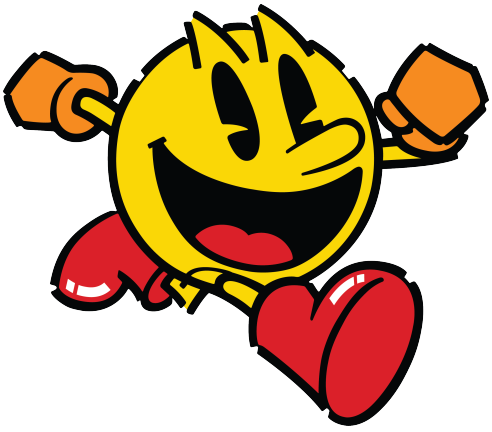
Instituto Superior Técnico

Programação de Sistemas

Projeto: Pacman distribuído

Grupo ??:
Mauro Pungo, Nº87467

Docentes:
Jõao Nuno Silva
Tiago Leão



Índice

1 Arquitetura do projeto

O projeto pode ser dividido em duas arquiteturas principais, o cliente e o servidor. Nas figuras 1 e 2 apresentadas em baixo estão representadas o fluxo(grama) de ações do cliente e do servidor durante uma sessão do programa.

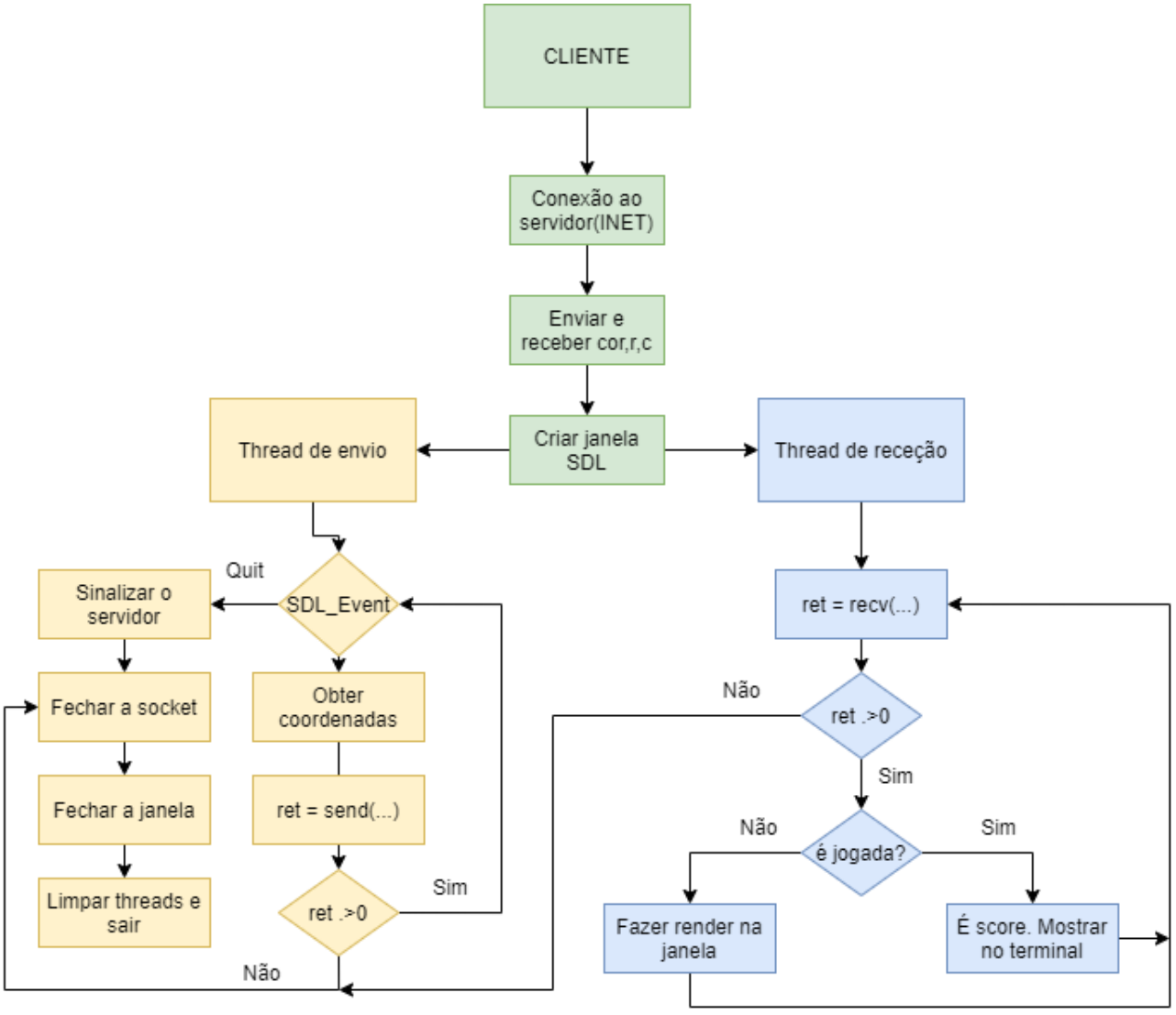


Figura 1: Arquitetura do cliente

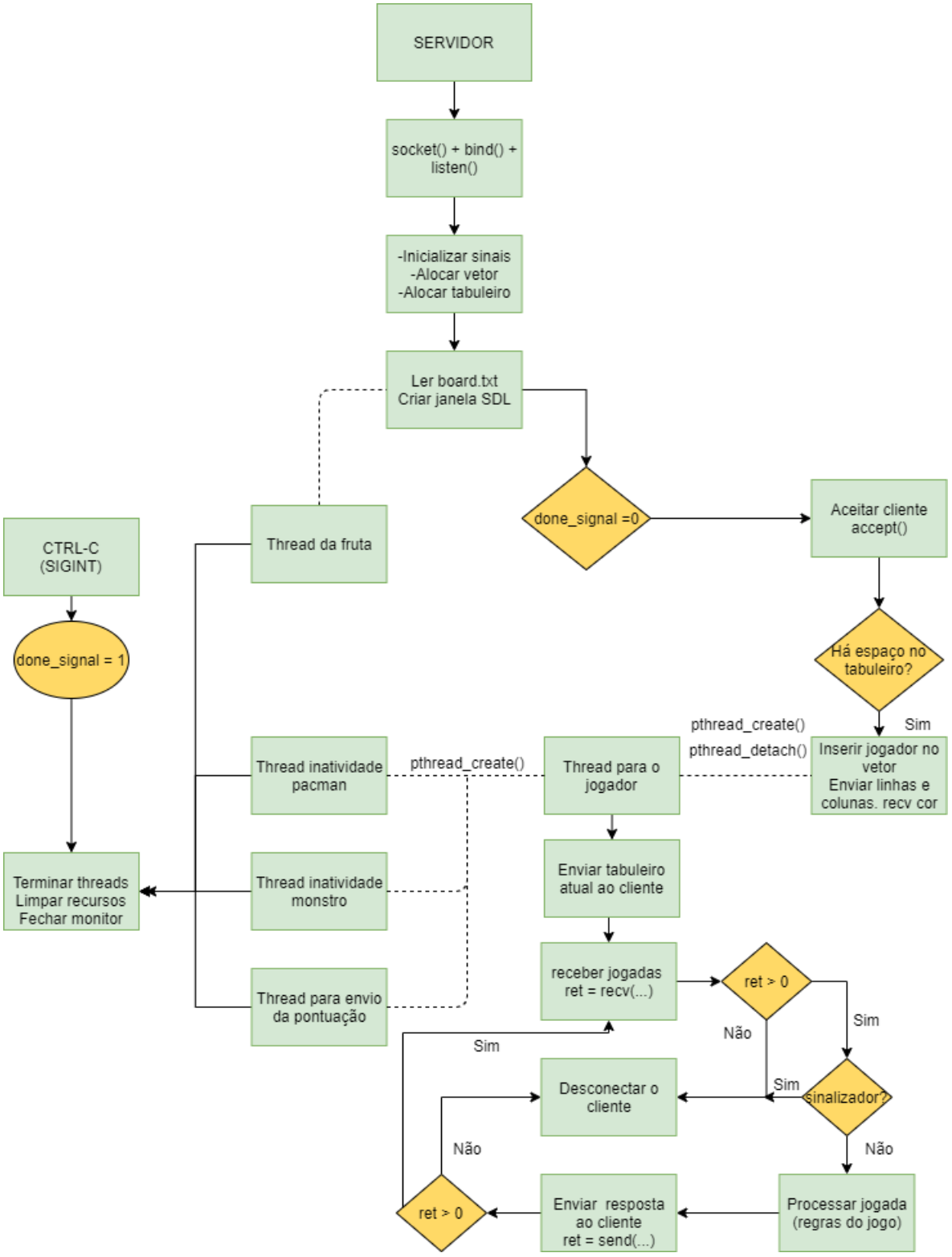


Figura 2: Arquitetura do servidor

2 Organização do código

O código encontra-se dividido em 4 módulos principais, constituídos cada um por diferentes ficheiros.

