universidade de aveiro

Sistemas Operativos

Simulação de jogo de futebol

Trabalho n.º 2

2020/2021

Diogo Pereira Henriques da Cruz, nº 98595, P2
Mariana Cabral Silva Silveira Rosa, nº 98390, P5

Conteúdo

Introdução

O 2º trabalho prático da unidade curricular Sistemas Operativos consiste em simular um jogo de futebol entre um grupo de amigos, através de um programa escrito na linguagem C, sob forma de tabela, onde cada número corresponderá a um estado atribuído. Sendo um jogo de futebol, claro que vamos precisar de jogadores (*players*), guarda-redes (*goalies*) e um árbitro (*referee*). Estas serão as entidades do nosso programa.
Assim, o objetivo principal deste trabalho é desenvolver um *script* na linguagem C que nos permita simular várias partidas de futebol (cada posição será aleatória), onde cada entidade (jogador, árbitro ou guarda-redes) é tratado como um processo independente, logo vamos ter de os sincronizar através de semáforos e memória partilhada (mais tarde iremos abordar como funcionará).

Abordagem ao problema

Em primeiro lugar, para conseguirmos resolver este projeto, precisamos de compreender todas as regras do jogo e o código fornecido pelo docente. Foi-nos disponibilizado um *zip* **semaphore_soccergame.tgz** com todos os *scripts* necessários, onde só precisamos alterar três destes: **semSharedMemPlayer.c ,semSharedMemGoalie.c e semSharedMemReferee.c** onde irão, cada um deles, mudar os estados do jogo e afetar o estado dos players, goalies e referee, respetivamente.

Resumidamente a maneira que este programa deve correr é a seguinte: Há um total de 10 jogadores de campo, 3 guarda-redes e 1 árbitro e vão chegando aleatoriamente. À medida que vão chegando ficam à espera de jogadores suficientes para formar uma equipa. Para ser formada são precisos 5 jogadores (4 players e 1 goalie), logo, como há 10 jogadores de campo e apenas 8 lugares para formar as 2 equipas, os últimos 2 players a chegar vão ficar ficar de fora e não participam no jogo (vão chegar atrasados). O mesmo acontece com os goalies, sendo que existem 3 e apenas 2 podem jogar, o último a chegar não irá jogar nesta partida. Mais tarde, quando houver jogadores suficientes para formar uma equipa, a última pessoa a chegar que vai completar a equipa de 5 jogadores vai formar a sua equipa e será o capitão. Este processo acontece 2 vezes, uma por cada equipa e algum tempo depois mais 4 jogadores passam para a equipa 1 e mais 4 para a equipa 2. Durante este processo não é necessário a presença do árbitro, por isso, caso o árbitro ainda não tenha chegado ao campo, os jogadores vão ficar todos à espera dele. O árbitro é o único que pode iniciar o jogo e sendo assim ele sempre que chega espera que as equipas se formem e, uma vez que estejam formadas, ele inicia o jogo. Após as equipas jogarem um bocado, o jogo termina (e consequentemente o programa) quando o árbitro apita declarando final do jogo.

Dentro do zip **semaphore_soccergame.tgz** encontramos 2 ficheiros distintos, o ficheiro **run** onde vamos correr o *script* principal do jogo no programa **probSemSharedMemSoccerGame.c** e o ficheiro **src** que é onde se encontram os códigos principais para a resolução do nosso problema; dentro deste último existe o **sharedDataSync.h** *[Fig. 1 e Fig. 2]* onde encontramos a criação dos semáforos e os seus IDS que iremos utilizar em conjunto com outras condições para chegar a uma solução final. Estes semáforos encontram-se dentro da estrutura **SHARED_DATA** assim como a estrutura **FULL_STAT** que será mais à frente abordada, sendo assim possível os seus dados serem usados em todos os códigos.

```c
/**
 *  \brief Definition of <em>shared information</em> data type.
 */
typedef struct
        { /** \brief full state of the problem */
          FULL_STAT fSt;

          /* semaphores ids */
          /** \brief identification of critical region protection semaphore - val = 1 */
          unsigned int mutex;
          /** \brief identification of semaphore used by players to wait for forming team teammate - val = 0 */
          unsigned int playersWaitTeam;
          /** \brief identification of semaphore used by goalies to wait for forming team teammate - val = 0 */
          unsigned int goaliesWaitTeam;
          /** \brief identification of semaphore used by players and goalies to wait for the match to start - val = 0 */
          unsigned int playersWaitReferee;
          /** \brief identification of semaphore used by players and goalies to wait for the match to end - val = 0 */
          unsigned int playersWaitEnd;
          /** \brief identification of semaphore used by referee to wait for teams to be formed - val = 0  */
          unsigned int refereeWaitTeams;
          /** \brief identification of semaphore used by players and goalies to acknowledge team registration - val = 0  */
          unsigned int playerRegistered;

        } SHARED_DATA;
```

Fig. 1 - ID de cada semáforo

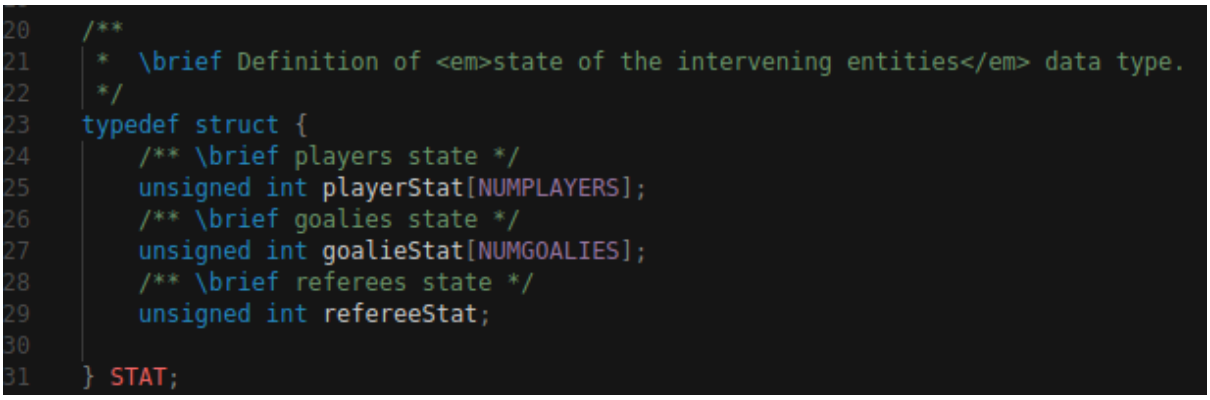
```c
/** \brief number of semaphores in the set */
#define SEM_NU                  7

#define MUTEX                   1
#define PLAYERSWAITTEAM         2
#define GOALIESWAITTEAM         3
#define PLAYERSWAITREFEREE      4
#define PLAYERSWAITEND          5
#define REFEREEWAITTEAMS        6
#define PLAYERREGISTERED        7

#endif /* SHAREDDATASYNC_H_ */
```

