Sistemas Operativos

Trabalho 2

Simulação de um Jogo de Futebol

Professor:

Nuno Lau (nunolau@ua.pt)

Pedro Sobral, 98491, P3

Daniel Figueiredo, 98498, P5

As distribuições de percentagem são iguais para os membros do grupo

21/01/2021

Índice

1 - Introdução

No âmbito da unidade curricular de Sistemas Operativos, foi-nos apresentado a realização deste trabalho prático, que consiste na realização de uma simulação de um jogo de futebol, com 3 envolventes, *player*, *goalie*, *referee*, todos os envolventes são processos independentes, sendo que a sua sincronização e comunicação é realizada através de semáforos e de memória partilhada.

A simulação é constituída por duas equipas, sendo que cada uma terá 5 jogadores, 4 de campo (*player*), e um guarda-redes (*goalie*), as equipas vão sendo formadas à medida que os jogadores vão chegando, se ambas as equipas já estiverem formadas, os jogadores que chegarem irão ser informados que as equipas se encontram completas e que por consequência não entraram no jogo. Por jogo, existe um *referee*, este envolvente é bastante importante pois é ele que dita o início da partida, e o fim da mesma. De maneira a que não ocorram *deadlocks*, vamos usar semáforos para que não ocorra "choque" de informação, e deste modo a simulação correr da maneira correta.

A implementação do código será feita através do IDE *VSCode*, pois é um editor com que ambos os membros do grupo já estão familiarizados derivados a unidades curriculares e projetos passados. A execução de todo o trabalho foi suportada através de um repositório no GitHub[1], o que facilita em muito todo o *workflow* da realização do mesmo.

Com a realização deste trabalho prático, esperamos conseguir cumprir todos os pontos essenciais que o guião propõe, e veemente alargar os nossos horizontes no que toca a programar com semáforos em C, pois são uma variável muito importante no que toca a controlar a acesso a determinadas regiões por parte de vários processos

[1] – O repositório encontra-se privado, pelo menos até ao dia da entrega do trabalho (21/01/2021)

2 - Introdução ao Problema

2.1 - Compilação e Execução

Para compilar o programa é necessário à partida ter um compilador de C instalado na máquina, por exemplo o gcc. Posto isto, para compilar basta executar, estando na pasta /semaphore_soccergame/src/, o comando:

make all

De seguida, temos de entrar na pasta /semaphore_soccergame/run/, para que possamos "simular o jogo de futebol", fazendo:

./probSemSharedMemSoccerGame

Para averiguar se havia a existência de algum *dead lock*, executamos o código *run.sh*, que nos permite executar um determinado número de vezes o *./probSemSharedMemSoccerGame*

Nota : Para se correr o programa *run.sh*, pela primeira vez, teremos de dar permissões ao utilizador, assim, das próximas vezes em que for necessário correr esse programa, não teremos de repetir esse passo de novo.

2.2 – O problema

Quando há um determinado conjunto de processos a correr ao mesmo tempo, e porventura, esses processos partilham uma ou mais variáveis, que por cada um deles haverá manipulação dessas variáveis, pode acontecer (muito provável), que o retorno do programa não seja o esperado. Ora, isto é, um **CONDIÇÃO DE CORRIDA**, para que este tipo de problema seja resolvido, é preciso escalonar a manipulação dessas variáveis, temos de criar uns **SINCRONIZAÇÃO**. Cada processo irá ter **REGIÕES CRÍTICAS**, que são regiões onde são alteradas variáveis a vários processos, o importante a tratar nestas situações é que quando um processo entra nessa determinada região critica, mais nenhum entra nessa mesma região.

Para que consigamos resolver todos estes problemas, temos os **SEMÁFOROS**, são muito uteis para sincronizar processos, permitindo uma comunicação muito eficaz entre os diferentes processos. No decorrer do trabalho iremos usar semáforos mutex, são semáforos que trabalham só com os números 0 e 1, onde asseguram sempre exclusão mútua, e também iremos usar semáforos contadores, que serviram para controlar o número de acessos à região critica do processo.

