CS 3410: MUD

Introduction

In this assignment, you will write a simple Multi-User Dungeon, or MUD, which is a kind of multi-player text role-playing game that eventually evolved into the MMORPGs of today. The goals of this assignment include learning:

- the basics of Go
- to work with a simple SQL database
- to work with basic network connections
- basic multi-threaded programming
- synchronization using an event loop

The assignment is in three parts.

Part 1

In the first part you will write a simple loop to accept commands from the keyboard and process them. A big part of this step is to get a working Go programming environment and learn to use the associated tools.

Getting started

Start by installing Go on the system where you intend to work. Start at the Go documentation page:

- https://golang.org/doc/

The “Getting Started” page talks about how to install and test a Go build system. You should read the following document to learn how to start a project, structure your directories, and use the build tools:

- https://golang.org/doc/code.html

The project

The purpose of this part is to write a little bit of Go code in the environment you have set up, and to explore the standard library a bit.

Write a command processing loop. It should:

- Print a prompt
- Let the user type a line of text
- Parse that line of text and process the command the user typed
- Repeat

Start by writing code to print the prompt (which can be anything) and accept a line of input. You will need help from the standard library, which is documented here:

- https://golang.org/pkg/

The `fmt` package contains code for printing formatted text. To use it, you will need to import it near the top of your source file:

```go
import "fmt"
```

or if it is one of multiple imports you can use:

```go
import {
    "fmt"
    "log"
}
```

Then to print something you would reference the package and the function:

```go
fmt.Println("Hello, world")
```

To read lines of input, I suggest starting with the `bufio` package and using the `Scanner` API. Note that the
package documentation has a complete example that does exactly what you need to do: it reads lines of input and processes each one.

Once you can read lines of input, you need to parse them to recognize commands. A command in a MUD always starts with the command itself, and then has additional text to support it if appropriate. Example commands:

- look
- look north
- sigh
- laugh
- north
- n
- say I’ll be right back
- tell alice I am at the shop

In each case the command is the first part of the line (separated by one or more space characters from the rest of the line). Look at the `strings` package to find helpful functions to parse and process strings. There are more complex parsing functions in the standard library, but `strings` will let you do simple tasks like splitting a string on whitespace into words (the `Fields` function), etc.

Next, you should implement a way to dispatch different commands in a way that lets you easily add to the list of supported commands. You can do this in many different ways, but the most straightforward way is to create a `map` from a command name to a function that processes the command. For example, if the function that processes the look command looks like:

```go
func doLook(line string) {
    // process the look command
}
```

You might have a map of this type:

```go
var commands map[string]func(string)
```

When the user types a command, you would look it up in the `map`. If it is present, it is a valid command and you can call the associated function. If not, tell the user you did not understand their request.

There are other ways to implement a command dispatcher, and you are welcome to use whatever makes sense to you. However:

- Make sure each command is implemented in its own function. Do not use a giant `switch` statement with all of the commands implemented in one giant function.
- Make sure it is easy to add new commands. I suggest having a function that adds commands to the list. Something like:

```go
func addCommand(command string, action func(string)) {...}
```

Then when your code is starting up, you would register all of the commands, each with a call to `addCommand`.

In addition, MUD users end up typing a lot and expect to be able to use shortcuts. You should be able to recognize a prefix of a command as being the same as the command. So these should all be recognized as the “north” command:

- north
- n
- no
- nor
- nort

If you are using a `map`, you could add duplicate entries for all prefixes of the command. There may be multiple commands with some of the same prefixes, in which case you should be able to prioritize which one wins. More important/command commands should be able to use the shortest abbreviates. So “e” and “ea” are short for “east”, not “eat”, because moving around is more common than eating.

The examples above all show the command processing function as taking a single string parameter. This is just for demonstration purposes: you should figure out what your command processor actually needs, and you may find that the requirements will change as the project progresses.
Part 2

In this part you will add a simple SQL database to your MUD to store the rooms and zones that make up the world.

