PUC Minas

Pontifícia Universidade Católica de Minas Gerais
Campus de Poços de Caldas

Departamento de Ciência da Computação
Curso de Ciência da Computação
Disciplina de Sistemas Operacionais

TRABALHO DE PIPES

Gabriel Henrique Garcia
Aluno

Tuanne Assenço
Aluna

Prof. Dr. Joao Carlos de Moraes Morselli Junior
Professor

Poços de Caldas – MG

Dezembro de 2022

TRABALHO DE SISTEMAS OPERACIONAIS - PIPES

TRABALHO ACADÊMICO

Trabalho referente à DISCIPLINA DE SISTEMAS OPERACIONAIS, 2° Semestre de 2022.

Poços de Caldas – MG

Dezembro de 2022

SUMÁRIO

1 INTRODUÇÃO

Este documento tem como objetivo apresentar a descrição do trabalho acadêmico, notadamente no campo das ***Ciências da Computação*** (CC), em atendimento à tarefa solicitada da disciplina de Sistemas Operacionais – TRABALHO DE PIPES (15 PTS) 15/12, da Pontifícia Universidade Católica de Minas Gerais (PUC Minas). O trabalho aqui reportado está inserido no contexto do uso de ferramentas de ***Inter Process Communication (IPC) e Canais de Comunicação de Processos Utilizando PIPES***, como parte do estudo do assunto previsto na ementa da referida disciplina, na Pontifícia Universidade Católica de Minas Gerais (PUC Minas), *campus* de Poços de Caldas, enquadrando-se num cenário de desenvolvimento e implementação de um software que realiza o uso do sistema de intercomunicação de processos. Pretende-se apresentar com o **Código Fonte Desenvolvido em Linguagem C**, os cenários: 1) compreensão do funcionamento de Ferramentas de Intercomunicação de Processos; 2) Funcionalidades e Threads; 3) desenvolvimento de um jogo que realiza o uso de PIPES para controle de dados enviados entre dois processos em um sistema de turnos de um jogo RPG (*Role-Playing Game*).

2 DESCRIÇÃO DA FERRAMENTA

Os pipes são meios de comunicação usados para enviar dados de um processo para outro. Eles são definidos como arquivos específicos que permitem a conexão entre dois processos. Se um processo deseja enviar uma mensagem para outro, ele escreve a mensagem no pipe que conecta os dois processos. O processo receptor então lê a mensagem do pipe. Dessa forma, os pipes permitem que processos diferentes troquem dados através de escrita e leitura [1].

Os pipes são ferramentas importantes para garantir a sincronização e a integridade dos dados em um software. Quando um processo lê um pipe, ele bloqueia a escrita no arquivo, impedindo que outros processos escrevam nele enquanto a leitura está sendo executada. Isso garante que os dados sejam lidos e escritos de forma consistente e que não haja conflito entre os processos [1].

As *threads* em C são uma forma de executar várias tarefas de forma paralela dentro de um programa. Elas permitem que o programa execute várias tarefas ao mesmo tempo, aumentando a performance e a eficiência do programa. As threads são criadas e gerenciadas usando a biblioteca pthread.h, que fornece funções para criar, gerenciar e sincronizar threads. As threads compartilham o mesmo espaço de endereçamento e recursos do processo principal, o que as torna mais rápidas e fáceis de gerenciar do que processos separados [1].

A proposta do programa é um jogo de RPG baseado em turnos que usa pipes para comunicação entre dois processos e também chama uma *thread* durante a execução. Ele também inclui vários arquivos de cabeçalho para funções específicas do jogo, como instruções, customização de raça e classe do jogador, baralhos, movimentos de ataque e defesa e mensagens de motivação. O código também contém várias funções de impressão de cores para tornar a interface do usuário mais atraente.

Na função “main()”, o programa começa exibindo instruções para o usuário e, em seguida, cria dois pipes para comunicação entre processos. Ele cria um processo filho chamando fork() e, em

seguida, o processo pai e o processo filho chamam suas respectivas funções “player_1()” e “player_2()” que implementam o jogo em si. A comunicação entre os processos é realizada usando os pipes criados anteriormente. O jogo continua até que um dos jogadores perca toda a sua vida.

Vale ressaltar que a posição 0 dos vetores dos pipes se refere ao processo descritor de leitura, enquanto a posição 1, escrita.

3 DESCRIÇÃO DA APLICAÇÃO

A aplicação foi dividida em arquivos *headers* para a sua organização, a seguir será explicado as competências de cada arquivo:

- rpg.c

O programa começa incluindo bibliotecas necessárias para sua operação, como ‘stdio.h’, ‘stdlib.h’, ‘string.h’, ‘time.h’, ‘unistd.h’ e ‘pthread.h’. Estas bibliotecas fornecem funções para entrada/saída padrão, operações em bibliotecas padrão, manipulação de ‘strings’, funções relacionadas ao tempo, funções padrão do sistema operacional e funções relacionadas a *threads*, respectivamente.

Em seguida, o programa inclui arquivos de cabeçalho para funções relacionadas a instruções, customização do jogador por raça e classe, baralhos, ataques e defesas, jogador 1 e jogador 2, e mensagens motivacionais. Esses arquivos de cabeçalho contêm declarações de função para cada um desses tópicos.

O programa então define macros para limpar o buffer de entrada, limpar a tela e colocar o programa em espera por 1 segundo, dependendo do compilador usado. Isso é feito usando a macro ‘#if’ para verificar se o programa está sendo compilado no ‘MinGW’ ou no ‘Visual Studio’, e definindo as macros de acordo. Se nenhum desses compiladores estiver sendo usado, o programa inclui ‘stdio_ext.h’ e define as macros de acordo.

Em seguida, o programa declara funções para imprimir texto em diferentes cores. Essas funções usam códigos de escape ANSI para mudar a cor do texto na saída padrão.

Depois disso, o programa declara funções globais para ataques, defesas, jogador 1 e jogador 2. Essas funções são definidas em outros arquivos de cabeçalho incluídos anteriormente e serão explicadas posteriormente.

Na função ‘main()’, o programa começa declarando algumas funções e inicializando variáveis. Ele então cria dois ‘pipes’ para comunicação entre o jogador 1 e o jogador 2 usando a função ‘pipe()’. Em seguida, cria um processo filho que é responsável por executar a função do jogador 1. Um subprograma é criado para executar a função do jogador 2.

O programa então entra em um loop infinito que executa as seguintes etapas em cada iteração:

1. Imprime as vidas atuais dos jogadores e pede que o jogador 1 escolha uma ação.
2. O jogador 1 escolhe uma ação e a envia para o jogador 2 através do pipe1.
3. O jogador 2 recebe a ação do jogador 1 através do pipe2 e escolhe uma ação.
4. As ações dos jogadores 1 e 2 são processadas e as vidas dos jogadores são atualizadas de acordo.
5. O loop é reiniciado e o processo se repete até que um jogador perca todas as suas vidas. Quando isso acontece, o jogo é encerrado e uma mensagem de vitória é impressa para o jogador vencedor.

- print_colors.h

Este *header* é um conjunto de funções em C que permitem imprimir texto em diferentes cores. Cada função imprime texto de uma cor específica, como rosa, ciano, amarelo, azul, verde, vermelho ou branco. A função “print_reset” permite *resetar* a cor do texto para a cor padrão.

Cada função começa declarando o nome da cor que ela imprime e recebe um parâmetro do tipo “char *” que representa o texto a ser impresso. Em seguida, ela usa a função printf para imprimir o texto usando uma sequência de escape ANSI. Cada sequência de escape é um código de cor que diz ao terminal para imprimir o texto em uma determinada cor. Por exemplo, a sequência de escape “\033[1;35m” é usada para imprimir o texto em rosa. A função então imprime o texto recebido como parâmetro e, por fim, usa outra sequência de escape para *resetar* a cor do texto para a cor padrão. Isso permite que o programa imprima texto em diferentes cores.

- Instructions.h

Este *header* é uma função que é responsável por imprimir na tela as instruções do jogo para os

jogadores. A função contém uma série de impressões de texto colorido na tela, que descrevem as regras do jogo e como ele funciona. A função também contém um chamado para a função "getchar", que aguarda o usuário pressionar a tecla "*Enter*" para continuar. (FIGURA 1);

- player_1.h

Este *header* é uma função que define duas estruturas para armazenar os atributos de um jogador e os valores de variáveis passadas através de pipes, sendo elas de dano causado e uma variável de controle para finalizar o processo. Ele também define dois *arrays* para as opções de classe e raça disponíveis.

A função "player_1" é a função principal para o jogador 1. Ela pede que o jogador escolha sua raça e classe, e então entra em um *loop* onde permite que o jogador escolha e execute um movimento de ataque e defesa. A função também verifica o status de vida do jogador e imprime uma mensagem motivacional através de uma *thread* se eles estiverem abaixo de 25%.

