CS4323 Assignment II Final Report

Group I

Collin Thornton - Ethan Vascellaro - Kazi Sharif - Caleb Goodart

Table of Contents

Information

Class: CS4323 Introduction to Operating Systems
Assignment: Assignment 2
Group: Group I
Members:

- Collin Thornton - A1173381
- Ethan Vascellaro -
- Kazi Sharif -
- Caleb Goodart -

Project Overview

Assignment 2 involves the emulation of the Bash shell in the C programming language utilizing a client-server process structure. It is divided into 5 distinct tasks:

1. Informational
2. Informational
3. Basic shell interface
4. Basic shell commands
5. Shell commands in background
6. History
7. Client-Server interface

The shell interface runs entirely on the server and is the only process allowed access to stdin and stdout. Basic shell commands include commands on the PATH, ‘cd’, ‘help’, ‘history’, ‘exit’, and ‘jobs’. These are to be executed on the server process. Commands should be executed in the background when ‘&’ is appended as the last character in the command. The server should track all commands entered and maintain a function to display the history.

Work Distribution

Initial

Caleb:	Part 6	Command history
Collin:	Part 5	Shell commands in background
Ethan:	Part 7	Client-server communication

Kazi:	Part 3 & 4	Shell interface and command execution

Final

Collin:	Parts 3-7	Sole contributor to Parts 3, 4, 5
Ethan:	Part 7	**Undergoing urgent family issues*
Kazi:	Part 6	
Caleb:	**NO CONTRIBUTION**	

Design Goals

Our high-level design goals focused on code structure such that integration might flow smoothly. To this end, we created process.h/c and msg.h/c to simplify communications between processes, threads, and functions. Each of these files abstracts a commonly used feature into a simplistic set of functions. Figure 1 depicts the general structure of our code. The Msg datatype server as an intermediary between the client and server. It contains all information necessary for the processes to communicate.

The code is divided into multiple distinct sections: client, server, sockets, and helper functions. This allowed for efficient work division, however the original work distribution did not stay in effect for long.

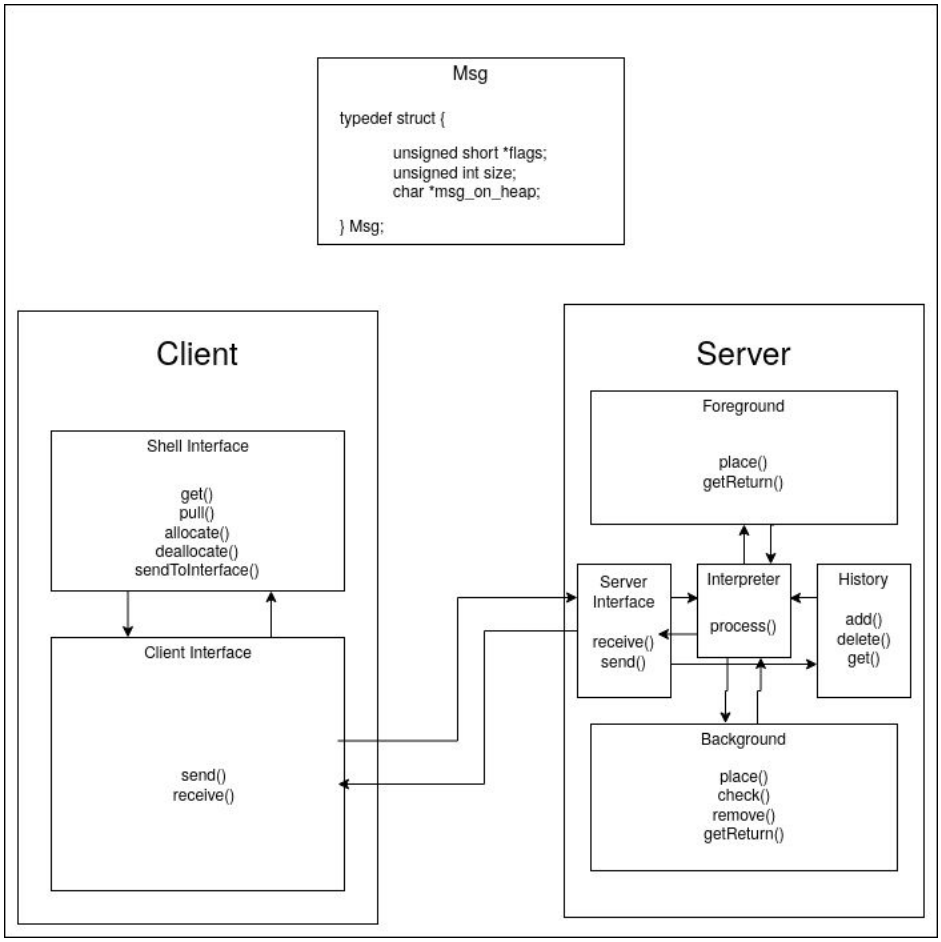

Figure 1: Block diagram of design

Individual Work Completed

This section outlines the work completed by individual members of the group.

Caleb:

No contribution.

Collin:

In the first week of the project, Ethan and I designed the hierarchy of the project and communication lanes between the sections. After this, I created the block diagram, set up a GitHub repository, added a branch for each member, and organized most meetings. As

evidenced in Canvas, I provided links to resources/tutorials on the utilization of git. Furthermore, I offered to push others' code to the branch myself.

My original section of the project was limited to Part 5, which I completed immediately following the Progress Report submission. After this I create the main, client, and server files such that the other could begin working with them on their branches. Soon after this, it became apparent I needed to produce more code as no one else had pushed code to the git repository.

After discussion with Kazi, we reallocated his work to Part 6 and I assumed Parts 3 and 4. Kazi had a basic framework setup, though it did not comply with the project requirements and was largely based on external sources. I removed third-party code, made it compatible with socket communications, and multithreaded it such that I/O operations are predominately nonblocking. This satisfied Part 3.

I then moved to the command interpreter, such that Part 4 might be completed. I implemented multiping, command execution with execvp(), and created a list of custom commands that are not found on the PATH. I began work with I/O redirection, but subsequently removed the progress after running out of time while debugging.

Ethan ran into family issues about a week before the project was due. He sent me what he had for socket communications and I assumed Part 7 thereafter. I created the Msg classes to streamline communications, made clientServerEV.h more robust, and created the acknowledgement algorithm between the server and client.

Barring the first meeting, Caleb never communicated with the group nor did he contribute code. Kazi submitted Part 6 the night of the project deadline. It was a good framework, though it only stored the most recent 10 commands, in reverse order, and as such did not fulfill the requirements of the project. Due to this, I extended the ProcessList struct (originally designed for tracking background processes) to track the full command history.

In sum, I wrote every file but clientServerEV.c. This file was originally written by Ethan and heavily modified by me. Kazi provided a framework for the shell interface and command history, though the shell interface was dependent upon the installation of third-party software.

Ethan:

Basic socket client-server communication designed Designed hierarchy of the project, as well as communication lanes between the sections

Kazi:

Designed layout for the shell interface, including startup, user input, directory location, and command execution. Design storage for previous commands to act as a reference for history

Technical Details

Testing Environment

All code was tested on the CSX machines with valgrind to verify memory performance. To our best knowledge, the code is memory-safe barring a few small memory leaks. The leaks are not large enough to be noticeable without a tool such as valgrind. More details are given in a subsequent subsection.