Fig. 2 - Número de semáforos

```c
/**
 *  \brief Definition of <em>full state of the problem</em> data type.
 */
typedef struct
{   /** \brief state of all intervening entities */
    STAT st;

    /** \brief total number of players */
    int nPlayers;

    /** \brief total number of goalies */
    int nGoalies;

    /** \brief total number of referees */
    int nReferees;

    /** \brief number of players that already arrived */
    int playersArrived;
    /** \brief number of goalies that already arrived */
    int goaliesArrived;
    /** \brief number of players that arrived and are free (no team) */
    int playersFree;
    /** \brief number of goalies that arrived and are free (no team) */
    int goaliesFree;

    /** \brief id of team that will be formed next - initial value=1 */
    int teamId;

} FULL_STAT;
```

No script **probStruct.h** encontramos mais dados importantes para a resolução do projeto, é-nos dado a estrutura **FULL_STAT** *[Fig. 3]* que guarda nela outra estrutura **STAT** e onde também contém variáveis que serão importantes na estruturação do código e no funcionamento de todos os semáforos.

Fig. 3 - Estrutura FULL_STAT



```
/**
 *  \brief Definition of <em>state of the intervening entities</em> data type.
 */
typedef struct {
    /** \brief players state */
    unsigned int playerStat[NUMPLAYERS];
    /** \brief goalies state */
    unsigned int goalieStat[NUMGOALIES];
    /** \brief referees state */
    unsigned int refereeStat;

} STAT;
```

Fig. 4 - Estrutura STAT

No *script* **probConst.h** encontramos as informações importantes, como os parâmetros que representam a quantidade de pessoas que existem e as necessárias para o jogo decorrer:

Número total de jogadores (NUMPLAYERS)	10
Número total de guarda-redes (NUMGOALIES)	3
Número de árbitros (NUMREFEREES)	1
Número de jogadores por equipa (NUMTEAMPLAYERS)	4
Número de guarda-redes por equipa (NUMTEAMGOALIES)	1

No mesmo *script*, encontramos também os diferentes estados que cada entidade vai percorrer ao longo da execução do programa:

Para o refere:

Estado 0 "ARRIVING": o árbitro inicia neste estado e permanece até chegar ao campo de futebol.

Estado 1 "WAITING_TEAMS": uma vez que o árbitro chegue ele fica neste estado até receber confirmação que ambas as equipas estão formadas

Estado 2 "STARTING_GAME": uma vez formadas o árbitro passa para este estado onde pode iniciar o jogo quando quiser.

Estado 3 "REFEREEING": o árbitro passa para este estado quando pretende iniciar o jogo e ficará neste até decidir acabar o jogo.

Estado 4 "ENDING_GAME": quando o árbitro quiser ele pode terminar o jogo mudando para este estado e o programa acaba.

Estados	Referee
ARRIVING	0
WAITING_TEAMS	1
STARTING_GAME	2
REFEREEING	3
ENDING_GAME	4

Para players/goalies:

Estado 0 "ARRIVING": todos os jogadores iniciam neste estado e permanecem até chegarem ao campo de futebol.
Estado 1 "WAITING_TEAM": uma vez que um player/goalie chega tem possibilidade de pertencer a qualquer uma das equipas (ou seja, tem lugar numa das 2 possíveis equipas a ser formadas) ele passa para o estado 1 e fica à espera que hajam jogadores suficientes para formar uma equipa.
Estado 2 "FORMING_TEAM": o jogador a chegar que perceba que pode formar uma equipa (quer seja player ou goalie), por exemplo, se for um player e este notar que estão outros 3 jogadores (no mínimo) e 1 goalie já à espera (no mínimo) ou chegar um goalie que note que há 4 players (no mínimo) para formar a sua equipa então vai passar do estado 0 para este estado, tornando-se o capitão de uma nova equipa a ser formada e vai chamar os outros 4 jogadores para pertencerem à sua equipa, não podendo estes depois fazer parte de outra equipa. Só 2 pessoas irão passar por este estado pois cada equipa só vai ter 1 capitão que irá ser responsável pela formação da equipa.
Estado 3 "WAITING_TEAM_1": Após o 1º capitão reunir os 5 jogadores, estes todos vão representar a equipa 1 e ficam à espera neste estado até que o árbitro dê início ao jogo.
Estado 4 "WAITING_TEAM_2": Após o 2º capitão reunir os 5 jogadores, estes todos vão representar a equipa 2 e ficam à espera neste estado até que o árbitro dê início ao jogo.
Estado 5: "PLAYING_1": Uma vez que o árbitro chegue ao seu estado 2 "STARTING GAME" e inicie o jogo os jogadores da equipa 1 passam do estado 3 para este estado sinalizando que estão a jogar pela equipa 1 e ficarão assim até o árbitro terminar o jogo.
Estado 6: "PLAYING_2": Uma vez que o árbitro chegue ao seu estado 2 "STARTING GAME" e inicie o jogo os jogadores da equipa 2 passam do estado 4 para este estado sinalizando que estão a jogar pela equipa 2 e ficarão assim até o árbitro terminar o jogo.
Estado 7 "LATE": quando um jogador chega e já não tem possibilidade de pertencer a nenhuma das 2 equipas passa logo para este estado que indica que não vai participar naquele jogo.

Estados	**Player/Goalie**
ARRIVING	0
WAITING_TEAM	1
FORMING_TEAM	2
WAITING_START_1	3
WAITING_START_2	4
PLAYING_1	5
PLAYING_2	6
LATE	7

Assim, após reunir um conjunto de informações sobre como podemos alcançar com sucesso o *script* (as regras do jogo de futebol, o que é que cada entidade faz, como jogar), podemos perceber o "mecanismo" por detrás do projeto e começar a trabalhar no código.

Estruturação do código

Para resolver este projeto, o docente implementou sete semáforos que iremos explicar pormenorizadamente cada um. Mas antes disso, achamos por bem explicar do que se trata e para que são úteis.

