Project 1

MPCS 52060 – Parallel Programming

Due: July 24th 2020, by 11:59 pm

Initial Setup

1. Make sure to perform a pull upstream inside your repository. This will grab the distribution code for proj1. The command is the following:

```
        $ git pull upstream master
```

2. The **proj1** directory should have the following structure (bold items represent a directory):

- **proj1**
  - go.mod
  - **twitter**
    - twitter.go
  - **feed**
    - feed.go
    - feed_test.go
  - **lock**
    - rwlock.go

As you complete this assignment, I will state what should go in each file. A few of these files are already created for you and others you will create on your own.

The name of the file must match exactly as specified. **Failing to name your files and directories as specified in the homework will incur point penalties**.

Assignment: Single-User Twitter Feed

For this assignment you are **only allowed** to use the following Go concurrent constructs:

- `go` statement
- `sync` package with its types and its associated methods
- `sync/atomic` package with its types and its associated methods

You **cannont** use Go channels (i.e., `chan`) or anything related to channels in this assignment!. **If you are unsure about whether you are able to use a language feature then please ask on pizza**.

Part 1: Twitter Feed

Imagine you are a new software engineer at Twitter and your first assignment is to redesign the data structure that represents a user's feed. Your implementation will redefine it as a linked list. If you are unaware about linked lists then I recommend you look over this resource:

- `https://en.wikipedia.org/wiki/Linked_list`

Your task to implement the remaining incomplete methods of a `feed` (i.e. the `Add`, `Remove`, and `Contains` methods). You **must** use the internal representations for `type feed struct` and and `type post struct` in your implementation. You can only add fields to the `struct` but cannot modify the original fields given.

Test your implementation of `feed` by using the test file called `feed_test.go`. You should only run the sequential tests first:

- TestSimpleSeq
- TestAdd
- TestContains
- TestRemove

For all parts of this assignment, you can run individual tests by using the `-run` flag when using `go test`. This flag takes in a regular expression to know which tests to run. Make sure are you in the directory that has the `*_test.go` file. For example,

```
//Run top-level tests matching "SimpleSeq", such as "TestSimpleSeq".
$ go test -run "SimpleSeq"
PASS
ok          proj1/feed        0.078s

//Run top-level tests matching "SimpleSeq" such as "TestSimpleSeq" and "TestAdd". -v is verbose
// which shows more information about the tests
$ go test -v -run "SimpleSeq|TestAdd"
=== RUN   TestSimpleSeq
--- PASS: TestSimpleSeq (0.00s)
=== RUN   TestAdd
--- PASS: TestAdd (0.00s)
PASS
ok          proj1/feed        0.078s
```

Part 2: Thread Safety using a Read-Write Lock

Condition variables will be covered in Week 4. However, if you would like to get started on completing Parts 2 and 3 then you can read about them here (ignore the information about channels):

`https://kaviraj.me/understanding-condition-variable-in-go/`

A read/write lock mechanism allows multiple readers to access a data structure concurrently, but only a single writer is allowed to access the data structures at a time. Implement a read/write lock library that **only** uses a **single condition variable** and **mutex** for its synchronization mechanisms. Go provides a Read/Write lock that is implemented using atomics:

`https://golang.org/pkg/sync/#RWMutex`

As with the Go implementation, you will need to provide four methods associated with your lock:

- `Lock()`
- `Unlock()`
- `RLock()`
- `RUnlock()`

These methods should function exactly like their Go counterparts. Documentation on each method's functionality is described by the link provided above. You **must** limit the max number of readers to `64`.

Now, go back to your `feed` library inside of `feed.go` and make it thread-safe by using your implementation of a read/write lock. You need to think about the appropriate places to call the various read/write locking and unlocking methods in the `feed` methods. Do not over think this step. It should be simple and straightforward. Test that this is working by running the remaining tests inside the `feed_test.go` file.

Part 3: A Twitter Feed Task Queue

Inside the `twitter.go` file, write a concurrent Go program that implements a *task queue*. This task queue is your first exposure to the producer-consumer model, where the *producer* will be the main goroutine and its job is to collect a series of tasks and place them in a queue structure to be executed by consumers (also known as *workers*). The consumers will be spawned goroutines. You **must** implement the parallelization as follows:

1. The main goroutine reads in two command line arguments: `<number of goroutines>` and `<block size>`. The `<number of goroutines>` is an integer and represents the number of goroutines to spawn (i.e., the number of consumers to spawn). The `<block size>` argument is an integer and represents the maximum amount of tasks a goroutine should be processing at any time. This argument is exaplained in more detail below.

2. The main goroutine begins by spawning a specified `<number of goroutines>` goroutines, where each will begin executing a function called `func consumer(...)`. The arguments you passed to this function is up to you to decide. Each goroutine will either begin doing work or go to sleep in a conditional wait if there is no work to begin processing yet. This "work" is explained in Steps 3 and 4. **Your program cannot spawn additional goroutines after this initial spawning by the main goroutine**.