Work Flow

General sequence

1. Perform setup
2. Command delivered on client stdin -> `ls -al | grep -i b | wc -c`
    a. Detected on inputThread()
    b. Allocated to global buffer
3. Client loop updates, detects update buffer
    a. Allocate Msg struct
    b. Copy buffer to msg.cmd
    c. Serializes msg
    d. Write to socket
4. ClientServerEV sockReadThread detects update
    a. Running multithreaded on server process
    b. Allocates msg to global buffer
5. Server loop updates, detects updated ClientServerEV buffer
    a. Deserializes msg
    b. Checks for empy msg
    c. Adds to history
    d. Send empty response to client
        i. Msg.show_prompt TRUE if in background
        ii. Msg.show_prompt FALSE if in foreground
    e. Create server_command_interpreter thread
6. Server_command_interpreter
    a. Break command into jobs (specific processes) with the pipe symbol as delimiter
        i. At least 2 pipes (stdin -> process) and (stdout -> buffer)
    b. Allocate each job as a Process struct
        i. Separates executable name from arguments
    c. Decide whether to execute in background
        i. Determines whether thread should wait(NULL) or not

   d. Iterate through jobs, execute as custom command or execvp()
      i. Custom commands include ‘exit’, ‘cd’, ‘history’, ‘jobs’, ‘help’
      ii. Execvp() automatically searches path
      iii. Pass Process.exec as filename
      iv. Pass Process.args as process arguments
7. Process returns, allocate Msg resp and set flag
   a. Server loop detects flag, serializes, and transmits to socket
8. Client receives Msg
   a. Deserialize and display msg.ret on stdout

Ethan

clientServerEV

Int socket_init(void)

- **Summary:** Initialize the server-side socket connection. Creates, binds, and listens for connections. Assigns file descriptors and buffers as global variables in clientServerEV.c.
- **Input:** void
- **Return:** unused

Int socket_write(Msg *msg)

- **Summary:** Serialize and write a Msg struct to the socket
- **Input:** msg (Msg*) Pointer to a Msg struct
- **Return:** unused

Collin

Main

Int main(int argc, char** argv)

- **Summary:** Fork two child processes. 1st executes server. 2nd sleeps 1 second, then executes client. The delay allows time for the server to initialize the socket.
- **Input**: Unused
- **Return:** Unused

Client

int client(void);

- **brief** Run the client process
- **return** (int) return code

void *inputThread(void *vargp);

- **brief** Threaded function for nonblocking stdin
- **param** vargp (void*) unused (NULL)

void *socketReadThread(void *vargp);

- **brief** Threaded function for nonblocking socket input
- **param** vargp (void*) unused (NULL)

Server

int server(void);
- **brief** run server process
- **return** (int) return code

void* server_command_interpreter(void* vargp);
- **brief** threaded function to interpret commands from client
- **param** vargp (void* -> Msg*) pointer to Msg structure
- **return** unused

void redirect(int fdfrom, int fdto);
- **brief** redirect filedescriptors
- **param** fdfrom (int) original file descriptor
- **param** fdto (int) new file descriptor

void run(Process *proc);
- **brief** execute a command with execvp()
- **param** proc (Process*) process to be executed

void run_cmd_list(Process *proc, char* outbuff);
- **brief** execute a custom command
- **param** proc (Process*) process to be executed
- **param** outbuff (char*) output buffer (eventually send to client)

bool in_cmd_list(Process *proc);
- **brief** check if process is in custom command list
- **param** proc (Process*) process to be checked
- **return** (bool) true if process found. else false.

void init_shutdown(void);
- **brief** set global flag to trigger shutdown of server

Msg

char* msg_serialize(Msg *msg, char *buff);
- **brief** Convert Msg to string for socket comms. Should be returned to Msg by msg_deserialize()
- **param** msg (Msg*) Msg to be converted
- **param** buff (char*) String buffer
- **Return** (char*) pointer to buff

Msg* msg_deserialize(const char* str);

- **brief** Conver string to Msg for socket comms. Should be preceded by msg_serialize()
- **param** str (const char*) String containing Msg information
- **return** (Msg*) msg struct allocated on heap

Kazi:

Multi-pipe command execution. Then I shared the problems with groupmates and
Then it is solved by setting it proper handling of pipes which will work for the build-in comments and other multiple kind of comments.

Incomplete Components

Memory cleanup

- Testing with valgrind shows that not all memory is freed
- Ran out of time debugging

Redirection

Needed mechanisms

- **Redirection detection** with string processing
  - if(strstr(buff, “>>”) != NULL)
  - Else if(strstr(buff, “<<”) != NULL)
  - Else if(strstr(buff, “>”) != NULL)
  - Etc
- **Redirection operation** with dup2() and helper functions
  - See redirect(int from_fd, int to_fd) in server.c
  - This was designed with pipes in mind, though could easily be used with files
  - Redirect to file instead of pipe
- Had most of these in place, though ran out of time during debugging. Changes made program astable and thus were not pushed to the master branch.

Issues And Solutions

1. Members not showing up
   a. Split work among whoever finishes their part first
   b. Otherwise, split evenly
2. Members plagiarizing
   a. Performed thorough review of everyone’s code against Google search
   b. Rewrote questionable sections

3. Unclear parts of assignment
    a. Messaged professor
4. Unclear how to approach problem individually
    a. Have group discussions to tackle any issues
    b. **Kazi** -> Lots of error found in the middle of the project creating shell command and then all colorabations of meeting such as setting multiple piping and building the connection of each server to the clients. Then discussion with Collin in the online meeting and library meetup works very well in the end to build the whole program.

Appendix A - Ethan's Code

clientServerEV.c

```
int PORT = 8081;
int ssockfd, connfd;
pthread_t tid = 0;

bool sock_exit_flag = false;
char sock_input[SOCKET_BUFF];
int sock_queue_len = 0;


/**
 * @brief Initialize the socket
 * @return (int) return code
 */
int socket_init(void) {
	int len;
	struct sockaddr_in servaddr, cli;

	// Create and verify socket
	ssockfd = socket(AF_INET, SOCK_STREAM, 0);
	if (ssockfd == -1) {
	printf("Socket Failed\n");
	exit(0);
	}

	// Clear servaddr
	bzero(&servaddr, sizeof(servaddr));

	// IP and PORT
	servaddr.sin_family = AF_INET;
	servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servaddr.sin_port = htons(PORT);

	// Binding socket
	if ((bind(ssockfd, (sockA*)&servaddr, sizeof(servaddr))) != 0) {
	printf("Bind Failed\n");
	exit(0);
	}
```

```
// Listening
if ((listen(ssockfd, 10)) != 0) {
printf("Listen Failed\n");
exit(0);
}

len = sizeof(cli);

// Accept client and begin chat
connfd = accept(ssockfd, (sockA*)&cli, &len);
if (connfd < 0) {
printf("Accept Failed\n");
exit(0);
}
```

Appendix B - Kazi's Code

Appendix C - Collin's Code

Main

```
// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 main
//   Date          -   10-14-20
//
// ##########################################

#define VERBOSE

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

```
#include <sys/wait.h>

#include "../include/client.h"
#include "../include/server.h"

void test(void);