Os semáforos, tais como os nossos na vida real (no trânsito), ajudam a sincronizar o que os rodeia: isto é, na vida real evita colisões entre veículos diferentes, aplicado ao código sincroniza a comunicação entre processos ou *threads* .

Para além do uso dos semáforos, foi definido um *mutex*, utilizado inúmeras vezes no código, responsável por aceder à região crítica de um processo, uma exclusividade mútua no acesso à região.

Semáforos aplicados

Quando começamos a trabalhar neste projeto, o docente durante uma aula prática sugeriu que construíssemos uma tabela, com a finalidade de nos ajudar a desenvolver o raciocínio, para entrarmos "mais no problema", para alcançar uma solução. E assim foi:

Semáforo	**Entidade que vai fazer Down**	**Em que função vai fazer down**	**#Downs**	**Entidade que vai fazer Up**	**Em que função vai fazer up**	**#Ups**
playersWaitTeam	Players	playerConstituteTeam()	1	Players	playerConstituteTeam()	3
goaliesWaitTeam	Goalie	goalieConstituteTeam()	1	Players	playerConstituteTeam()	1
playersWaitReferee	Goalie + Players	waitReferee()	2	Referee	StartGame()	1
playersWaitEnd	Players+Goalies	playUntilEnd()	2	Referee	endGame()	1
refereeWaitTeams	Referee	waitForTeams()	1	Goalie + Player (capitão)	GoaliesConstituteTeam() + playerConstituteTeam()	2, um por equipa
playersRegistered	Goalie/Players	goaliesContituteTeam()+ playersContituteTeam()	4	Goalie + Player	GoalieConstituteTeam() + playerConstituteTeam()	1

Grande parte do código neste projeto já veio disponibilizado pelo docente, a parte mais desafiadora foi decifrar o que estava escrito em cada script para saber onde começar a encontrar a solução.

Foi-nos pedido para escrever só dentro de certas funções que faziam o código correr de forma correta, vamos explicar as mesmas posteriormente. Todas as outras importantes foram inicializadas e geradas previamente pelo docente.

Explicação do código semSharedMemoryReferee.c

Há 5 funções neste código em que podemos trabalhar, cada uma vai utilizar um estado diferente do Referee.

Função ***arrive()***:

Dentro da região crítica do mutex vamos atualizar o estado do árbitro por um daqueles que estão atribuídos a esta entidade. Começamos com o estado **ARRIVING**, onde indicamos que é nesta função que o árbitro está para sair para o campo de futebol. Para aceder à variável responsável pela mudança de estado do árbitro (*RefereeStat*), que se encontra na estrutura STAT, temos de aceder às estruturas fornecidas pelo professor: começando primeiro por utilizar o ponteiro para a memória partilhada SHARED_DATA, de seguida acedemos à estrutura FULL_STAT para daí chegar à estrutura STAT e atualizar a variável. Este ponteiro (sh) vai ser usado muito frequentemente pois é o que permite aceder aos dados todos guardados em memória partilhada. Após mudar o estado temos também o passo importante de atualizar e guardar o novo estado e podemos assim sair da região crítica do mutex.

```
137   */
138   static void arrive ()
139   {
140       if (semDown (semgid, sh->mutex) == -1) {
141           perror ("error on the up operation for semaphore access (RF)");
142           exit (EXIT_FAILURE);
143       }
144
145       sh->fSt.st.refereeStat = ARRIVING;
146       saveState(nFic, &sh->fSt);
147
148       if (semUp (semgid,   sh->mutex) == -1) {
149           perror ("error on the up operation for semaphore access (RF)");
150           exit (EXIT_FAILURE);
151       }
152
153       usleep((100.0*random())/(RAND_MAX+1.0)+10.0);
154
155   }
156
```

Fig. 5- Função arrive()

Função ***waitForTeams()***:

Dentro da região crítica do mutex vamos atualizar o estado do árbitro agora para o estado 1 definido por **WAITING_TEAMS** e é nesta função que o árbitro chega e inicia a sua espera até que ambas as equipas estejam formadas. Mais uma vez temos de guardar o estado atual do árbitro e temos tudo para poder sair da região crítica do mutex. De seguida vamos usar pela 1ª vez um semáforo que irá ser o *refereeWaitTeams*, relembro que, estando este semáforo dentro da estrutura da memória partilhada temos de usar o ponteiro *sh* para aceder ao semáforo. O árbitro vai dar down 2 vezes neste semáforo, sinalizando que está à espera que as duas equipas se formem. Uma vez que cada equipa se formar, cada uma delas vai dar up a este semáforo, que também será feito 2 vezes, uma por capitão.

```
163     */
164    static void waitForTeams ()
165    {
166        if (semDown (semgid, sh->mutex) == -1) {
167            perror ("error on the up operation for semaphore access (RF)");
168            exit (EXIT_FAILURE);
169        }
170
171        sh->fSt.st.refereeStat = WAITING_TEAMS;
172        saveState(nFic, &sh->fSt);
173
174        if (semUp (semgid, sh->mutex) == -1) {
175            perror ("error on the up operation for semaphore access (RF)");
176            exit (EXIT_FAILURE);
177        }
178
179        /* espera equipa 1 e 2*/
180        for (int i = 1; i < NUMTEAMGOALIES-1; i++) {
181            if (semDown (semgid, sh->refereeWaitTeams) == -1) {
182                perror ("error on the up operation for semaphore access (RF)");
183                exit (EXIT_FAILURE);
184            }
185        }
186    }
187
```

Fig. 6- Função waitForTeams()

Função ***startGame()***:

```
static void startGame ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.refereeStat = STARTING_GAME;
    saveState(nFic, &sh->fSt);


    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }


    for (int i = 0; i < NUMPLAYERS; i++) {
        if (semUp(semgid, sh->playersWaitReferee) == -1)
        { /* leave critical region */
            perror("error on the up operation for semaphore access (RF)");
            exit(EXIT_FAILURE);
        }
    }


}
```

Fig. 7- Função startGame()

Dentro da região crítica do mutex vamos atualizar o estado do árbitro agora para o estado 2 definido por **STARTING_GAME** e é nesta função que o árbitro prepara-se para iniciar a partida. Mais uma vez temos de guardar o estado atual do árbitro e podemos sair da região crítica do mutex. De seguida vamos dar *up* ao semáforo *playersWaitReferee* 2 vezes, sinalizando que cada uma das 2 equipas já se formou e o árbitro pode dar início ao jogo quando quiser.