Para que a sincronização ocorra de forma correta, temos se seguir algumas instruções:

- Enquanto não houver um número de elementos completo, os elementos ficam em WAITING_TEAM.
- O último elemento da equipa a chegar, "acorda os outros", forma a equipa, informa o árbitro, e ficam à esperam que o jogo comece.
- O árbitro tem de receber duas notificações, uma de cada equipa formada.
- É o árbitro quem dá início ao jogo, arbitra, e também é ele que o acaba.
- Quando estão a jogar os jogadores, ficam à espera de que o árbitro termine o jogo.

Como base de implementação para este trabalho prático usamos o código-fonte fornecido pelo Professor, código esse onde já se encontravam as variáveis e os semáforos inicializados, entre outras funções.

O estado que cada individuo pode abarcar está definido no ficheiro *probConst.h*, os estados serão abordados mais à frente.

Existem 3 entidades, sendo que dispõem das seguintes funções:

- ***Player*** **–** Chegar, pertencer a uma equipa e jogar.
- ***Goalie*** **–** Chegar, pertencer a uma equipa e jogar.
- ***Referee*** **–** Iniciar, arbitrar, e terminal o jogo.

Relativamente à memória partilhada, esta encontra-se no ficheiro *shareDataSync.h*, nas estruturas *FULL_STAT*, e *STAT*.

A utilização de semáforos serve essencialmente para controlar o acesso à memoria partilhada, evitando assim choques de informação entre os 3 indivíduos que compõem o trabalho prático. As notificações entre indivíduos foram feitas através de semáforos, para que desta forma o programa executasse sem problemas maiores.

Para facilitar todo o processo de implementação, dos semáforos e da memoria partilhada, elaboramos uma tabela, com todos os semáforos que foram alterados por nós, desta forma a implementação dos semáforos no código tornou-se mais simples e rápida, pois a tabela estrutura de forma bastante rigorosa, o que, e onde as alterações no código-fonte serão feitas.

SEMAFORO	**Entidade Down**	**Função Down**	**#Downs**	**Entidade Up**	**Função Up**	**#Ups**
playersWaitTeam	Jogadores ou guarda-redes	playerConstituteTeam() goalieConstituteTeam()	3/4	Jogador formador da equipa	playerConstituteTeam() goalieConstituteTeam()	3/4
goaliesWaitTeam	Jogadores ou guarda-redes	playerConstituteTeam() goalieConstituteTeam()	1/0	Jogador formador da equipa	playerConstituteTeam() goalieConstituteTeam()	1/0
playersWaitReferee	Jogador formador da equipa	playerConstituteTeam() goalieConstituteTeam()	10	Árbitro	startGame()	10
playersWaitEnd	Árbitro	endGame()	10	Jogadores ou guarda-redes	playUntilEnd()	10
refereeWaitTeams	Árbitro	waitForTeams()	2	Jogador formador da equipa	playerConstituteTeam() goalieConstituteTeam()	2
playersRegistered	Jogadores ou guarda-redes	playerConstituteTeam() goalieConstituteTeam()	1 cada	Jogador formador da equipa	playerConstituteTeam() goalieConstituteTeam()	1 cada

3 – Implementação

3.1 - Referee

Para a nossa simulação do jogo de futebol, como já referido anteriormente, iremos necessitar de um árbitro, sendo que, o nosso árbitro, ao longo do código irá assumir cinco estados diferentes. Sendo eles (Fig.1):

- ***ARRIVING***, tomando o valor 0, que significa que está a chegar ao campo .
- ***WAITING_TEAMS***, tomando o valor 1, que significa que o árbitro está à espera de que se formem as duas equipas.
- ***STARTING_GAME***, tomando o valor 2, que significa que, as equipas já estão formadas, logo o árbitro pode iniciar o jogo.
- ***REFEREEING***, tomando o valor 3, que significa que o árbitro está a arbitrar o jogo
- ***ENDING_GAME***, tomando o valor 4, que significa que o árbitro termina o jogo.

```
/** \brief referee initial state, arriving  */
#define  ARRIVING          0
/** \brief referee waiting for both teams  */
#define  WAITING_TEAMS     1
/** \brief referee starting game  */
#define  STARTING_GAME     2
/** \brief referee refereeing  */
#define  REFEREEING        3
/** \brief referee ending game  */
#define  ENDING_GAME       4
```

Figura 1 - Constantes de estado do árbitro

Para que o árbitro percorra, todos estes estados mencionados, serão utilizadas cinco funções, sendo elas *arrive(), waitForTeams(), startGame(), play() e endGame()*.