2.1 Módulo Principal e de Comunicação

Estes módulos representam o ponto de convergência para os outros modulos e o programa se desenrola. É constituído pelos ficheiros *server.c* , *client.c* e *com.c*.

server.c: Onde corre o server e onde se encontra função *receivePlays*. É nesta função que se encontra o ciclo principal de receção e tratamento das jogadas.

client.c: Neste ficheiro se encontram as funções das threads do cliente:

```
void * receiveThread(void *)
```

(função que recebe a atualização do tabuleiro) e

```
void * sendThread(void *)
```

(função que envia as jogadas para o servidor).

com.c: Neste módulo encontram-se todas as funções, ligadas diretamente às regras do jogo ou não, destinadas a transmitir dados ao cliente.

```
40  int send_data(int id,int sock_fd, void* _buff, size_t _n);
41  int send_board(BOARD board,array arr,int id,int sock_fd);
42  int process_play(int id,board_cell * old_play,board_cell *play);
43
44  void send_scoreboard();
45  void *send_score_thread(void *n);
46  void send_initial_data(int id, int sock_fd);
47  void disconnect_client(int id,int sock_fd);
48  void *poll_monster_inactivity(void * poll_time);
49  void *poll_pacman_inactivity(void * poll_time);
50  void *fruit_spawn_thread(void *);
51  void remove_characters(int player_id);
52  void eat_character(board_cell *old_play,board_cell *play,int flag);
53  void update_pos(board_cell *old_play,board_cell *play);
54  void send_cell_to_all(BOARD board,array arr, board_cell *cell);
```

Figura 3: API do módulo de comunicação

```
int send_data(int id,int sock_fd, void* _buff, size_t _n);
```

Esta função faz encapsulmento da função send(), fazendo internamente o tratamento de erros durante o envio. A sua implementação é a seguinte:

```c
int send_data(int id,int sock_fd, void* _buff, size_t _n)
{
    int ret = -1;

    if((ret = send(sock_fd, _buff, _n,0)) < 0)
    {
        puts("Unable to send data to socket");
        disconnect_client(id,sock_fd);
    }
    return ret;
}
```

Figura 4: Função send_data

```c
int send_board(BOARD board,array arr,int id,int sock_fd);
```

Envia o tabuleiro atual aos novos clientes.

```c
int process_play(int id,board_cell *old_play,board_cell *play);
```

Função que processa a jogada recebida, aplicando às respetivas regra de jogo. Envia indiretamente(através de outras funções) a resposta aos clientes conectados.

```c
void send_scoreboard();
```

Envia a pontuação atual a todos os clientes conectados.

```c
void* send_score_thread(void *n);
```

Thread que chama a função **send_scoreboard** a cada 60s.

```c
void remove_characters(int player_id);
```

Remove do tabuleiro os 2 bonecos do cliente identificado pelo argumento player_id.

```c
void disconnect_client(int id,int sock_fd);
```

Procede à desconexão do cliente: faz chamada à função **remove_characters**, decrementa a variável global que guarda o número de clientes conectados e fecha a socket do cliente.

```
void *poll_monster_inactivity(void * poll_time);
```

```
void *poll_pacman_inactivity(void * poll_time);
```

As 2 funções acima servem para verificar continuamente se o respetivo boneco esteve ativo nos últimos 30s. Caso afirmativo, gera uma posição aleatória para o mesmo.

```
void *fruit_spwan_thread(void *);
```

Thread que é acordada quando um boneco come uma fruta. Dorme 2s e de seguida gera uma nova fruta no tabuleiro.

```
void eat_character(board_cell *old_play,board_cell *play,int flag);
```

Função que implementa as seguintes interações entre bonecos de jogadores diferentes: Monstro/Pacman, Pacman/Monstro, SuperPacman/Monstro e Monstro/SuperPacman.