Função ***play()***:

Dentro da região crítica do mutex vamos atualizar o estado do árbitro agora para o estado 3 definido por **REFEREEING** e é nesta função que o árbitro reconhece que as equipas estão formadas e inicia o jogo quando quiser. Mais uma vez temos de guardar o estado atual do árbitro e temos tudo para poder sair da região crítica do mutex.

Ficará aqui até decidir terminar o jogo.

```
229    */
230    static void play ()
231    {
232        if (semDown (semgid, sh->mutex) == -1) {
233            perror ("error on the up operation for semaphore access (RF)");
234            exit (EXIT_FAILURE);
235        }
236
237        sh->fSt.st.refereeStat = REFEREEING;
238        saveState(nFic, &sh->fSt);
239
240        if (semUp (semgid, sh->mutex) == -1) {
241            perror ("error on the up operation for semaphore access (RF)");
242            exit (EXIT_FAILURE);
243        }
244
245        usleep((100.0*random())/(RAND_MAX+1.0)+900.0);
246    }
247
```

Fig. 8- Função play()

Função ***endGame()***:

Dentro da região crítica do mutex vamos atualizar o estado do árbitro agora para o estado 4 definido por **ENDING_GAME** e é nesta função que o árbitro vai dar o apito final para o fim do jogo. Mais uma vez temos de guardar o estado atual do árbitro e temos tudo para poder sair da região crítica do mutex. De seguida vamos dar *up* do semáforo *playersWaitEnd* dez vezes, uma por cada jogador na partida, para avisar cada um individualmente que o jogo terminou. Este semáforo foi *down* nos códigos do *semSharedMemPLayer.c* e do *semSharedMemGoalie.c* para cada jogador que entrava na partida.

```
254     */
255     static void endGame ()
256     {
257         if (semDown (semgid, sh->mutex) == -1) {
258             perror ("error on the up operation for semaphore access (RF)");
259             exit (EXIT_FAILURE);
260         }
261 
262         sh->fSt.st.refereeStat = ENDING_GAME;
263         saveState(nFic, &sh->fSt);
264 
265         if (semUp (semgid, sh->mutex) == -1) {
266             perror ("error on the up operation for semaphore access (RF)");
267             exit (EXIT_FAILURE);
268         }
269 
270         for (int i = 0; i < NUMPLAYERS; i++) {
271             if (semUp (semgid, sh->playersWaitEnd) == -1) {
272                 perror ("error on the up operation for semaphore access (RF)");
273                 exit (EXIT_FAILURE);
274             }
275         }
276 
277     }
278 
```

Fig. 9- Função endGame()

Explicação do código semSharedMemoryPlayer.c

Existem 4 funções neste código, cada uma vai utilizar um estado diferente do jogador.

Função ***arrive()***:

Dentro da região crítica do *mutex* vamos atualizar o estado do jogador e guardá-la, tal como fizemos no Referee, a indicar que está a chegar.

```
139    */
140   static void arrive(int id)
141   {
142       if (semDown (semgid, sh->mutex) == -1)  {
143           perror ("error on the up operation for semaphore access (PL)");
144           exit (EXIT_FAILURE);
145       }
146
147       sh->fSt.st.playerStat[id] = ARRIVING;
148       saveState(nFic, &sh->fSt);
149
150       if (semUp (semgid, sh->mutex) == -1) {
151           perror ("error on the down operation for semaphore access (PL)");
152           exit (EXIT_FAILURE);
153       }
154
155       usleep((200.0*random())/(RAND_MAX+1.0)+50.0);
156   }
157
```

Fig. 10- Função arrive()

Função ***playerConstituteTeam()***:

```
static int playerConstituteTeam (int id)
{
    int ret = 0;

    if (semDown (semgid, sh->mutex) == -1)  {                                                   /* enter critical region */
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.playersFree++;      //os que vao entrar nas equipas
    sh->fSt.playersArrived++;   //os que chegam

    if (sh->fSt.playersArrived <= NUMPLAYERS-2){
        if (sh->fSt.playersFree >= NUMTEAMPLAYERS && sh->fSt.goaliesFree >= NUMTEAMGOALIES){ //capitao player
            sh->fSt.st.playerStat[id] = FORMING_TEAM;
            sh->fSt.playersFree = sh->fSt.playersFree-4;
            sh->fSt.goaliesFree--;
            saveState(nFic, &sh->fSt);

            for(int i=1; i < NUMTEAMPLAYERS; i++){
                if (semUp (semgid, sh->playersWaitTeam) == -1) {
                    perror ("error on the down operation for semaphore access (PL)");
                    exit (EXIT_FAILURE);
                }
            }

            if (semUp (semgid, sh->goaliesWaitTeam) == -1) {
                perror ("error on the down operation for semaphore access (PL)");
                exit (EXIT_FAILURE);
            }

            for(int i=1; i < NUMTEAMPLAYERS+1; i++){
                if (semDown (semgid, sh->playerRegistered) == -1) {
                    perror ("error on the down operation for semaphore access (PL)");
                    exit (EXIT_FAILURE);
                }
```

Fig. 11- Função playerConstituteTeam() - parte 1

Como foi explicado anteriormente, uma equipa é constituída por 5 elementos, quatro jogadores e um guarda-redes. Esta função é direcionada para distinguir entre os players que vão formar as equipas, aqueles que vão ficar só em estado de espera e aqueles que vão chegar tarde. Quando o último jogador a chegar (o capitão) é um player e é este que vai formar a equipa. Logo, sabemos que tem que estar já à espera, no mínimo, 3 jogadores e 1 guarda-redes.

Assim, implementamos um contador que vai ser incrementado para o número de jogadores livres (sem equipa formada, *playersFree*) e os que chegaram (*playersArrived*). Dentro de um ciclo de condição ***if*,** enquanto o número de jogadores de campo que já tenham chegado for menor que o número de jogadores total que podem estar nas duas equipas (8) , verifica se pelo menos 3 players e 1 guarda-redes estão livres. Se esta condição for verdadeira, atribuímos ao jogador a ser avaliado o atributo **FORMING_TEAM,** ou seja, ele vai ser o capitão e vai fechar uma equipa, digamos assim.

Depois, alteramos o número de jogadores livre, subtraímos quatro inteiros, uma vez que 4 jogadores vão para uma equipa, existem menos 4 jogadores livres. De igual

modo, subtraímos uma unidade ao número de guarda-redes livres (sem equipa). E por fim guardamos o estado de todos.