3.1.1 - *arrive()*

Nesta função (Fig.2), é pedido para que se atualize o estado do árbitro. Logo, alteramos o estado do árbitro para *ARRIVING*, dentro da região crítica do *mutex* e em seguimos guardamos esse estado.

```
144 static void arrive()
145 {
146     if (semDown(semgid, sh->mutex) == -1)
147     { /* enter critical region */
148         perror("error on the up operation for semaphore access (RF)");
149         exit(EXIT_FAILURE);
150     }
151 
152     // TODO: insert your code here
153     sh->fSt.st.refereeStat = ARRIVING;
154     saveState(nFic, &sh->fSt);
155 
156     if (semUp(semgid, sh->mutex) == -1)
157     { /* leave critical region */
158         perror("error on the down operation for semaphore access (RF)");
159         exit(EXIT_FAILURE);
160     }
161 
162     usleep((100.0 * random()) / (RAND_MAX + 1.0) + 10.0);
163 }
```

Figura 2 – Função arrive() do árbitro

3.1.2 - *waitForTeams()*

Na função *waitForTeams()* (Fig.3) é pedido que o árbitro espere que as equipas estejam completamente formadas, ou seja, quatro jogadores e apenas um guarda-redes em cada equipa. Para isso, é necessário que o árbitro atualize o seu estado e que seja utilizado o semáforo *refereeWaitTeams.*

```
172  static void waitForTeams()
173  {
174      if (semDown(semgid, sh->mutex) == -1)
175      { /* enter critical region */
176          perror("error on the up operation for semaphore access (RF)");
177          exit(EXIT_FAILURE);
178      }
179
180      // TODO: insert your code here
181      sh->fSt.st.refereeStat = WAITING_TEAMS;
182      saveState(nFic, &sh->fSt); //para "escrever" no log
183
184      if (semUp(semgid, sh->mutex) == -1)
185      { /* leave critical region */
186          perror("error on the down operation for semaphore access (RF)");
187          exit(EXIT_FAILURE);
188      }
189
190      // TODO: insert your code here
191      // usar semaforo refereeWaitTeams
192      // Ta ca um 2 mas devia de tar o Nº de Teams
193
194      for (int i = 0; i < 2; i++)
195      {
196          if (semDown(semgid, sh->refereeWaitTeams) == -1)
197          {
198              perror("error on the up operation for semaphore access ()");
199              exit(EXIT_FAILURE);
200          }
201      }
202  }
```

Figura 3 - Função waitForTeams()

Para a implementação deste semáforo, utilizamos um ciclo *for*, iterando-o duas vezes, visto que existem duas equipas. Este semáforo, utilizado pelo árbitro, vai fazer dois *downs*, representando que vai estar à espera pela confirmação do último membro de cada equipa (o que formou a equipa) com a informação que as equipas estão completas.

3.1.3 - *startGame()*

A função *startGame()* (Fig.4), pede que o árbitro comece o jogo. Para isso, o árbitro tem de alterar o seu estado atual e terá de notificar os jogadores todos que o jogo vai começar.

```
211 static void startGame()
212 {
213     if (semDown(semgid, sh->mutex) == -1)
214     { /* enter critical region */
215         perror("error on the up operation for semaphore access (RF)");
216         exit(EXIT_FAILURE);
217     }
218 
219     // TODO: insert your code here
220     sh->fSt.st.refereeStat = STARTING_GAME;
221     saveState(nFic, &sh->fSt);
222 
223     if (semUp(semgid, sh->mutex) == -1)
224     { /* leave critical region */
225         perror("error on the down operation for semaphore access (RF)");
226         exit(EXIT_FAILURE);
227     }
228 
229     // TODO: insert your code here
230     for (int i = 0; i < NUMPLAYERS; i++)
231     {
232         if (semUp(semgid, sh->playersWaitReferee) == -1)
233         { /* leave critical region */
234             perror("error on the down operation for semaphore access (RF)");
235             exit(EXIT_FAILURE);
236         }
237     }
238 }
```

Figura 4 - Função startGame()

Dentro da região crítica, atualizamos o estado do árbitro para *STARTING_GAME*, significando que o jogo vai começar. Para que o árbitro pudesse avisar os jogadores que o jogo iria começar, fora da região crítica, utilizamos o semáforo *playersWaitReferee*, implementando este, dentro de um ciclo *for*, iterando *NUMPLAYERS(10)* vezes, fazendo em cada iteração *semUp*, que permite, ao árbitro, notificar cada jogador que o jogo vai começar.