Os valores das variáveis passadas através dos “pipes” são lidos e escritos na função "player_1" para permitir que o jogador 1 comunique-se com o jogador 2. Isso é feito através do uso de funções de leitura e escrita em “pipes”, como "read" e "write".

- player_2.h

Este *header* define uma função para o jogador 2 em um jogo. A função permite que o jogador 2 escolha sua raça e classe, e então inicie um combate contra o jogador 1. A função usa “pipes” para comunicar-se com o jogador 1 e trocar informações sobre o jogo. A função repete ciclos de defesa e ataque, onde o jogador 2 pode defender-se contra os ataques do jogador 1 e causar dano ao jogador 1. A função também verifica o status de vida do jogador e imprime uma mensagem motivacional através de uma *thread* se eles estiverem abaixo de 25%. O ciclo continua até que a vida do jogador 2 ou do jogador 1 atinja 0, momento em que o jogo termina e um vencedor é declarado.

- player_customization_race.h

Este arquivo *header* exerce a função para customização do jogador em um jogo. Ela permite que o jogador escolha sua raça a partir de uma lista de opções e, em seguida, retorna a raça escolhida pelo jogador como um valor inteiro. A função também imprime mensagens para o jogador informando sua escolha e tratando a entrada inválida. (FIGURA 2 E 4);

A função começa inicializando uma variável para a escolha da raça do usuário. Em seguida, ela exibe opções para a seleção de raça. O jogador é então solicitado a inserir sua escolha de raça, que é lida usando a função ‘scanf()’. O buffer de entrada é limpo com a função ‘getchar()’.

A função então entra em um loop que só é encerrado quando o jogador escolhe uma opção válida. Dentro do loop, a escolha do jogador é verificada com a instrução 'switch’. Se a opção for válida, uma mensagem é impressa para confirmar a escolha e uma variável é definida para sair do loop. Se a opção for inválida, uma mensagem de erro é impressa.

Quando o loop é encerrado, a função retorna a escolha da raça do jogador.

- player_customization_class.h

Este *header* tem a função de personalização da classe do jogador. Ele permite que o jogador escolha sua classe de uma lista de opções e, em seguida, retorna a classe escolhida pelo jogador como um valor inteiro. A função também imprime mensagens para o jogador para informá-los sobre sua escolha. Quando a função é chamada, ela imprime instruções para o jogador e pede que eles escolham uma classe de jogo. Em seguida, a função lê a escolha do jogador e verifica se é válida. Se a escolha for válida, a função imprime uma mensagem informando a classe escolhida e retorna o número da classe escolhida. Se a escolha for inválida, a função pede ao jogador que escolha novamente até que uma opção válida seja selecionada. (FIGURA 3 E 5);

- decks.h

Este header é uma função que define uma estrutura de dados para uma carta de ataque, que contém o nome do ataque, seu poder e uma descrição. Ele também define uma estrutura de dados para um array de cartas de ataque e uma função para selecionar um deck de cartas de ataque com base na classe do jogador. A função tem um comando "switch" que retorna um deck diferente de cartas de ataque para cada classe (guerreiro, mago, assassino, clérigo). Os decks de cartas de ataque também são definidos no código.

A estrutura de dados "attack" é usada para representar uma carta de ataque e contém três campos: um nome para o ataque, seu poder e uma descrição.

A seguir, a estrutura de dados "attack_arr" é usada para representar um array de cartas de ataque. Ele contém um único campo, que é um array de 10 cartas de ataque.

A função "selected_deck" é usada para selecionar um deck de cartas de ataque com base na classe do jogador. Ela recebe um índice de classe como argumento e retorna um array de cartas de ataque adequado. A função contém um comando "switch" que retorna um deck diferente de cartas de ataque para cada classe
função em C que retorna o poder de ataque do ataque escolhido pelo usuário. Ela tem como entrada uma “struct” do tipo “Attack_arr”.

- attack_move.h

Este *header* é uma função inicializa variáveis para os limites inferior e superior da geração de números aleatórios, o contador de ataques para escolher e o array de ataques. Ela então inicializa o gerador de números aleatórios com o tempo atual e gera três números aleatórios dentro dos limites especificados. (FIGURAS 6 E 8);

Em seguida, a função escolhe três ataques de acordo com os índices aleatórios gerados. Ela imprime os ataques disponíveis para o usuário e solicita que eles escolham um. Isso é feito em um loop até que o usuário selecione um ataque válido. Quando o jogador escolhe um ataque

válido, a função retorna o poder de ataque do ataque escolhido.

- defense_move.h

Este header é uma função que retorna a quantidade de dano que o jogador recebe de um ataque. Inicia-se declarando algumas variáveis:

- “lower” e “upper” são os limites inferior e superior, respectivamente, para a geração de números aleatórios. Neste caso, eles estão inicializados com 1 e 3, o que significa que o gerador de números aleatórios irá gerar números entre 1 e 3, inclusive.
- “d3_dice” é a variável que armazena o resultado do lançamento do dado de defesa. Inicialmente, ela é inicializada com 0.
- “damage” é a variável que armazena a quantidade final de dano que o jogador receberá. Inicialmente, ela é inicializada com a quantidade de dano de entrada.

Em seguida, a função imprime para o usuário os possíveis resultados do lançamento do dado de defesa.

O usuário é solicitado a pressionar a tecla "Enter" para rolar o dado de defesa.

Quando o usuário pressiona "Enter", a função inicializa o gerador de números aleatórios com a hora atual usando a função “srand” e então rola o dado de defesa usando a função rand. O resultado do lançamento é armazenado na variável “d3_dice”. (FIGURA 7 E 9);

Finalmente, a função usa um “switch statement” para retornar a quantidade adequada de dano baseada no resultado do lançamento do dado. Se o resultado for 1, a função retorna 0, o que significa que o jogador evitou o ataque e não sofreu dano. Se o resultado for 2, a função retorna 50% da quantidade inicial de dano, o que significa que o jogador sofreu metade do dano inicial. Se o resultado for 3, a função retorna a quantidade inicial de dano, o que significa que o jogador sofreu todo o dano inicial.

- motivation_message.h

Este *header* é uma função de thread que imprime uma mensagem motivacional para os jogadores, se o status de vida dele for menor ou igual a 25% de sua vida inicial. A mensagem impressa depende da classe do jogador. A função recebe como parâmetro "arg", que é um inteiro representando a classe do jogador. A função então imprime a mensagem motivacional correspondente e aguarda que o usuário pressione uma tecla antes de continuar. (FIGURA 10);

A função começa fazendo um casting do parâmetro "arg" para o tipo de dados correto "int". Em seguida, ela usa um “switch statement” para determinar qual mensagem motivacional imprimir baseada na classe do jogador.

4 CÓDIGO

- rpg.c

```c
/**
* Description:
* The following program takes the conception of pipes to communicate between 2
* process and also calls a thread during execution time.
*
* Project developed for the Operational Systems for the Computer Science degree.
*
* Autors:
* Tuanne Assenço - tuanne.assenco@gmail.com
* Gabriel Garcia - gabrielhgarcia7@gmail.com
*
* Compiled as follows:
* $ gcc -Wall rpg.c -o rpg; ./rpg
**/


// Inclusion of libraries necessary for the operation of the program
#include <stdio.h> // For standard input/output functions
#include <stdlib.h> // For standard library functions
#include <string.h> // For string manipulation functions
#include <time.h> // For time-related functions
#include <unistd.h> // For POSIX-standard functions
#include <pthread.h> // For thread-related functions
#include "instructions.h" // For instructions-related functions
#include "player_customization_race.h" // For player customization race-related functions
#include "player_customization_class.h" // For player customization class-related
functions
#include "decks.h" // For deck-related functions
#include "attack_move.h" // For attack move-related functions
#include "defense_move.h" // For defense move-related functions
#include "player_1.h" // For player 1-related functions
#include "player_2.h" // For player 2-related functions
#include "print_colors.h" // For print color-related functions
#include "motivation_message.h" // For motivation message-related functions


// This code defines macros for cleaning the input buffer, clearing the screen, and
sleeping for 1 second,
// depending on the compiler being used.
#if defined(__MINGW32__) || defined(_MSC_VER)
#define clean_input() fflush(stdin)
#define clear_screen() system("cls")
#define sleep() system("timeout 1")
#else
#include <stdio_ext.h>
#define clean_input() __fpurge(stdin)
#define clear_screen() system("clear")
#define sleep() system("sleep 1")
#endif
```

```c
// Color control functions for user preview
void print_pink(char *s);
void print_yellow(char *s);
void print_blue(char *s);
void print_green(char *s);
void print_green(char *s);
void print_red(char *s);
void print_white(char *s);
void print_cyan(char *s);


// Global functions declaration of 'attack moves', 'defense', 'player 1' and 'player 2'
int attack_move(Attack_arr player_attack);
float defense_move(int damage);
int player_1(int readfd, int writefd);
int player_2(int readfd, int writefd);


int main() {
	// Declarations of functions
	void instructions();
	int player_customization_race();
	int player_customization_class();
	Attack_arr selected_deck(int deck_index);


	// Variables
	int descriptor,
	pipe1[2], // array of size 2 to store 2 file descriptors for pipe 1
	pipe2[2]; // array of size 2 to store 2 file descriptors for pipe 2


	// Print blue text and prompt user to press enter to enter the game
	print_blue("Press 'Enter' to entry the game...\n");
	getchar();


	// Clear the terminal screen
	system("clear");


	// Call the instructions function
	instructions();


	// Create two pipes for communication between player 1 and player 2
	if (pipe(pipe1) < 0 || pipe(pipe2) < 0) {


		// Print error message if pipe call fails
		print_red("\nrpg.c: Pipe call error");
```

```
        printf("\nerror: pipe1 = %d pipe2 = %d", pipe(pipe1), pipe(pipe2));

        // Exit the program
        exit(0);
    }

    // Create a child process
    if((descriptor = fork()) < 0) {

        // Print error message if fork call fails
        print_red("Error: Call FORK()");

        // Exit the program
        exit(0);
    }

    // If the descriptor is greater than 0, it is the parent process
    else if(descriptor > 0) {

        // Close the reading end of pipe1 and the writing end of pipe2
        close(pipe1[0]);
        close(pipe2[1]);

        // Call player_1 function with the reading end of pipe2 and the writing end
        of pipe1
        player_1(pipe2[0], pipe1[1]);

        // Close the writing end of pipe1 and the reading end of pipe2
        close(pipe1[1]);
        close(pipe2[0]);
    }

    // If the descriptor is 0, it is the child process
    else {

        // Close the writing end of pipe1 and the reading end of pipe2
        close(pipe1[1]);
        close(pipe2[0]);

        // Call player_2 function with the reading end of pipe1 and the writing end
        of pipe2
        player_2(pipe1[0], pipe2[1]);
```

```
            // Close the reading end of pipe1 and the writing end of pipe2
            close(pipe1[0]);
            close(pipe2[1]);
        }


        // Flush the stdout buffer
        fflush(stdout);


        // Prompt user to press enter to end the program
        getchar();


        return 0;
}
```

