Program 2 – CS 344

This assignment asks you to write a simple game akin to old text adventure games. You'll write two programs that will introduce you to programming in C on UNIX based systems, and will get you familiar with reading and writing files.

Overview

This assignment is split up into two C programs (no other languages is allowed). The first program (hereafter called the "rooms program") will be contained in a file named "<*STUDENT ONID USERNAME*>.buildrooms.c", which when compiled with the same name (minus the extension) and run creates a series of files that hold descriptions of the in-game rooms and how the rooms are connected.

The second program (hereafter called the "game") will be called "<*STUDENT ONID USERNAME*>.adventure.c" and when compiled with the same name (minus the extension) and run provides an interface for playing the game using the most recently generated rooms.

In the game, the player will begin in the "starting room" and will win the game automatically upon entering the "ending room", which causes the game to exit, displaying the path taken by the player.

During the game, the player can also enter a command that returns the current time - this functionality utilizes mutexes and multithreading.

For this assignment, do not use the C99 standard: this should be done using raw C (which is C89). In the complete example and grading instructions below, note the absence of the -c99 compilation flag.

Specifications

Rooms Program

The first thing your rooms program must do is generate 7 different room files, one room per file, in a directory called "<*STUDENT ONID USERNAME*>.rooms.<*PROCESS ID*>*"*. You get to pick the names for those files, which should be hard-coded into your program. For example, the directory, if John Smith was writing the program, should be hard-coded (except for the process id number) as:

smithj.rooms.19903

Each room has a Room Name, at least 3 outgoing connections (and at most 6 outgoing connections, where the number of outgoing connections is random) from this room to other rooms, and a room type. The connections from one room to the others should be randomly assigned – i.e. which rooms connect to each other one is random - but note that if room A connects to room B, then room B must have a connection back to room A. Because of these specs, there will always be at least one path through. Note that a room cannot connect to itself, nor can a room connect to another room more than once: i.e. room A cannot have more than one outbound connection to room B.

Each file that stores a room must have exactly this format, where the … is additional room connections, as randomly generated:

```
ROOM NAME: <room name>
CONNECTION 1: <room name>
…
ROOM TYPE: <room type>
```

You must hard code a list of ten different Room Names into your rooms program and have your rooms program randomly assign one of these to each room generated. For a given run of your program, 7 of the 10 hardcoded room names will be used. Note that a room name cannot be used in more than one room, and each name can be at max 8 characters long, with only uppercase and lowercase letters allowed (thus, no numbers, special characters, or spaces).

The possible room type entries are: START_ROOM, END_ROOM, and MID_ROOM. The assignment of which room gets which type should be random. Naturally, only one room should be assigned as the start room, and only one room should be assigned as the end room.

Here are the contents of files representing three *sample* rooms from a full set of room files (remember, you must use this same format). My list of room names includes the following, among others: XYZZY, PLUGH, PLOVER, twisty, Zork, Crowther, and Dungeon.

```
ROOM NAME: XYZZY
CONNECTION 1: PLOVER
CONNECTION 2: Dungeon
CONNECTION 3: twisty
ROOM TYPE: START_ROOM


ROOM NAME: twisty
CONNECTION 1: PLOVER
CONNECTION 2: XYZZY
CONNECTION 3: Dungeon
CONNECTION 4: PLUGH
ROOM TYPE: MID_ROOM


… (Other rooms) …

ROOM NAME: Dungeon
CONNECTION 1: twisty
CONNECTION 2: PLOVER
CONNECTION 3: XYZZY
CONNECTION 4: PLUGH
CONNECTION 5: Crowther
CONNECTION 6: Zork
ROOM TYPE: END_ROOM
```

The ordering of the connections from a room to the other rooms, in the file, does not matter. Note that the randomization you do here to define the layout is not all that important: just make sure the connections between rooms, the room names themselves and which room is which type, is somewhat different each time, however you want to do that. We're not evaluating your randomization procedure, though it's not acceptable to just randomize the room names but use the same room structure every time.

I highly recommend building up the room graph in this manner: adding connections two at a time (forwards and backwards), to randomly chosen room endpoints, until the graph satisfies the requirements.

It's easy, requires no backtracking, and tends to generate sparser layouts. As a warning, the method of choosing the number of connections *beforehand* that each room will have is *not recommended*, as it's hard to make those chosen numbers match the constraints of the graph. To help do this correctly, please read the article 2.2 Program Outlining in Program 2 and consider using the room-generating pseudo-code listed!

Here is an example of the rooms program being compiled and then run. Note that it returns NO OUTPUT, unless there is an error:

```
$ gcc -o smithj.buildrooms smithj.buildrooms.c
$ smithj.buildrooms
$
```

The Game

Now let’s describe what should be presented to the player in the game. Once the rooms program has been run, which generates the room files, the game program can then be run. This program should present an interface to the player. Note that the room data must be read back into the program from the previously-generated room files, for use by the game. Since the rooms program may have been run multiple times before executing the game, your game should use the most recently created files: perform a `stat()` function call (Links to an external site.)Links to an external site. on rooms directories in the same directory as the game, and open the one with most recent `st_mtime` component of the returned `stat` struct.

This player interface should list where the player current is, and list the possible connections that can be followed. It should also then have a prompt. Here is the form that must be used:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >
```

The cursor should be placed just after the > sign. Note the punctuation used: colons on the first two lines, commas on the second line, and the period on the second line. All are required.

When the user types in the exact name of a connection to another room (Dungeon, for example), and then hits return, your program should write a new line, and then continue running as before. For example, if I typed twisty above, here is what the output should look like:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >twisty

CURRENT LOCATION: twisty
POSSIBLE CONNECTIONS: PLOVER, XYZZY, Dungeon, PLUGH.
WHERE TO? >
```

If the user types anything but a valid room name from this location (case matters!), the game should return an error line that says “HUH? I DON’T UNDERSTAND THAT ROOM. TRY AGAIN.”, and repeat the current location and prompt, as follows:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
```

```
WHERE TO? >Twisty

HUH? I DON’T UNDERSTAND THAT ROOM. TRY AGAIN.

CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >
```

Trying to go to an incorrect location does not increment the path history or the step count. Once the user has reached the End Room, the game should indicate that it has been reached. It should also print out the path the user has taken to get there, the number of steps taken (*not* the number of rooms visited, which would be one higher because of the start room), a congratulatory message, and then exit.

Note the punctuation used in the complete example below: we're looking for the same punctuation in your program.

When your program exits, set the exit status code to 0, and leave the rooms directory in place, so that it can be examined.

If you need to use temporary files (you probably won't), place them in the directory you create, above. Do not leave any behind once your program is finished. We will not test for early termination of your program, so you don’t need to watch for those signals.

Time Keeping

Your game program must also be able to return the current time of day by utilizing a second thread and mutex(es). I recommend you complete all other portions of this assignment first, then add this mutex-based timekeeping component last of all. Use the classic pthread library for this simple multithreading, which will require you to use the "-lpthread" compile option switch with gcc (see below for compilation example).

When the player types in the command "time" at the prompt, and hits enter, a second thread must write the current time in the format shown below (use strftime() (Links to an external site.)Links to an external site. for the formatting) to a file called "currentTime.txt", which should be located in the same directory as the game. The main thread will then read this time value from the file and print it out to the user, with the next prompt on the next line. I recommend you keep the second thread running during the execution of the main program, and merely wake it up as needed via this "time" command. In any event, at least one mutex must be used to control execution between these two threads.

Using the time command does not increment the path history or the step count. Do not delete the currentTime.txt file after your program completes - it will be examined by the grading TA.