CSCI 4210 — Operating Systems
Homework 4 (document version 1.0)
Network Programming and UDP

- This homework is due in Submitty by 11:59PM on Friday, April 18, 2025
- You can use at most three late days on this assignment
- This homework is to be done individually, so **please do not share your code with others**
- Place your code in `hw4.c` for submission; do not include a `main()` function; instead, provide a `wordle_server()` function; you may 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
- All submitted code **must** successfully compile and run on Submitty, which currently uses Ubuntu v22.04.5 LTS and `gcc` version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- You will have **eight** penalty-free submissions on Submitty, after which points will slowly be deducted, e.g., `-1` on submission #9, etc.
- You will have at least **three** days before the due date to submit your code to Submitty; if the auto-grading is not available three days before the due date, the due date will be 11:59PM three days after auto-grading becomes available

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` (or `drmemory` or other dynamic memory checkers) 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 (section 3), system calls (section 2), etc.

Homework specifications

In this fourth assignment, you will use C to implement a single-process UDP server for the Wordle word game. Your server must handle multiple games simultaneously.

You can design and implement either an iterative server or a multi-threaded server using POSIX threads (Pthreads). This decision is entirely up to you!

Regardless of what you decide, your server must manage game play for multiple clients.

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 this 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 but in an incorrect position (if any), and which guessed letters are not in the word at all.

Duplicate (or triplicate) letters can be tricky. If a guess contains duplicate letters and only one such letter is in the hidden word, only one letter of the guess will be marked in the correct or incorrect position. As an example, if the guess is "paper" and the hidden word is "spark," the first 'p' in the guess will be marked as in an incorrect position.

If both the guess and the hidden word contain the same duplicate letters, each will be marked accordingly. For example, if the guess is "paper" and the word is "apple," the first 'p' in the guess will be marked as in an incorrect position, and the second 'p' will be marked as in a correct position.

These duplication rules extend out to triplicates, e.g., "poppy," "mummy," "esses," and so on.

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

Game play stops when the player guesses the word correctly or runs out of guesses.

Command-line arguments

There are four required command-line arguments. The first command-line argument specifies the UDP port number for your server.

The second command-line argument specifies the path of the input file containing all valid words, while the third command-line argument specifies the number of words in this input file. The input file will contain words delimited by newline characters. Case does not matter. Here is an example file with four words: `"ready\nheavy\nUPPER\nVaguE\n"`

The fourth 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()`. So, when a game starts, select the nth word from the given input file of valid words, where $n =$ `rand() % num-words`.

Validate all inputs as necessary; if invalid, display the following to `stderr` and return `EXIT_FAILURE` from `wordle_server()`:

```
ERROR: Invalid argument(s)
USAGE: ./hw4.out <UDP-server-port> <word-file-path> <num-words> <seed>
```

Global variables and compilation

Similar to Homework 3, the given `hw4-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 `hw4.c` source file. The `hw4-main.c` code is provided on the last page.

Submitty will compile your `hw4.c` code as follows:

```
bash$ gcc -Wall -Werror -o hw4.out hw4-main.c hw4.c
```

Do **not** include a `main()` function in your `hw4.c` code.

You are **required** to make use of the four global variables in the given `hw4-main.c` source file. To do so, declare them as external variables in your `hw4.c` code as follows:

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

Initialized to zero, the first global variable, `game_token`, is used to uniquely number games as they are started. At the end of your server's execution, this variable will have a count of the number of games both in progress and completed.

The next two global variables count the total number of games won and the total number of games lost, respectively. Initialized to zero, these totals are accumulated across all games.

Finally, the `words` array is a dynamically allocated array of character strings representing the words actually used in game play. (Note that this is **not** a dictionary of valid words.) This array is initially set in `hw4-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 game words can be displayed using a simple loop, as shown below. (Also refer to the `command-line-args.c` example.)

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

Submitty test cases will check these global variables when your `wordle_server()` function returns. As with Homework 3, return either `EXIT_SUCCESS` or `EXIT_FAILURE`.

Feel free to use additional global variables in your own code, especially if you use a multi-threaded approach. And if you do use a multi-threaded approach, synchronization is required.

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 shutting down any running child threads (if necessary), freeing up dynamically allocated memory, closing any open descriptors, and returning from the `wordle_server()` function with `EXIT_SUCCESS`. Each active game must be ended simply with a line of output; see the **Program execution and required output** section on page 7.

No square brackets allowed!

To emphasize the use of pointers and pointer arithmetic, **you are not allowed to use square brackets** anywhere in your code! If a `'['` or `']'` character is detected, including within comments, Submitty will completely remove that line of code before running `gcc`.

Dynamic Memory Allocation

As with Homework 3, you must use `calloc()` to dynamically allocate memory. Further, you must use `realloc()` to extend the size of the global `words` array, one word at a time.

Of course, you must also use `free()` and have no memory leaks.

Do **not** use `malloc()` or `memset()`. Similar to square brackets, any line containing either of these substrings will be removed before compiling your code.

Application-layer protocol

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

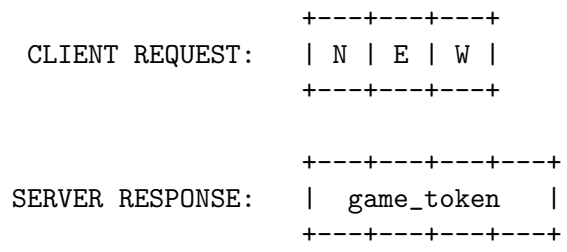
Since UDP is connection-less, when a UDP datagram is received, extract the client's host address and port number (via `sin_addr` and `sin_port`, as shown in the `udp-server.c` example); then, send your UDP datagram response to this same address and port, which may change for each UDP datagram your server receives.