3. After spawning the consumer goroutines, the main goroutine will call a function `func producer(...)`. Again, what you pass to this function is for you to decide. Inside the `producer` function, the main goroutine reads in from `os.Stdin` a series of tasks. For the sake of explicitness, the tasks will be `feed` operations for a single user-feed that the program manages. The main goroutine will place the tasks inside of a queue data structure and do the following:
   - If there is at least one consumer goroutine waiting for work then place a task inside the queue and wake one consumer up.
   - Otherwise, the main gorountine continues to place tasks into the queue. Eventually, the consumers will grab the tasks from the queue at later point in time.

4. Inside the `func consumer(...)` function, each consumer goroutine will try to grab at least a `<block size>` amount of tasks from the queue. If there is less than a `<block size>` amount of tasks in the queue then the consumer grabs all the tasks from the queue and starts to execute them. When a consumer finishes executing its block of tasks, it checks the queue to grab another `<block size>` amount of tasks. If there are no tasks in the queue then it will need to wait for more tasks to process or exit its function if there are no remaining tasks to complete.

Additional Queue Requirements

You must implement this queue data structure so that both the main and worker goroutines have access to retrieve and modify it. All work is placed in this queue so workers can grab a block of tasks when necessary. Along with the requirements defined in this section, the actual enqueuing and dequeuing of items must also be done in a **unbounded lock-free manner** (i.e., **non-blocking**). However, the code to make the producer signal to consumers, and consumers to wait on work must be done using a condition variable.

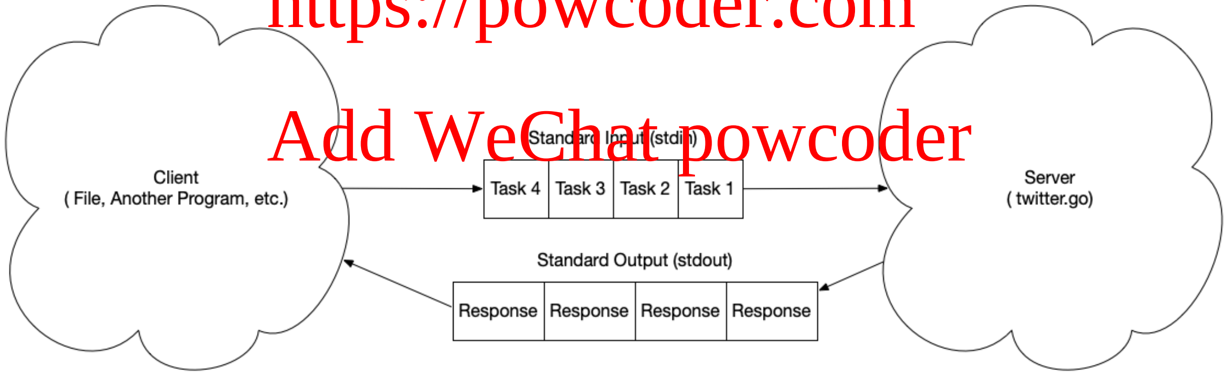
You may want to separate out the queue implementation into its own package and then have `twitter.go` import it. This design is up to you. However, the `producer` and `consumer` functions must always remain in `twitter.go`. I will also allow you to separate out the producer/consumer condition variable code from the unbounded lock-free queue code. Again, this is for you to decide.

Sequential Version

You will need to write a sequential version of this program where the main goroutine processes and executes all the requested tasks without spawning any gorountines. How this version will be run is provided in the "Program Usage" section.

Formatting and Processing Tasks

At a high-level, you can think of your program as a simple "server" in the client-server model illustrated below:



Requests (i.e., tasks in our program) are sent from a "client" (e.g., a redirected file on the command line, a task generator program piped into your program, etc.) via `os.Stdin`. The "server" (i.e., your program) will process these requests and send their results back to the client via `os.Stdout`. This model is a simplified version of a real-life client-server model used heavily in many domains such as web development.

A task is a single string in JSON format. If you are unfamiliar with the JSON standard then you should read up on it before beginning this portion of the assignment. JSON is a widely used serialization format that you will mostly encounter at some point in your career. Thus, it's better that you learn it now so you can be comfortable with it in the future. How to work with JSON data inside Go is described here:

`https://blog.golang.org/json-and-go`

The basic format for the task requests coming in from `os.Stdin` will be of the following format:

```
{
  "command": string,
  "id": integer,
  ... data key-value pairings ...
}
```

A request will always have a “command” and “id” key. The “command” key holds a string value that represents the type of feed task. The “id” represents an unique identification number for this request. Requests are processed asynchronously by the server so requests can be processed out of order from how they are received from `os.Stdin`; therefore, the “id” acts as a way to tell the client that result coming back from the server is a response to an original request with this specific “id” value. Thus, it is not your responsibility to maintain this order and you must not do anything to maintain it in your program.

The remaining key-value pairings represent the data for a specific request. The following subsections will go over the various types of requests.