//#define EXEC_MAIN_TEST


#ifdef EXEC_MAIN_TEST
#include <string.h>
#endif // EXEC_MAIN_TEST


int main(int argc, char** argv) {
        #ifdef EXEC_MAIN_TEST
        test();
        return 0;
        #endif // EXEC_MAIN_TEST

        pid_t pids[2] = { 0, 0 };
        for(int i=0; i<2; ++i) {
        if((pids[i] = fork()) < 0) {        // SPAWN PROCESSES
        perror("Fork failed");
        }

        if(pids[0] == 0) {
        int server_ret = server();      // EXECUTE SERVER PROCESS
        //execl("sock", "sock", NULL);
        exit(server_ret);
        }
        if(pids[1] == 0) {
        sleep(1);
        int client_ret = client();          // EXECUTE CLIENT PROCESS
        exit(client_ret);
        }
        }

        for(int i=0; i<2; ++i) wait(NULL);      // WAIT FOR BOTH TO EXIT

        exit(0);                            // EXIT
}
```

```
#ifdef EXEC_MAIN_TEST
void test(void) {
	char test1[100] = "this is";
	char test2[100] = "this | is";

	printf("\r\ntmp1\r\n");
	char *tmp1 = strtok(test1, "|");
	do {
	printf("%s\r\n", tmp1);
	tmp1 = strtok(NULL, "|");
	} while(tmp1 != NULL);

	printf("\r\ntmp2\r\n");
	char *tmp2 = strtok(test2, "|");
	do {
	printf("%s\r\n", tmp2);
	tmp2 = strtok(NULL, "|");
	} while(tmp2 != NULL);
}
#endif // EXEC_MAIN_TEST
```

Client

Client.h

```
// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 Client include
//   Date          -   10-27-20
//
// ##########################################

#ifndef CLIENT_H
#define CLIENT_H


/**
 * @brief Run the client process
```

```
 * @return (int) return code
 */
int client(void);

/**
 * @brief Manage input on stdin. Multithreaded for nonblocking.
 * @param vargp (void*) unused (NULL)
 */
void *inputThread(void *vargp);

/**
 * @brief Manage input from socket-server. Multithreaded for nonblocking.
 * @param vargp (void*) unused (NULL)
 */
void *socketReadThread(void *vargp);

#endif // CLIENT_H
```

Client.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdbool.h>
#include <pthread.h>
#include <time.h>

#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

#define sockA struct sockaddr
#define SOCKET_BUFF 5000

#include "../include/client.h"
#include "../include/msg.h"

//#define EXEC_CLIENT

bool exit_flag = false;             // FLAG TO CONTINUE EXECUTING

pthread_t sock_tid, stdin_tid;              // THREAD IDs
```

```
int sockfd;                     // FILE DESCRIPTOR FOR SOCKET

char stdin_input[1024];         // BUFFER FOR stdin
bool stdin_input_read = false;  // FLAG FOR NEW MESSAGE ON stdin

char socket_input[SOCKET_BUFF];     // BUFFER FOR SOCKET
int socket_queue_len = 0;           // FLAG FOR NEW MESSAGE ON socket


/**
 * @brief Run the client process
 * @return (int) return code
 */
int client(void) {

        // BEGIN SETUP

        #ifdef VERBOSE
        printf("|----- CLIENT:\t%d, %d\r\n", getppid(), getpid());
        #endif // VERBOSE

        bzero(socket_input, sizeof(socket_input));
        bzero(stdin_input, sizeof(stdin_input));

        // SETUP SOCKET
        const int PORT = 8081;
        struct sockaddr_in servaddr, cli;

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if(sockfd == -1) {
        perror("client: socket()");
        exit(1);
        }

        struct timeval timeout;
        timeout.tv_sec = 3;
        timeout.tv_usec = 0;

        if(setsockopt(sockfd, SOL_SOCKET, SO_SNDTIMEO, (char*)&timeout, sizeof(timeout))
< 0) {
        perror("client: setsockopt");
        exit(1);
        }
```

```
bzero(&servaddr, sizeof(servaddr));

servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

if(connect(sockfd, (sockA*)&servaddr, sizeof(servaddr)) != 0) {
perror("client: connect()");
exit(1);
}


// SETUP SHELL INTERFACE

const char *user = getenv("USER");

char home_dir[256];
strcpy(home_dir, "/home/");
strcat(home_dir, user);


// LAUNCH THREADS FOR stdin AND socket

pthread_create(&stdin_tid, NULL, inputThread, NULL);
pthread_create(&sock_tid, NULL, socketReadThread, NULL);


// DISPLAY ASSIGNMENT INFORMATION

printf("CS4323 ASSIGNMENT II GROUP I\r\n");
printf("C SHELL EMULATOR\r\n\r\n");
printf("Collin Thornton\r\nEthan Vascellaro\r\nKazi Sharif\r\nCaleb Goodart\r\n\r\n");
printf("HELP DISPLAY:\r\n\r\n");


// SEND INITIAL COMMAND TO DISPLAY help

Msg *init_msg;
init_msg = msg_allocate("help", NULL, NULL);
char init_msg_buff[SOCKET_BUFF];
bzero(init_msg_buff, sizeof(init_msg_buff));
msg_serialize(init_msg, init_msg_buff);

msg_deallocate(init_msg);
```

```
if(write(sockfd, init_msg_buff, SOCKET_BUFF) < 0) {
perror("client: write()");
exit(1);
}


while(1) {

// HANDLE STDIN
if(stdin_input_read == true) {
stdin_input_read = false;

// REMOVE NEW LINE CHARACTER
if(stdin_input[strlen(stdin_input)-1] == '\n') stdin_input[strlen(stdin_input)-1] = '\0';

// ALLOCATE MSG
Msg *msg = msg_allocate(stdin_input, NULL, NULL);

// SERIALIZE MSG
char msg_buff[SOCKET_BUFF];
bzero(msg_buff, SOCKET_BUFF);
msg_serialize(msg, msg_buff);

// WRITE TO SOCKET
if(write(sockfd, msg_buff, SOCKET_BUFF) < 0) {
        perror("client: write()");
        exit(1);
}

// DEALLOCATE MSG
msg_deallocate(msg);

// QUIT IF q RECEIVED
if(strcmp(stdin_input, "exit") == 0) break;
}

// HANDLE SOCKET
if(socket_queue_len > 0) {
--socket_queue_len;

// DESERIALZE MESSAGE
Msg *msg = msg_deserialize(socket_input);
printf("%s", msg->ret);
```

```
        fflush(stdout);

        // SHOW PROMPT IF DESIRED
        if(msg->show_prompt) {
                if(strcmp(msg->dir, home_dir) == 0) {
                printf("%s@CS4323shell:%s~$ ", user, msg->dir);
                } else {
                printf("%s@CS4323shell:%s$ ", user, msg->dir);
                }
                fflush(stdout);
        }

        // DEALLOCATE MESSAGE
        msg_deallocate(msg);
        }


        // Sleep for 0.01 second
        struct timespec ts;
        ts.tv_sec  = 1E-2;
        ts.tv_nsec = 1E7;
        nanosleep(&ts, &ts);
        }

        close(sockfd);
        exit_flag = true;
        pthread_join(stdin_tid, NULL);
        pthread_join(sock_tid, NULL);
        printf("Goodbye\r\n");

        return 0;
}



/**
 * @brief Manage input on stdin. Multithreaded for nonblocking.
 * @param vargp (void*) unused (NULL)
 */
void *inputThread(void *vargp) {
        while(!exit_flag) {
        char *ret = fgets(stdin_input, sizeof(stdin_input), stdin);
        if(ret == NULL) strcpy(stdin_input, "eof\n");
        stdin_input_read = true;
```

```
        struct timespec ts;
        ts.tv_sec  = 1E-2;
        ts.tv_nsec = 1E7;
        nanosleep(&ts, &ts);
        }
}

/**
 * @brief Manage input from socket-server. Multithreaded for nonblocking.
 * @param vargp (void*) unused (NULL)
 */
void *socketReadThread(void *vargp) {
        while(!exit_flag) {
        char buff[SOCKET_BUFF];
        bzero(buff, SOCKET_BUFF);

        if(read(sockfd, buff, sizeof(buff)) == -1) {
        perror("client:  socket closed");
        exit(1);
        }
        strcpy(socket_input, buff);

        ++socket_queue_len;
        }
}


#ifdef EXEC_CLIENT
int main(int argc, char** argv) {
        client();
        return 0;
}
#endif // EXEC_CLIENT
```