- print_colors.h

```
// function to print pink text
void print_pink(char *s)
{
        printf("\033[1;35m%s\033[0m", s);
}


// function to print cyan text
void print_cyan(char *s)
{
        printf("\033[1;36m%s\033[0m", s);
}


// function to print yellow text
void print_yellow(char *s)
{
        printf("\033[1;33m%s\033[0m", s);
}


// function to print blue text
void print_blue(char *s)
{
        printf("\033[1;34m%s\033[0m", s);
}


// function to print green text
void print_green(char *s)
{
        printf("\033[1;32m%s\033[0m", s);
```

```
}


// function to print red text
void print_red(char *s)
{
        printf("\033[1;31m%s\033[0m", s);
}


// function to print white text
void print_white(char *s)
{
        printf("\033[1;29m%s\033[0m", s);
}


// function to reset the text color to default
void print_reset(char *s)
{
        printf("\033[0m%s\033[0m", s);
}
```

- instructions.h

```
/**
* This function prints the game instructions for the players
**/


void instructions() {


        print_blue(" ------------------------------------------------------------------------------\n |");
        print_green(" Operational System ");
        print_blue("|\n +------------------------------------------------------------------------------+\n"
" | |\n |");
        print_green(" Game Instructions ");
        print_blue("|\n | |\n"
" +--------------------------------------------------------------------------+\n"
" | |\n"
" | - You and your opponent will choose your class and race. |\n"
" | |\n"
" | - Each one of you will receive a special deck with 10 power cards related |\n"
" | with the class you choosed. |\n"
" | |\n"
" | - Both of players have initial 50 points of Life. |\n"
" | |\n"
" | - When your life is equal or less than 25% a thread will send you a |\n"
" | motivational message personalized by the class you chose. |\n"
" | |\n"
" | - On your turn you will have to choose between 3 randomly cards of your deck |\n"
" | to attack your opponent. Based on how much Power you have. |\n"
" | |\n"
```

```
" | - Your opponent will turn a d3 dice. If the result is: |\n |");
        print_yellow(" 1 -> The defender will succesfully avoid your attack, not taking any ");
        print_blue("|\n |");
        print_yellow(" damage. ");
        print_blue("|\n |");
        print_yellow(" 2 -> The defender will partially block your attack, receiving 50\% of ");
        print_blue("|\n |");
        print_yellow(" full damage. ");
        print_blue("|\n |");
        print_yellow(" 3 -> The defender failed to avoid your attack, receiving full damage. ");
        print_blue("|\n"
" | |\n"
" | - Win who kills the opponent first! |\n"
" ----------------------------------------------------------------------------\n");


        print_green("\n\nPress 'Enter' to continue...\n");


        getchar();
}
```

- player_1.h

```
/**
* The code defines two structs for storing the attributes of a player and the values of variables passed through pipes.
* It also defines two arrays for the available class and race options.
*
* The player_1() function is the main function for player 1. It asks the player to choose their race
* and class, and then enters a loop where it allows the player to choose and perform an attack and defense move.
* The function also checks the player's life points and prints a motivational message if they are low.
**/


// Define a struct for holding the values of variables passed through pipes
typedef struct pipe_variables {
int damage;
int end_game;
} Pipe_variables;


// Define a struct for holding the attributes of a player
typedef struct player_attr
{
char player_class[10];
char race[8];
float life_status;
} Player_attr;


// Define arrays for the available class and race options
char classes_opt[4][10] = {"Warrior", "Mage", "Assassin", "Cleric"};
char races_opt[3][8] = {"Human", "Elf", "Dwarf"};
```

```c
// Declare thread function for printing a motivational message
void *motivation_message(void *arg);


    // Define the main function for player 1
    int player_1(int readfd, int writefd) {


    // Initialize variables for the game
    int end_game = 0, damage; // Flags for ending the game and storing damage dealt
    Pipe_variables var_pipes = {0, 0}; // Struct for passing values through pipes
    Attack_arr deck; // Array for storing the player's attack moves


    // Clear the terminal
    system("clear");


    // Print a message to the player
    print_green("\n Process 1: Player 1 - Customization - Race\n\n");


    // Ask the player to choose their race
    int player_race_index = player_customization_race();


    // Consume the newline character left in the input buffer after
    // the player enters their choice
    getchar();


    system("clear");
    print_green("\n Process 1: Player 1 - Customization - Class\n\n");


    // Ask the player to choose their class
    int player_class_index = player_customization_class();
    getchar();


    // Write the initial values of the pipe variables to the pipe
    write(writefd, &var_pipes, sizeof(var_pipes));


    // Read the values of the pipe variables from the pipe
    read(readfd, &var_pipes, sizeof(var_pipes));


    // Create a struct for storing the attributes of player 1
    Player_attr player_1;


    // Set the player's race and class based on their chosen indexes
```

```
strcpy(player_1.race, races_opt[player_race_index - 1]);
strcpy(player_1.player_class, classes_opt[player_class_index - 1]);


// Set the player's initial life points
player_1.life_status = 50;


system("clear");


// Loop until either player's life points reach 0 or the end_game flag is set
while(player_1.life_status > 0 || end_game == 1) {


        system("clear");


        // Select the player's attack moves based on their chosen class
        deck = selected_deck(player_class_index);


        // Print a message to the player
        print_green("\n Process 1: Player 1 - Attack time!\n\n");


        // Print the player's attributes
        printf("\n\n Player 1 Attributes:\n\n\tRace: %s\n\tClass: %s\n\tLife Status: %.2f",
        player_1.race, player_1.player_class, player_1.life_status);


        // Allow the player to choose and perform an attack move
        damage = attack_move(deck);


        // Store the amount of damage dealt in the pipe variable struct
        var_pipes.damage = damage;


        // Write the updated values of the pipe variables to the pipe
        write(writefd, &var_pipes, sizeof(var_pipes));


        // Read the updated values of the pipe variables from the pipe
        read(readfd, &var_pipes, sizeof(var_pipes));


        // Store the updated values of the end_game and damage variables
        end_game = var_pipes.end_game;
        damage = var_pipes.damage;


        // If the end_game flag is set, break out of the loop
        if(end_game == 1) {
                break;
```

```
}

// Print a message to the player
print_green("\n Process 1: Player 1 - Defense time!\n\n");

float damage_took;

// Print the player's attributes
printf("Player 1 Attributs:\n\n\tRace: %s\n\tClass: %s\n\tLife Status: %.2f\n\n",
player_1.race, player_1.player_class, player_1.life_status);

// Allow the player to choose and perform a defense move
damage_took = defense_move(damage);

// Print the amount of damage taken by the player
printf("\n\n Player 1 took %.2f of damage", damage_took);

// Reduce the player's life points by the amount of damage taken
player_1.life_status -= damage_took;

// If the player's life points are low, print a motivational message
if(player_1.life_status <= 12.5 && player_1.life_status > 0) {

// Create a new thread for printing the motivational message
pthread_t thread;

// Start the thread and pass the player's class as an argument
pthread_create(&thread, NULL, motivation_message, (void*)player_class_index);

// Wait for the thread to finish
pthread_join(thread, NULL);

}

// Print the player's current life points
printf("\n\n Life bar: %.2f", player_1.life_status);

// Consume the newline character left in the input buffer
getchar();

// If the player's life points have reached 0, set the end_game flag
```

```
            // and write the updated values of the pipe variables to the pipe
            if (player_1.life_status <= 0){
                var_pipes.end_game = 1;
                var_pipes.damage = 0;
                write(writefd, &var_pipes, sizeof(var_pipes));
                break;
            }
        }


        system("clear");


        // If player 1's life points are greater than 0, print a win message
        if(player_1.life_status > 0) {
            print_green("\n\n ***************  PLAYER 1 WIN ***************  ");
        }


        // Print a message to the player informing that the Process 1 is finished
        print_green("\n\n ...end of process 1.\n\n");


        // Return 0 to indicate that the function ran successfully
        return(0);
}
```