Posteriormente, elaboramos um ciclo *for*, que irá dar *up* 3 vezes (número de jogadores por equipa, excluindo o capitão) ao semáforo *playersWaitTeam* indicando que já não estão mais à espera de equipa, o semáforo estava *down* inicialmente para todos os jogadores que ainda estavam há espera.

Do mesmo modo, ativamos o semáforo *goaliesWaitTeam* pois o guarda-redes selecionado para a equipa já não está mais há espera de equipa.

Por último neste ciclo de condição *if*, fazemos 4 *downs* ao semáforo *playersRegistered*, uma vez que já registámos 4 jogadores novos.

Atribuímos à variável de retorno, *ret*, o teamID de cada equipa, e incrementamos cada vez que passa naquele ciclo, ou seja, no 1º loop a equipa 1 terá o teamID 1 e no 2º loop, a equipa 2 terá o teamID 2.

```
            ret = sh->fSt.teamId;
            sh->fSt.teamId++; // teamId do capitao 1 e 2

        }else{  // jogador normal
            sh->fSt.st.playerStat[id] = WAITING_TEAM;
            saveState(nFic, &sh->fSt);
        }

    }else{
        sh->fSt.st.playerStat[id] = LATE;
        saveState(nFic, &sh->fSt);
        sh->fSt.playersFree--;
    }
```

Fig. 12- Função playerConstituteTeam() - parte 2

É de notar que este loop a verificar se existe o número de jogadores mínimo para formar equipa só vai ser usado 2 vezes, uma por cada entidade que chega e tem oportunidade de formar equipa.

Para os restantes casos, elaboramos um *else*, onde vamos tratar dos casos onde um jogador está simplesmente à espera de formar equipa, atribuímos-lhe o estado WAITING_TEAM e guardamos o seu estado.

Voltando ao primeiro ciclo de condição mencionado, formamos uma condição *else*, que indica se o jogador chegou depois de todos os amigos já incluídos em equipas, então temos de lhe atribuir o estado LATE e diminuímos o número de jogadores livres.

```
    if (semUp (semgid, sh->mutex) == -1) {                                    /* exit critical region */
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (sh->fSt.st.playerStat[id] == WAITING_TEAM){
        if (semDown (semgid, sh->playersWaitTeam) == -1) {
            perror ("error on the down operation for semaphore access (PL)");
            exit (EXIT_FAILURE);
        }

        ret = sh->fSt.teamId; // teamId do jogador

        if (semUp (semgid, sh->playerRegistered) == -1) {
            perror ("error on the down operation for semaphore access (PL)");
            exit (EXIT_FAILURE);
        }
    }

    /* acorda equipa 1 e 2*/
    if (sh->fSt.st.playerStat[id] == FORMING_TEAM){
        if (semUp (semgid, sh->refereeWaitTeams) == -1) {
            perror ("error on the up operation for semaphore access (RF)");
            exit (EXIT_FAILURE);
        }
    }

    return ret;
}
```

Fig. 13- Função playerConstituteTeam() - parte 3

Com todos os estados já definidos, damos *up* no nosso *mutex* para sair da região crítica.

De seguida atualizamos semáforos, para cada jogador que estiver no estado **WAITING_TEAM** damos *down* ao semáforo *playersWaitTeam*, indicando que esse jogador está à espera de equipa. E se assim for, atribuímos à variável de retorno *ret* o teamID do jogador e damos *up* ao semáforo *playerRegistered*, indicando que o jogador já está registado numa equipa.

Para "ativar" ambas as equipas formadas, ou seja, para os jogadores com o estado **FORMING_TEAM** (os capitães), ativamos (damos *up*) ao semáforo *refereeeWaitTeams* para informar ao árbitro que as equipas estão prontas a jogar e devolvemos a variável de retorno.

Função ***waitReferee()***:

Esta função tem como principal objetivo esperar pelo árbitro para iniciar o jogo e alterar os estados dos jogadores de ambas as equipas do estado **FORMING_TEAM** para o estado **WAITING_START_1** para a equipa 1 e **WAITING_START_2.** Através do acesso à região crítica, verificamos com uma condição *if* qual das equipas estamos a lidar e alterámos os estados. Por último, damos *down* ao semáforo *playersWaitReferee.*

```
static void waitReferee (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (team == 1){ // muda para o estado 3
        sh->fSt.st.playerStat[id] = WAITING_START_1;
        saveState(nFic, &sh->fSt);
    }

    if (team == 2){ // muda para o estado 4
        sh->fSt.st.playerStat[id] = WAITING_START_2;
        saveState(nFic, &sh->fSt);
    }

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (semDown(semgid, sh->playersWaitReferee) == -1){
    perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }


}

/**
```

Fig. 14- Função waitReferee()

Função ***playUntilEnd()***:

```
static void playUntilEnd (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (team == 1){ // muda para estado 5
        sh->fSt.st.playerStat[id] = PLAYING_1;
        saveState(nFic, &sh->fSt);
    }
    if (team == 2){ // muda para estado 6
        sh->fSt.st.playerStat[id] = PLAYING_2;
        saveState(nFic, &sh->fSt);
    }


    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->playersWaitEnd) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

}
```

Fig. 15- Função waitReferee()

A função **playUntilEnd()** procede de maneira muito semelhante à função descrita anteriormente, mas tem como principal objetivo esperar que o árbitro acabe a partida. Fazemos isso, de igual modo, verificando de qual das equipas estamos a falar e alterámos do estado **WAITING_TO_START_1** ou **WAITING_TO_START_2** para **PLAYING_1** ou **PLAYING_2**.

Explicação do código semSharedMemoryGoalie.c

Este código é bastante semelhante ao *semSharedMemoryPlayer.c* onde também existem 4 funções iguais, cada uma vai utilizar um estado diferente do goalie.

Função ***arrive()***:

Dentro da região crítica do *mutex* vamos atualizar o estado do goalie e guardá-la, tal como fizemos no Referee e no Player, a indicar que está a chegar.

```
137     */
138    static void arrive(int id)
139    {
140        if (semDown (semgid, sh->mutex) == -1)  {
141            perror ("error on the up operation for semaphore access (GL)");
142            exit (EXIT_FAILURE);
143        }
144
145        sh->fSt.st.goalieStat[id] = ARRIVING;
146        saveState(nFic, &sh->fSt);
147        /* TODO: insert your code here */
148
149        if (semUp (semgid, sh->mutex) == -1) {
150            perror ("error on the down operation for semaphore access (GL)");
151            exit (EXIT_FAILURE);
152        }
153
154        usleep((200.0*random())/(RAND_MAX+1.0)+60.0);
155    }
```

Fig. 16- Função arrive()

Função ***goalieConstituteTeam()***:

De forma igual ao *playerConstituteTeam()* esta função é direcionada para distinguir entre os guarda-redes que vão formar as equipas, aqueles que vão ficar só em estado de espera e aqueles que vão chegar tarde. Quando o último jogador a chegar (o capitão) é um goalie é este que vai formar a equipa. Para isto sabemos que tem de estar já à espera, no mínimo, 4 jogadores de campo.