Add Request
An add request adds a new post to the feed data structure. The “command” value will always be the string `"ADD"`. The data fields include a key-value pairing for the message body (`"body": string`) and timestamp (`"timestamp": number`). For example,

```
{
  "command": "ADD",
  "id": 342,
  "body": "just setting up my twttr",
  "timestamp": 43242423
}
```

After completing a `"ADD"` task, the goroutine assigned the task will send a response back to the client via `os.Stdout` acknowledging the add was successful. The response is a JSON object that includes a success key-value pair (`"success": boolean`). For an add request, the value is always `true` since you can add an infinite number of posts. The original identification number should also be included in the response. For example, using the add request shown above, the response message is

```
{
  "success": true,
  "id": 342
}
```

Remove Request
A remove request removes a post from the feed data structure. The “command” value will always be the string `"REMOVE"`. The data fields include a key-value pairing for the timestamp (`"timestamp": number`) that represents the post that should be removed. For example,

```
{
  "command": "REMOVE",
  "id": 2361,
  "timestamp": 43242423
}
```

After completing a `"REMOVE"` task, the goroutine assigned the task will send a response back to the client via `os.Stdout` acknowledging the remove was successful or unsuccesful. The response is a JSON object that includes a success key-value pair (`"success": boolean`). For a remove request, the value is `true` if the post with the requested timestamp was removed, otherwise assign the key to `false`. The original identification number should also be included in the response. For example, using the remove request shown above, the response message is

```
{
  "success": true,
  "id": 2361
}
```

Contains Request
A contains request checks to see if a feed post is inside the feed data structure. The "command" value will always be the string `"CONTAINS"`. The data fields include a key-value pairing for the timestamp (`"timestamp": number`) that represents the post to check. For example,

```
{
  "command": "CONTAINS",
  "id": 2362,
  "timestamp": 43242423
}
```

After completing a `"CONTAINS"` task, the goroutine assigned the task will send a response back to the client via `os.Stdout` acknowledging whether the feed contains that post. The response is a JSON object that includes a success key-value pair (`"success": boolean`). For a contains request, the value is `true` if the post with the requested timestamp is inside the feed, otherwise assign the key to `false`. The original identification number should also be included in the response. For example, using the contains request shown above, the response message is

```
{
  "success": false,
  "id": 2362
}
```

Note: Assuming we removed the post previously.

Feed Request

A feed request returns all the posts within the feed. The "command" value will always be the string `"FEED"`. Their are no data fields for this request. For example,

```
{
  "command": "FEED",
  "id": 2,
}
```

After completing a `"FEED"` task, the goroutine assigned the task will send a response back to the client via `os.Stdout` with all the posts currently in the feed. The response is a JSON object that includes a success key-value pair (`"feed": [objects]`). For a feed request, the value is a JSON array that includes a JSON object for each feed post. Each JSON object will include a "body" key (`"body": string`) that represents a post's body and a "timestamp" key (`"timestamp": number`) that represents the timestamp for the post. The original identification number should also be included in the response. For example, assuming we inserted a few posts into the feed, the response should look like

```
{
  "id": 2
  "feed":[
         {
           "body": "This is my second twitter post",
           "timestamp": 43242423
         },
         {
           "body": "This is my first twitter post",
           "timestamp": 43242420
         }
         ]
}
```

Done Request
If client will no longer send requests then it sends a done request. The “command” value will always be the string "DONE". Their are no data fields for this request. For example,

```
{
  "command": "DONE"
}
```

This notifies server it needs to “shutdown” (i.e., close down the program). A done request signals to the main goroutine that no further processing is necessary after this request is received. No response is sent back to the client. Make sure to handle all remaining requests in the and responses before shutting down the program.

Program Usage

The program should have the following usage and required command-line argument:

```
Usage: twitter <number of goroutines> <block size>
    <number of goroutines> = the number of goroutines to be part of the queue
    <block size> = the maximum number of tasks a goroutine can process at any given point in time.
```

If `<number of goroutines>` and `<block size>` is not entered then this means you need to run your sequential version of the program.

Assumptions: No error checking is needed. All tasks read-in will be in the correct format with all its specified data. All command line arguments will be given as specified.

Sample Files/Test Cases

You may want to create a few sample files with a few tasks within them (all on seperate lines). For example, you could create a file called `tasks.txt` with the following contents:

```
{"command": "ADD", "id": 1, "body": "just setting up my twttr", "timestamp": 43242423}
{"command": "ADD", "id": 2, "body": "Another post to add", "timestamp": 43242421}
{"command": "REMOVE", "id": 3, "timestamp": 43242423}
{"command": "DONE"}
```

You could then use file redirection to supply those files to your twitter program as such:

```
$ ./twitter 2 1 < tasks.txt
```

Note you are not reading from a file in this assignment! You are using file redirection that is part of terminal to supply `os.Stdin` with the contents of the file specified on the command line.

I will supply you with a few test cases at the end of the Week 4 to help you test for correctness. However, as programmers, I want you to be able to test your programs yourself before I provide you with additional test cases.

Program Decomposition

Pay close attention to your program decomposition! You may need to add functionality to this program in homework 5. Make sure you appropriately split your programs into functions to ensure modularity.