Hangman Game Documentation

Overview

This project implements a networked Hangman game in C, consisting of a client-server architecture. Players can connect to the server to play the Hangman game. The server manages game state, user accounts, and game logic, while the client provides an interface for players to interact with the game.

Link to GitHub repository:

https://github.com/nmocko/hangman-game

Directory Structure

- `client/`
  - `Makefile`: Instructions to compile the client program.
  - `animation.txt`: Contains animation frames for the client.
  - `client.c`: Source code for the client application.
- `server/`
  - `Makefile`: Instructions to compile the server program.
  - `game.c`: Implements game logic and mechanics.
  - `game.h`: Header file for `game.c`.
  - `main.c`: Entry point for the server application.
  - `multicast.c`: Handles multicast communication.
  - `multicast.h`: Header file for `multicast.c`.
  - `read_write.c`: Manages reading from and writing to files.
  - `read_write.h`: Header file for `read_write.c`.
  - `utils.c`: Utility functions used by the server.
  - `utils.h`: Header file for `utils.c`.
  - `ranking`: Stores ranking information.
  - `words`: Contains the word list for the game.
  - `users/`: Directory containing user account files.

Compilation

Client

To compile the client application, navigate to the `client` directory and run `make`:

```
cd client
make
```

This will generate the executable `client`.

Server

To compile the server application, navigate to the `server` directory and run `make`:

```
cd server
make
```

This will generate the executable `server`.

Running the Game

Server

Start the server by running the compiled server executable:

```
./server
```

The server will initialize the game, load the word list, and listen for client connections.

Client

Start the client by running the compiled client executable

```
./client
```

The client will connect to the server, and the player can start interacting with the game.

Code Description

Client

`client.c`

The client code handles the following:

- Connecting to the server.
- Sending player inputs to the server.
- Receiving game state updates from the server.
- Displaying the game state and animations.

Server

`main.c`

The entry point of the server application, which:

- Initializes the server.
- Loads necessary resources (word list, user accounts).
- Manages client connections.
- Coordinates game state and logic.

`game.c` / `game.h`

Contains the core game logic, including:

- Word selection.
- Checking player guesses.
- Updating game state.
- Tracking player progress and lives.

`multicast.c` / `multicast.h`

Handles multicast communication, including:

- Broadcasting messages to multiple clients.
- Managing multicast groups.

`read_write.c` / `read_write.h`

Manages file I/O operations, such as:

- Reading and writing user accounts.
- Saving and loading game state.
- Handling the ranking system.

`utils.c` / `utils.h`

Provides utility functions used across the server codebase, such as:

- String manipulation.
- Logging.

User Accounts and Ranking

The server maintains user accounts in the `users/` directory. Each file corresponds to a user and stores their game progress and ranking. The `ranking` file maintains a leaderboard of players based on their performance in the game.

Word List

The `words` file contains the list of words used in the game. The server randomly selects words from this list for each game session.

Animations

The `animation.txt` file in the `client` directory contains frames for the hangman animation. The client reads from this file to display the game progression visually.

Server

`server/main.c`

Global variable `MAXLINE`:

Description: Maximum length of a single line or buffer in the program.

Usage: This macro defines the maximum number of characters that can be stored or processed in a single line or buffer within the program.

Global variable `LISTENQ`:

Description: Maximum number of pending connections that can be queued up in the server's listen queue.

Usage: Determines the backlog parameter when calling `listen()` on a server socket, specifying the maximum number of incoming connections that can wait in the queue to be accepted.

Global variable `MAXFD`:

Description: Maximum number of file descriptors to iterate through in a loop.

Usage: Specifies the upper limit for iterating through file descriptors, typically used in loops that manage or manipulate file descriptors.

Function `main`:

Description

The `main` function sets up a network server for a Hangman game, handling logging, socket creation, signal processing, and client connections. The function configures a multicast socket and listens for incoming client connections. Each client connection is handled in a separate child process.

Arguments

- `int argc`: The number of command-line arguments.
- `char **argv`: An array of command-line arguments. If an IP address is provided as the first argument, the server will bind to that address; otherwise, it binds to `INADDR_ANY`.