2.2 Módulo de gestão de jogadores

Constituído apenas pelo ficheiro playerManagement.c, é aqui que se encontram as várias funções para a gestão do vetor de jogadores.

```
array init_array(int n_players)
```

Faz a inicialização de um vetor de tamanho **MAXPLAYERS** (constante). Cada posição do vetor é uma estrutura usada para guardar informações sobre o jogador (playerStruct). Devolve o vetor inicializado.

```
array insert_player(array arr ,int player_id,int player_fd, int player_color)
```

Preenche o vetor na posição atribuida ao jogador recém conectado com as informações iniciais necessarias.

searchList: procura na lista o playerNumber que recebe como parâmetro.

```
void free_array()
```

Liberta a memória alocada para o vetor e destrói os *rw_locks* de cada posição.

```
void reset_all()
```

Coloca todas as pontuações do vetor ,ou tabuleiro, a zero, esta função é chamada quando o último cliente desconecta-se.

```
void generate_coords(int id,board_cell *_monster, board_cell *_pacman)
```

Função que gera uma posição aleatória para o monstro e pacman do jogador recém conectado.

```
int check_speed(int id,int SET,int FIRST)
```

Função booleana que verifica se um dos bonecos está em *"excesso de velocidade"*. Caso esteja, a jogada não é processada.

Todas as funções seguinte são essencialmente "setters"e "getters"que permitem ler ou escrever no vetor de forma encapsulada e sincronizada:

```
playerStruct *get_player_ptr(array arr, int id);

clock_t get_char_time(int id, char ch,char);

int get_field(char field,int id);

int get_vect_color(array arr,int id);

int check_speed(int id,int SET,int FIRST);
```

2.3 Módulo de regras de jogo

Este módulo é composto pelo ficheiro board.c, onde estão as funções das regras e lógica de jogo, aquelas que permitem avaliar as jogadas. As funções neste ficheiro mexem diretamente no tabuleiro de jogo, de forma a encapsular o mesmo para os outros módulos. As exceções são as funções que enviam dados para os clientes, estas estão no ficheiro *com.c*

```
96   /*****Abstracao /API do tabuleiro de jogo****/
97   void free_board();
98   BOARD init_board(char *file);
99   BOARD alloc_board(int n_rows,int n_cols);
100
101  void set_cell_id(BOARD *board,int id,int r,int c);
102  void set_cell_type(BOARD* board,char elem,int r,int c);
103  void set_cell_clr(BOARD* board,int color,int r,int c);
104
105  int get_cell_clr(BOARD board,int r,int c);
106  int get_cell_id(BOARD board,int r,int c);
107  char get_cell_type(BOARD board,int r,int c);
108
109  int cell_is_brick(BOARD board,int r,int c);
110  int cell_is_fruit(BOARD board,int r,int c);
111  int cell_is_character(BOARD board,int r,int c);
112  int cell_is_mine(BOARD board,int id,int r, int c );
113  int same_cells(board_cell *c1,board_cell *c2);
114  board_cell* get_cell_ptr(BOARD board,int r,int c);
115  void update_cell(BOARD board,int id,board_cell *pos);
```

Figura 5: API do tabuleiro de jogo

As funções depois da linha 101 são "setters"e "getters", isto é, permitem encapsular os dados do tabuleiro. Também permitem obter ou escrever dados do tabuleiro de forma sincronizada.

`BOARD alloc_board(int n_rows,int n_Cols)`

Função que aloca memória para uma matriz de tipo BOARD (já veremos mais à frente do que se trata) com n_rows linhas e n_cols colunas.

`BOARD init_board(char *file)`

Esta função chama a função acima e lê o ficheiro "board.txt", preenchendo assim no tabuleiro as posições com tijolos.

`int same_cells(board_cell* c1, board_cell *c2`

Função booleana que verifica se as duas células que recebe como argumentos são iguais.

`board_cell* get_cell_ptr(BOARD board,int r,int c)`

Função que devolve um apontador para a celula (r,c) do tabuleiro.

`void update_cell(BOARD board,int id,board_cell *pos)`

Função que atualiza uma célula do tabuleiro com os conteúdos da célula *pos

2.4 Módulo de interface gráfica

É constituído pelo ficheiro **UI_library.c**, são as funções baseadas no SDL2 que permitem a criação e constante atualização da interface gráfica utilizada pelos jogadores. Este módulo foi nos disponibilizado. Foi muito pouco modificado.

2.5 Módulo comum

Este módulo é constituído pelo ficheiro **utils.c** e disponibiliza uma interface entre o cliente e o servidor. As funções e estruturas de dados disponíveis neste módulo estão visíveis tanto para o cliente como para o servidor.

```
void checkMem(void *ptr);
void render_cell(board_cell *);
void serialize(board_cell *cell);
void unserialize(board_cell *cell);
void initSigHandlers(void (*handler)(int));
void parseArgs(unsigned int *color,int *port, int argc, char const *argv[]);
void start_socket(int * sock_fd,const char *address,struct sockaddr_in *_addr, int port, short is_server);
```

Figura 6: API do módulo comum

`void checkMem(void *ptr);`

Verifica se o ponteiro $*ptr$ é valido. Caso contrário, termina-se o programa.