Server

Server.h

```
// ##############################################
//
//   Author  -   Collin Thornton, Kazi Sherif
```

```
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 msg struct header
//   Date          -   10-27-20
//
// ##########################################

#ifndef SERVER_H
#define SERVER_H

#include "msg.h"
#include "process.h"

/**
 * @brief run server process
 * @return (int) return code
 */
int server(void);



/**
 * @brief threaded function to interpret commands from client
 * @param vargp (void* -> Msg*) pointer to Msg structure
 * @return unused
 */
void* server_command_interpreter(void* vargp);

/**
 * @brief redirect filedescriptors
 * @param fdfrom (int) original file descriptor
 * @param fdto (int) new file descriptor
 */
void redirect(int fdfrom, int fdto);

/**
 * @brief execute a command with execvp()
 * @param proc (Process*) process to be executed
 */
void run(Process *proc);

/**
 * @brief execute a custom command
 * @param proc (Process*) process to be executed
 * @param outbuff (char*) output buffer (eventually send to client)
```

```
 */
void run_cmd_list(Process *proc, char* outbuff);

/**
 * @brief check if process is in custom command list
 * @param proc (Process*) process to be checked
 * @return (bool) true if process found. else false.
 */
bool in_cmd_list(Process *proc);

/**
 * @brief set flag to trigger shutdown of server
 */
void init_shutdown(void);

#endif // SERVER_H
```