- player_2.h

```
/**
* This code is a function that handles the actions of player 2 in a game. It allows player 2 to customize their
* race and class, and then engage in a battle against player 1. The function uses pipes to communicate with
* player 1 and exchange information about the game. The function repeatedly loops through defense and attack
* phases, where player 2 can defend against player 1's attacks and deal damage to player 1. The loop continues
* until either player 2 or player 1's life status reaches 0, at which point the game ends and a winner is declared.
**/


// function to handle player 2's actions
int player_2(int readfd, int writefd) {


        // variables
        int end_game = 0, // variable to track if the game has ended
        damage, // variable to store the amount of damage
        player_race_index, // variable to store the selected race index
        player_class_index; // variable to store the selected class index


        Pipe_variables var_pipes_2 = {0, 0}; // create a Pipe_variables object to store variables for pipe
```

```
communication
Attack_arr deck; // create an Attack_arr object to store attack deck


// read from the pipe the variables stored in var_pipes_2
read(readfd, &var_pipes_2, sizeof(var_pipes_2));


// clear the terminal screen
system("clear");


// print a message in green text
print_green("\n Process 2: Player 2 Customization - Race\n\n");


// call the player_customization_race function to let player 2 choose their race
player_race_index = player_customization_race();
// wait for user to press enter
getchar();


// clear the terminal screen
system("clear");


// print a message in green text
print_green("\n Process 2: Player 2 Customization - Class\n\n");


// call the player_customization_class function to let player 2 choose their class
player_class_index = player_customization_class();


// wait for user to press enter
getchar();


// create a Player_attr object to store player 2's attributes
Player_attr player_2;


// store player 2's selected race in the race attribute
strcpy(player_2.race, races_opt[player_race_index - 1]);


// store player 2's selected class in the player_class attribute
strcpy(player_2.player_class, classes_opt[player_class_index - 1]);


// initialize player 2's life status to 50
player_2.life_status = 50;


// write the variables in var_pipes_2 to the pipe
```

```
write(writefd, &var_pipes_2, sizeof(var_pipes_2));


// keep executing the loop while player 2's life status is greater than 0 and the game has not ended
while(player_2.life_status > 0 || end_game == 1){


// read from the pipe the variables stored in var_pipes_2
read(readfd, &var_pipes_2, sizeof(var_pipes_2));


// call the selected_deck function to get the selected attack deck
deck = selected_deck(player_class_index);


// store the damage received from player 1
damage = var_pipes_2.damage;
// store the value of end_game
end_game = var_pipes_2.end_game;


// if the game has ended, break the loop
if(end_game == 1) {
        break;
}


// print the amount of damage received
printf("\n\n Damage: %d\n\n", damage);


// wait for user to press enter
getchar();


// print a message in green text
print_green("\n Process 2: Player 2 - Defense time!\n\n");


// declare a variable to store the amount of damage taken
float damage_took;


// print player 2's attributes
printf("\n\n Player 2 Attributs:\n\n\tRace: %s\n\tClass: %s\n\tLife Status: %.2f\n\n", player_2.race,
player_2.player_class, player_2.life_status);


// call the defense_move function to calculate the amount of damage taken
damage_took = defense_move(damage);


// print the amount of damage taken
printf(" You took %.2f of damage", damage_took);
```

```
// subtract the damage taken from player 2's life status
player_2.life_status -= damage_took;


// if player 2's life status is less than or equal to 12.5, print a motivational message
if(player_2.life_status <= 12.5 && player_2.life_status > 0) {
        // create a pthread object to handle the motivational message
        pthread_t thread;


        // create the thread
        pthread_create(&thread, NULL, motivation_message, (void*)player_class_index);


        // wait for the thread to finish
        pthread_join(thread, NULL);
}


// print player 2's life status
printf("\n\n Player 2 life bar: %.2f", player_2.life_status);


// wait for user to press enter
getchar();


// if player 2's life status is less than or equal to 0, end the game
if (player_2.life_status <= 0){


        // set the end_game variable to 1
        var_pipes_2.end_game = 1;


        // set the damage variable to 0
        var_pipes_2.damage = 0;


        // write the variables in var_pipes_2 to the pipe
        write(writefd, &var_pipes_2, sizeof(var_pipes_2));


        // break the loop
        break;
}


// clear the terminal screen
system("clear");


// print a message in green text
```

```
        print_green("\n Process 2: Player 2 - Attack time!\n\n");


        // print player 2's attributes
        printf("\n\n Player 2 Attributes:\n\n\tRace: %s\n\tClass: %s\n\tLife Status: %.2f", player_2.race,
        player_2.player_class, player_2.life_status);


        // call the attack_move function to calculate the amount of damage to be dealt
        damage = attack_move(deck);


        // print the amount of damage dealt
        printf("\n\n Damage: %d\n\n", damage);
        // wait for user to press enter
        getchar();


        // store the damage dealt in the damage variable of var_pipes_2
        var_pipes_2.damage = damage;


        // write the variables in var_pipes_2 to the pipe
        write(writefd, &var_pipes_2, sizeof(var_pipes_2));
}


        // clear the terminal screen
        system("clear");


        // if player 2's life status is greater than 0, print a message that player 2 has won the game
        if(player_2.life_status > 0) {
                // print a message in green text
                print_green("\n\n **********  PLAYER 2 WIN! **********  \n\n");
                }


                // print a message in green text
                print_green("\n\n ...end of process 2.\n\n");


                // return 0
                return (0);
        }
```