`void render_cell(board_cell *);`

Função que "pinta"uma dada célula na janela SDL com o devido elemento (fruta, pac-man,etc), cor e posição, todas contida no argumento board_cell.

```
void serialize(board_cell *cell);
```

```
void unserialize(board_cell *cell);
```

Estas duas funções acima permitem respetivamente, serilizar e desserializar os dados enviados, ie, chamar as funções **htonl** e **ntohl** no envio e receção respetivamente;

```
void initSigHandlers(void (*handler)(int));
```

Função que permite "apanhar"o sinal **SIGINT** (e tratar com *handler) e ignorar **SIGPIPE**

```
void parseArgs(unsigned int *color,int *port, int argc, char const *argv[]);
```

Função que verifica a validade dos argumentos de compilação do cliente (numero do porto, cor,etc).

```
void start_socket(int * sock_fd,const char *address,struct sockaddr_in *_addr, int por
```

Função que inicializa um socket, tanto para o cliente como para o servidor. Para o servidor também é feito o **bind**.

3 Estruturas de dados

Para além dos tipos de dados básicos disponibilizados pela linguagem C,as estruturas de dados essenciais usadas no projeto foram uma matriz de estruturas para representar o tabuleiro de jogo e um vetor de estruturas para armazenar informação sobre os clientes.

3.1 Estrutura para a matriz

A estrutura que compõe a matriz é a **board_cell** e encontra-se definida no ficheiro *utils.h*. É a mesma estrutura enviada e recebida continuamente pelos clientes. A sua estrutura (no pun intended) é a seguinte:

```c
/* Esta estrutura tem de estar aqui
para o cliente tambem ter acesso*/
typedef struct pt
{
  unsigned short row,col;
}point;

typedef struct bc
{
  point coord;
  char element;
  int color;
  int id;
  pthread_rwlock_t rwlock;
}board_cell;
```

Figura 7: Estrutura principal do programa

Campos:

- ◊ **point color**: ponto com as coordenadas (linha,coluna) da estrutura. Só é util para o cliente pois no servidor usam-se os índices no tabuleiro.
- ◊ **int id**: ID do cliente que efetuou a jogada
- ◊ **pthread_rwlock_t rwlock**: "read-write lock"que permite sincronizar o acesso a uma célula do tabuleiro.
- ◊ **char element**: define o tipo de elemento que ocupa a célula. Valores possíveis para este campo:

```
#define EMPTY 'o'
#define PACMAN 'P'
#define MONSTER 'M'
#define CHERRY 'C'
#define LEMON 'L'
#define BRICK 'B'
#define POWERED 'Z'
#define FINISHED 'F'
```

Figura 8: Valores possíveis para o campo *.element*

3.2 Estrutura para o vetor

Para o vetor, cada posição corresponde à estrutura **playerStruct** (playerManagement.h) cuja estrutura é a seguinte:

```
typedef struct _playerStruct
{
  int player_fd;
  int score;
  int playerID;
  int color;
  unsigned int monster_eaten;
  int speed_pac;
  int speed_monster;
  clock_t first_move[2];
  clock_t monster_time;
  clock_t pacman_time;
  pthread_rwlock_t rwlock;
}playerStruct;
```

Figura 9: Estrutura com informação sobre o cliente

Campos:

- ◊ **int player_fd** : *"file descriptor"* do jogador
- ◊ **int playerID**: Número do jogador, esse número corresponde à ordem em que o jogador entrou num jogo
- ◊ **int score**: Pontuação do jogador
- ◊ **int color**: Cor atribuída, pela linha de comandos, ao jogador. Vai servir para pintar o seu monstro e pacman.
- ◊ **int score**: Pontuação do jogador
- ◊ **int monster_eaten**: variável para contar o número de monstros comido pelo boneco (só faz sentido para o pacman portanto). É reinicializada a cada 2 monstros comidos

- ◊ **int speed_pac e speed_monster**: variável que conta o número de jogadas do boneco. Se alguma destas variáveis for maior do que 2 em menos de 1s, a jogada é descartada (regra da velocidade).
- ◊ **clock_t first_move[2]**: Dois temporizadores ,um para cada boneco, que são acionados quando uma jogada é feita. Serve para a regra da velocidade máxima
- ◊ **pthread_rwlock_t rwlock**: "read-write lock"que permite sincronizar o acesso a uma posição do vetor.

