```
/****//**
      @file       main.c
      @author           Stephen Brennan
      @date       Thursday, 8 January 2015
      @brief            LSH (Libstephen Shell
********/

/****//**

      Ian Sanchez Munoz
      005824893
      October 3, 2023
      Purpose: This program will simulate a Linux shell and support a number of
built-in and external services.

********/

#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define INPUT_LENGTH 1024
#define MAX_ALIASES 10

// Function declarations for builtin commands:
int ms_cd(char **args);
int ms_help(char **args);
int ms_stop(char **args);
int ms_setshellname(char **args);
int ms_setterminator(char **args);
int ms_newname(char **args);
int ms_listnewnames(char **args);
int ms_savenewnames(char **args);
int ms_readnewnames(char **args);

char shellName[] = "myshell";
char terminator[] = ">";
char aliases[MAX_ALIASES][2][INPUT_LENGTH];

// List of built-in commands followed by their corresponding functions.
char *builtin_str[] = {
      "cd",
      "help",
      "stop",
      "setshellname",
      "setterminator",
      "newname",
      "listnewnames",
      "savenewnames",
      "readnewnames"
};

int (*builtin_func[]) (char **) = {
      &ms_cd,
      &ms_help,
      &ms_stop,
```

```
      &ms_setshellname,
      &ms_setterminator,
      &ms_newname,
      &ms_listnewnames,
      &ms_savenewnames,
      &ms_readnewnames
};

// Purpose: Returns the number of built-in functions
// @return Integer representing count of built-in functions
int num_builtins()
{
      return sizeof(builtin_str) / sizeof(char *);
}

// Built-in function implementations

// Purpose: Built-in command: change directory
// @param args List of arguments
// @return Returns 1 to continue executing
int ms_cd(char **args)
{
      if (args[1] == NULL){
            fprintf(stderr, "myshell: expected argument to \"cd\"\n");
      } else {
            if (chdir(args[1]) != 0){
                  perror("myshell");
            }
      }

      return 1;
}

// Purpose: Built-in command: print help
// @param args List of arguments
// @return Returns 1 to continue executing
int ms_help(char **args)
{
      int i;
      printf("Ian Sanchez Munoz's Toy Shell\n");
      printf("Type program names and arguments, and hit enter.\n");
      printf("The following are built in:\n");

      for (i = 0; i < num_builtins(); i++){
            printf("  %s\n", builtin_str[i]);
      }

      printf("Use the man command for information on other programs.\n");

      return 1;
}

// Purpose: Built-in command: exit
// @param args List of arguments
// @return Always return 0 to stop execution
int ms_stop(char **args)
{
      return 0;
}
```

```
// Purpose: Built-in command: setshellname (changes the name of the shell)
// @param args List of arguments
// @return Always return 1 to continue execution
int ms_setshellname(char **args)
{
      if (args[1] != NULL){
            strcpy(shellName, args[1]);
      } else{
            printf("(error) %s: please provide a name for the shell\n", shellName);
      }

      return 1;
}

// Purpose: Built-in command: setterminator (changes the terminator operator for
the current shell session)
// @param args List of arguments
// @return Always return 1 to continue execution
int ms_setterminator(char **args)
{
      if (args[1] != NULL){
            strcpy(terminator, args[1]);
      } else{
            printf("(error) %s: please provide a terminator for the shell\n",
shellName);
      }

      return 1;
}

// Purpose: Built-in command: newname (Creates a new alias for a command)
// @param args List of arguments
// @return Always return 1 to continue execution
int ms_newname(char **args)
{
      if (args[1] == NULL){
            printf("(error): %s: please provide a name for the alias\n",
shellName);
            return 1;
      }

      if (args[2] == NULL){
            int found = 0;
            for (int i = 0; i < MAX_ALIASES; i++){
                  if (strcmp(aliases[i][0], args[1]) == 0){
                        aliases[i][0][0] = '\0'; // Mark alias as empty
                        found = 1;
                        break;
                  }
            }
            if (!found) {
                  printf("(error): %s: this alias does not exist\n", shellName);
            }
      } else {
            int replaced = 0;
            for (int i = 0; i < MAX_ALIASES; i++) {
                  if (aliases[i][0][0] == '\0'){
                        // Empty index
```

```
                        strcpy(aliases[i][0], args[1]);
                        strcpy(aliases[i][1], args[2]);
                        replaced = 1;
                        break;
                  } else if (strcmp(aliases[i][0], args[1]) == 0){
                        // Replace alias
                        strcpy(aliases[i][1], args[2]);
                        replaced = 1;
                        break;
                  }
            }
            if (!replaced) {
                  printf("(error) %s: maximum amount of aliases reached\n",
shellName);
            }
      }

      return 1;
}

// Purpose: Built-in command: listnewnames (List all of the current aliases in this
shell session)
// @param args List of arguments
// @return Always return 1 to continue execution
int ms_listnewnames(char **args)
{
      int some = 0;
      for (int i = 0; i < MAX_ALIASES; i++){
            if (aliases[i][0][0] != '\0'){
                  printf("%s %s\n", aliases[i][0], aliases[i][1]);
                  some = 1;
            }
      }
      if (!some) {
            printf("(error) %s: no entries in the alias list\n", shellName);
      }

      return 1;
}

// Purpose: Built-in command: savenewnames (Saves all of the current aliases in
this shell session to a file)
// @param args List of arguments
// @return Always return 1 to continue execution
int ms_savenewnames(char **args)
{
      if (args[1] == NULL){