Server.c

```
// ##########################################
//
//   Author  -   Collin Thornton, Kazi Sherif
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 msg struct header
//   Date          -   10-27-20
//
// ##########################################

#include <time.h>

#include "../include/server.h"
#include "../include/background.h"
#include "../include/clientServerEV.h"

//#define EXEC_SERVER

bool loop = true;                   // FLAG TO CONTINUE IN WHILE LOOP
bool in_foreground = false;             // FLAG IF FOREGROUND CURRENTLY OCCUPIED

Msg *resp;                          // RESPONSE FROM COMMAND_INTERPRETER

ProcessList background_list, history;   // LINKED LISTS OF PROCESSES

int foreground_stdin[2];                // PIPE TO COMMAND IN FOREGROUND
```

```
int stdin_bak;          // BACKUP OF stdin FILE DESCRIPTOR
int stderr_bak;         // "
int stdout_bak;         // "


/**
 * @brief run server process
 * @return (int) return code
 */
int server(void) {

        // BEGIN SETUP

        pthread_t cmd_int;
        Msg *msg = NULL;

        stdin_bak  = dup(STDIN_FILENO);
        stderr_bak = dup(STDERR_FILENO);
        stdout_bak = dup(STDOUT_FILENO);

        #ifdef VERBOSE
        printf("|----- SERVER:\t%d, %d\r\n", getppid(), getpid());
        #endif // VERBOSE

        msg = NULL;

        #ifndef EXEC_SERVER
        socket_init();
        #endif // EXEC_SERVER

        process_list_init(&background_list, NULL, NULL);    // LINKED LIST FOR
BACKGROUND
        process_list_init(&history, NULL, NULL);             // LINKED LIST FOR HISTORY


        // BEGIN LOOP

        while(loop) {
        #ifdef EXEC_SERVER
        char dir_str[1024];
        getcwd(dir_str, sizeof(dir_str));
        printf("%s$> ", dir_str);
        fflush(stdout);
```

```
char input_buff[1024];
bzero(input_buff, sizeof(input_buff));

fgets(input_buff, sizeof(input_buff), stdin);
input_buff[strlen(input_buff)-1] = '\0';

msg = msg_allocate(input_buff, NULL, NULL);
#else
msg = socket_read();
#endif // EXEC_SERVER


// IF MESSAGE RECEIVED
if(msg != NULL ) {

// ADD TO HISTORY
Process history_node;
process_init(&history_node, msg);
strcpy(history_node.exec, msg->cmd);
process_list_add_node(&history, &history_node);
process_rem(&history_node);


// MAKE SURE IT'S REAL
bool only_whitespace = true;

for(int i=strlen(msg->cmd)-1; i>=0; --i) {
        if(msg->cmd[i] == ' ' || msg->cmd[i] == '\t') msg->cmd[i] = '\0';
        else {
        only_whitespace = false;
        break;
        }
}

char dir[500];
bzero(dir, sizeof(dir));
getcwd(dir, sizeof(dir));

// IF THE MESSAGE WAS MISSENT
if(only_whitespace) {
        // ACKNOWLEDGE RECEIPT

        resp = msg_allocate(msg->cmd, "\0", dir);
```

```
            resp->show_prompt = true;

            // SEND RESPONSE

            #ifndef EXEC_SERVER
            socket_write(resp);
            #else
            msg_deallocate(resp);
            #endif // EXEC_SERVER

            resp = NULL;
            msg_deallocate(msg);
        }

        // IF THERE'S CURRENTLY A PROCESS EXECUTING IN FOREGROUND
        else if(in_foreground) {
            // SETUP PIPE TO EXECUTING PROCESS
            strcat(msg->cmd, "\n");
            close(foreground_stdin[0]);
            write(foreground_stdin[1], msg->cmd, sizeof(msg->cmd));
            if(strcmp(msg->cmd, "eof\n") == 0) {
            close(foreground_stdin[1]);
            }
            msg_deallocate(msg);
        }
        else {
            // CHECK IF THE MESSAGE SHOULD RUN IN BACKGROUND
            bool background = (msg->cmd[strlen(msg->cmd)-1] == '&') ? true : false;

            // ALLOCATE ACKNOLWEDGEMENT
            resp = msg_allocate(msg->cmd, "\0", dir);
            resp->show_prompt = background;

            // SEND ACKOWLEDGEMENT
            #ifndef EXEC_SERVER
            socket_write(resp);
            #else
            msg_deallocate(resp);
            #endif // EXEC_SERVER

            resp = NULL;

            // PROCESS MESSAGE
            pthread_create(&cmd_int, NULL, server_command_interpreter, (void*)msg);
```

```
}
}

// IF RESPONSE RECEIVED FROM COMMAND INTERPRETER
if(resp != NULL) {
// GET CURRENT DIRECTORY
char dir[500];
bzero(dir, sizeof(dir));
getcwd(dir, sizeof(dir));
strcpy(resp->dir, dir);

// FREE THE TERMINAL
resp->show_prompt = true;

// SEND THE RESPONSE
#ifdef EXEC_SERVER
printf("%s\r\n", resp->ret);
msg_deallocate(resp);
#else
socket_write(resp);
#endif // EXEC_SERVER

resp = NULL;
}


// SLEEP FOR 0.01 SECOND TO LOWER CPU USAGE
struct timespec ts;
ts.tv_sec  = 1E-2;
ts.tv_nsec = 1E7;
nanosleep(&ts, &ts);
}

// BEGIN TEARDOWN
redirect(STDOUT_FILENO, stdout_bak);
redirect(STDERR_FILENO, stderr_bak);
redirect(STDIN_FILENO, stdin_bak);

process_list_del_list(&background_list);
process_list_del_list(&history);


#ifdef VERBOSE
printf("Exiting server\r\n");
```

```
	#endif // VERBOSE

	return 0;
}



/**
 * @brief threaded function to interpret commands from client
 * @param vargp (void* -> Msg*) pointer to Msg structure
 * @return unused
 */
void* server_command_interpreter(void* vargp) {

	// GET THE MESSAGE
	Msg *msg = (Msg*)vargp;

	// PIPE FROM STDIN TO PROCESS
	// PIPE FROM PROCESS TO STDOUT
	// PLUS ANY INTERNAL PIPES
	int num_pipes = 2;
	for(int i=0; i<strlen(msg->cmd); ++i) {
	if(msg->cmd[i] == '|') ++num_pipes;
	}

	// DECIDE WHETHER TO RUN IN FOREGROUND OR BACKGROUND
	bool background = (msg->cmd[strlen(msg->cmd)-1] == '&') ? true : false;

	// REMOVE '&' CHARACTER
	if(background) {
	msg->cmd[strlen(msg->cmd)-1] = '\0';
	}

	// SETUP PROCESSES (DIVIDED BY PIPE CHARACTER)
	Process procs[num_pipes-1];
	int pipes[num_pipes][2];
	for(int i=0; i<num_pipes; ++i) pipe(pipes[i]);


	// MAKE A COPY OF THE COMMAND
	char cmd[1024];
	strcpy(cmd, msg->cmd);

	char *pos = cmd;
```

```
char *tmp_cmd = (char*)calloc(1024, sizeof(char));

// DIVIDE COMMAND INTO JOBS BY PIPE CHARACTER
for(int i=0; i<num_pipes-1; ++i) {
char *oldpos;
if(i==0) oldpos = pos;
else oldpos = pos+2;

pos = strchr(pos+1, '|');

if(pos != NULL) strncpy(tmp_cmd, oldpos, strlen(oldpos)-strlen(pos)-1);
else strncpy(tmp_cmd, oldpos, strlen(cmd)-(strlen(oldpos)-strlen(cmd)-1));

Msg *tmp_msg = msg_allocate(tmp_cmd, NULL, NULL);
process_init(&procs[i], tmp_msg);
msg_deallocate(tmp_msg);
}

free(tmp_cmd);


// ITERATE THROUGH THE JOBS
for(int i=1; i<num_pipes; ++i) {
if(!background) {
in_foreground = true;
foreground_stdin[0] = pipes[i-1][0];
foreground_stdin[1] = pipes[i-1][1];
}

#ifdef VERBOSE
printf("num pipes: %d\r\n", num_pipes-1);
printf("cmd: %s\r\n", cmd);
printf("procs_exec: %s\r\n", procs[i].exec);
for(int j=0; j<procs[i].num_args; ++j) {
printf("procs_args: %s\r\n", procs[i].args[j]);
}
printf("\r\n\r\n");
#endif // VERBOSE

if(in_cmd_list(&procs[i-1])) {
// HANDLE OWN COMMANDS
if(background) {
        procs[i-1].pid = getpid();
        background_place_proc(&background_list, &procs[i-1]);
```

```
        }

        char outbuff[5000];
        bzero(outbuff, sizeof(outbuff));

        run_cmd_list(&procs[i-1], outbuff);
        write(pipes[i][1], outbuff, sizeof(outbuff));
        close(pipes[i][1]);
        }
        else {
        // HANDLE SYSTEM COMMANDS
        pid_t child = fork();
        switch(child) {
                case -1:
                // ERROR:

                break;

                case 0: {
                // CHILD
                close(pipes[i][0]);
                redirect(STDERR_FILENO, pipes[i][1]);
                redirect(STDOUT_FILENO, pipes[i][1]);
                close(pipes[i][1]);

                redirect(STDIN_FILENO, pipes[i-1][0]);
                close(pipes[i-1][0]);

                run(&procs[i-1]);
                process_rem(&procs[i-1]);
                exit(0);
                }

                default:
                // PARENT
                close(pipes[i][1]);
                close(pipes[i-1][0]);
                if(background) {
                procs[i-1].pid = child;
                background_place_proc(&background_list, &procs[i-1]);
                } else {
                wait(NULL);
                }
                break;
```

```
        }
        }
        }
        in_foreground = false;

        char buff[5000];

        bzero(buff, sizeof(buff));
        close(pipes[num_pipes-1][1]);
        while(read(pipes[num_pipes-1][0], buff, sizeof(buff)) != 0) {
        // Sleep for 0.001 second
        struct timespec ts;
        ts.tv_sec  = 1E-3;
        ts.tv_nsec = 1E6;
        nanosleep(&ts, &ts);
        }
        close(pipes[num_pipes-1][0]);

        strcpy(msg->ret, buff);

        for(int i=0; i<num_pipes-1; ++i) {
        process_rem(&procs[i]);
        }

        resp = msg_allocate(msg->cmd, msg->ret, NULL);
        resp->show_prompt = true;
}



/**
 * @brief redirect filedescriptors
 * @param fdfrom (int) original file descriptor
 * @param fdto (int) new file descriptor
 */
void redirect(int fdfrom, int fdto) {
        dup2(fdto, fdfrom);
}

/**
 * @brief execute a command with execvp()
 * @param proc (Process*) process to be executed
 */
void run(Process *proc) {
```

```
        execvp(proc->exec, proc->args);
        exit(-1);
        // HANDLE FAILURE
}

/**
 * @brief execute a custom command
 * @param proc (Process*) process to be executed
 * @param outbuff (char*) output buffer (eventually send to client)
 */
void run_cmd_list(Process *proc, char *outbuff) {
        if(strcmp(proc->exec, "exit") == 0) {
        // HANDLE EXIT
        strcpy(outbuff, "shutdown");
        init_shutdown();
        return;
        }
        else if(strcmp(proc->exec, "cd") == 0) {
        // HANDLE CD
        if(proc->num_args != 2) {
        strcpy(outbuff, "cd: incorrect number of arguments");
        return;
        }
        if(chdir(proc->args[1]) != 0) {
        strcpy(outbuff ,"cd: not a valid folder");
        return;
        }
        strcpy(outbuff, "cd");
        return;
        }
        else if(strcmp(proc->exec, "history") == 0) {
        // HANDLE HISTORY
        process_list_to_string(&history, outbuff);
        return;
        }
        else if(strcmp(proc->exec, "jobs") == 0) {
        background_update_procs(&background_list);
        process_list_to_string(&background_list, outbuff);
        // printf("%s\r\n", outbuff);
        return;
        }
        else if(strcmp(proc->exec, "help") == 0) {
        sprintf(outbuff, "CMD\t\tDESCRIPTION\r\n\r\n");
        sprintf(outbuff + strlen(outbuff), "exit\t\texit shell\r\n");
```

```
        sprintf(outbuff + strlen(outbuff), "cd\t\tchange directory\r\n");
        sprintf(outbuff + strlen(outbuff), "history\t\tdisplay command history\r\n");
        sprintf(outbuff + strlen(outbuff), "jobs\t\tlist jobs in background\r\n");
        sprintf(outbuff + strlen(outbuff), "help\t\tdisplay this help message\r\n\r\n");
        return;
        }
}

/**
 * @brief check if process is in custom command list
 * @param proc (Process*) process to be checked
 * @return (bool) true if process found. else false.
 */
bool in_cmd_list(Process *proc) {
        int num_cmds = 5;
        const char* cmd_list[num_cmds];
        cmd_list[0] = "exit";
        cmd_list[1] = "cd";
        cmd_list[2] = "history";
        cmd_list[3] = "jobs";
        cmd_list[4] = "help";

        for(int i=0; i<num_cmds; ++i) {
        if(strcmp(cmd_list[i], proc->exec) == 0) return true;
        }
        return false;
}

/**
 * @brief set flag to trigger shutdown of server
 */
void init_shutdown(void) {
        // SEND SHUTDOWN MSG TO CLIENT
        loop = false;
}



#ifdef EXEC_SERVER
int main() {
        server();
        exit(0);
}
#endif // EXEC_SERVER
```