4 Comunicação: dados trocados

A comunicação entre o cliente e o servidor é feito exclusivamente através de sockets. Durante a execução do programa existem vários momentos onde existe troca de informação entre o servidor e o cliente:

No início do jogo, ou seja, sempre que um novo cliente se conecta ao servidor, é lhe enviado o tamanho do tabuleiro. E o mesmo enviar um inteiro que representa a cor dos seus bonecos.

De resto somente estruturas do tipo board_cell são trocadas entre o cliente o servidor.

5 Validação dos dados

A validação dos inteiros trocados inicialmente é feito antecipadamente: verifica-se que a cor é válida e que o número de linhas e colunas não são negativos.

Todas às chamadas às funções send/recv estão protegidas por tratamento de erros caso o resultado seja negativo.

Também é feito a verificação do valor de retorno das chamadas às funções da biblioteca pthread e quando se opera sobre os mutexes, rwLocks e semáforos.

Globalmente o tratamento e validamento dos dados está bem efetuado.

6 Sincronização

Para tentar implementar uma sincronização funcional, começamos por identificar as regiões críticas do código. Definimos como região critica todo código que faz escrita nas 2 variáveis globais, o tabuleiro e o vetor afim de evitar uma leitura ou escrita concorrente.

Utilizei 3 mecanismos de sincronização: semáforos,mutexes e rwLocks. Nas regiões críticas, o seu acesso é bloqueado da seguinte forma:

```
1               void someFunction(Args)
2               {
3                       pthread_mutex_lock(&mutex); //
4
5                           // Some data-race suscepitble code
6                           ...
7                           // end code
8                       pthread_mutex_unlock(&mutex);
9                       return;
10              }
```

É claro que o código funciona igual utilizando os rwLocks.

6.1 rwLocks: Quero ler! Eu também!

O mecanismo predominante que escolhi para sincronizar o acesso ao tabuleiro e vetor foi o rwLocks. Todas as células do vetor e do tabuleiro têm um rwLock inicializado. Este mecanismo permite ter, de forma concorrente, vários acessos em modo de leitura mas um único em modo de escrita de cada vez. Fez-me mais sentido dar preferência aos rwLocks do que aos mutexes pois há várias secções criticas que somente acedem a posições em modo leitura, ou seja não é preciso esperar pelo desbloqueio de um mutex, tornando assim o programa mais rápido e igualmente seguro.

6.2 Mutexes: Yummy, dados partilhados! São exclusivamente meus!

Somente um mutex foi efetivamente utilizado no programa, 3 foram inicializados mas só 1 acabou por ser utilizado. Este mutex foi utilizado para bloquear as variáveis globais que guardam o número de frutas,jogadores,etc a cada instante

6.3 Semáforos: Está vermelho, pare!

Um semáforo foi utilizado para acordar a thread que faz aparecer frutas aleatoriamente, depois de 2s. O semáforo acorda a thread quando um dos bonecos come uma fruta.

7 Implementação do jogo

7.1 Posições aleatórias

Foram implementados 3 mecanismos para gerar posições aleatórias vazias no tabuleiro na função **generate_pos** no ficheiro board.c, de forma a otimizar a velocidade do mesmo. O primeiro método é o aleatório puro, ie, geram-se coordenadas (r,c) até que Tabuleiro[r][c] seja uma posição vazia. Este método é acionado quando pelo menos metade das células do tabuleiro estão vazias.

```
int free_cells = get_empty_cells();

if (free_cells > (n_cells/2)) // Metodo aleatorio
{

    while (get_cell_type(game_board,x,y) != EMPTY)
    {
        x = rand()%(rows-1);
        y = rand()%(cols-1);
    }
    pos->row = x;
    pos->col = y;
}
```

Figura 10: Método aleatório

O segundo método é o pseudo-aleatório: escolhe-se uma linha do tabuleiro aleatoriamente e de seguida percorre-se essa linha linearmente. Repete-se o processo até encontrar uma célula vazia. este método é acionado quando pelo menos 1/5 das células estão vazias.

O terceiro e último método é o linear. Percorre-se o tabuleiro linearmente o tabuleiro até encontrar a primeira célula vazia.