Using sqlite

Go has a standard SQL database driver ([database/sql](https://golang.org/docpkg/github.com/mattn/go-sqlite3)), but to use it you must include a driver for the specific database you are using. We will use sqlite3 in this project, which is an embedded database. To install the driver, use:

```
go get github.com/mattn/go-sqlite3
```

In one of your source files (technically it does not matter which), add the driver to your imports using:

```go
_ "github.com/mattn/go-sqlite3"
```

The underscore means that you will not be referring to the package directly in your code (you will work with the database/sql package of the standard library), but you need it to load so it can register itself.

To see the basics of how to use the driver, see the simple example from the driver package:

* [https://github.com/mattn/go-sqlite3/blob/master/_example/simple/simple.go](https://github.com/mattn/go-sqlite3/blob/master/_example/simple/simple.go)

Sqlite keeps all of its data in a single file; when you open the database, you supply the name of that file. At the time you open the database, you can set various configuration options. I suggest something like the following:

```
// the path to the database--this could be an absolute path
path := "world.db"
options :=
    "?" + "_busy_timeout=10000" +
    "&" + "_foreign_keys=ON"

db, err := sql.Open("sqlite3", path+options)
if err != nil {
    // handle the error here
}
```

**The world database**

The world is organized into **zones**, and each zone has some number of **rooms**. I have extracted the basic world information from the classic MERC mud distribution so that you do not have to start from scratch. That data is supplied here:

* [world.sql](https://github.com/mattn/go-sqlite3/blob/master/_example/simple/simple.go)

This is a sqlite database dump. It is a text file containing a series of commands that, when executed in order, reproduce an entire database. Search through it and look at the **CREATE** statements to see the schema. The **INSERT** statements provide the data. Note that there are three tables: zones, rooms, and exits (the connections between rooms).

You can create the database from this file using the following command:

```
sqlite3 world.db < world.sql
```

This assumes that you have sqlite3 installed on your system (you should install it if you do not already have it).

The starting point of this world is room 3001, The Temple of Midgaard. The original data included items, mobs (short for mobile objects, another name for monsters), shops, etc., but we will only be using the basic room descriptions and connections.

**An example**

A Linux binary demonstrating the functionality is available here:

* [mud-part2-example.tar.gz](https://github.com/mattn/go-sqlite3/blob/master/_example/simple/simple.go)
Download it, untar it, and run it in the same directory as the `world.db` file. The supported commands include:

- north, east, west, south, up, down
- look, look north, look east, etc.
- recall (to return to the starting location)
- smile

To quit, type ctrl-d to end the stream of input.

**Queries**

To retrieve a kind of object from the database, you will need to run a *query*. You can run these by hand within the sqlite3 tool:

```bash
sqlite3 world.db
```

and then type a query, such as:

```sql
select id, zone_id, name, description from rooms where id = 3001;
```

You can change the way the data is presented, see the database schema, and various other operations as well. Type `.help` for details.

To run a query within your Go program, you need to use the database driver. See the example cited above for the basics of how to run commands. As an example, you could retrieve all of the zones using:

```sql
SELECT id, name FROM zones ORDER BY id
```

An example in the standard library documentation shows how to issue a query and read in the results using a `Rows` object:

- [https://golang.org/pkg/database/sql/#example_Rows](https://golang.org/pkg/database/sql/#example_Rows)

I suggest the following datatypes in Go:

```go
type Zone struct {
    ID    int
    Name  string
    Rooms []*Room
}

type Room struct {
    ID    int
    Zone  *Zone
    Name  string
    Description string
    Exits  [6]Exit
}

type Exit struct {
    To    *Room
    Description string
}
```

To get started, I suggest the following:

- Write a Go function that opens the database and reads a single room (use the query given earlier to load room 3001). Store the `ID`, `Name`, and `Description` fields in a `Room` object and prints it out.

- Add a new function to read all of the zones (see the `Rows` examples from the standard library documentation and the example from the sqlite driver library). Collect all of the zones into a map where the keys are zone IDs and the values are `Zone` pointers. Print them all out.

- Refactor your code so that `main` opens the database and starts a transaction, calls the function to read in all the zones with that transaction, then commits the transaction when it returns (or rolls it back if there is an error).

- Write a new function to read in all of the rooms. Have it accept an open transaction as a parameter and
return a map from IDs to \texttt{Room} pointers. In addition, have it accept the map of zones as a parameter. When you get a zone ID from the database, use it to find the corresponding \texttt{Zone} pointer and store it in the \texttt{Room} object. Make sure that \texttt{main} opens a transaction before calling this function, then either commits or rolls back the transaction depending on whether or not there were errors.