Functionality

1. **Logging Setup**:
    - Configures the logging level and opens a syslog for logging messages with the identifier "Hangman_server".
2. **Signal Handling**:
    - Sets up a handler for the `SIGCHLD` signal to properly handle terminated child processes.
3. **Multicast Setup**:
    - Configures a multicast socket on the specified interface ("wlo1").
4. **Socket Creation and Configuration**:
    - Creates a TCP socket for listening to incoming connections.
    - Sets socket options to allow reuse of local addresses.
    - Configures the server address structure (`servaddr`). If an IP address is provided as a command-line argument, it binds to that address; otherwise, it binds to `INADDR_ANY`.
5. **Binding and Listening**:
    - Binds the socket to the configured server address.
    - Puts the socket into listening mode to accept incoming client connections.
6. **Daemon Initialization**:
    - Converts the process into a daemon process, detaching it from the terminal.
7. **Accepting Connections**:
    - Enters an infinite loop to accept client connections.
    - For each accepted connection:
        - Logs the client's address.
        - Forks a child process to handle the client connection.
        - In the child process, closes the listening socket and calls the `game` function to handle the game logic.
        - In the parent process, closes the connected socket.
8. **Cleanup**:
    - Closes the syslog and the multicast socket upon termination.

Returns

The function returns an integer value:

- `0` on successful execution.
- `1` if any error occurs during setup (e.g., socket creation, binding, listening, or signal handling).
- `-1` if there is an error in address format conversion when an IP address is provided as an argument.

Function `daemon_init`:

Description: Initializes the program as a daemon process. This involves forking twice to detach from the controlling terminal, becoming a session leader, ignoring SIGHUP signals, closing unnecessary file descriptors except for a specified socket, redirecting standard input, output, and error to `/dev/null`, and optionally changing the root directory to `/tmp`.

- **Parameters:**
  - `const char *pname`: Name of the daemon process.
  - `int facility`: Facility for syslog logging (e.g., `LOG_USER`).
  - `uid_t uid`: User ID to switch to after daemon initialization.
  - `int socket`: File descriptor of a socket that should not be closed during file descriptor cleanup.
- **Returns:**
  - `0`: On successful initialization as a daemon process.
  - `-1`: If an error occurs during forking or session leader creation.

Function `sig_chld`:

Description: Handles the `SIGCHLD` signal, which is sent to a parent process when a child process terminates. This function waits for and logs information about terminated child processes.

- **Parameters:**
  - `int signo`: The signal number (`SIGCHLD` in this case).
- **Returns:** None.

Function `sig_pipe`:

Description: Handles the `SIGPIPE` signal, which is sent to a process when it attempts to write to a pipe or socket that has no readers. This function logs a message indicating that the server received a `SIGPIPE` signal, and then exits the process with an error code (`1`).

- **Parameters:**
  - `int signo`: The signal number (`SIGPIPE` in this case).
- **Returns:** None.

`server/multicast.c`

Function `snd_udp_socket`:

Description: Creates a UDP socket and prepares a `struct sockaddr_in` for sending UDP packets to a specified server address and port.

- **Parameters:**
  - `const char *serv`: Server address in string format.
  - `int port`: Port number.
  - `SA **saptrarg`: Pointer to a pointer to a socket address structure (`struct sockaddr`).
  - `socklen_t *lenparg`: Pointer to the length of the socket address structure.
- **Returns:**
  - `int`: Socket file descriptor (`sockfd`) on success, or `-1` on failure.

Function `setup_multicast`:

Description: Sets up a multicast UDP socket by calling `snd_udp_socket`, then configures it to bind to a specific network interface (`interface`) and sets the multicast interface to listen on (`224.2.2.2`).

- **Parameters:**
  - `const char *serv`: Server address in string format.
  - `int port`: Port number.
  - `SA **saptrarg`: Pointer to a pointer to a socket address structure (`struct sockaddr`).
  - `socklen_t *lenparg`: Pointer to the length of the socket address structure.
  - `const char *interface`: Network interface name.
- **Returns:**
  - `int`: Multicast socket file descriptor (`mul_sockfd`) on success.

server/read_write.c

Global variable `MAXLINE`:

Description: Maximum length of a single line or buffer in the program.

Usage: This macro defines the maximum number of characters that can be stored or processed in a single line or buffer within the program.

Struct `TLV` (Type-Length-Value):

Description: Represents messages in TLV (Type-Length-Value) format, commonly used for structured data representation.