clientServerEV

clientServerEV.h

```
// ##########################################
//
//   Author  -   Collin Thornton, Ethan Vascellaro
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 socket include
//   Date          -   10-27-20
//
// ##########################################


#ifndef CLIENT_SERVER_EV_H
#define CLIENT_SERVER_EV_H

#include <stdio.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <pthread.h>

#include "msg.h"
#define sockA struct sockaddr
#define SOCKET_BUFF 5000


/**
 * @brief Initialize the socket
 * @return (int) return code
 */
int socket_init();

/**
 * @brief Read from the socket (nonblocking)
 * @return (Msg*) NULL if nothing to read
```

```
 */
Msg* socket_read();

/**
 * @brief Write to the socket (blocking)
 * @return (int) return code
 */
int socket_write(Msg *msg);

/**
 * @brief Shutdown the socket
 */
void socket_close();


/**
 * @brief Thread to managing blocking read calls
 * @param vargp (void*) unused (NULL)
 */
void *sockReadThread(void *vargp);

#endif // CLIENT_SERVER_EV_H
```

clientServerEV.c

```
        pthread_create(&tid, NULL, sockReadThread, NULL);
}



/**
 * @brief Read from the socket (nonblocking)
 * @return (Msg*) NULL if nothing to read
 */
Msg* socket_read() {
        if(sock_queue_len <= 0) return NULL;
        --sock_queue_len;

        char buff[SOCKET_BUFF];
        bzero(buff, sizeof(SOCKET_BUFF));
        strcpy(buff, sock_input);

        Msg *msg = msg_deserialize(buff);
        return msg;
```

```
}


/**
 * @brief Write to the socket (blocking)
 * @return (int) return code
 */
int socket_write(Msg *msg) {
        char buff[SOCKET_BUFF];
        bzero(buff, sizeof(buff));
        msg_serialize(msg, buff);

        write(connfd, buff, sizeof(buff));
        msg_deallocate(msg);
}


/**
 * @brief Shutdown the socket
 */
void socket_close() {
        if(tid != 0) pthread_cancel(tid);
        close(connfd);
}


/**
 * @brief Thread to managing blocking read calls
 * @param vargp (void*) unused (NULL)
 */
void *sockReadThread(void *vargp) {
        while(!sock_exit_flag) {
        char buff[SOCKET_BUFF];
        bzero(buff, SOCKET_BUFF);

        if(read(connfd, buff, sizeof(buff)) == -1) {
        perror("client:  socket closed");
        exit(1);
        }
        strcpy(sock_input, buff);
        ++sock_queue_len;
        }
}
```

Msg

msg.h

```
// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 msg struct header
//   Date    -   10-14-20
//
// ##########################################

#ifndef MSG_H
#define MSG_H

#include <stdbool.h>

#define MAX_CMD_SIZE 1024
#define MAX_RETURN_SIZE 1024

/**
 * @brief msg stuct used for comms on socket
 */
typedef struct {
    char *cmd;
    char *ret;

    char* dir;
    bool show_prompt;
} Msg;


/**
 * @brief allocate a Msg on heap. Must subsequently free memory with msg_deallocate()
 * @param cmd (char*) string command. set to NULL if unused
 * @param ret (char*) string return. set to NULL if unused
 * @param dir (char*) current working directory set to NULL if unused
 * @return (Msg*) Pointer to Msg allocated on heap
 */
Msg* msg_allocate(char* cmd, char *ret, char* dir);      \
```

```
/**
 * @brief Deallocate Msg previously allocated by msg_allocate()
 * @param msg (Msg*) msg allocated on heap
 */
void msg_deallocate(Msg *msg);                \

/**
 * @brief Convert Msg to string for socket comms. Should be returned to Msg by
msg_deserialize()
 * @param msg (Msg*) Msg to be converted
 * @param buff (char*) String buffer
 */
char* msg_serialize(Msg *msg, char *buff);             \

/**
 * @brief Conver string to Msg for socket comms. Should be preceded by msg_serialize()
 * @param str (const char*) String containing Msg information
 * @return (Msg*) msg struct allocated on heap
 */
Msg* msg_deserialize(const char* str);

#endif //MSG_H
```

Msg.c

```
// ##############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Assign  -   Assignment 02 msg struct source
//   Date    -   10-14-20
//
// ##############################################

#include "../include/msg.h"

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

#define SIZE 5000
```

```
/**
 * @brief allocate a Msg on heap. Must subsequently free memory with msg_deallocate()
 * @param cmd (char*) string command. set to NULL if unused
 * @param ret (char*) string return. set to NULL if unused
 * @param dir (char*) current working directory set to NULL if unused
 * @return (Msg*) Pointer to Msg allocated on heap
 */
Msg* msg_allocate(char* cmd, char* ret, char *dir) {
    Msg *msg = (Msg*)malloc(sizeof(Msg));

    if(msg == NULL) {
        perror("msg: allocation");
        return NULL;                    // Return NULL if allocation fails
    }

    msg->cmd = (char*)calloc(SIZE, sizeof(char));
    msg->ret = (char*)calloc(SIZE, sizeof(char));
    msg->dir = (char*)calloc(SIZE, sizeof(char));
    msg->show_prompt = true;

    if(msg->cmd == NULL) {
        perror("msg->command: allocation");
        return NULL;
    }
    if(msg->ret == NULL) {
        perror("msg->ret: allocation");
        return NULL;
    }
    if(msg->dir == NULL) {
        perror("msg->dir: allocation");
        return(NULL);
    }

    if(cmd != NULL) {
        strcpy(msg->cmd, cmd);
    }
    if(ret != NULL) {
        strcpy(msg->ret, ret);
    }
    if(dir != NULL) {
        strcpy(msg->dir, dir);
    }
```

```
    return msg;
}


/**
 * @brief Deallocate Msg previously allocated by msg_allocate()
 * @param msg (Msg*) msg allocated on heap
 */
void msg_deallocate(Msg *msg) {
    if(msg != NULL) {
        if(msg->cmd != NULL) free(msg->cmd);
        if(msg->ret != NULL) free(msg->ret);
        if(msg->dir != NULL) free(msg->dir);

        free(msg);
        msg = NULL;
    }
}


/**
 * @brief Convert Msg to string for socket comms. Should be returned to Msg by
msg_deserialize()
 * @param msg (Msg*) Msg to be converted
 * @param buff (char*) String buffer
 */
char* msg_serialize(Msg *msg, char *buff) {

    if(msg->cmd == NULL) return NULL;

    strcpy(buff, msg->cmd);

    if(msg->ret == NULL) {
        strcat(buff, ":-:null");
    } else {
        strcat(buff, ":-:");
        strcat(buff, msg->ret);
    }

    if(msg->dir == NULL) {
        strcat(buff, ":-:null");
    } else {
        strcat(buff, ":-:");
        strcat(buff, msg->dir);
```

```
    }

    strcat(buff, ":-:");
    if(msg->show_prompt) strcat(buff, "t");
    else strcat(buff, "f");

    strcat(buff, "\r\n");
    return buff;
}



/**
 * @brief Conver string to Msg for socket comms. Should be preceded by msg_serialize()
 * @param str (const char*) String containing Msg information
 * @return (Msg*) msg struct allocated on heap
 */
Msg* msg_deserialize(const char* str) {
    //printf("%s\r\n", str);

    char *cmd = (char*)calloc(SIZE, sizeof(char));
    char *ret = (char*)calloc(SIZE, sizeof(char));
    char *dir = (char*)calloc(SIZE, sizeof(char));
    char prmpt[2];
    bzero(prmpt, sizeof(prmpt));

    char *pos1 = strstr(str, ":-:");
    strncpy(cmd, str, strlen(str)-strlen(pos1));

    char *pos2 = strstr(str+(strlen(str)-strlen(pos1)+3), ":-:");
    strncpy(ret, pos1+3, strlen(pos1)-strlen(pos2)-3);

    char *pos3 = strstr(str+(strlen(str)-strlen(pos2)+3), ":-:");
    strncpy(dir, pos2+3, strlen(pos2)-strlen(pos3)-3);

    char *pos4 = strstr(str+(strlen(str)-strlen(pos3)+3), "\r\n");
    strncpy(prmpt, pos3+3, strlen(pos3)-strlen(pos4)-3);

    Msg *msg = msg_allocate(cmd, ret, dir);

    msg->show_prompt = (strcmp(prmpt, "t") == 0) ? true : false;

    if(cmd != NULL) free(cmd);
    if(ret != NULL) free(ret);
```

```
    if(dir != NULL) free(dir);

    return msg;
}
```