            printf("(error) %s: please enter the name of a file\n", shellName);
            return 1;
      }

      FILE* f = fopen(args[1], "w");
      if (f != NULL){
            for (int i = 0; i < MAX_ALIASES; i++){
                  if (aliases[i][0][0] != '\0'){
                        fprintf(f, "%s %s\n", aliases[i][0], aliases[i][1]);
                  }
            }
            fclose(f);
```

```
	} else {
		printf("(error) %s: unable to open file\n", shellName);
	}

	return 1;
}

// Purpose: Built-in command: readnewnames (Reads all of the aliases in the
provided file)
// @param args List of arguments
// @return Always return 1 to continue execution
int ms_readnewnames(char **args)
{
	if (args[1] == NULL){
		printf("(error) %s: please enter the name of a file\n", shellName);
		return 1;
	}

	FILE* f = fopen(args[1], "r");
	if (f != NULL){
		int i = 1;
		while (fscanf(f, "%s %s", aliases[i][0], aliases[i][1]) != EOF){
			i++;
		}
		fclose(f);
	} else {
		printf("(error) %s: unable to open file\n", shellName);
	}

	return 1;
}

// Purpose: Launch a program and wait for it to terminate
// @param args Null terminated list of arguments
// @return Always return 1 to continue execution
int ms_launch(char **args)
{
	pid_t pid;
	int status;

	pid = fork();
	if (pid == 0){
		// Child process
		if (execvp(args[0], args) == -1)
		{
			perror("myshell");
		}
		exit(EXIT_FAILURE);
	}
	else if (pid < 0){
		// Error forking
		perror("myshell");
	}
	else{
		// Parent process
		do {
			waitpid(pid, &status, WUNTRACED);
		} while (!WIFEXITED(status) && !WIFSIGNALED(status));
	}
```

```
      return 1;
}

// Purpose: Execute shell built-in or launch program
// @param args List of arguments
// @return 1 if the shell should continue running, or 0 if it should terminate
int ms_execute(char **args)
{
      int i;

      if (args[0] == NULL){
            // Empty command
            return 1;
      }

      for (int i = 0; i < MAX_ALIASES; i++){
            if (strcmp(args[0], aliases[i][0]) == 0) {
                  // Found an alias
                  // Replace args[0] with aliases[i][1]
                  strcpy(args[0], aliases[i][1]);
                  break;
            }
      }

      for (i = 0; i < num_builtins(); i++){
            if (strcmp(args[0], builtin_str[i]) == 0){
                  return (*builtin_func[i])(args);
            }
      }


      return ms_launch(args);
}


// Purpose: Read a line of input froms stdin
// @return The line from stdin
char *read_line(void)
{
      #ifdef MS_USE_STD_GETLINE
       char *line = NULL;
       ssize_t bufsize = 0; // have getline allocate a buffer for us
       if (getline(&line, &bufsize, stdin) == -1) {
            if (feof(stdin)){
                        exit(EXIT_SUCCESS);  // We received an EOF
            } else {
                        perror("myshell: getline\n");
                  exit(EXIT_FAILURE);
            }
       }
       return line;
      #else
      #define MS_RL_BUFSIZE 1024
       int bufsize = MS_RL_BUFSIZE;
       int position = 0;
       char *buffer = malloc(sizeof(char) * bufsize);
       int c;
```

```
       if (!buffer){
            fprintf(stderr, "myshell: allocation error\n");
            exit(EXIT_FAILURE);
       }

       while (1) {
            // Read a character
            c = getchar();

            if (c == EOF) {
                  exit(EXIT_SUCCESS);
            } else if (c == '\n') {
                        buffer[position] = '\0';
                        return buffer;
            } else {
                        buffer[position] = c;
            }
            position++;

             // If we have exceeded the buffer, reallocate.
            if (position >= bufsize) {
                        bufsize += MS_RL_BUFSIZE;
                        buffer = realloc(buffer, bufsize);
                        if (!buffer) {
                        fprintf(stderr, "myshell: allocation error\n");
                        exit(EXIT_FAILURE);
                        }
            }
       }
      #endif
}

#define MS_TOK_BUFSIZE 64
#define MS_TOK_DELIM " \t\r\n\a"
/**
      Purpose: Split a line into tokens (very naively).
      @param line The line.
      @return Null-terminated array of tokens.
 */
char **split_line(char *line)
{
      int bufsize = MS_TOK_BUFSIZE, position = 0;
      char **tokens = malloc(bufsize * sizeof(char*));
      char *token, **tokens_backup;

      if (!tokens) {
            fprintf(stderr, "myshell: allocation error\n");
            exit(EXIT_FAILURE);
      }

      token = strtok(line, MS_TOK_DELIM);
      while (token != NULL) {
            tokens[position] = token;
            position++;

            if (position >= bufsize) {
                        bufsize += MS_TOK_BUFSIZE;
                        tokens_backup = tokens;
                  tokens = realloc(tokens, bufsize * sizeof(char*));
```

```
                if (!tokens) {
                        free(tokens_backup);
                        fprintf(stderr, "myshell: allocation error\n");
                        exit(EXIT_FAILURE);
                }
            }

            token = strtok(NULL, MS_TOK_DELIM);
        }

        tokens[position] = NULL;
        return tokens;
}

// Purpose: Loop getting input and executing it
void loop(void)
{
        char *line;
        char **args;
        int status;

        do {
            // printf("> ");
            printf("%s%s ", shellName, terminator);
            line = read_line();
            args = split_line(line);
            status = ms_execute(args);

            free(line);
            free(args);
        } while (status);
}

// Purpose: Main entry point
// @param argc Argument count
// @param argv Argument vector
// @return Status code
int main(int argc, char **argv)
{
        // Load config files, if any

        // Run command loop
        loop();

        // Perform any shutdown / cleanup.

        return EXIT_SUCCESS;
}
```