- Note: if you detect an error, you should return an error. Have \texttt{main} actually report the error to the console and decide what to do next (usually quitting). The functions that it calls should just return errors. If you want to add additional information, make your own error. For example:

```go
if err != nil {
    return fmt.Errorf("reading a room from the database: %v", err)
}
```

\texttt{fmt.Errorf} is a version of \texttt{Printf} that constructs an error, so this example uses it to construct an error message with useful information in it that also embeds the error message it detected.

- Write another similar function to read in all of the exits. For each one, find the room it leaves from and fill in the corresponding exit field of the room. Note that this is a rare occasion where using an array (not a slice) of objects makes sense. Every room has exactly six exits (though some may not be used), and declaring them this way embeds the storage for the exits directly in the room record, which simplifies memory a bit. Because the type has exactly six exits in the array, you never have to check the length (as you would normally do with a slice). Note that the index of the exit tells its direction. You should use:

  - 0 → north
  - 1 → east
  - 2 → west
  - 3 → south
  - 4 → up
  - 5 → down

I suggest making lookup tables that make it easy to translate from a direction number to a name and vice versa.

- Now go and integrate this into your command handler. Implement at least:
  
  - The movement commands “north”, “east”, “west”, “south”, “up”, and “down”.
  
  - A “look” command that displays the current room name, description, and exits (see my example implementation for details).
  
  - The ability to “look” in a direction and get the description of the exit in that direction.
  
  - A “recall” command that takes you back to the starting room, which is room 3001.

You should create a \texttt{Player} object that holds information about the player. For now, it will mainly contain the player’s location. Update your command processor to accept a pointer to the \texttt{Player} object as a parameter to each command.

\textbf{Part 3}

In this part you will make your MUD into a multi-player environment. You will introduce the following:

- Concurrency
- Networking
- Multiple players who can interact

\textbf{Goroutines}

A goroutine is Go’s name for a lightweight thread. Each goroutine runs independent of the others, so multiple things can be happening at the same time. A few important notes:

- When you launch a new goroutine using \texttt{go f()}, the code that makes the call continues running and does NOT wait for the called function to complete.
- All goroutines share the same address space, meaning that they can share global variables and even local variables (when pointers or other references are shared).
- Each goroutine has its own system stack, so local variables of different goroutines are independent of each other. Likewise, the call and return tree of one goroutine is not tied in any way to the call and return tree of another goroutine.
Your mud should include the following goroutines:

- The main goroutine that the program starts with. This one is special, in that when it quits, i.e., when `main` returns, the entire program quits. You will use this goroutine initially to set things up, and then it will be the main goroutine of the mud (discussed below).
- One goroutine will accept incoming network connections. It will spend most of the time blocked and waiting for a connection. When it receives one, it will launch a new goroutine that will be dedicated to serving that connection.
- Each player connection will need two goroutines. One will accept incoming commands from the player (and will block and wait the rest of the time between commands). When it receives a command it will hand it over to the main goroutine to execute the command.
- The second goroutine for a connection will accept messages from the main goroutine, e.g., the result of running a command or a message sent from another player, and will pass it on to the outgoing network connection. It will spend most of its time blocked waiting for messages, but may also block while waiting on the network (when messages are being delivered faster than the network can deliver them).

Concurrency allows you to think about different parts of the code that are logically independent of each other as independent entities. Your program can run in many places at once. This is often a natural way to think about a program, since it may involve many different moving parts whose actions do not follow an obvious sequence.

An important danger in concurrent code is that of data races. A data race is a fault where multiple concurrent goroutines access the same data, and one or more of them may change that data. When this happens, the outcome may depend on luck of timing between the goroutines, including incorrect results when the timing works out poorly. Correct code makes such data races impossible.

To avoid data races, we will institute a simple rule: no two goroutines may access the same data. This means you cannot have any variables or objects that can be used by multiple goroutines at the same time. You should be able to point clearly at any piece of data in your program and identify which goroutine owns it. This includes local and global data.