3.1.4 - play()

Para a função *play()* (Fig.5), era pedido, apenas, que o árbitro deixasse passar algum tempo para que pudesse terminar o jogo e que alterasse o seu estado enquanto o jogo estivesse a decorrer. Para isso, dentro da região crítica, alteramos o estado do árbitro para *REFEREEING*, estando este, assim, a arbitrar o jogo.

```
247 static void play()
248 {
249     if (semDown(semgid, sh->mutex) == -1)
250     { /* enter critical region */
251         perror("error on the up operation for semaphore access (RF)");
252         exit(EXIT_FAILURE);
253     }
254 
255     // TODO: insert your code here */
256     sh->fSt.st.refereeStat = REFEREEING;
257     saveState(nFic, &sh->fSt);
258 
259     if (semUp(semgid, sh->mutex) == -1)
260     { /* leave critical region */
261         perror("error on the down operation for semaphore access (RF)");
262         exit(EXIT_FAILURE);
263     }
264 
265     usleep((100.0 * random()) / (RAND_MAX + 1.0) + 900.0);
266 }
```

Figura 5 - Função play()

3.1.5 - *endGame()*

Na função *endGame()* (Fig.6), era pedido que o árbitro alterasse o seu estado e que notificasse todos os jogadores que o jogo terminou. Alteramos, dentro da região crítica, o estado do árbitro para *ENDING_GAME*, terminando assim o jogo.

```
275 static void endGame()
276 {
277     if (semDown(semgid, sh->mutex) == -1)
278     { /* enter critical region */
279         perror("error on the up operation for semaphore access (RF)");
280         exit(EXIT_FAILURE);
281     }
282 
283     // TODO: insert your code here */
284     sh->fSt.st.refereeStat = ENDING_GAME;
285     saveState(nFic, &sh->fSt);
286 
287     if (semUp(semgid, sh->mutex) == -1)
288     { /* leave critical region */
289         perror("error on the down operation for semaphore access (RF)");
290         exit(EXIT_FAILURE);
291     }
292 
293     // TODO: insert your code here
294     for (int i = 0; i < NUMPLAYERS; i++)
295     {
296         if (semUp(semgid, sh->playersWaitEnd) == -1)
297         {
298             perror("error on the down operation for semaphore access (RF)");
299             exit(EXIT_FAILURE);
300         }
301     }
302 }
```

Figura 6 – Função endGame()

Para que o árbitro pudesse notificar os jogadores acerca do término do jogo, fora da região crítica, utilizamos um ciclo *for*, iterando sobre ele *NUMPLAYERS*(10) vezes, fazendo em cada iteração *semUp*, que permite, ao árbitro, notificar cada jogador que o jogo terminou.

3.2 - Goalie

Quanto aos *goalies*, no jogo pode haver 3, no entanto só irão jogar 2, ficando desta forma sempre um que não irá jogar. O guarda-redes, é um envolvente do jogo que pode estar em diversos estados, estados esses que estão representados no ficheiro *probConst.h* (Fig. 7).

```
/** \brief player/goalie initial state, arriving */
#define  ARRIVING          0
/** \brief player/goalie waiting to constitute team */
#define  WAITING_TEAM      1
/** \brief player/goalie waiting to constitute team */
#define  FORMING_TEAM      2
/** \brief player/goalie waiting for referee to start game in team 1 */
#define  WAITING_START_1   3
/** \brief player/goalie waiting for referee to start game in team 2 */
#define  WAITING_START_2   4
/** \brief player/goalie playing in team 1 */
#define  PLAYING_1         5
/** \brief player/goalie playing in team 2 */
#define  PLAYING_2         6
/** \brief player/goalie playing */
#define  LATE              7
```

Figura 7 - Constantes de estado do Goalie

Como a figura. 7, demonstra o *goalie* pode arcar 8 estados diferentes:

- **ARRIVING 0** - significa que está a chegar ao campo.
- **WAITING_TEAM 1 -** espera que a equipa seja constituída.
- **FORMING TEAM 2 -** *goalie* forma a equipa.
- **WAITING_START_1 3 -** *goalie* da equipa 1 espera pelo início do jogo.
- **WAITING_START_2 4 -** *goalie* da equipa 2, espera pelo início do jogo.
- **PLAYING_1 5 -** goalie da equipa 1, está a jogar.
- **PLAYING_2 6 -** goalie da equipa 2, está a jogar.
- **LATE 7 -** *goalie chegou atrasado, não irá jogar.*

3.2.1 - *arrive()*

Na função *arrive()* (Fig.8), as alterações feitas foram minutas, uma vez que o necessário a fazer é somente alterar o estado do *goalie* em concreto, para isso usamos o *id* do mesmo para ARRIVING, esta alteração foi é feita dentro da região crítica. Fizemos uso da função *saveState()* para que o estado agora já atualizado aparecesse no terminal.

```
147 static void arrive(int id)
148 {
149     if (semDown(semgid, sh->mutex) == -1)
150     { /* enter critical region */
151         perror("error on the up operation for semaphore access (GL)");
152         exit(EXIT_FAILURE);
153     }
154 
155     /* TODO: insert your code here */
156     sh->fSt.st.goalieStat[id] = ARRIVING;
157     saveState(nFic, &sh->fSt);
158 
159     if (semUp(semgid, sh->mutex) == -1)
160     { /* exit critical region */
161         perror("error on the down operation for semaphore access (GL)");
162         exit(EXIT_FAILURE);
163     }
164 
165     usleep((200.0 * random()) / (RAND_MAX + 1.0) + 60.0);
166 }
```

Figura 8 - Função arrive() do Goalie

3.2.2 - *goaliesConstituteTeam()*

Na função *goaliesConstituteTeam()* (Fig.9) realizamos mais alterações dentro da zona crítica do *mutex* comparando com as restantes funções. Para esta função, dentro da região crítica, começamos por fazer incrementos às variáveis *goaliesFree* e *goaliesArrived*, visto que, ao chegar a este ponto da função, vamos ter mais um guarda-redes que está livre (ainda não foi colocado em nenhuma equipa) e esse mesmo é mais um guarda-redes que chegou(por isso o incremento na variável *goaliesArrived*).

Ainda dentro da região crítica, avaliamos se já chegaram pelo menos dois guarda-redes, pois para se terem duas equipas precisamos de ter dois guarda-redes (um para cada equipa) e caso já tenham chegado pelo menos dois vamos avaliar se existem quatro ou mais jogadores livres, visto que para se formar uma equipa precisamos de quatro jogadores e de um guarda-redes. Se essa condição se verificar, então alteramos o estado do guarda-redes para *FORMING_TEAM* e fazemos um decremento da variável *goaliesFree*, pois foi um guarda-redes que passou de estar livre para pertencer a uma equipa. Para além disso, se for viável formar uma equipa, então o guarda-redes, ao iterar sobre o ciclo *for NUMTEAMPLAYERS(4) vezes*, faz em cada iteração um *semUp* ao semáforo *playersWaitTeam*, simbolizando a entrega da confirmação que quatro jogadores pertencem agora à equipa do guarda-redes e deste modo faz também noutro ciclo *for, NUMTEAMPLAYERS(4) semDowns* ao semáforo *playersRegistered*, significando que recebeu as confirmações que quatro jogadores ingressaram numa equipa. Caso ainda não haja jogadores livres ou guarda-redes livres suficientes, então irá ficar no estado WAITING_TEAM.