- **Members:**
  - **`uint8_t type`:** Type of the TLV message.
  - **`uint16_t length`:** Length of the value in bytes.
  - **`void *value`:** Pointer to the value data.

Function `Writen`:

Description: Writes `n` bytes to a file descriptor (`fd`). Handles interruptions and retries if `write` is interrupted by a signal (`EINTR`).

- **Parameters:**
  - `int fd`: File descriptor to write to.
  - `const void *vptr`: Pointer to the buffer containing data to be written.
  - `size_t n`: Number of bytes to write.
- **Returns:**
  - `ssize_t`: Number of bytes written on success (`n`).
  - `-1`: If an error occurs during writing (`write` returns <= 0).

Function `encode_tlv`:

Description: Encodes a TLV (Type-Length-Value) structure into a byte array. Allocates memory for the encoded data and sets the encoded length.

- **Parameters:**
  - `uint8_t type`: Type of the TLV.
  - `uint16_t length`: Length of the value.
  - `void *value`: Pointer to the value data.
  - `uint16_t *encoded_length`: Pointer to store the length of the encoded data.
- **Returns:**
  - `void *`: Pointer to the encoded TLV data.

Function `send_tlv`:

Description: Sends a TLV structure over a socket. Calls `encode_tlv` to encode the TLV data and uses `Writen` to send it.

- **Parameters:**
  - `int sockfd`: Socket file descriptor.
  - `uint8_t type`: Type of the TLV.
  - `uint16_t length`: Length of the value.
  - `void *value`: Pointer to the value data.
- **Returns:**
  - `int`: `0` on success, `1` if an error occurs during writing.

Function `Readn`:

Description: Reads `n` bytes from a socket into a buffer. Handles interruptions and retries if `read` is interrupted by a signal (`EINTR`).

- **Parameters:**
  - `int sockfd`: Socket file descriptor to read from.
  - `void *buf`: Pointer to the buffer where data will be read into.
  - `size_t n`: Number of bytes to read.
- **Returns:**
  - `ssize_t`: Number of bytes read on success.
  - `-1`: If an error occurs during reading (`read` returns < 0).

Function `decode_tlv`:

Description: Decodes an encoded TLV byte array into a `TLV` structure. Allocates memory for the value data.

- **Parameters:**
  - `void *encoded`: Pointer to the encoded TLV data.
  - `TLV *tlv`: Pointer to a `TLV` structure where decoded data will be stored.
- **Returns:** None.

Function `recv_tlv`:

Description: Receives a TLV structure from a socket. Reads the TLV header and value using `Readn` and stores them in the `TLV` structure.

- **Parameters:**
  - `int sockfd`: Socket file descriptor.
  - `TLV *tlv`: Pointer to a `TLV` structure where received data will be stored.
- **Returns:** None.

`server/game.c`

Global variable `MAXLINE`:

Description: Maximum length of a single line or buffer in the program.

Usage: This macro defines the maximum number of characters that can be stored or processed in a single line or buffer within the program.

`MAX_NICKNAME_LENGTH`

Description: Maximum length for a player's nickname.

Usage: Specifies the maximum number of characters allowed in a player's nickname within the system.

`MAX_MULTICAST_MESSAGE_LENGTH`

Description: Maximum length for a multicast message.

Usage: Defines the maximum size in bytes for a multicast message that can be sent or received.

Function `user_exit`:

Description: Closes the connection (`connfd`) and terminates the process.

- **Parameters:**
  - `int connfd`: File descriptor of the connection to close.
- **Returns:** None.

Function `send_announcement`:

Description: Sends a multicast announcement containing a message (`name`) in TLV (Type-Length-Value) format to a specified multicast address (`224.2.2.2:5554`).

- **Parameters:**
  - `int multicastfd`: Multicast socket file descriptor.
  - `const char *name`: Message to be sent as an announcement.
- **Returns:** None.

Function `view_ranking`:

Description: Reads a ranking file ("ranking") and sends its contents through a socket (`sockfd`) in TLV format.

- **Parameters:**
  - `int sockfd`: Socket file descriptor for sending data.
- **Returns:** None.

Function `play_game`:

Description: Handles the gameplay logic for a player identified by `nickname`. This function involves:

1. Generating a random word.
2. Sending the initial state of the word (underscores for letters) to the player.
3. Handling player guesses and updating the word state accordingly.
4. Timing the player's attempts and updating the ranking if necessary.
5. Sending announcements over multicast if a player achieves the fastest time.