Process

Process.h

```
// ###########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 process struct header
//   Date          -   10-14-20
//
// ###########################################

#ifndef PROCESS_H
#define PROCESS_H

#include "msg.h"
#include <stdbool.h>


/**
// @brief Stores data for single processes. Allocated on heap
*/
typedef struct {
        bool initialized;       // Is process initialized
        bool returned;          // Has process returned

        int pid;
        int num_args;

        char *exec;
        char **args;
        char *ret;
} Process;


/**
// @brief Node in linked list of processes
```

```
*/
struct ProcessNode{
        Process *node;
        struct ProcessNode *next;
        struct ProcessNode *prev;
};
typedef struct ProcessNode ProcessNode;


/**
// @brief Linked list of processes
*/
typedef struct {
        ProcessNode *HEAD;
        ProcessNode *TAIL;

        int num_processes;
} ProcessList;


/**
 * @brief Initialize fields of Process struct
 * @param proc (Process*) process to be initialized
 * @param msg (Msg*) Msg from which to fill Process
 * @return (Process*) pointer to proc
 */
Process* process_init(Process* proc, Msg *msg);

/**
 * @brief Deallocated all fields of Process struct
 * @param proc (Process*) process to be deallocated
 * @return (int) return code
 */
int process_rem(Process* proc);

/**
 * @brief initalize process linked list
 * @param list (ProcessList*) Pointer to list to be initialized
 * @param HEAD (ProcessNode*) Pointer to HEAD of list. NULL if unused.
 * @param TAIL (ProcessNode*) Pointer to TAIL of list. NULL if unused.
 * @return (ProcessList*) same as list
 */
ProcessList* process_list_init(ProcessList *list, ProcessNode *HEAD, ProcessNode *TAIL);
```

```
/**
 * @brief add node to linked list
 * @param list (ProcessList*) pointer to list
 * @param proc (Process*) process to be added
 * @return (int) size of list
 */
int process_list_add_node(ProcessList *list, Process *proc);

/**
 * @brief remove node from linked list
 * @param list (ProcessList*) pointer to list
 * @param proc (Process*) process to be removed
 * @return (int) length of list
 */
int process_list_rem_node(ProcessList *list, Process *proc);

/**
 * @brief Deallocate list, all nodes, and all process in list
 * @param list (ProcessList*) list to be removed
 * @return (int) return code
 */
int process_list_del_list(ProcessList *list);


/**
 * @brief convert process list to string
 * @param list (ProcessList*) list to be converted
 * @param buff (char[]) string buffer
 * @return (char*) pointer to buff
 */
const char* process_list_to_string(ProcessList *list, char buff[]);

#endif // PROCESS_H
```