Caso já tenham chegado dois guarda-redes (*goaliesArrived*), então o próximo guarda-redes a chegar irá passar para o estado *LATE*, e irá permanecer nesse estado até ao final do jogo.

```
180     sh->fSt.goaliesFree++;
181     sh->fSt.goaliesArrived++;
182 
183     if (sh->fSt.goaliesArrived <= 2 * NUMTEAMGOALIES) {
184         if (sh->fSt.playersFree >= 4 && sh->fSt.goaliesFree >= NUMTEAMGOALIES) {
185             sh->fSt.st.goalieStat[id] = FORMING_TEAM;
186             sh->fSt.goaliesFree--;
187 
188             for (int i = 0; i < NUMTEAMPLAYERS; i++) {
189                 if (semUp(semgid, sh->playersWaitTeam) == -1) {
190                     perror("error on the up operation for semaphore access (GL)");
191                     exit(EXIT_FAILURE);
192                 }
193             }
194 
195             for (int i = 0; i < NUMTEAMPLAYERS; i++) {
196                 if (semDown(semgid, sh->playerRegistered) == -1) {
197                     perror("error on the up operation for semaphore access (GL)");
198                     exit(EXIT_FAILURE);
199                 }
200             }
201 
202             sh->fSt.playersFree = sh->fSt.playersFree - 4;
203             ret = sh->fSt.teamId++;
204             saveState(nFic, &sh->fSt);
205         } else {
206             sh->fSt.st.goalieStat[id] = WAITING_TEAM;
207             saveState(nFic, &sh->fSt);
208         }
209     } else {
210         ret = 0;
211         sh->fSt.st.goalieStat[id] = LATE;
212         saveState(nFic, &sh->fSt);
213     }
```

Figura 9 – Função goaliesConstituteTeam() dentro da zona crítica

Fora da região crítica (Fig.10), caso o guarda-redes se encontre no estado *FORMING_TEAM*, ou seja, este está a constituir uma equipa, iremos fazer um *semUp* ao semáforo *refereeWaitTeams*, para o guarda-redes notificar o árbitro que uma equipa foi formada.

Caso não esteja no estado *FORMIMG_TEAM*, vamos ver se está no estado *WAITING_TEAM* e caso esteja faz um *semDown* do semáforo *goaliesWaitTeam*, que notifica que o guarda-redes está à espera que a equipa fique formada. De seguida, iremos à variável *ret* e colocamos nela o valor de *teamId*. Outro semáforo necessário para esta função é o *playerRegistered*, fazendo um *semUp*, simbolizando que houve um jogador que se registou na equipa.

```
215     if (semUp(semgid, sh->mutex) == -1) { /* exit critical region */
216         perror("error on the down operation for semaphore access (GL)");
217         exit(EXIT_FAILURE);
218     }
219 
220     // TODO: insert your code here
221     if (sh->fSt.st.goalieStat[id] == FORMING_TEAM) {
222         if (semUp(semgid, sh->refereeWaitTeams) == -1) {
223             perror("error on the up operation for semaphore access (GL)");
224             exit(EXIT_FAILURE);
225         }
226     }
227 
228     else if (sh->fSt.st.goalieStat[id] == WAITING_TEAM) {
229         if (semDown(semgid, sh->goaliesWaitTeam) == -1) {
230             perror("error on the up operation for semaphore access (GL)");
231             exit(EXIT_FAILURE);
232         }
233 
234         ret = sh->fSt.teamId;
235 
236         if (semUp(semgid, sh->playerRegistered) == -1) {
237             perror("error on the up operation for semaphore access (GL)");
238             exit(EXIT_FAILURE);
239         }
240     }
241 
242     return ret;
243 }
```

Figura 10 - Función goaliesConstituteTeam() fora da zona crítica

3.2.3 - *waitReferee()*

Quanto à função *waitReferee() (*Fig.11*)*, é dito que é necessário atualizar o estado do *goalies*, e esperar que o *referee* comece o jogo.

Relativamente à parte de atualizar o estado, este é feito dentro da região crítica, utilizamos um *if*, para distinguir a team a que o goalie pertence, para a *team* 1, o estado com que o *goalie* fica é WAITING_START_1, para a *team* 2, o estado com que o *goalie* fica é WAITING_START_2. Ao fim usamos a função *saveState()*, para que o estado seja atualizado.

```
286 static void waitReferee(int id, int team)
287 {
288     if (semDown(semgid, sh->mutex) == -1)
289     { /* enter critical region */
290         perror("error on the up operation for semaphore access (GL)");
291         exit(EXIT_FAILURE);
292     }
293 
294     // TODO: insert your code here
295     if (team == 1)
296     {
297         sh->fSt.st.goalieStat[id] = WAITING_START_1;
298     }
299     else if (team == 2)
300     {
301         sh->fSt.st.goalieStat[id] = WAITING_START_2;
302     }
303     saveState(nFic, &sh->fSt);
304 
305     if (semUp(semgid, sh->mutex) == -1)
306     { /* exit critical region */
307         perror("error on the down operation for semaphore access (GL)");
308         exit(EXIT_FAILURE);
309     }
310 
311     // TODO: insert your code here
312     if (semDown(semgid, sh->playersWaitReferee) == -1)
313     {
314         perror("error on the up operation for semaphore access(GL)");
315         exit(EXIT_FAILURE);
316     }
317 }
```

Figura 11 - Função waitReferee() do Goalie

Agora já fora da região crítica, vamos usar o semáforo *playersWaitReferee*, através de um *Down*, pois os *goalies* vão estar à espera da confirmação do árbitro para que o possam começar a jogar.