When goroutines need to share data, they will do so by communicating explicitly. In Go we achieve this using channels. A channel is a special data type that allows us to safely communicate between goroutines. A goroutine can write a data item to a channel, and another goroutine can read that value from the channel. It is safe to have many goroutines writing to a channel and/or many goroutines reading from a channel, and the system will ensure that no conflicts occur.

**Networking**

Start by adding networking. The examples in the `net` package show how to listen on a TCP port and accept connections, and also how to dispatch a goroutine to handle an individual connection. Recall that your main goroutine should launch a new goroutine dedicated to accepting connections. To launch a goroutine you can either have an named function and call it will a `go` in front:

```
go foo(a, b)
```

or you may find it more convenient to create the function and call it at the same time:

```
go func() {
    // do stuff
}()
```

This code defines an anonymous function (also called a function literal) and calls it in a new goroutine. It can access the local variables that existed when it was defined and called, so be careful to follow the single-owner rule discussed above (every variable should have a single clear owner, and no other goroutine should access it). The exception is channels, which may be safely accessed by any number of goroutines.

Start by accepting connections and writing simple messages back to the user. You can use `fmt.Fprintf` to write to a connection as you would any `io.Writer` objects:

```
fmt.Fprintf(conn, "Hello, world!\n")
```

Confirm that you can in fact connect to your server using `telnet`. If you are listening on port 9001, then from the same machine you can use:

```
telnet localhost 9001
```
and it should connect. You may have to install telnet, as it is not always installed by default on modern operating systems.

Whenever you create a network connection or a goroutine, you should think about its complete lifecycle. When is it created? Which goroutine creates it? When does its lifetime end? Can you ensure that every connection and every goroutine associated with it always closes cleanly? With concurrent code, there is no substitute for thinking through a problem.

When you can successfully accept connections and dispatch commands from the player that made the connection. A TCP connection is an `io.Reader`, so you can use the same `bufio` package from part 1 to read a line at a time.

The world data is global data, and so it must have a single owner. The owner cannot be the goroutine associated with a single player connection, because there can be many player connections. Instead, it should be owned by the main goroutine (the one that launched all of the others), which will do most of the work in the MUD. It should:

- Create a `struct` type representing an event going from the player to the mud.
- Create a channel to accept events from players.
- Pass this channel to each connection handler so that they can pass commands into it to have them processed. The event should include the `Player` object as well as the command, and may need additional fields later.
- Run in a loop that accepts events from the channel and processes each one in sequence. This allows many players to send commands into the event channel, but they are processed in sequence by a single goroutine, and it happens to be the goroutine that owns the world data.

This will be simpler if the `Player` object itself is owned by the main goroutine. You will create it in the connection-specific goroutine, but then hand over control to the main goroutine after everything is set up. Create the player object in the connection goroutine, and start out by having the player type a name as the first line of input (before tying in to the rest of the system).

Note: there is no formal notion of “ownership” or “control” in Go as we are using the terms here, so handing control from one goroutine to another is just something that you should keep track of mentally (and in the comments) to avoid making errors.

Finally, you need a way for messages from the goroutine to be sent back to the player:

- Create a new `struct` type representing an event going from the MUD to the player.
- Create a channel to accept events from the mud.
- Create a goroutine that accepts events from this channel and processes them, writing data to the connection when appropriate.
- Add this channel to the `Player` object so the main MUD goroutine can find it and send data to the player whenever it needs to.

Note that there is only one channel that all players share to send events to the MUD, but each player connection has its own channel to accept events going in the other direction.

Let us review the goroutines and what each one does:

- The main goroutine processes events from players. It owns all of the `Player` objects (after they have been initialized) and all of the world data (which it initializes before everything else). It spends most of its time waiting for commands from players.
- One goroutine accepts connections from players and launches a fresh goroutine for each one. It spends most of its time waiting for new connections.
- Each connection goroutine starts by setting up the `Player` (more on that later) and then waits for commands from the player that it can pass on to the main goroutine through a shared channel. It spends most of its time waiting for the player to type a new command.
- Each connection has a second goroutine that accepts events from the main goroutine that should be passed on to the player. It spends most of its time blocked waiting for output from the mud to the player.