process.c

```
// ############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 process struct source
//   Date          -   10-14-20
//
// ############################################
```

```
#include "../include/process.h"

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

//#define EXEC_PROCESS


/**
 * @brief Initialize fields of Process struct
 * @param proc (Process*) process to be initialized
 * @param msg (Msg*) Msg from which to fill Process
 * @return (Process*) pointer to proc
 */
Process* process_init(Process *proc, Msg *msg) {
	proc->initialized = false;
	proc->returned = false;
	proc->pid = -1;
	proc->ret = (char*)calloc(5000, sizeof(char));
	proc->args = NULL;
	proc->exec = (char*)calloc(1000, sizeof(char));
	proc->num_args = 0;

	char buff[MAX_CMD_SIZE];
	strcpy(buff, msg->cmd);
	char *token = strtok(buff, " ");
	if(token == NULL) {
	strcpy(proc->exec, msg->cmd);
	proc->args = (char**)malloc(sizeof(char*));
	proc->args[0] = NULL;
	proc->num_args = 0;
	}
	else {
	int i = 0;

	do {
	token = strtok(NULL, " ");
	++i;
	} while(token != NULL);
	proc->num_args = i;

	proc->args = (char**)malloc((proc->num_args+1)*sizeof(char*));
```

```
        strcpy(buff, msg->cmd);
        token = strtok(buff, " ");
        proc->exec = (char*)malloc(sizeof(char[50]));
        strcpy(proc->exec, token);

        int j=0;
        proc->args[j] = (char*)malloc(sizeof(char[50]));
        strcpy(proc->args[j++], token);

        do {
        token = strtok(NULL, " ");

        if(token != NULL) {
                proc->args[j] = (char*)calloc(50, sizeof(char));
                strcpy(proc->args[j++], token);
        }
        } while(token != NULL);


        proc->args[j] = NULL;
        }

        return proc;
}


/**
 * @brief Deallocated all fields of Process struct
 * @param proc (Process*) process to be deallocated
 * @return (int) return code
 */
int process_rem(Process *proc) {
        if(proc->ret != NULL) free(proc->ret);
        if(proc->exec != NULL)free(proc->exec);

        if(proc->args != NULL) {
        for(int i=0; i<=proc->num_args; ++i) {
        free(proc->args[i]);
        }
        free(proc->args);
        }
}
```

```
/**
 * @brief initalize process linked list
 * @param list (ProcessList*) Pointer to list to be initialized
 * @param HEAD (ProcessNode*) Pointer to HEAD of list. NULL if unused.
 * @param TAIL (ProcessNode*) Pointer to TAIL of list. NULL if unused.
 * @return (ProcessList*) same as list
 */
ProcessList* process_list_init(ProcessList *list, ProcessNode *HEAD, ProcessNode *TAIL) {
	list->num_processes = 0;

	if(HEAD != NULL) ++list->num_processes;
	if(TAIL != NULL && TAIL!= HEAD) ++list->num_processes;

	list->HEAD = HEAD;
	list->TAIL = TAIL;
}


/**
 * @brief add node to linked list
 * @param list (ProcessList*) pointer to list
 * @param proc (Process*) process to be added
 * @return (int) size of list
 */
int process_list_add_node(ProcessList *list, Process *proc) {
	if(list == NULL) return -1;
	if(list->num_processes < 0) return -2;

	Process *new_proc = (Process*)malloc(sizeof(Process));
	ProcessNode *new_node = (ProcessNode*)malloc(sizeof(ProcessNode));
	new_node->node = new_proc;
	new_node->next = NULL;
	new_node->prev = NULL;

	new_proc->initialized = proc->initialized;
	new_proc->pid = proc->pid;
	new_proc->returned = proc->returned;
	new_proc->num_args = proc->num_args;

	new_proc->exec = (char*)calloc(5000, sizeof(char));
	new_proc->ret = (char*)calloc(5000, sizeof(char));
	new_proc->args = (char**)malloc((proc->num_args+1)*sizeof(char*));
```

```
	for(int i=0; i<new_proc->num_args; ++i) new_proc->args[i] = (char*)calloc(5000,
sizeof(char));
	new_proc->args[new_proc->num_args] = NULL;

	if(proc->exec != NULL) {
	strcpy(new_proc->exec, proc->exec);
	}
	if(proc->ret != NULL) {
	strcpy(new_proc->ret, proc->ret);
	}
	if(proc->args != NULL) {
	for(int i=0; i<new_proc->num_args; ++i) {
	strcpy(new_proc->args[i], proc->args[i]);
	}
	}

	if(list->num_processes == 0) {
	list->HEAD = new_node;
	list->TAIL = new_node;
	}
	else {
	list->TAIL->next = new_node;
	new_node->prev = list->TAIL;
	list->TAIL = new_node;
	}

	++list->num_processes;
	return list->num_processes;
}


/**
 * @brief remove node from linked list
 * @param list (ProcessList*) pointer to list
 * @param proc (Process*) process to be removed
 * @return (int) length of list
 */
int process_list_rem_node(ProcessList *list, Process *proc) {
	if(list == NULL) return -1;
	if(list->HEAD == NULL || list->TAIL == NULL || list->num_processes <= 0) return -2;


	ProcessNode *tmp = list->HEAD;
	while(tmp != list->TAIL) {
```

```
        tmp = tmp->next;
        }

        while(tmp->node->pid != proc->pid) {
        if(tmp == list->TAIL) return -3;
        tmp = tmp->next;
        }

        if(list->num_processes == 1) {
        list->HEAD = NULL;
        list->TAIL = NULL;
        free(tmp->node->exec);
        free(tmp->node->ret);
        for(int i=0; i<tmp->node->num_args; ++i) free(tmp->node->args[i]);
        free(tmp->node->args);

        free(tmp->node);
        free(tmp);
        --list->num_processes;
        return list->num_processes;
        }

        if(tmp == list->HEAD) {
        list->HEAD = tmp->next;
        list->HEAD->prev = NULL;
        }
        else if(tmp == list->TAIL) {
        list->TAIL = tmp->prev;
        list->TAIL->next = NULL;
        }
        else {
        tmp->prev->next = tmp->next;
        tmp->next->prev = tmp->prev;
        }

        --list->num_processes;
        free(tmp->node);
        free(tmp);
        return list->num_processes;
}


/**
 * @brief Deallocate list, all nodes, and all process in list
```

```
 * @param list (ProcessList*) list to be removed
 * @return (int) return code
 */
int process_list_del_list(ProcessList *list) {
	if(list == NULL) return -1;
	if(list->HEAD == NULL || list->TAIL == NULL || list->num_processes < 0) return -2;

	ProcessNode *tmp = list->HEAD;
	while(tmp != list->TAIL) {
	tmp = tmp->next;
	free(tmp->prev->node);
	free(tmp->prev);
	--list->num_processes;
	}
	free(tmp->node);
	free(tmp);
	--list->num_processes;
	return list->num_processes;
}


/**
 * @brief convert process list to string
 * @param list (ProcessList*) list to be converted
 * @param buff (char[]) string buffer
 * @return (char*) pointer to buff
 */
const char* process_list_to_string(ProcessList *list, char buff[]) {
	sprintf(buff, "%d jobs\r\n", list->num_processes);

	if(list->HEAD == NULL || list->TAIL == NULL) return buff;

	sprintf(buff + strlen(buff), "CMD\t\t\tPID\r\n\r\n");

	ProcessNode *tmp = list->HEAD;
	sprintf(buff + strlen(buff), "0.) %s\t\t%d\r\n", tmp->node->exec, tmp->node->pid);

	int i=0;
	while(tmp != list->TAIL) {
	tmp = tmp->next;
	++i;
	sprintf(buff + strlen(buff), "%d.) %s\t\t%d\r\n", i, tmp->node->exec, tmp->node->pid);
	}
```

```
	return buff;
}



#ifdef EXEC_PROCESS
int main() {
	Msg msg;
	msg.cmd = "ps -A";
	msg.ret = NULL;

	Process proc;
	process_init(&proc, &msg);

	printf("%s\r\n", proc.exec);

	for(int i=0; i<proc.num_args+1; ++i) printf("%s\r\n", proc.args[i]);

	process_rem(&proc);
}
#endif // EXEC_PROCESS
```

Background

Background.h

```
// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 Part 5 include
//   Date          -   10-14-20
//
// ##########################################


#ifndef BACKGROUND_H
#define BACKGROUND_H

#include <stdio.h>       // for printf()
#include <unistd.h>      // for fork()
#include <stdlib.h>      // for exit()
#include <fcntl.h>
```

```
#include <string.h>

#include <sys/wait.h>   // for wait()

#include "../include/msg.h"
#include "../include/process.h"

/**
 * @brief Places a process into the background list
 * @param list (ProcessList*) Pointer to the list
 * @param proc (Process*) Pointer to the process to be added
 * @return (int) Length of list
 */
int background_place_proc(ProcessList *list, Process *proc);

/**
 * @brief Remove exited processes from list
 * @param list (ProcessList*) Pointer to the list
 * @return (int) Length of list
 */
int background_update_procs(ProcessList *list);

#endif // BACKGROUND_H
```

Background.c

```
// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Assignment 02 Part 5 source
//   Date          -   10-14-20
//
// ##########################################

#include "../include/background.h"

//#define EXEC_BACK


/**
 * @brief Places a process into the background list
 * @param list (ProcessList*) Pointer to the list
 * @param proc (Process*) Pointer to the process to be added
```

```
 * @return (int) Length of list
 */
int background_place_proc(ProcessList *list, Process *proc) {

	proc->initialized = true;
	proc->returned = false;

	process_list_add_node(list, proc);
}


/**
 * @brief Remove exited processes from list
 * @param list (ProcessList*) Pointer to the list
 * @return (int) Length of list
 */
int background_update_procs(ProcessList *list) {
	if(list->num_processes == 0) return 0;

	ProcessNode *tmp = list->HEAD;
	while(tmp != list->TAIL) {
	tmp = tmp->next;

	int exited = waitpid(tmp->prev->node->pid, NULL, WNOHANG);
	if(exited != 0) {
	process_list_rem_node(list, tmp->prev->node);
	}
	}

	int exited = waitpid(tmp->node->pid, NULL, WNOHANG);
	if(exited != 0) {
	process_list_rem_node(list, tmp->node);
	}

}


#ifdef EXEC_BACK
int main() {
	background_test();
}
#endif // EXEC_BACK
```

Compile.sh

```
#! /bin/bash

gcc msg.c process.c background.c clientServerEV.c server.c client.c main.c -lpthread -lrt -o main -g
```

Appendix D - Caleb's Code