org/software/make/manual/make.html
- http://mrbook.org/blog/tutorials/make/
- YouTube video: How to Create a Simple Makefile - Introduction to Makefiles
- Starter Information about Python Makefiles

C Program Makefile (50%)

The C Program Makefile must compile the file `c_program.c` into the executable `c_program`.

Assume that your C Program Makefile is called `Makefile` and is in the same directory as the `c_program.c`, and therefore, when you run `make` in that directory, the compiled program `c_program` is created using `gcc`.

Also, your C Program Makefile must recompile the binary when `c_program.c` changes when you run `make` again.

This means that:

1. Run `make`.
2. Execute `./c_program`.
3. Edit the source code `c_program.c` (change the strings, for instance).
4. Run `make`.
5. Execute `./c_program` and verify that the changes you made in step 3 are reflected.

Python Makefile (50%)

Use a separate directory than the C Program Makefile.

The Python Makefile must use the file `python_program.py` to create an executable file called `python_program`.

Assume that your Python Makefile is called `Makefile` and is in the same directory as the `python_program.py`, and therefore, when you run `make` in that directory, your Python Makefile will create a `python_program` executable.

Also, your Python Makefile must recreate `python_program` when `python_program.py` changes and you run `make` again.

This means that:

1. Run `make`.
2. Execute `./python_program`.
3. Edit the source code `python_program.py` (change the strings, for instance).
4. Run `make`.
5. Execute `./python_program` and verify that the changes you made in step 3 are reflected.

There are several ways to approach this (because there's nothing to compile). For instance, you can take advantage of the shebang functionality, which allows a file that can be interpreted to be executed. This requires that the file is executable (`chmod +x filename`).

Submission Instructions

Submit on GradeScope to the Python Makefile assignment and to the C Makefile assignment.

Part 2 — Commands (25 points)

One of the ways that programs receive input from users is through command line arguments. We will also use this in future assignments.

Your goal is to write, in any language, a program which first prints out the number of command line arguments and the next line prints them out in reverse order, separated by space (so that the last command line argument is printed first).

The name of your program will be called `command`.

Examples

When your program is executed with the following:

```
./command foo bar
```

It must output exactly:

```
2
bar foo
```

Other examples:

```
./command
```

Output:

```
0
```

Running:

```
./command a b "test input" c d e
```

Output:

```
6
e d c test input b a
```

Implementation

Your program must work on Ubuntu 18.04 64-bit with the default packages installed.

In addition to the default packages, the following packages for languages are also installed:

- C (`gcc`)
- C++ (`g++`)
- Java (`default-jre` and `default-jdk`)
- Rust (`rustc`)
- Ruby (`ruby-full`)
- Node.js (`nodejs` and `npm`)
- Mono (Custom install)
- Go (1.15.6, custom install)

If there's a package that you need, please ask on the course piazza and I'll have it installed for everyone. Java is already installed.

You'll also need to write a Makefile that, when the `make` command is run, will create the executable called `command`.

Submission Instructions

Submit on GradeScope your source code, along with a Makefile (called `Makefile`) and a README. The Makefile must create your executable, called `command`, when `make` is ran. Your README file must be plain text and should contain your name, ASU ID, and a description of how your program works.

Part 3 — Secure this house (50 points)

Your goal is to write, in any language, a program which implements the given security policy. The security policy will be based on our in-class discussion of the security policy for a house.

The name of your house simulator will be called `secure_house`.

Policy

Only users with an `authorized key` can enter the house. To enter the house, the user must first:

1. Insert their key in the lock
2. Turn the key in the lock
3. Enter the house

Note that testing if a key is valid is done only when the key is turned.

For turning the key and entering the house, it must be the same user that puts the key in the lock, turns the key, and enters the house.

A house can be rekeyed (this means that the old keys are no longer useable) with new keys only by the owner.

It is only physically possible to rekeyed the house when there is no key in the lock. For instance, there will never be a rekey between inserting a key and turning a key, or after turning a key successfuly and entering the house.

There is only one lock and one door. The lock will always be accessed in the following way:

1. insert key
2. turn the key
3. enter the house

Other commands can be issued in between insert, turn, and enter.

For example, the following situations will never happen:

- insert, enter
- insert, turn, insert
- turn, enter
- insert, turn, turn, enter

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 all authorized keys for the house.

All inputs to the program (keys and names) will be `[a-zA-Z0-9_\-]` (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 a newline. Your program must track these events and respond appropriately, while enforcing the security policy.

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

Your program must continue to process input until there is no input left. How you do this will vary by programming language, try Googling for "End of file" (or EOF) and your programming language.

You can simulate EOF on the command line by pressing CTRL-D (more info on this stackoverflow post).

```
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>` or `FAILURE <user_name> UNABLE TO TURN KEY <key>`

```
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: `ACCESS DENIED` or `OK`

```
LEAVE HOUSE <user_name>
```

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

Note that all test cases will be valid according to this specification. To make it easier, if any events are received that are not according to this specification, the response should be: `ERROR`.

Example

Running the program as follows:

```
./secure_house selina foobar key2 key3
```

Given the input:

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

The program must produce the following output:

```
KEY key INSERTED BY adam
FAILURE adam UNABLE TO TURN KEY key
ACCESS DENIED
KEY foobar INSERTED BY pat
SUCCESS pat TURNS KEY foobar
ACCESS ALLOWED
pat
```

Implementation

Your program must work on Ubuntu 18.04 64-bit with the default packages installed.

In addition to the default packages, the following packages for languages are also installed:

- C (`gcc`)
- C++ (`g++`)
- Kotlin 1.4.21 (Custom install)

- PHP 7.2 (`php-cli`)
- Python 2 (`python`)
- Python 3 (`python3`)
- Java (`default-jre` and `default-jdk`)
- Rust (`rustc`)
- Ruby (`ruby-full`)
- Node.js (`nodejs` and `npm`)
- Mono (Custom install)
- Go (1.15.6, custom install)

If there's a package that you need, please ask on the course piazza and I'll have it installed for everyone. Java is already installed.

If you need to set up a virtual machine for your development: VirtualBox is a free and open-source VM system.

We've created a test script called `test.sh` to help you test your program before submitting.

1. Download test.sh to the directory where your code lives (including `README` and `Makefile`).
2. Ensure that `test.sh` is executable: `chmod +x test.sh`
3. Run: `./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 GradeScope your source code, along with a `Makefile` and `README` . The Makefile must create your executable, called `secure_house` , when the command `make` is ran. Your README file must be plain text and should contain your name, ASU ID, and a description of how your program works.

CSE 365: Introduction to Information Assurance S22 **Adam Doupé and Tiffany Bao**

Website Design Acknowledgement: Carolina Zarate and Zilin Jiang