An important detail is how to close a connection safely and make sure that both connection goroutines close down safely with it. Here is one approach:

- The main goroutine is the only one that decides when a player connection is finished. It communicates this by closing the channel used to communicate events from the main goroutine to the player (see the built-in `close` function for Go channels for details).
- When the goroutine that accepts events from the main MUD goroutine and passes them on to the user sees that the channel is closed, it logs this to the console (for debugging purposes) and closes the
connection (using its \texttt{Close()} method). The goroutine then exits normally by allowing the function to return. Note that this is simple and natural structure for this function: be sure to look up how to use a \texttt{for} loop with \texttt{range} with channels and what happens when the channel closes.

- Closing the connection will cause an error to occur in the code that is reading commands from the user. It should respond by sending an event to the main MUD goroutine requesting that the connection be shut down. You may wish to add a new field to the event struct to distinguish these requests from normal commands. This goroutine should then exit (by ending the loop it was in and letting the function return). Be sure to log a message so that you can verify that this happens.

The main MUD goroutine has the most subtle rules:

- It should always check if the channel in a \texttt{Player} object is nil before trying to access it. If it is nil, then the player connection is in the process of shutting down and any commands should be ignored and events discarded. This includes requests to shut down the connection, which might come after a shutdown has already been initiated.
- When it gets an event to shut down a connection, it should close the channel and then set the channel reference in the \texttt{Player} object to nil (to prevent trying to close it a second time which would cause a panic).
- When the player’s connection channel is closed, the player should be removed from any other data structures, for example a list of all players, or a list of all players in a zone, etc.

Here are a few common sequences of events that might happen:

- A player types “quit”. The command is passed to the main goroutine, which responds by closing the player event channel. The goroutine processing those events responds to the channel close by closing the network connection and exiting the goroutine. The closed network connection causes the command-reading goroutine to send a close event to the main MUD goroutine and then exit. The main MUD goroutine receives that event but ignores it since it has already closed the channel and set it to nil.
- A player’s network connection fails. The goroutine reading commands detects the network error and sends a quit request to the main goroutine. It then exits normally. The main goroutine receives the request, closes the outgoing event channel for the player, sets the channel to nil, and removes the player from all global data structures. The goroutine processing outgoing messages for the player detects the closed channel and closes the network connection and exits.

Since the main goroutine always initiates the actual disconnect, it can do so in response to events other than a player request if it needs to. For example, if a player has not typed a command for a long time and the MUD decides to timeout and disconnect, or if an administrator kicks the player off, or if the same player logs in with a fresh connection and the old connection needs to be terminated.

**Player names**

When a player connection first starts and before the main loops begin, ask the player to log in. For simplicity:

- Print a message requesting the player’s name
- Read a single line of input

This should happen in the connection goroutine before starting the main commands loops. You should introduce the player to the main event loop with a special event (add a field to the event struct to signal when this happens) so the main MUD goroutine can add the player to global data structures that it owns and possibly terminate an existing connection for the same player.

**Multiplayer features**

Once you have the ability to connect multiple players, you should add a few interactive features. This is the fun part and is much easier than the infrastructure we have focused on so far. A few suggestions:

- A gossip command, which sends a message to everyone currently logged in, e.g., “Sage gossips: good evening everyone”
- A say command, which sends a message to everyone in the same room.
- A tell command, which sends a message to a specific other player by name, regardless of where that player is.
- A shout command, which sends a message to other players in the same zone.
- Notify other players in a room when a player arrives.
- The look command should show other players in the same room.
- A where command, which displays a list of other players in the same zone along with the name of the room each is in (but not the description).
- Social commands that send a fixed message to other players in the same room: sigh, laugh, chuckle,
glare, wink, poke, etc. Some of these commands may have a target player, e.g., "poke eve".

**An example**

A Linux binary demonstrating the functionality is available here:

- [mud-part3-example.tar.gz](#)

Download it, untar it, and run it in the same directory as the `world.db` file. You should also create the `players` table before running it. The supported commands include:

- north, east, west, south, up, down
- look, look north, look east, etc.
- recall (to return to the starting location)
- say, shout, gossip, tell
- who, where
- smile
- quit

To connect, use `telnet localhost 9001` from the same machine, or put the IP address/hostname in place of "localhost" to connect from a different machine.

Note that the demo asks you to create a password and it adds players to the database. You do not need to implement this functionality. Your code should just prompt for the player’s name and create the new player on the fly each time.