- decks.h

```
/**
* This code defines a struct for an attack card, which contains the name of the attack, its power, and a
description.
* It also defines a struct for an array of attack cards, and a function for selecting a deck of attack cards
based
* on the player's class. The function has a switch statement that returns a different deck of attack cards
for each
```

```
* class (warrior, mage, assassin, cleric). The decks of attack cards are defined in the code as well.
**/


// Define a struct for an attack card
typedef struct attack
{
char attack_name[20]; // Attack name
int attack_pwr; // Attack power
char description[200]; // Description of the attack
} Attack;


// Define a struct for an array of attack cards
typedef struct attack_arr {
Attack attack_cards[10];
} Attack_arr;


// Define a function for selecting a deck of attack cards
Attack_arr selected_deck(int deck_index) {


int player_class_index = deck_index;


    // Define a deck of attack cards for a Warrior
    Attack_arr warrior_deck;
    warrior_deck.attack_cards[0] = (Attack){.attack_name = "Thundering Strike", .attack_pwr = 10,
    .description = "A powerful attack that causes the ground to shake and the air to crackle with
    electricity."};
    warrior_deck.attack_cards[1] = (Attack){.attack_name = "Mighty Blow", .attack_pwr = 13,
    .description = "An incredibly strong attack that deals massive damage to the enemy."};
    warrior_deck.attack_cards[2] = (Attack){.attack_name = "Shield Bash", .attack_pwr = 9,
    .description = "The warrior uses their shield to bash the enemy, stunning them and leaving them
    vulnerable to further attacks."};
    warrior_deck.attack_cards[3] = (Attack){.attack_name = "Whirlwind Attack", .attack_pwr = 3,
    .description = " A spinning attack that hits all enemies in a wide area around the warrior."};
    warrior_deck.attack_cards[4] = (Attack){.attack_name = "Battle Cry", .attack_pwr = 6, .description
    = "The warrior lets out a mighty roar, inspiring their allies and demoralizing their enemies."};
    warrior_deck.attack_cards[5] = (Attack){.attack_name = "Hamstring Slash", .attack_pwr = 4,
    .description = "The warrior targets the enemy's legs, crippling them and slowing them down."};
    warrior_deck.attack_cards[6] = (Attack){.attack_name = "Sundering Strike", .attack_pwr = 7,
    .description = "A powerful attack that shatters the enemy's armor and weaponry."};
    warrior_deck.attack_cards[7] = (Attack){.attack_name = "Stunning Blow", .attack_pwr = 5,
    .description = "The warrior lands a powerful strike that stuns the enemy, leaving them unable to
    defend themselves."};
    warrior_deck.attack_cards[8] = (Attack){.attack_name = "Taunt", .attack_pwr = 8, .description =
    "The warrior taunts the enemy, drawing their attention and aggression onto themselves."};
    warrior_deck.attack_cards[9] = (Attack){.attack_name = "Shield Wall", .attack_pwr = 15,
    .description = "The warrior raises their shield, forming a barrier that protects them and their
    allies from enemy attacks."};


    // Define a deck of attack cards for a Mage
```

```
Attack_arr mage_deck;
mage_deck.attack_cards[0] = (Attack){.attack_name = "Fireball", .attack_pwr = 10, .description = "The mage summons a ball of flame and hurls it at the enemy, causing burning damage."};
mage_deck.attack_cards[1] = (Attack){.attack_name = "Frost Bolt", .attack_pwr = 13, .description = "The mage shoots a bolt of ice at the enemy, freezing them in place and dealing frost damage."};
mage_deck.attack_cards[2] = (Attack){.attack_name = "Lightning Bolt", .attack_pwr = 9, .description = "The mage unleashes a bolt of electricity at the enemy, shocking them and dealing lightning damage."};
mage_deck.attack_cards[3] = (Attack){.attack_name = "Arcane Blast", .attack_pwr = 3, .description = "The mage unleashes a wave of pure magical energy at the enemy, dealing heavy arcane damage."};
mage_deck.attack_cards[4] = (Attack){.attack_name = "Chain Lightning", .attack_pwr = 6, .description = "The mage summons a bolt of lightning that bounces from the initial target to nearby enemies, dealing lightning damage to all of them."};
mage_deck.attack_cards[5] = (Attack){.attack_name = "Cone of Cold", .attack_pwr = 4, .description = "The mage creates a cone of freezing air that damages and slows all enemies in its path."};
mage_deck.attack_cards[6] = (Attack){.attack_name = "Polymorph", .attack_pwr = 7, .description = "The mage transforms the enemy into a harmless creature, rendering them unable to attack."};
mage_deck.attack_cards[7] = (Attack){.attack_name = "Blink", .attack_pwr = 5, .description = "The mage teleports a short distance, allowing them to quickly reposition themselves and avoid enemy attacks."};
mage_deck.attack_cards[8] = (Attack){.attack_name = "Mana Drain", .attack_pwr = 8, .description = "The mage drains the enemy's magical energy, weakening their spells and abilities."};
mage_deck.attack_cards[9] = (Attack){.attack_name = "Arcane Barrier", .attack_pwr = 15, .description = "The mage creates a barrier of arcane energy that protects them and their allies from incoming attacks."};


// Define a deck of attack cards for a Assassin
Attack_arr assassin_deck;
assassin_deck.attack_cards[0] = (Attack){.attack_name = "Venom Strike", .attack_pwr = 10, .description = "The assassin coats their blade with poison and strikes the enemy, dealing damage over time."};
assassin_deck.attack_cards[1] = (Attack){.attack_name = "Shadow Step", .attack_pwr = 13, .description = "The assassin disappears into the shadows and reappears behind the enemy, surprising them and allowing the assassin to get in a quick attack."};
assassin_deck.attack_cards[2] = (Attack){.attack_name = "Backstab", .attack_pwr = 9, .description = "The assassin sneaks up on the enemy and strikes them from behind, dealing extra damage."};
assassin_deck.attack_cards[3] = (Attack){.attack_name = "Smoke Bomb", .attack_pwr = 3, .description = "The assassin tosses a smoke bomb at the enemy, blinding them and allowing the assassin to escape or set up a surprise attack."};
assassin_deck.attack_cards[4] = (Attack){.attack_name = "Gouge", .attack_pwr = 6, .description = "The assassin rakes their claws across the enemy's eyes, blinding them and leaving them vulnerable to further attacks."};
assassin_deck.attack_cards[5] = (Attack){.attack_name = "Sprint", .attack_pwr = 4, .description = "The assassin temporarily increases their speed, allowing them to quickly close the distance with the enemy or escape from danger."};
assassin_deck.attack_cards[6] = (Attack){.attack_name = "Ambush", .attack_pwr = 7, .description = "The assassin sets up a hidden trap that triggers when the enemy walks over it, dealing damage and stunning them."};
assassin_deck.attack_cards[7] = (Attack){.attack_name = "Disarm", .attack_pwr = 5, .description = "The assassin uses their skills to disarm the enemy, leaving them unable to attack."};
```

```
    assassin_deck.attack_cards[8] = (Attack){.attack_name = "Crippling Blow", .attack_pwr = 8,
    .description = "The assassin strikes a vital spot on the enemy's body, weakening them and
    making them easier to defeat."};
    assassin_deck.attack_cards[9] = (Attack){.attack_name = "Assassin's Mark", .attack_pwr = 15,
    .description = "The assassin marks the enemy, allowing the assassin and their allies to track the
    enemy and deal extra damage to them."};


    // Define a deck of attack cards for a Cleric
    Attack_arr cleric_deck;
    cleric_deck.attack_cards[0] = (Attack){.attack_name = "Healing Word", .attack_pwr = 10,
    .description = "The cleric utters a word of divine power that heals the wounds of an ally."};
    cleric_deck.attack_cards[1] = (Attack){.attack_name = "Bless", .attack_pwr = 13, .description =
    "The cleric blesses an ally, increasing their defenses and making them more resistant to
    damage."};
    cleric_deck.attack_cards[2] = (Attack){.attack_name = "Smite", .attack_pwr = 9, .description =
    "The cleric channels divine energy into their weapon, causing it to glow with holy light and deal
    extra damage to evil enemies."};
    cleric_deck.attack_cards[3] = (Attack){.attack_name = "Divine Shield", .attack_pwr = 3,
    .description = "The cleric creates a shield of divine energy that protects an ally from incoming
    attacks."};
    cleric_deck.attack_cards[4] = (Attack){.attack_name = "Purify", .attack_pwr = 6, .description =
    "The cleric cleanses an ally of any negative effects, such as poison or disease."};
    cleric_deck.attack_cards[5] = (Attack){.attack_name = "Mass Cure", .attack_pwr = 4, .description
    = "The cleric channels divine energy to heal the wounds of all allies within a certain radius."};
    cleric_deck.attack_cards[6] = (Attack){.attack_name = "Turn Undead", .attack_pwr = 7,
    .description = "The cleric channels divine energy to repel undead enemies, causing them to flee
    in fear."};
    cleric_deck.attack_cards[7] = (Attack){.attack_name = "Divine Intervention", .attack_pwr = 5,
    .description = "The cleric calls upon their deity to intervene on their behalf, granting them a
    temporary boost to their abilities."};
    cleric_deck.attack_cards[8] = (Attack){.attack_name = "Judgment", .attack_pwr = 8, .description
    = "The cleric pronounces judgment on the enemy, dealing extra damage to evil enemies and
    reducing the damage they can deal to the cleric and their allies."};
    cleric_deck.attack_cards[9] = (Attack){.attack_name = "Resurrection", .attack_pwr = 15,
    .description = "The cleric brings a fallen ally back to life, restoring them to full health and
    allowing them to continue fighting."};



    switch (player_class_index) {
    case 1:
        return warrior_deck; // Return the warrior deck if the player's class is warrior
    case 2:
        return mage_deck; // Return the warrior deck if the player's class is mage
    case 3:
        return assassin_deck; // Return the warrior deck if the player's class is assassin
    case 4:
        return cleric_deck; // Return the warrior deck if the player's class is cleric
    default:
        break;
    }
}
```

- player_customization_race.h

```
/**
* This code is a function for player customization in a game. It allows the player to choose their
* race from a list of options, and then returns the player's chosen race as an integer value.
* The function also prints messages to the player to inform them of their choice and to handle invalid
input.
**/


int player_customization_race() {
        // Initialize variable for user's choice of race
        int opt, valid_option = 1;


        // Display options for race selection
        print_blue(" Choose your race by its number:\n\n"
        " 1) Human\n"
        " 2) Elf\n"
        " 3) Dwarf\n\n");


        // Loop until user selects a valid option
        while(valid_option) {


                // Prompt user to enter their choice of race
                printf(" Your choice: ");


                // Read user's choice of race
                scanf("%d", &opt);


                // Clear input buffer
                getchar();


                                // Switch case to handle user's choice of race
                switch(opt) {
                case 1: // User chose Human
                        // Display message to confirm user's choice of race
                        print_blue("\n You choose to be a Human!\n");
                        // Set flag to exit loop
                        valid_option = 0;
                        break;
                case 2: // User chose Elf
                        // Display message to confirm user's choice of race
                        print_blue("\n You choose to be an Elf!\n");
                        // Set flag to exit loop
                        valid_option = 0;
                        break;
                case 3: // User chose Dwarf
                        // Display message to confirm user's choice of race
                        print_blue("\n You choose to be a Dwarf!\n");
                        // Set flag to exit loop
```

```
                valid_option = 0;
                break;
            default: // User entered an invalid option
                // Display error message
                print_red("\n\n Please insert a valid option.\n\n");
                break;
            }
        }


        // Return user's choice of race
        return opt;
}
```