3.2.4 - *playUntilEnd()*

Relativamente à função *playUntilEnd()* (Fig.12), o que fizemos foi: atualizar o estado o *goalie* e esperar que o árbitro termine o jogo.

Dentro da região crítica, vamos atualizar o estado do *goalies*, a atualização é feita com um *if*, para distinguir a team a que o goalie pertence, para a *team* 1, o estado com que o *goalie* fica é PLAYING_1, para a *team* 2, o estado com que o *goalie* fica é PLAYING_2. Ao fim usamos a função *saveState()*, para que o estado seja atualizado.

```
328 static void playUntilEnd(int id, int team)
329 {
330     if (semDown(semgid, sh->mutex) == -1)
331     { /* enter critical region */
332         perror("error on the up operation for semaphore access (GL)");
333         exit(EXIT_FAILURE);
334     }
335 
336     // TODO: insert your code here
337     if (team == 1)
338     {
339         sh->fSt.st.goalieStat[id] = PLAYING_1;
340     }
341     else if (team == 2)
342     {
343         sh->fSt.st.goalieStat[id] = PLAYING_2;
344     }
345     saveState(nFic, &sh->fSt);
346 
347     if (semUp(semgid, sh->mutex) == -1)
348     { /* exit critical region */
349         perror("error on the down operation for semaphore access (GL)");
350         exit(EXIT_FAILURE);
351     }
352 
353     // TODO: insert your code here
354     if (semDown(semgid, sh->playersWaitEnd) == -1)
355     {
356         perror("error on the up operation for semaphore access (GL)");
357         exit(EXIT_FAILURE);
358     }
359 }
```

Figura 12 - Função playUntilEnd() do Goalie

Já fora da região crítica , fazemos uso do semáforo *playersWaitEnd*, com recurso à função *semDown()*, fazemos um *Down no semáforo, pois desta forma os jogadores vão estar à espera que o jogo acabe.*

3.3 - Player

Quanto aos *players*, no jogo pode haver 10, no entanto só irão no jogar 8, 4 em cada equipa, ficando desta forma sempre dois que não irá jogar. O *player*, é um envolvente do jogo que pode estar em diversos estados, estados esses que estão representados no ficheiro *probConst.h* (Fig. 13).

```
/** \brief player/goalie initial state, arriving */
#define  ARRIVING          0
/** \brief player/goalie waiting to constitute team */
#define  WAITING_TEAM      1
/** \brief player/goalie waiting to constitute team */
#define  FORMING_TEAM      2
/** \brief player/goalie waiting for referee to start game in team 1 */
#define  WAITING_START_1   3
/** \brief player/goalie waiting for referee to start game in team 2 */
#define  WAITING_START_2   4
/** \brief player/goalie playing in team 1 */
#define  PLAYING_1         5
/** \brief player/goalie playing in team 2 */
#define  PLAYING_2         6
/** \brief player/goalie playing */
#define  LATE              7
```

Figura 13 - Constantes de estado do Player

Como a figura 13, demonstra o *player* pode assumir 8 estados diferentes:

- **ARRIVING 0** - significa que está a chegar ao campo.
- **WAITING_TEAM 1** - espera que a equipa seja constituída.
- **FORMING TEAM 2** - *player* forma a equipa.
- **WAITING_START_1 3** - *player* da equipa 1 espera pelo início do jogo.
- **WAITING_START_2 4** - *player* da equipa 2, espera pelo início do jogo.
- **PLAYING_1 5** - *player* da equipa 1, está a jogar.
- **PLAYING_2 6** - *player* da equipa 2, está a jogar.
- **LATE 7** - *player* chegou atrasado, não irá jogar.