Assim, implementamos um contador que vai ser incrementado para o número de guarda-redes livres (sem equipa formada, *goaliesFree*) e os que chegaram (*goaliesArrived*). De igual forma como fizemos com os players, dentro de um ciclo de condição *if*, vemos enquanto o número de guarda-redes que já tenham chegado for

menor que o número de goalies total que podem estar nas duas equipas (3) , verifica se pelo menos 4 players estão livres. Se esta condição for verdadeira, atribuímos ao jogador a ser avaliado o atributo **FORMING_TEAM,** ou seja, ele vai ser o capitão e vai fechar uma equipa.

Depois, alteramos o número de jogadores livres e de guarda-redes livres de maneira igual que fizemos no *playersConstituteTeam*() no código do Player. E por fim guardamos o estado de todos.

Posteriormente, elaboramos um ciclo *for*, que irá dar *up* 4 vezes (número de jogadores de campo na equipa) ao semáforo *playersWaitTeam* indicando que já não estão mais à espera de equipa e fazemos também mais 4 *downs* ao semáforo *playersRegistered*, uma vez que já registámos 4 jogadores novos.

Da mesma maneira atribuímos à variável de retorno, *ret*, o teamID de cada equipa, igual ao código do outro script.

```
171     */
172    static int goalieConstituteTeam (int id)
173    {
174        int ret = 0;
175
176        if (semDown (semgid, sh->mutex) == -1)  {
177            perror ("error on the up operation for semaphore access (GL)");
178            exit (EXIT_FAILURE);
179        }
180
181        sh->fSt.goaliesFree++;      //os q vao entrar nas equipas
182        sh->fSt.goaliesArrived++;   //os q chegam
183
184        if (sh->fSt.goaliesArrived < NUMGOALIES){
185            if (sh->fSt.playersFree >= NUMTEAMPLAYERS){ //capitao goalie
186                sh->fSt.st.goalieStat[id] = FORMING_TEAM;
187                sh->fSt.playersFree = sh->fSt.playersFree-4;
188                sh->fSt.goaliesFree--;
189                saveState(nFic, &sh->fSt);
190
191                for(int i=1; i < NUMTEAMPLAYERS+1; i++){
192                    if (semUp (semgid, sh->playersWaitTeam) == -1) {
193                        perror ("error on the down operation for semaphore access (PL)");
194                        exit (EXIT_FAILURE);
195                    }
196
197                    if (semDown (semgid, sh->playerRegistered) == -1) {
198                        perror ("error on the down operation for semaphore access (PL)");
199                        exit (EXIT_FAILURE);
200                    }
201                }
202                ret = sh->fSt.teamId;
203                sh->fSt.teamId++; // teamId do capitao 1 e 2
204
```

Fig. 17- Função goaliesConstituteTeam() - parte 1

Para os restantes casos, elaboramos um *else*, onde vamos tratar dos casos onde um goalie está simplesmente à espera de formar equipa, atribuímos-lhe o estado **WAITING_TEAM** e guardamos o seu estado.

Voltando ao primeiro ciclo de condição mencionado, formamos uma condição *else*, que indica que o goalie chegou depois de todos os amigos já incluídos em equipas, então temos de lhe atribuir o estado LATE e diminuímos o número de guarda-redes livres.
Com todos os estados já definidos, damos *up* do *mutex* para sair da região crítica.

```
204
205             }else{  // guarda redes normal
206                 sh->fSt.st.goalieStat[id] = WAITING_TEAM;
207                 saveState(nFic, &sh->fSt);
208             }
209
210         }else{
211             sh->fSt.st.goalieStat[id] = LATE;
212             saveState(nFic, &sh->fSt);
213             sh->fSt.goaliesFree--;
214         }
215
216         if (semUp (semgid, sh->mutex) == -1) {
217             perror ("error on the down operation for semaphore access (GL)");
218             exit (EXIT_FAILURE);
219         }
```

Fig. 18- Função goaliesConstituteTeam() - parte 2

De seguida atualizamos semáforos, para cada guarda-redes que estiver no estado **WAITING_TEAM** damos *down* ao semáforo *goaliesWaitTeam*, indicando que esse guarda-redes está à espera de equipa. E se assim for, atribuímos à variável de retorno *ret* o teamID do jogador e damos *up* ao semáforo *playerRegistered*, indicando que o jogador já está registado numa equipa.

```
220
221     if (sh->fSt.st.goalieStat[id] == WAITING_TEAM){
222         if (semDown (semgid, sh->goaliesWaitTeam) == -1) {
223             perror ("error on the down operation for semaphore access (PL)");
224             exit (EXIT_FAILURE);
225         }
226
227         ret = sh->fSt.teamId; // teamId do goalie
228
229         if (semUp (semgid, sh->playerRegistered) == -1) {
230             perror ("error on the down operation for semaphore access (PL)");
231             exit (EXIT_FAILURE);
232         }
233     }
234
235     /* acorda equipa 1 e 2*/
236     if (sh->fSt.st.goalieStat[id] == FORMING_TEAM){
237         if (semUp (semgid, sh->refereeWaitTeams) == -1) {
238             perror ("error on the up operation for semaphore access (RF)");
239             exit (EXIT_FAILURE);
240         }
241     }
242
243     return ret;
244 }
```

Para "ativar" ambas as equipas formadas, ou seja, para os goalies com o estado **FORMING_TEAM** (os capitães), ativamos (damos *up*) ao semáforo *refereeeWaitTeams* para informar ao árbitro que as equipas estão prontas a jogar e devolvemos a variável de retorno.