- player_customization_class.h

```
/**
* This code is a function for player customization in a game. It allows the player to choose their
* class from a list of options, and then returns the player's chosen class as an integer value.
* The function also prints messages to the player to inform them of their choice.
**/


int player_customization_class() {


        // Declare an integer variable for the player's class choice
        // and a variable for keeping track of whether the player's choice is valid
        int opt, valid_option = 1;


        // Print instructions for the player
        print_blue(" Choose your class by its number: \n\n");
        print_yellow(" 1) Warrior\n");
        print_cyan(" 2) Mage\n");
        print_red(" 3) Assassin\n");
        print_green(" 4) Cleric\n\n");


        // Keep asking the player to enter a valid class until they do so
        while(valid_option) {
        printf(" Your choice: ");
        scanf("%d", &opt);
        switch(opt) {
            case 1:

                // The player chose to be a Warrior
                print_yellow("\nYou choose to be a Warrior!\n");
                // Set the valid_option variable to 0 to exit the loop
                valid_option = 0;
                break;
            case 2:

                // The player chose to be a Mage
                print_cyan("\nYou choose to be a Mage!\n");
```

```
                // Set the valid_option variable to 0 to exit the loop
                valid_option = 0;
                break;
            case 3:

                // The player chose to be an Assassin
                print_red("\nYou choose to be an Assassin!\n");
                // Set the valid_option variable to 0 to exit the loop
                valid_option = 0;
                break;
            case 4:

                // The player chose to be a Cleric
                print_green("\nYou choose to be a Cleric!\n");
                // Set the valid_option variable to 0 to exit the loop
                valid_option = 0;
                break;
            default:

                // The player entered an invalid option
                print_red("\nPlease insert a valid option.\n\n");
                break;
        }
}


    // Consume the newline character left in the input buffer after
    // the player enters their choice
    getchar();


    // Return the player's chosen class
    return opt;
}
```

- motivation_message.h

```
/**
* This code is a thread function that prints a motivational message to player 2 if their life status
* is less than or equal to 12.5. The message printed depends on the player's class. The function takes
* in a parameter "arg" which is expected to be an integer representing the player's class. The function
* then prints the corresponding motivational message and waits for the user to press a key before
continuing.
**/


void *motivation_message(void *arg) {


    // Casting the "arg" parameter to the correct data type "int"
    int class_id = (int) arg;


    print_pink("\n\n\tHere's a Thread!\n\n");
```

```
	// Switch statement to determine which motivational message to print based on player's class
	switch(class_id) {
		// If player is a warrior
		case 1:
			// Print a message in pink text
			print_pink(" The only way to fail is to give up, so keep fighting until the very end Warrior!!!\n\n");
			break;
		// If player is a mage
		case 2:
			// Print a message in pink text
			print_pink("\n\n Though your body may be weak, your mind and spirit are unbreakable. Hold onto those, and you can overcome any obstacle Mage\n\n");
			break;
		// If player is an assassin
		case 3:
			// Print a message in pink text
			print_pink("\n\n Your cunning and stealth may have failed you this time, but they have served you well throughout your life. Hold onto the lessons you have learned, and use them to guide you in your next life Assassin!!!\n\n");
			break;
		// If player is a cleric
		case 4:
			// Print a message in pink text
			print_pink("\n\n Your faith has been your rock, and it will continue to be so even in death. Hold onto it and let it guide you on your journey beyond the veil Cleric!!!\n\n");
			break;
		// If player's class is not recognized
		default:
			// Do nothing
			break;
	}


	// Wait for the user to press a key before continuing
	getchar();


	return NULL;
}
```

