PSCOM - Relatório do Projeto Grupo 1

1. Compilar e correr ao programa (makefile):

Para compilar todos os nossos programas, basta correr “make” na diretoria principal. Para compilar cada programa individualmente, basta correr “make serv”, “make log” ou “make apl”, respectivamente. Existe também a opção de compilar e correr cada programa individualmente com: “make run_s”, “make run_l” e “make run_a” respectivamente. (Sempre executado na diretoria principal).

2. Estruturas de comunicação e sincronização:

2.1. JMMapl e JMMserv:

A comunicação entre JMMapl e JMMserv é feita de dois modos. Quando o programa JMMapl é corrido, este tenta abrir um socket datagram em `extern DATAGRAM create_sock(void)`. Se houver sucesso e algum dos servidores estiver vinculado, o cliente poderá fazer envio de comandos através da estrutura `coms_t`:

```
typedef struct     //estrutura para enviar comandos e seus argumentos
{
  commands_t command;

  union
  {
    char name[4];
    char move[6];
    unsigned int j;
    unsigned short int n;
  } arg1;

  union
  {
    unsigned short int n;
    time_t t;
  } arg2;

} coms_t;
```

O servidor JMMserv poderá responder apropriadamente, mesmo sem o JMMapl ter qualquer jogo iniciado. Para garantir a comunicação que permite o funcionamento do jogo, o cliente e o servidor (JMMapl e JMMserv) comunicam por socket stream.

De modo a garantir que o JMMapl não fique indefinidamente à espera de uma resposta do servidor, adicionámos um timeout com recurso a:

```
setsockopt(strmsock.sd_stream, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));
```

Isto permite definir o tempo máximo que o socket datagram poderá estar aberto para efetuar leituras (do lado do JMMapl). Em caso de o tempo definido na estrutura `struct timeval timeout` ser ultrapassado, na próxima tentativa de leitura, o cliente deverá receber uma mensagem de erro. O mesmo se sucede com o socket datagram mas para este será necessário utilizar o descriptor de ficheiro adequado `datsock.sd_datagram`.

O servidor possuí também um mecanismo de timeout que tenta continuamente comunicar com o cliente para verificar se ele pretende fazer uma jogada (o timeout em si é feito da maneira anteriormente referida). Após executar essa tentativa, verifica se o cliente ainda tem tempo para jogar. Se essa verificação determinar que o jogador pode jogar, a tentativa de leitura volta a ser feita. Caso não seja esse o caso, a conexão é encerrada.

2.2. JMMserv e JMMlog:

A única comunicação entre JMMserv e JMMlog acontece através de uma queue (`#define JMMLOGQ "/JMMLOGQ"`) criada pelo JMMlog. O JMMserv quando tenta guardar um jogo, verifica se esta queue está aberta, se não estiver, este lança o JMMlog e envia o jogo a ser guardado. O envio do registo de jogo acontece através da estrutura `rjg_t`;

```
typedef struct
{
  int nd;     /* nível de dificuldade do jogo */
  char nj[4]; /* nome do jogador (3 caracteres) */
  int nt;     /* número de tentativas usadas */
  time_t ti;  /* estampilha temporal início do jogo */
  time_t tf;  /* estampilha temporal fim do jogo */

} rjg_t; /* estrutura de um registo de jogo */
```

2.3. JMMlog e JMMapl:

A comunicação entre JMMlog e JMMapl acontece através de socket datagramas (`#define JMMLOGSD "/tmp/JMMLOGS"`).

O JMMapl envia os comandos para o JMMlog através de uma estrutura do tipo `coms_t` (como explicado na secção 2.1. JMMapl e JMMserv).

E o JMMlog responde ao JMMapl de duas formas dependendo do comando executado:

1. Caso o comando seja “ltc” responde com a estrutura `log_single_tab_t`

```
typedef struct
{
  rjg_t tb[10];
  int tb_n_games;
  int tb_diff;

} log_single_tab_t;
```

Esta estrutura contém apenas uma das tabelas de jogo. Caso o jogador (JMMapl) peça as duas tabelas, o JMMlog faz dois envios, um com cada tabela. Decidimos criar esta estrutura para garantir que apenas a informação necessária é enviada, evitando enviar as duas tabelas em casos em que apenas uma é necessária.

