CSCI 4210 — Operating Systems
Homework 3 (document version 1.1)
Multi-Threaded Network Programming and Wordle

- **(v1.1)** You can optionally work in teams of at most two students (from either section)
- This homework is due in Submitty by 11:59PM EST on Wednesday, August 16, 2023
- You can use at most three late days on this assignment; **(v1.1)** in such cases, each team member must use a late day
- **(v1.1)** Beyond your team (or yourself if working alone), **do not share your code**
- Place your code in `hw3.c` for submission; you may also optionally include your own header files
- You **must** use C for this assignment, and all submitted code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- You **must** use the POSIX thread (Pthread) library by appending the `-pthread` flag to `gcc`
- All submitted code must successfully compile and run on Submitty, which (as of July 24) uses Ubuntu v22.04.2 LTS and `gcc` version 11.3.0 (`Ubuntu 11.3.0-1ubuntu1~22.04.1`)

Hints and reminders

To succeed in this course, do **not** rely on program output to show whether your code is correct. Consistently allocate **exactly** the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code.

Make use of `valgrind` to check for errors with dynamic memory allocation and dynamic memory usage. As another helpful hint, close open file descriptors as soon as you are done using them.

Finally, always read (and re-read!) the `man` pages for library functions, system calls, etc.

Homework specifications

In this third and final assignment, you will use C to implement a single-process multi-threaded TCP server for the Wordle word game. You will use POSIX threads to implement a TCP server that handles multiple client connections **in parallel**.

Specifically, your top-level main thread blocks on the `accept()` system call, listening on the port number specified as a command-line argument. For each connection request received by your server, create a child thread via `pthread_create()` to handle that specific connection.

Each child thread manages game play for one client and only for one hidden word. Both during and after game play ends, child threads update a set of global variables.

Note that child threads are **not** joined back in to the main thread.

And remember that all threads run within one process.

Wordle game play

To learn how to play this one-player game, visit `https://www.nytimes.com/games/wordle`. In brief, a five-letter word is selected at random, then a player has up to six guesses to guess the hidden word. For each guess, the player sees which guessed letters are in the correct position (if any), which guessed letters are in the word in an incorrect position (if any), and which guessed letters are not in the word at all.

Note that only valid five-letter words are allowed as guesses! Therefore, if a guess is not in the given dictionary file, it does not count as a guess. In general, expect your server to receive anything, including erroneous data.

Game play stops when the player guesses the word correctly or runs out of guesses. In either case, the server thread closes the TCP connection, then that corresponding thread terminates.

Global variables and compilation

The given `hw3-main.c` source file contains a short `main()` function that initializes four global variables, then calls the `wordle_server()` function, which you must write in your own `hw3.c` source file.

Submitty will compile your `hw3.c` code as follows:


```
bash$ gcc -Wall -Werror hw3-main.c hw3.c -pthread
```


You are **required** to make use of the four global variables in the given `hw3-main.c` source file. To do so, declare them as external variables in your `hw3.c` code as follows:


```
extern int total_guesses;
extern int total_wins;
extern int total_losses;
extern char ** words;
```

The first three global variables shown above count the total number of **valid** guesses, the total number of games won, and the total number of games lost, respectively. These totals are accumulated across all active players during game play for the entire lifetime of the server.

The `words` array is a dynamically allocated array of character strings representing all of the hidden words actually used in game play. This array is initially empty—specifically, it is set (in `hw3-main.c`) to be an array of size 1, with `*words` initialized to `NULL`.

Similar to `argv`, the last entry in this array must always be `NULL` so that the list of hidden words can be displayed using a loop, as shown below. (Refer to `command-line-args.c` for an example using this technique.)

```
for ( char ** ptr = words ; *ptr ; ptr++ )
{
  printf( "HIDDEN WORD: %s\n", *ptr );
}
```

Submitty test cases will check these global variables when your `wordle_server()` function returns. Be sure your server returns either `EXIT_SUCCESS` or `EXIT_FAILURE`.

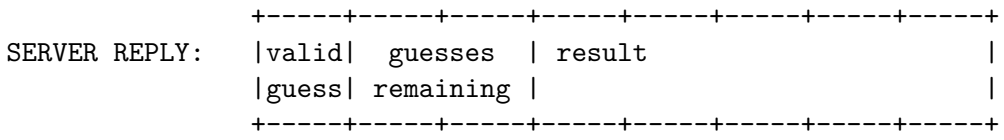
Feel free to use additional global variables in your own code. And since multiple threads will be accessing and changing these global variables, **synchronization is required**.

Application-layer protocol

The specifications below focus on the application-layer protocol that your server must implement to successfully communicate with multiple clients simultaneously.

Once a connection is accepted, the client sends a five-byte packet containing a guess, e.g., `"ready"`; the guessed word can be a mix of uppercase and lowercase letters, as case does not matter.

The server replies with an eight-byte packet that is formatted as follows:



The `valid guess` field is a one-byte `char` value that is either `'Y'` (yes) or `'N'` (no).

The `guesses remaining` field is a two-byte `short` value that indicates how many guesses the client has left. Since a client starts with six guesses, this counts down from five to zero for each valid guess made.