5 APRESENTAÇÃO GRÁFICA

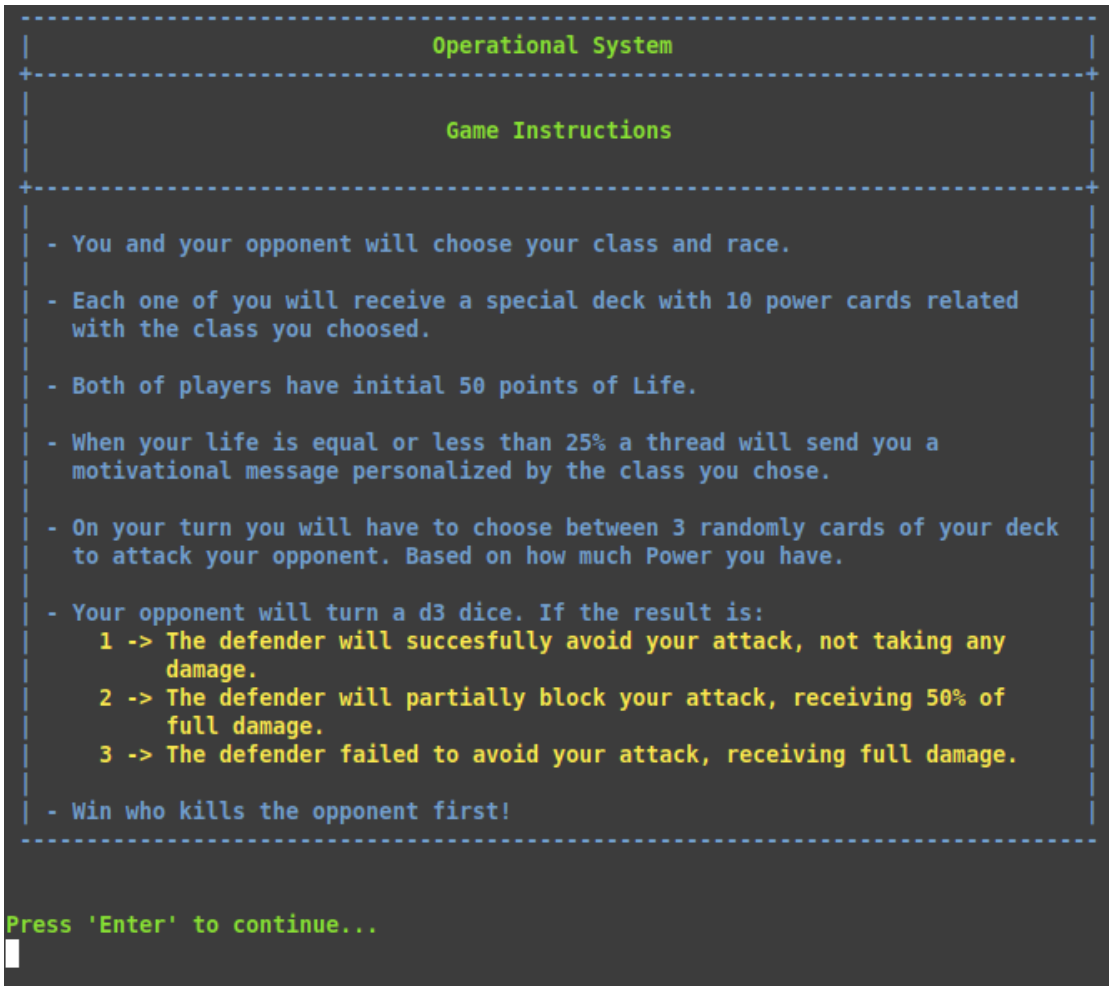

Figura 1: Intruções aos jogadores

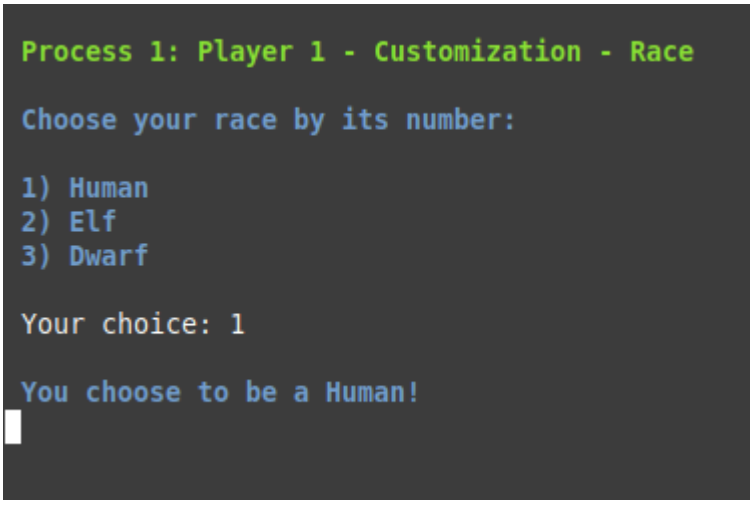

Figura 2: Processo 1 (Pai) customizando o personagem

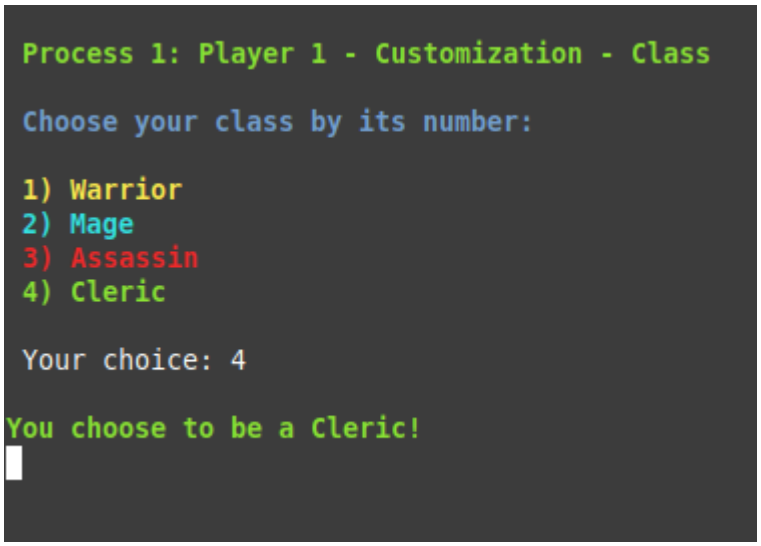

Figura 3: Processo 1 (Pai) customizando o personagem

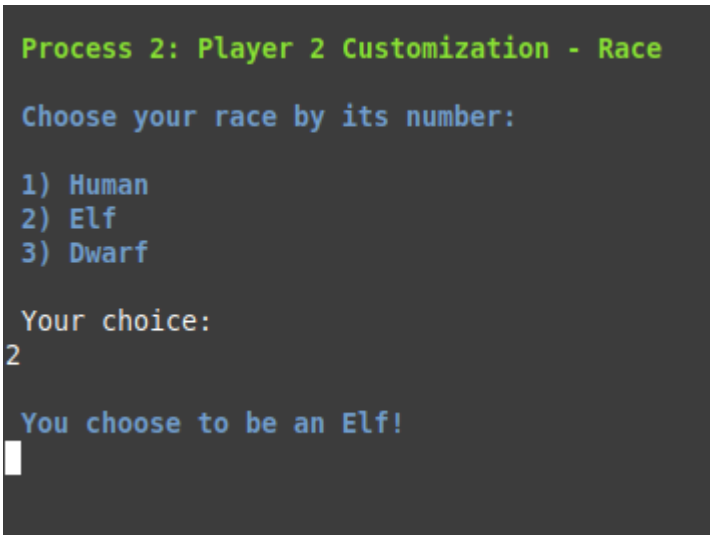

Figura 4: Processo 2 (Filho) customizando o personagem

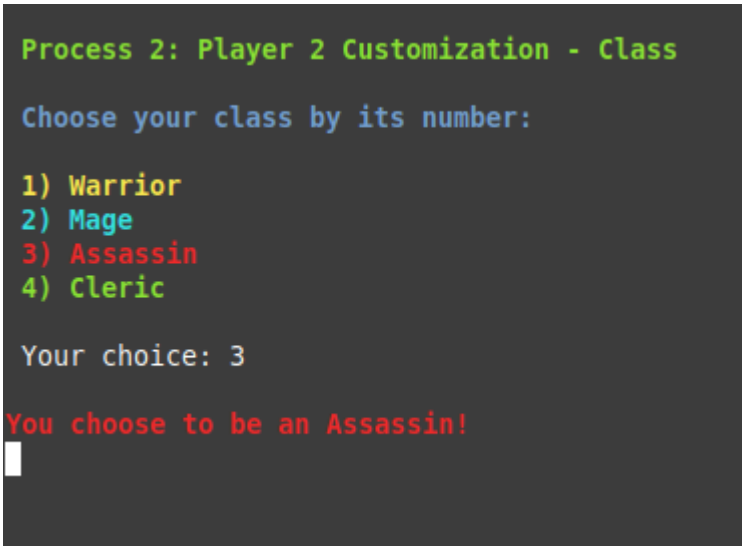

Figura 5: Processo 2 (Filho) customizando o personagem

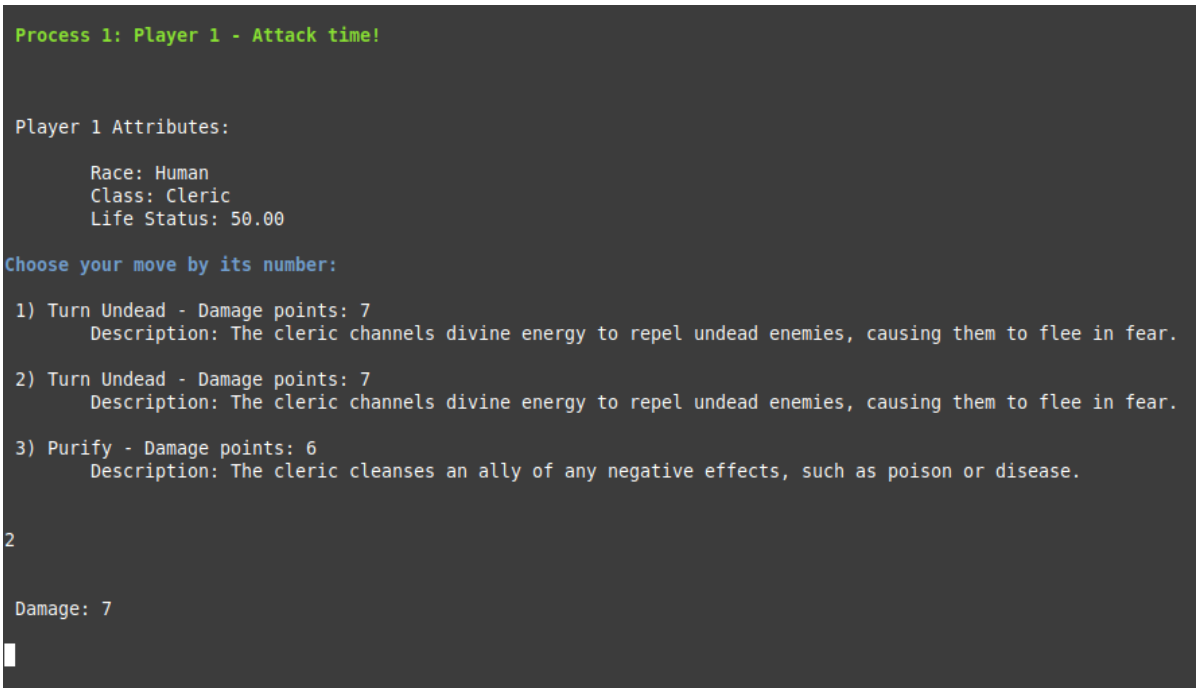

Figura 6: Processo 1 (Pai) ataque (seleciona 1 das 3 opções fornecidas de um deck de 10 cartas)

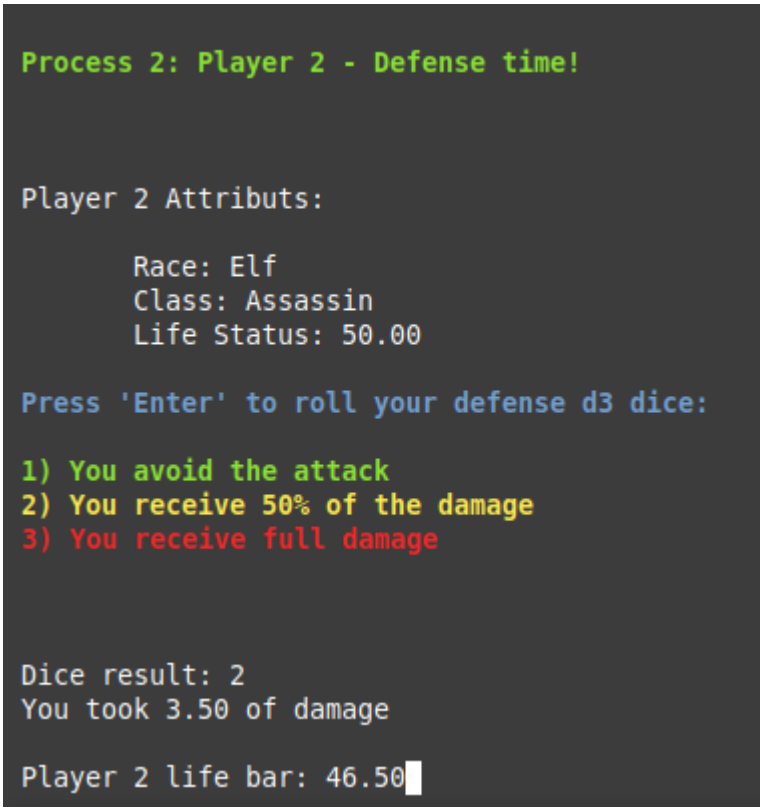

Figura 7: Processo 2 (Filho) defesa (gira um dado de 3 lados para receber *status* de dano)