Fig. 19- Função goaliesConstituteTeam() - parte 3

Função ***waitReferee()***:

```
static void waitReferee (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    if (team == 1){ // muda p estado 3
        sh->fSt.st.goalieStat[id] = WAITING_START_1;
        saveState(nFic, &sh->fSt);
    }

    if (team == 2){ // muda p estado 4
        sh->fSt.st.goalieStat[id] = WAITING_START_2;
        saveState(nFic, &sh->fSt);
    }

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (semDown(semgid, sh->playersWaitReferee) == -1){
    perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

}
```

Fig. 20 - Função waitReferee()

Tal como explicado anteriormente, esta função tem como principal objetivo esperar pelo árbitro para iniciar o jogo e alterar os estados dos jogadores de ambas as equipas do estado **FORMING_TEAM** para o estado **WAITING_START_1** para a equipa 1 e **WAITING_START_2**. Através do acesso à região crítica, verificamos com uma condição *if* qual das equipas estamos a lidar e alterámos os estados. Por último, dá-mos *down* ao semáforo *playersWaitReferee*.

Função ***playUntilEnd()***:

Esta função também funciona como a que descrevemos anteriormente no ***semSharedMemoryGoalie.c***, tem como principal objetivo esperar que o árbitro acabe a partida. Fazemos isso, de igual modo, verificando de qual das equipas estamos a falar e alterámos do estado **WAITING_TO_START_1** ou **WAITING_TO_START_2** para **PLAYING_1** ou **PLAYING_2.**

```
static void playUntilEnd (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }


    if (team == 1){ // muda p estado 5
        sh->fSt.st.goalieStat[id] = PLAYING_1;
        saveState(nFic, &sh->fSt);
    }
    if (team == 2){ // muda p estado 6
        sh->fSt.st.goalieStat[id] = PLAYING_2;
        saveState(nFic, &sh->fSt);
    }

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);

    if (semDown (semgid, sh->playersWaitEnd) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

}
```

Fig. 21 - Função playUntilEnd()

Testes realizados

Para compilar os *scripts* mencionados anteriormente, no terminal, dirigimo-nos para a pasta *src* e compilamos ao escrever `make` *[Fig.22]*. Depois, direcionamo-nos para a pasta *run* e executamos o código `./probSemSharedMemSoccerGame` *[Fig.23]*.

```
mariana@mariana-Lenovo-Legion-5-15IMH05:~/Desktop$ cd ./semaphore_soccergame/
mariana@mariana-Lenovo-Legion-5-15IMH05:~/Desktop/semaphore_soccergame$ cd ./src
mariana@mariana-Lenovo-Legion-5-15IMH05:~/Desktop/semaphore_soccergame/src$ make
rm -f *.o
gcc -Wall -g   -c -o semSharedMemPlayer.o semSharedMemPlayer.c
gcc -Wall -g   -c -o sharedMemory.o sharedMemory.c
gcc -Wall -g   -c -o semaphore.o semaphore.c
gcc -Wall -g   -c -o logging.o logging.c
gcc -o ../run/player semSharedMemPlayer.o sharedMemory.o semaphore.o logging.o -lm
gcc -Wall -g   -c -o semSharedMemGoalie.o semSharedMemGoalie.c
gcc -o ../run/goalie semSharedMemGoalie.o sharedMemory.o semaphore.o logging.o
gcc -Wall -g   -c -o semSharedMemReferee.o semSharedMemReferee.c
gcc -o ../run/referee semSharedMemReferee.o sharedMemory.o semaphore.o logging.o -lm
gcc -Wall -g   -c -o probSemSharedMemSoccerGame.o probSemSharedMemSoccerGame.c
gcc -o ../run/probSemSharedMemSoccerGame probSemSharedMemSoccerGame.o sharedMemory.o semaphore.o logging.o -lm
mariana@mariana-Lenovo-Legion-5-15IMH05:~/Desktop/semaphore_soccergame/src$ cd ../run
```

Fig. 22 e 23 - Terminal + Output

```
mariana@mariana-Lenovo-Legion-5-15IMH05:~/Desktop/semaphore_soccergame/run$ ./probSemSharedMemSoccerGame
                    SoccerGame - Description of the internal state

 P00 P01 P02 P03 P04 P05 P06 P07 P08 P09  G00 G01 G02  R01
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   0   0   0   0   0   0   0   0   0    0   0   0    0
   0   1   0   0   0   0   0   0   0   0    0   0   0    0
   0   1   0   0   0   0   0   0   0   0    0   0   0    0
   0   1   0   0   0   0   0   0   0   0    0   0   0    0
   1   1   0   0   0   0   0   0   0   0    0   0   0    0
   1   1   0   0   0   0   0   0   0   0    0   0   0    0
   1   1   0   0   1   0   0   0   0   0    0   0   0    0
   1   1   0   0   1   0   0   0   0   0    0   0   0    0
   1   1   1   0   1   0   0   0   0   0    0   0   0    0
   1   1   1   0   1   0   1   0   0   0    0   0   0    0
   1   1   1   0   1   1   1   0   0   0    0   0   0    0
   1   1   1   0   1   1   1   1   0   0    0   0   0    0
   1   1   1   1   1   1   1   1   0   0    0   0   0    0
   1   1   1   1   1   1   1   1   7   0    0   0   0    0
   1   1   1   1   1   1   1   1   7   7    0   0   0    0
   1   1   1   1   1   1   1   1   7   7    2   0   0    0
   1   1   1   1   1   1   1   1   7   7    2   2   0    0
   1   1   1   1   1   1   1   1   7   7    2   2   7    0
   1   1   1   1   1   1   1   1   7   7    2   2   7    1
   1   3   1   1   1   1   1   1   7   7    2   2   7    1
   3   3   1   1   1   1   1   1   7   7    2   2   7    1
   3   3   1   1   3   1   1   1   7   7    2   2   7    1
   3   3   3   1   3   1   1   1   7   7    2   2   7    1
   3   3   3   1   3   1   1   1   7   7    3   2   7    1
   3   3   3   1   3   1   4   1   7   7    3   2   7    1
   3   3   3   1   3   4   4   1   7   7    3   2   7    1
   3   3   3   1   3   4   4   4   7   7    3   2   7    1
   3   3   3   4   3   4   4   4   7   7    3   2   7    1
   3   3   3   4   3   4   4   4   7   7    3   4   7    1
   3   3   3   4   3   4   4   4   7   7    3   4   7    2
   3   5   3   4   3   4   4   4   7   7    3   4   7    2
   5   5   3   4   3   4   4   4   7   7    3   4   7    2
   5   5   3   4   5   4   4   4   7   7    3   4   7    2
   5   5   5   4   5   4   4   4   7   7    3   4   7    2
   5   5   5   4   5   4   4   4   7   7    5   4   7    2
   5   5   5   4   5   4   6   4   7   7    5   4   7    2
   5   5   5   4   5   4   6   4   7   7    5   4   7    3
   5   5   5   6   5   4   6   4   7   7    5   4   7    3
   5   5   5   6   5   4   6   6   7   7    5   4   7    3
   5   5   5   6   5   4   6   6   7   7    5   6   7    3
   5   5   5   6   5   6   6   6   7   7    5   6   7    3
   5   5   5   6   5   6   6   6   7   7    5   6   7    4
```

Com este primeiro exemplo, podemos comprovar que tudo o que dissemos anteriormente é verdadeiro e de facto acontece. Na última linha encontramos 5 jogadores no estado 5 (significa que estão a jogar pela equipa 1), 5 jogadores no estado 6 (significa que estão a jogar pela equipa 2), que 2 chegaram atrasados (7). Termina quando o árbitro anuncia fim do jogo (estado 4).