The `result` field is a five-byte character string that corresponds to the client's guess. If a guess is not valid, simply send `"?????"`; for a valid guess, encode the results as follows. Use an uppercase letter to indicate a matching letter in the correct position. Use a lowercase letter to indicate a letter that is in the word but not in the correct position. And use a `'-'` character to indicate an incorrect letter not in the word at all.

As an example, if the hidden word is "wears," and a client guesses "ready," the server replies with `"rEA--"`; note that both guesses and words may contain duplicate letters. In such cases, there are additional rules to follow.

As an example, assume the hidden word is "radar." If a client guesses a word with duplicate letters, the response indicates how many such duplicates appear in the hidden word. For example, guessing "whirr," the response is `"---rR"`. If instead the hidden word is "sewer," the response is `"w---R"` for guess "whirr."

As a trickier example, if the guess is "error" (for hidden word "radar"), the response is `"-r--R"` since the matching letter is shown, then duplicates are evaluated left-to-right.

As another example, if a client guesses a word with a duplicate letter that only appears once in the hidden word (e.g., guessing "muddy" when the hidden word is "radar"), only one duplicate letter guessed will be in the response (e.g., `"--D--"`).

Finally, if a client guesses a word with a duplicate letter that appears at least twice in the hidden word (e.g., guessing "muddy" when the hidden word is "udder"), both duplicate letters guessed will be in the response (e.g., `"-uDd-"`).

The game ends when the client sends the correct word or the number of guesses remaining reaches zero. When the game is over, the server closes the TCP connection.

Command-line arguments

There are four required command-line arguments. The first command-line argument specifies the TCP listener port number.

The second command-line argument specifies the seed value for the pseudo-random number generator—this is used to "randomly" select words in a predictable and repeatable manner via `rand()`.

The third command-line argument specifies the name (or path) of the input file containing valid words (i.e., the dictionary file), and the fourth command-line argument specifies the number of words in this input file.

Validate the inputs as necessary. If invalid, display the following to `stderr` and return `EXIT_FAILURE`:

```
ERROR: Invalid argument(s)
USAGE: hw3.out <listener-port> <seed> <dictionary-filename> <num-words>
```

The input file should contain words delimited by newline characters. Case does not matter. Here is an example file: `"ready\nheavy\nUPPER\nVaguE\n"`

Signals and server termination

Since servers are typically designed to run forever without interruption, ignore signals `SIGINT`, `SIGTERM`, and `SIGUSR2`.

Still, we need a mechanism to shut down the server. Set up a signal handler for `SIGUSR1` that gracefully shuts down your server by terminating any running child threads, freeing up dynamically allocated memory, and returning from the `wordle_server()` function with `EXIT_SUCCESS`.

Dynamic memory allocation

As with previous homeworks, you must use `calloc()` to dynamically allocate memory. For the global `words` array, you must also use `realloc()` to extend the size of the array.

Do **not** use `malloc()`. And of course, be sure your program has no memory leaks.

No square brackets allowed!

As per usual, **you are not allowed to use square brackets** anywhere in your code!

If a `'['` or `']'` character is detected, including within comments, that line of code will be removed before running `gcc`.

Program execution and required output

To illustrate via an example, you could execute your program as shown below, which has your server process listening on port 8192 for incoming TCP connection requests.

```
bash$ ./hw3.out 8192 111 wordle-words.txt 5757
MAIN: opened wordle-words.txt (5757 words)
MAIN: seeded pseudo-random number generator with 111
MAIN: Wordle server listening on port {8192}
MAIN: rcvd incoming connection request
THREAD 139711842105088: waiting for guess
THREAD 139711842105088: rcvd guess: stare
THREAD 139711842105088: sending reply: --ArE (5 guesses left)
THREAD 139711842105088: waiting for guess
THREAD 139711842105088: rcvd guess: brade
THREAD 139711842105088: invalid guess; sending reply: ????? (5 guesses left)
THREAD 139711842105088: waiting for guess
MAIN: rcvd incoming connection request
THREAD 139711833601792: waiting for guess
THREAD 139711842105088: rcvd guess: brake
THREAD 139711842105088: sending reply: BRA-E (4 guesses left)
THREAD 139711842105088: waiting for guess
THREAD 139711842105088: rcvd guess: brace
THREAD 139711842105088: sending reply: BRACE (3 guesses left)
THREAD 139711842105088: game over; word was BRACE!
...
MAIN: SIGUSR1 rcvd; Wordle server shutting down...
```


To display thread IDs, use `"\%lu"` and `pthread_self()` in your `printf()` calls. Note that you might see duplicate thread IDs for threads not running in parallel.

If a client closes the TCP connection, your server must detect that and display the following, counting the game as a loss:

```
THREAD 139711833601792: client gave up; closing TCP connection...
```

Match the above output format **exactly as shown above**, though note that thread IDs will certainly vary. Also, interleaving output across multiple child threads is expected, though the first four lines and the last line must be first and last, respectively.

Error handling

In general, if an error is encountered in any thread, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution by calling `pthread_exit()`. Only use `perror()` if the given library function or system call sets the global `errno` variable.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere including Submitty, use the techniques below.

First, make use of the `DEBUG_MODE` technique to make sure that Submitty does not execute any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of q is %d\n", q );
    printf( "here12\n" );
    printf( "why is my program crashing here?!\n" );
    printf( "aaaaaaaaaaaaaggggggggghhhh square brackets!\n" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw3.c
```

Second, output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.