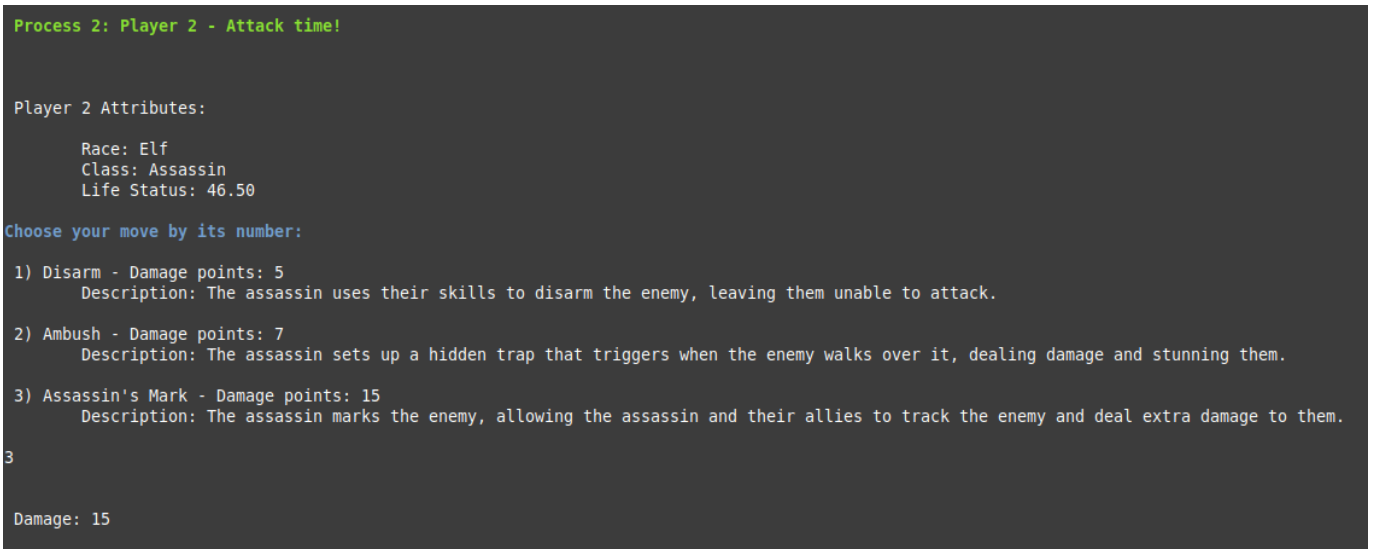

Figura 8: Processo 2 (Filho) ataque (seleciona 1 das 3 opções fornecidas de um deck de 10 cartas)

Figura 9: Processo 1 (Pai) defesa (gira um dado de 3 lados para receber *status* de dano)

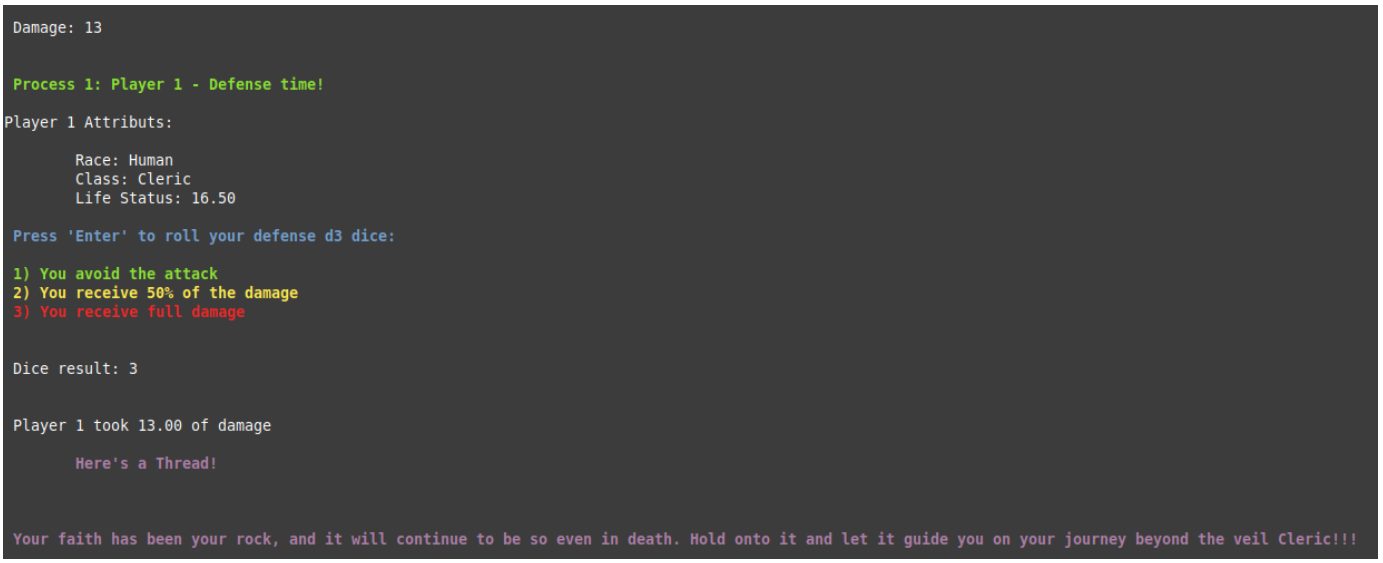

Figura 10: Exemplo de chamada da thread, quando um jogador atinge 25% ou menos de sua vida, a thread envia uma mensagem motivacional (com base na classe que este escolheu) para p jogador

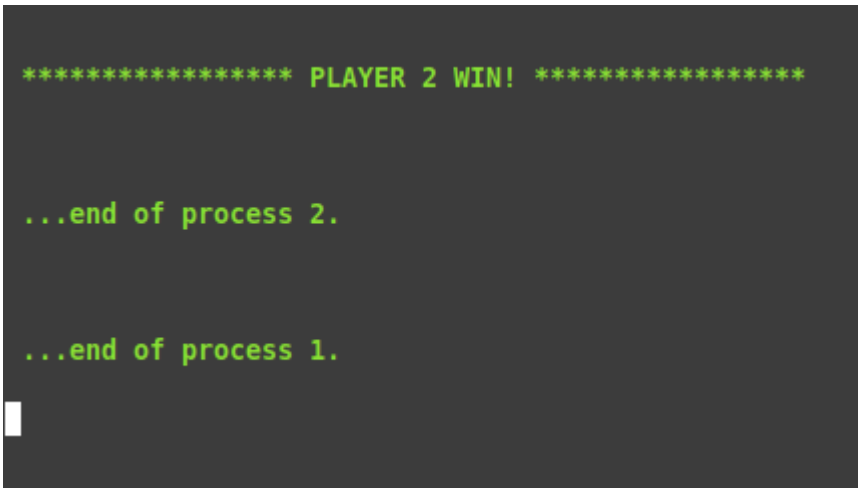

Figura 11: Fim de jogo, será printado o jogador vencedor e o encerramento dos processos

6 CONCLUSÃO

Em resumo, a proposta desse programa é criar um jogo de RPG baseado em turnos que usa pipes e threads para a comunicação e a execução de tarefas. O jogo inclui várias bibliotecas e arquivos de cabeçalho para diferentes funções relacionadas ao jogo, como instruções, customização do jogador, baralhos, movimentos de ataque e defesa e mensagens motivacionais. O código também contém funções de impressão de cores para tornar a interface do usuário mais agradável.

Este projeto foi desenvolvido com o intuito de praticar conceitos vistos na disciplina de Sistemas Operacionais do curso de Ciências da Computação. O programa é escrito em C e usa a biblioteca “pthread.h” para gerenciar *threads*. Isso permite que o jogo execute tarefas de forma paralela e ajude a melhorar a performance do jogo.

O jogo inclui vários arquivos de cabeçalho que contêm funções específicas do jogo, como instruções, customização de raça e classe do jogador, baralhos, movimentos de ataque e defesa e mensagens de motivação. Isso permite que o código principal seja mais limpo e organizado, pois as funções específicas do jogo estão em arquivos separados.

O jogo usa pipes para comunicação entre dois processos. Isso permite que o jogo seja jogado por dois jogadores em diferentes terminais. Cada jogador executa o jogo em um processo separado e os processos se comunicam através de pipes. Abordando conceitos como concorrentemente buscar acesso na CPU.

7 REFERÊNCIAS

[1] TANENBAUM, Andrew S.; BOS, Herbert. Sistemas Operacionais Modernos. 4. ed. rev. São Paulo, Brasil: Pearson Education do Brasil, 2016. 757 p. ISBN 978-85-4301-818-8.

[2] JÚNIOR, João Benedito dos Santos. Linguagens de Programação - Linguagem C. Poços de Caldas, Minas Gerais, Pontifícia Universidade Católica – PUC Minas, 2º semestre de 2020. Disponibilizado pela disciplina de Algoritmos e Estruturas de Dados II.