- **Parameters:**
  - `int sockfd`: Socket file descriptor for communication with the player.
  - `int multicastfd`: Multicast socket file descriptor for sending announcements.
  - `char* nickname`: Player's nickname.
- **Returns:** None.

Function `game`:

Description: Manages the interaction with a player connected via `connfd`. This includes logging in the player, handling game choices (playing, viewing ranking, exiting), and invoking the appropriate functions (`play_game`, `view_ranking`, `user_exit`) based on player actions.

- **Parameters:**
  - `int connfd`: Socket file descriptor for communication with the player.
  - `int multicastfd`: Multicast socket file descriptor for sending announcements.
- **Returns:** None.

server/utils.c

Global variable `MAX_PLAYERS`:

Description: Maximum number of players that can be stored in the ranking.

Usage: Determines the maximum capacity of the ranking array (`Player players[MAX_PLAYERS+1]`). This variable ensures that the ranking does not exceed a specified number of entries.

Global variable `MAX_NICK_LENGTH`:

Description: Maximum length allowed for a player's nickname.

Usage: This constant defines the maximum number of characters that can be used for a player's nickname throughout the program.

Global variable `MAX_WORD_LENGTH`:

Description: Maximum length allowed for a word read from a file.

Usage: Specifies the maximum number of characters that can be read from a file when retrieving words for game purposes.

Global variable `RANKING_FILE`:

Description: Name of the file storing the game's ranking data.

Usage: Specifies the filename where the ranking data of players is stored.

Struct `Player`:

Description: Represents a player in the ranking with a nickname and the time taken in seconds.

- **Members:**
  - `char nickname[MAX_NICK_LENGTH]`: Array storing the player's nickname.
  - `double seconds`: Time taken by the player in seconds.

Function `count_lines`:

Description: Counts the number of lines in a specified file (`filename`).

- **Parameters:**
  - `const char *filename`: Name of the file to count lines in.
- **Returns:**
  - `int`: Number of lines in the file.
  - `-1` if there was an error opening the file or if no lines were counted.

Function `draw_word`:

Description: Randomly selects and retrieves a word from a file (`filename`). The word is stripped of any newline characters.

- **Parameters:**
  - `const char *filename`: Name of the file containing words.
- **Returns:**
  - `char *`: Pointer to a dynamically allocated string containing the randomly selected word.
  - `NULL` if there was an error opening the file or if no words were found.

Function `create_underscore_word`:

Description: Creates a new string where each character of the input `word` is replaced with an underscore ('_'). This is useful for initializing the display of a word in a game where letters are gradually revealed.

- **Parameters:**
  - `char *word`: Input word for which underscores will be created.
- **Returns:**
  - `char *`: Pointer to a dynamically allocated string containing underscores corresponding to each character in `word`, terminated with a null character.
  - `NULL` if there was an error allocating memory.

Function `check_letter`:

Description: Checks if a given `letter` exists in the `word`. If found, updates the corresponding position in `underscore_word` with the `letter`.

- **Parameters:**
  - `char letter`: The letter to check.
  - `char *underscore_word`: Pointer to a string representing the word with underscores.
  - `char *word`: Pointer to the original word.
- **Returns:**
  - `int`: Returns `1` if the `letter` was found in `word` and updated in `underscore_word`, otherwise `0`.

Function `check_ranking`:

Description: Checks if a player's `seconds` score qualifies for inclusion in the ranking. If the score is better than an existing entry or the ranking is not full (`MAX_PLAYERS`), updates the ranking file accordingly.

- **Parameters:**
  - `const char *nickname`: The nickname of the player.
  - `double seconds`: The time score in seconds achieved by the player.
- **Returns:**
  - `int`: Returns `2` if the player achieves a new first place in the ranking, `1` if the player beats an existing ranking entry, and `0` if the player's score does not qualify for inclusion.

Client

client/client.c

Struct `TLV` (Type-Length-Value):

Description: Represents messages in TLV (Type-Length-Value) format, commonly used for structured data representation.

- **Members:**
  - **`uint8_t type`:** Type of the TLV message.
  - **`uint16_t length`:** Length of the value in bytes.
  - **`void *value`:** Pointer to the value data.