7.2 Lamentamos, estamos cheios!

Durante o processo de conexão de um cliente, depois de feito o accept(), verifica-se que há espaço livre que chegue para o novo cliente. O espaço livre mínimo são 4 células, 2 para os bonecos e 2 para frutas. O espaço livre está contido numa variável global que vai mudando ao decorrer do programa.

```
if((client_fd = accept(server_sock_fd, NULL, NULL)) == -1)
    break;

//Aceita um cliente, mas so se houver espaco no tabuleiro
if((get_empty_cells() < 4) || (playerCounter >= MAXPLAYERS))
{
    puts("No space available on the board, try later!");
    close(client_fd);
    continue;
}
```

Figura 11: Código para expulsar um cliente

7.3 Gameplay:'we play by the rules here'

Todas as regras do jogo são processadas da seguinte forma: vai-se buscar o tabuleiro a posição atual do boneco, a que ainda está na janela e avalia-se o que se encontra na célula que se acabou de receber afim de se perceber qual é o tipo de jogada. Este mecanismo de jogo está implementado na função **play_is_valid**. Esta função recebe um inteiro, cada um para um tipo de jogada diferente. Os valores possíveis são:

```
/* 0 - BOUNCE
   1 - EAT FRUIT
   2 - EAT CHAR
   3 - STEP (normal move)
   -1 - Not valid
*/
int play_is_valid(BOARD board,int playerID,board_cell *old_play,board_cell *new_play)
```

Figura 12: Tipo de jogadas

7.3.1 Let's bounce

No caso do bounce, se a posição recebida for um tijolo ou estiver fora do tabuleiro, calcula-se a posição para a qual se faz bounce. O bounce só é efetivamente concluido se a nova posição for válida, ie, se for uma posição vazia ou uma fruta. Por escolha, não se pode dar bounce para cima de outros bonecos.

7.4 Veloz e furioso

No vetor temos 2 variáveis **int speed_pac** e **int speed_monster** para cada boneco. Para cada um dos boneco, quando se recebe uma jogada e esta variável está a 0, mete-se a 1 e inicia-se um dos temporizadores **first_move**[] (0 para o monstro e 1 para o pacman). Depois quando se recebe a 3a jogada e a diferença com a primeira é inferior a 1s, a jogada é descartada. Caso contrário é processada.

Funcionalidade implementada na funcao **check_speed**, chamada na **process_play**.

7.5 Se a vida te dá limões...

Gerar uma fruta resume-se a gerar uma posição aleatória e escolher aleatoriamente entre uma cereja e um limão, meter tudo numa estrutura board_cell e fazer render. Este processo é feito na função **spawn_fruit**.

7.6 Inatividade: 'Sr, não pode dormir aqui'

Cada cliente tem, no vetor, um temporizador para o pacman e outro para o monstro. Sempre que uma jogada é recebida, no respetivo temporizador mete-se o tempo de receção. Depois nas threads **poll_pacman_inactivity** e **poll_monster_inactivity** vai se verificando se a diferença entre o tempo no momento da verificação e o tempo guardado no vetor é maior ou inferior a 30s. Se for, gera-se uma posição nova para o respetivo boneco.

```
while(!done_signal)
{
    /********** REGRA DA INATIVIDADE (30s) ********************/


    // Calculo do tempo entre a ultima jogada do monstro
    dt = time_ms((clock() - get_char_time(id,MONSTER,1)));

    // Se passaram mais de 30s, mudar para uma posicao aleatoria
    if(dt > WAIT_30_SECONDS)
    {
        board_cell old_pos,new_pos;
        init_cell(&new_pos,get_field(GET_CLR,id),id,MONSTER,0,0);


        // Gerar nova posicao para o jogador
        generate_pos(&new_pos.coord);

        get_curr_pos(id,&old_pos,&new_pos);

        // Enviar a todos a nova posicao
        update_pos(&old_pos,&new_pos);

        // Reinicializar o timer
        set_field(TIME_M,id,RST);
    }
}
```

Figura 13: Implementação da regra da inatividade

7.7 Superpacman: 'Redbull dá-te dentes!'

Como cada elemento está representado por um char no tabuleiro (ver figura 8). Transformar um pacman em superpacman é tão simples como meter o char a 'Z' depois do pacman comer uma fruta.

```
// Comer a fruta e transformar o pacman e superpacman
if (is_pacman(old_play))
    set_char(play,POWERED);
```