2. Caso o comando seja quer “rtc”, “trh” ou for inválido, a resposta é uma string no buffer `char buffer_send[100]` (do lado de JMMlog) e, `char rtc_msg[MAX_RCV_SIZE]` e `char trh_msg[MAX_RCV_SIZE]` (do lado de JMMapl, para rtc e trh respetivamente).

3. Sincronização Interna:

3.1. JMMapl:

O JMMapl não contém mecanismos de sincronização interna pois é single thread.

3.2. JMMserv:

O JMMserv é composto por 3 threads:

- Thread “main”: inicializações de mecanismos de sincronização e socket datagrama; definição do handling de sinais; criação da thread “acceptgames”; processamento de datagramas.
- Thread “acceptgames”: inicialização de socket stream; criação de conexões com novos clientes e estabelecimento do jogo, guarda as informações de jogo numa estrutura como esta:

```
typedef struct      //informação sobre cliente a enviar para criar um novo jogo
{
  int sd;           //descriptor do cliente
  int game_number;  //número do jogo que lhe foi associado
  int sock_stream;  //socket stream para receber jogadas
  coms_t buffer_s;  //buffer com informações de jogo recebidas pela thread acceptgames

} new_game_info;
```

Passa posteriormente esta informação a uma nova thread “gameinstance” criada pela thread “acceptgames”. Fica neste processo até ser dada ordem de término e rejeita conexões de novos jogadores caso o limite `NJMAX` já tenha sido atingido.

- Thread “gameinstance”: thread que gere cada jogo sendo responsável por inicialização do jogo com a seguinte estrutura:

```
typedef struct // variável que guarda o jogo
{
  char correct_sequence[MAX_SEQUENCE_SIZE]; // sequência correta definida no início do jogo
  rjg_t log;                                // log do jogo para ser enviado depois ser armazenado
  unsigned short int n_char;                // número de caracteres na sequência
  char player_move[MAX_SEQUENCE_SIZE];      // sequência enviada pelo utilizador
  unsigned short int np;                    // número de letras certas no sítio certo
  unsigned short int nb;                    // número de letras certas no sítio errado<
  game_state_t game_state;                  // estado do jogo = {ONGOING,PLAYER_WIN,PLAYER_LOST}
  rules_t game_rules;                       // regras do jogo
  double elapsed_time;
  int sd;                                   //socket descriptor associado a este jogador
  struct sockaddr_un player_addr;           //address do cliente
  socklen_t addr_len;                       //comprimento do address

} game_t;
```

Processa também jogadas através duma conexão stream. Esta thread por ter várias threads irmãs que acedem a recursos partilhados possuí 2 mecanismos de sincronização.

O primeiro é um mutex (`pthread_mutex_t rules_mutex`) que bloqueia o acesso à variável onde estão definidas as regras (`rules_t global_game_rules`).

```
typedef struct
{
  int maxj; // número máximo de jogadas
  int maxt; // tempo máximo de jogo (em minutos)

} rules_t;
```

Isto deve-se ao facto de clientes poderem mudar as regras do jogo antes de criarem o seu jogo. Essa escrita não deve ser feita enquanto outros jogadores estão a tentar iniciar o seu jogo visto que a thread “gameinstance” lê os valores das regras e usa os mesmos para inicializar o jogo do cliente que lhe foi atribuído pela thread “acceptgames”.

O segundo é outro mutex (`pthread_mutex_t save_mutex`), responsável por impedir que duas threads “gameinstance” diferentes tentem iniciar o processo JMMlog caso este ainda não esteja inicializado, evitando assim instâncias duplicadas do processo.

3.3. JMMlog:

O JMMlog tem duas threads, a “main()” que recebe e processa datagramas e inicia a thread “queue_handler()” que está responsável pela queue.

Como ambas estas threads lêem e escrevem no ficheiro de dados, (`#define JMMLOG "../JOGOS.LOG"`), usámos um mutex associado ao seu acesso: `pthread_mutex_t file_mux`. Este mutex também é usado para garantir que quando o programa vai ser interrompido, quer por um signal quer pelo comando “trh”, o programa apenas fecha após garantir que ninguém está a usar o ficheiro.