Function `void snd_tlv(uint8_t type, uint16_t length, void *value, uint16_t *encoded_length, int socket)`:

Description: Sends a TLV-encoded message over a socket.

- **Parameters**:
  - `type`: The type of the message.
  - `length`: The length of the value.
  - `value`: The value to be sent.
  - `encoded_length`: Pointer to store the total length of the encoded message.
  - `socket`: The socket file descriptor to send the message through.
- **Returns**: None.

Function `void rcv_tlv(TLV *tlv, int socket)`:

Description: Receives a TLV-encoded message from a socket.

- **Parameters**:
  - `tlv`: Pointer to a TLV structure to store the received message.
  - `socket`: The socket file descriptor to receive the message from.
- **Returns**: None.

Function `int connect_to_hangman()`:

Description: Establishes a connection to the Hangman server.

- **Parameters**: None.
- **Returns**: The socket file descriptor for the connection.

Function `int connect_to_announcements()`:

Description: Establishes a connection to receive multicast announcements.

- **Parameters**: None.
- **Returns**: The socket file descriptor for the multicast connection.

Function `void get_nickname(int socket, char *nickname)`:

Description: Prompts the user to enter their nickname and sends it to the server using a TLV-encoded message. This function initializes the game by setting the player's nickname.

- **Parameters**:
  - `socket`: The socket file descriptor for the connection to the server.
  - `nickname`: Pointer to a character array where the entered nickname will be stored.
- **Returns**: None.

Function `int hangman(int socket)`:

Description: Manages the main game loop for the Hangman game. This function handles user inputs, sends guesses to the server, receives updates, and displays the game state. It continues until the player either wins, loses, or exits the game.

- **Parameters**:
  - `socket`: The socket file descriptor for the connection to the server.
- **Returns**: 0 if the game is won or lost, 1 if the player chooses to exit.

Function `void see_ranking(int socket)`:

Description: Requests and displays the ranking from the server. This function sends a TLV-encoded message to request the ranking and prints the received ranking data.

- **Parameters**:
  - `socket`: The socket file descriptor for the connection to the server.
- **Returns**: None.

Function `void clear_stdin()`:

Description: Clears the standard input buffer. This function is used to remove any unwanted characters from the input buffer to prevent them from being read by subsequent input operations.

- **Parameters**: None.
- **Returns**: None.

Function `void user_exit(int socket)`:

Description: Sends a request to the server to indicate that the user is exiting the game. This function sends a TLV-encoded message to inform the server of the user's intention to exit.

- **Parameters**:
  - `socket`: The socket file descriptor for the connection to the server.
- **Returns**: None.

Function `void menu(int socket, char *nickname)`:

Description: Displays the main menu and handles user choices. This function allows the user to choose between playing the game, viewing the ranking, or exiting. It calls the appropriate function based on the user's choice.

- **Parameters**:
  - `socket`: The socket file descriptor for the connection to the server.
  - `nickname`: The player's nickname.
- **Returns**: None.

Function `void read_announcment(int mtd)`:

Description: Reads and displays multicast announcements from the server. This function receives TLV-encoded messages from the multicast socket and prints the announcements.

- **Parameters**:
  - `mtd`: The socket file descriptor for the multicast connection.
- **Returns**: None.

Function `void comment_problem(int sig)`:

Description: Signal handler that prints a message indicating a problem with the comment section. This function is triggered by a specific signal and prints a predefined message.

- **Parameters**:
  - `sig`: The signal number triggering this handler.
- **Returns**: None.

Function `void clear_comment(int sig)`:

Description: Signal handler that clears the comment section on the screen. This function is triggered by a specific signal and clears the printed message.

- **Parameters**:
  - `sig`: The signal number triggering this handler.
- **Returns**: None.

Function `void sigchld_handler(int sig)`:

Description: Signal handler that reaps zombie processes. This function is triggered by the SIGCHLD signal and ensures that child processes are properly cleaned up to prevent them from becoming zombies.

- **Parameters**:
  - `sig`: The signal number triggering this handler.
- **Returns**: None.

Function `int main(int argc, char **argv)`:

Description: The main function of the client application. It sets up signal handlers, forks a process to handle multicast announcements, connects to the Hangman server, and manages the main menu loop.

- **Parameters**:
  - `argc`: Number of command-line arguments.
  - `argv`: Array of command-line arguments.
- **Returns**: Exit status of the program.