Figura 14: Transformar pacman em superpacman

A transformação de superpacman para pacman ocorre na função **eat_character**, depois de comer 2 monstros

```
// Atualizar quantidade de monstros comidos e transformar o superpacman em normal, caso tenha comido 2 monstros
if(eater.element == POWERED)
{
    set_field(SET_EAT,eater_id,INC);

    if( get_field(SET_EAT,eater_id) == 2)
    {
        set_field(SET_EAT,eater_id,RST); // Monstros comidos = 0

        set_char(&eater,PACMAN); /// Transformar SP -> P

        // Apagar character comido e render SP em P
        send_cell_to_all(game_board,player_arr,&eater);

    }
}
```

Figura 15: Transformar superpacman em pacman

7.8 Tabela de pontuação: 'Aposte no Placard!'

O envio da pontuação é feita da seguinte forma no servidor:

```
void *send_score_thread(void *_fd)
{
    while (!done_signal)
    {
        sleep(WAIT_60_SECONDS);
        send_scoreboard();
    }
    pthread_exit(NULL);
    return NULL;
}
```

Figura 16: Enviar pontuação

A receção no cliente é feita graças a um sinalizador:

```
if (to_recv.element == RECEIVE_SCORE)// Recebi pontuacao
    printf("PlayerID: %d  | Score:  %d\n\n",(to_recv.id),(to_recv.color));
else
    render_cell(&to_recv); // Pintar a celula
```

Figura 17: Receber pontuação

7.9 Hora da limpeza

7.9.1 Threads

No cliente, a única memória 'alocadas' são das 2 threads de envio e receção. As threads são fechadas com o pthread_join.

No servidor, todas as threads têm um loop infinito do genero:

```
1          volatile int done_signal = 0; // global
2
3
4          void * thread_func(void *arg)
5          {
6              while(!done_signal)
7              {
8                  do_stuff();
9              }
10
11         }
```

Quando se termina o servidor com um SIGINT, a variável done_signal passa para 1 de forma atómica, terminando os ciclos principais das threads. De seguida procede-se à terminação das threads usando **pthread_cancel()** e **pthread_join()** ou simplesmente a **pthread_exit()** para as threads que foram préviamente "detached"com **pthread_detach()**

7.9.2 Memória alocada: 'heap heap hurray'

A memória alocada para o tabuleiro e vetor é libertada logo depois de se destruir todos os rwLocks nas suas células.

7.10 Game Over!

O jogo só termina quando o servidor for desligado. No entanto quando só há 1 jogador, a sua pontuação é reinicializada.

7.11 Remoção de um cliente: 'thank you,bind again!'

Ambos os lados do programa se podem desconectar e como tal existem dois protocolos, um do lado do servidor e outro do lado do cliente.

7.11.1 Servidor

De forma a saber se um cliente se desconectou avaliamos o valor de retorno da função send(), quando este é menor ou igual a 0 iniciamos o processo de limpeza do cliente. Este processo ocorre na função **disconnect_client**. Primeiro tiram-se os bonecos do jogador do tabuleiro, de seguida fecha-se a socket deste cliente e logo o mesmo é 'removido' do vetor

de jogadores. Na verdade não é removido do vetor pois realocar o tamanho de um vetor é uma operação custosa. O que acontece é que um dos campos é marcado com -1 e sempre que quisermos aceder ao vetor, temos que ver se a posição acedida não tem este campo a -1.

De seguida a thread é fechada deixando assim de estar á espera de mensagens deste jogador. Este é o caso de uma desconexão 'acidental'. A desconexão 'civilizada' é explicada a seguir.

7.11.2 Cliente

Quando o cliente fecha a janela de jogo, envia uma estrutura board_Cell em que o campo .element = 'F' e assim o server sabe que tem de remover o cliente. No client.c a socket é fechada, o tabuleiro também e os recursos são libertados. Outro caso onde pode existir uma desconexão é quando o cliente perde a ligação ao servidor. Isto é avaliado através do return da função recv, e a socket é fechada.

8 Um olhar crítico

Globalmente sinto que a implementação do projeto foi um sucesso e cumpre todos os requisitos avaliados. A única coisa que não me pareceu estar a 100% é a regra das 2 células por segundo. Quanto à arquitetura do projeto, está funcional mas bastante simples. Uma abordagem mais sofisticada seria a utilização de 2 sockets por clientes a fim de aumentar de se ter o pacman e monstro completamente concorrentes. A sincronização teve especial cuidado, sendo que se recorreu a algumas ferramentas do gcc para verificar que a todo custo é impossível ocorrer deadlock nem livelock.

9 Referências

- threads e sincronização: https://docs.oracle.com/cd/E53394_01/pdf/E54803.pdf
- https://github.com/angrave/SystemProgramming/wiki
- Unix man page
- StackOverflow.com