Starting a game

To start a new game, expect a client to initiate a game by sending a three-byte UDP datagram containing the `"NEW"` string. When your server receives this request, use the global `game_token` variable to assign this game a unique token. Since this global is initialized to zero, increment `game_token` before using it, i.e., the first game should be assigned token 1, the second game should be assigned token 2, etc.

Once a game is initiated, send a UDP datagram containing `game_token`; specifically, send this as a four-byte `int` value.

The protocol is shown below.

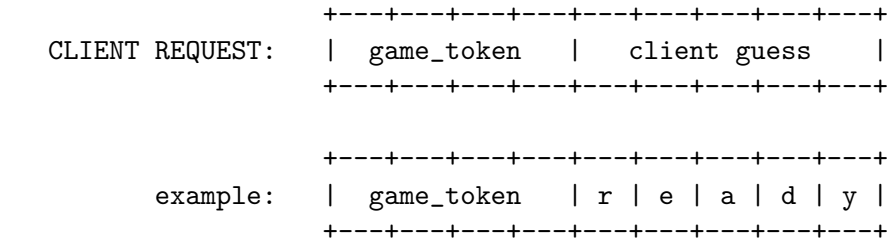


Part of creating a new game is determining the hidden word, which is described in the **Command-line arguments** section above.

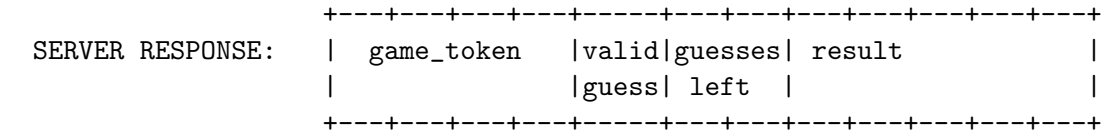
Game play

Once a game is underway, the client sends a nine-byte datagram containing the game token and a guess, e.g., `"ready"`. The guessed word can be a mix of uppercase and lowercase letters; for consistency, downcase the guess before any output or processing.

The protocol format and an example is shown below.



The server replies with a 12-byte datagram 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 left` field is a two-byte `short` value that indicates how many guesses the player has left. Since a player 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 player'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 words may contain duplicate letters, e.g., "muddy" and "radar" and so on (see the **Wordle game play** section for more details).

If the player guesses the correct word or the number of guesses remaining is zero, the server discards the game token after sending the last datagram to the client. Specifically, this datagram has either the `guesses left` field set to zero or the five-letter hidden word in ALL-CAPS as a `result`.

Program execution and required output

To illustrate via an example, you could execute your program as shown below. The server process loads the `knuth.txt` file, verifies it contains 5757 words, then runs a UDP server on port `8192`.

```
bash$ ./hw4.out 8192 knuth.txt 5757 111
Opened knuth.txt; read 5757 valid words
Wordle UDP server started
New game request; assigned game token 1
GAME 1: Received guess: stare
GAME 1: Sending reply: --ArE (5 guesses left)
GAME 1: Received guess: brade
GAME 1: Invalid guess; sending reply: ????? (5 guesses left)
New game request; assigned game token 2
GAME 1: Received guess: brake
GAME 1: Sending reply: BRA-E (4 guesses left)
GAME 2: Received guess: merit
GAME 2: Sending reply: -er-- (5 guesses left)
GAME 1: Received guess: brace
GAME 1: Sending reply: BRACE (3 guesses left)
GAME 1: Game over; word was brace!
SIGUSR1 received; Wordle server shutting down...
GAME 2: Game over; word was spare!
```

Match the above output format **exactly as shown above**. Note that interleaving output across multiple games is expected, though the first two lines and the last line will always be first and last, respectively. The code given in `hw4-main.c` will then print as shown below.

```
games: 2
wins: 1
losses: 0

Word(s) played: - brace - spare
```

Error handling

If improper command-line arguments are given, report the appropriate error message to `stderr` and abort further program execution; refer to the **Command-line arguments** section on page 2 for specific requirements.

In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then abort further execution. Only use `perror()` if the given library 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 perform final testing of your code), we will use Submitty.

To help make sure that your program executes properly, use the techniques below.

First, make use of the `DEBUG_MODE` preprocessor technique that helps avoid accidentally displaying extraneous output in Submitty. 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 -g -D DEBUG_MODE hw4-main.c hw4.c
```

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

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

You would not generally do this in practice since this can slow down your program, but to ensure you see as much output as possible in Submitty, this is a good technique to use.

```
/* hw4-main.c */

/* bash$ gcc -Wall -Werror -o hw4.out hw4-main.c hw4.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* global variables */
int game_token;
int total_wins;
int total_losses;
char ** words;

/* write the wordle_server() function and place all of your code in hw4.c */
int wordle_server( int argc, char ** argv );

int main( int argc, char ** argv )
{
  game_token = total_wins = total_losses = 0;
  words = calloc( 1, sizeof( char * ) );
  if ( words == NULL ) { perror( "calloc() failed" ); return EXIT_FAILURE; }

  int rc = wordle_server( argc, argv );

  /* on Submitty, there will be more code here that validates the
   *  global variables when your wordle_server() function returns...
   */

  printf( "\ngames: %d\nwins: %d\nlosses: %d\n", game_token, total_wins, total_losses );
  printf( "\nWord(s) played:" );

  /* display and deallocate memory for the list of words played */
  for ( char ** ptr = words ; *ptr ; ptr++ )
  {
    printf( " - %s", *ptr );
    free( *ptr );
  }
  printf( "\n" );
  free( words );

  return rc;
}
```