Aqui quem forma as equipas é o G00 (1º goalie a chegar) e forma a equipa 1 e o G01 (2º goalie a chegar) forma a equipa 2. Concluímos que, neste caso, ambos capitães são goalies. Vemos também que os primeiros 4 players a chegar ao campo de futebol, após algum tempo no estado 1 à espera de um capitão vão para a equipa 1 com a chegada do 1º goalie, passando do estado 1 para 3, indicando que estão na equipa 1. Os seguintes 4 players que chegaram ficaram também à espera de um capitão e quando o 2ª goalie chegou foi possível gerar a equipa, passando então do estado 1 de espera para o estado 4, indicando que pertencem à equipa 2. Os dois jogadores e guarda-redes que chegaram em último passam do estado 0 para o 7 porque chegaram tarde para o jogo.

Vemos aqui também que o árbitro chega pouco depois dos guarda-redes e fica em estado de espera até que as 2 equipas estejam formadas e que o capitão de cada uma o informe que pode começar o jogo. Uma vez com equipas formadas o árbitro passa para o estado 3 em que está a arbitrar o jogo e vemos os jogadores a passar do estado 3 para 5 indicando que estão a jogar pela equipa 1 e os outros passam do estado 4 para o 6 indicando que estão a jogar pela equipa 2. Quando o árbitro acaba o jogo passa para o estado 4 e avisa os jogadores, por fim o programa acaba.

Neste exemplo os 2 guarda-redes é que estão a formar equipas, que é o caso mais comum.

Realizamos 100 partidas e em muitas delas obtivemos um output parecido com o exemplo da figura 22.

Fig. 24 – Outro exemplo

```
ariana@mariana-Lenovo-Legion-5-15IMH05:~/Desktop/semaphore_soccergame/run$ ./run.sh 100

un n.º 1
                    SoccerGame - Description of the internal state

P00 P01 P02 P03 P04 P05 P06 P07 P08 P09  G00 G01 G02  R01
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   0   0   0   0   0   0   0   0    0   0   0    0
  0   0   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   0   0   0   0   0   0    0   0   0    0
  0   1   1   0   1   0   0   0   0   0    0   0   0    0
  0   1   1   0   1   1   0   0   0   0    0   0   0    0
  0   1   1   0   1   1   1   0   0   0    0   0   0    0
  0   1   1   1   1   1   1   0   0   0    0   0   0    0
  1   1   1   1   1   1   1   0   0   0    0   0   0    0
  1   1   1   1   1   1   1   1   0   0    0   0   0    0
  1   1   1   1   1   1   1   1   0   0    2   0   0    0
  1   1   1   1   1   1   1   1   7   0    2   0   0    0
  1   1   1   1   1   1   1   1   7   0    2   0   0    1
  1   1   1   1   1   1   1   1   7   7    2   0   0    1
  1   1   1   1   1   1   1   1   7   7    2   2   0    1
  1   1   1   1   1   1   1   1   7   7    2   2   7    1
  1   1   3   1   1   1   1   1   7   7    2   2   7    1
  1   3   3   1   1   1   1   1   7   7    2   2   7    1
  1   3   3   1   3   1   1   1   7   7    2   2   7    1
  1   3   3   1   3   3   1   1   7   7    2   2   7    1
  1   3   3   1   3   3   1   1   7   7    3   2   7    1
  1   3   3   1   3   3   1   1   7   7    3   2   7    2
  1   3   3   1   3   3   4   1   7   7    3   2   7    2
  1   3   3   4   3   3   4   1   7   7    3   2   7    2
  4   3   3   4   3   3   4   1   7   7    3   2   7    2
  4   3   3   4   3   3   4   4   7   7    3   2   7    2
  4   3   3   4   3   3   4   4   7   7    3   4   7    2
  4   3   5   4   3   3   4   4   7   7    3   4   7    2
  4   3   5   4   5   3   4   4   7   7    3   4   7    2
  4   3   5   4   5   3   6   4   7   7    3   4   7    2
  4   3   5   4   5   3   6   4   7   7    3   4   7    3
  4   3   5   4   5   3   6   4   7   7    5   4   7    3
  4   5   5   4   5   3   6   4   7   7    5   4   7    3
  4   5   5   6   5   3   6   4   7   7    5   4   7    3
  6   5   5   6   5   3   6   4   7   7    5   4   7    3
  6   5   5   6   5   5   6   4   7   7    5   4   7    3
  6   5   5   6   5   5   6   6   7   7    5   4   7    3
  6   5   5   6   5   5   6   6   7   7    5   6   7    3
  6   5   5   6   5   5   6   6   7   7    5   6   7    4
```

Posto a execução de todas as linhas de código ditas, dos testes realizados e das validações, podemos ver que de facto, cumprimos com o que nos foi pedido, um *script* onde é possível várias partidas válidas de jogos de futebol.

Conclusão

Ao longo deste trabalho, a nível teórico, consolidamos os nossos conhecimentos sobre os processos, semáforos, o que são e como funcionam, a sua utilidade e diversidade. A nível prático melhoramos o nosso conhecimento sobre a linguagem C, aprimorando as nossas habilidades de programação ao implementar novas metodologias de trabalho e pesquisa.

Em suma, chegamos a uma conclusão pessoal onde a combinação de escrever um *script* e escrever um relatório com base na criação desse código permite-nos otimizar o código e ter, ainda, um maior entendimento sobre o que estamos a criar, adquirindo ainda conhecimentos novos sobre diversos assuntos relacionados com a linguagem C.

Fontes

Para a concretização deste trabalho, para além das nossas bases adquiridas em semestres transatos, também nos baseamos na matéria lecionada durante as aulas teóricas e práticas da unidade curricular Sistemas Operativos, recorremos ao docente e pesquisamos em fóruns online quando algo não corria muito bem.