3.3.1 - *arrive()*

Na função *arrive()* (Fig.14), a única alteração necessária é atualizar o estado do *player* em concreto com o seu id, para ARRIVING, esta alteração é efetuada dentro da região crítica. De seguida, usamos a função *saveState()*, para o estado ser atualizado.

```
146 static void arrive(int id)
147 {
148     if (semDown(semgid, sh->mutex) == -1)
149     { /* enter critical region */
150         perror("error on the up operation for semaphore access (PL)");
151         exit(EXIT_FAILURE);
152     }
153 
154     /* TODO: insert your code here */
155     sh->fSt.st.playerStat[id] = ARRIVING;
156     saveState(nFic, &sh->fSt);
157 
158     if (semUp(semgid, sh->mutex) == -1)
159     { /* exit critical region */
160         perror("error on the down operation for semaphore access (PL)");
161         exit(EXIT_FAILURE);
162     }
163 
164     usleep((200.0 * random()) / (RAND_MAX + 1.0) + 50.0);
165 }
```

Figura 14 - Função arrive() do Player

3.3.2 - *playerConstituteTeam()*

Na função *playerConstituteTeam()* (Fig.15) realizamos mais alterações dentro da zona crítica do *mutex* comparando com as restantes funções. Para esta função, dentro da região crítica, começamos por fazer incrementos às variáveis *playersFree* e *playersArrived*, visto que, ao chegar a este ponto da função, vamos ter mais um jogador que está livre (ainda não foi colocado em nenhuma equipa) e esse mesmo é mais um jogador que chegou(por isso o incremento na variável *goaliesArrived*).

Ainda dentro da região crítica, avaliamos se já chegaram pelo menos oito guarda-redes, pois para se terem duas equipas precisamos de ter oito jogadores (quatro para cada equipa) e caso já tenham chegado pelo menos oito vamos avaliar se existem quatro ou mais jogadores livres, visto que para se formar uma equipa precisamos de quatro jogadores e de um guarda-redes. Se essa condição se verificar, então alteramos o estado do jogador para *FORMING_TEAM* e fazemos um decremento da variável *playersFree*, pois foi um jogador que passou de estar livre para pertencer a uma equipa. Para além disso, se for viável formar uma equipa, então o jogador, ao iterar sobre o ciclo *for* (*NUMTEAMPLAYERS(4) – 1) vezes (*faz menos uma vez pois ele já pertence à equipa *)*, faz em cada iteração um *semUp* ao semáforo *playersWaitTeam*, simbolizando a entrega da confirmação que três jogadores pertencem agora à equipa e deste modo faz também (*NUMTEAMPLAYERS(4) – 1) semDowns* ao semáforo *playersRegistered*, significando que recebeu as confirmações que três jogadores ingressaram numa equipa , fazendo o decremento da variável *playersFree* em cada iteração pelo ciclo *for*, sendo esta parte apenas para jogadores mas ainda precisamos de colocar o guarda-redes.

Para colocar o guarda-redes na equipa, como entrou na condição *if*, então vai fazer um *semUp* ao semáforo *goaliesWaitTeam*, que simboliza que o guarda-redes está à espera que a equipa que este ingressou fique completa. Faz-se, também, um *semUp* ao semáforo *playerRegistered* para notificar que o guarda-redes faz agora parte de uma equipa. De seguida faz-se o decremento da variável *goaliesFree*, visto que o guarda-redes ingressou numa equipa e já não se encontra livre. Na variável ret colocamos o *teamid++*.

```
193     // TODO: insert your code here */
194     sh->fSt.playersFree++;
195     sh->fSt.playersArrived++;
196 
197     if (sh->fSt.playersArrived <= 2 * NUMTEAMPLAYERS)
198     {
199         if (sh->fSt.playersFree < NUMTEAMPLAYERS || sh->fSt.goaliesFree < NUMTEAMGOALIES)
200         {
201             sh->fSt.st.playerStat[id] = WAITING_TEAM;
202             saveState(nFic, &sh->fSt);
203         }
204         else
205         { /* playersFree >= 4 && goaliesFree >= 1 -> able to constitute team */
206             sh->fSt.st.playerStat[id] = FORMING_TEAM;
207             sh->fSt.playersFree--;
208 
209             for (int i = 0; i < NUMTEAMPLAYERS - 1; i++)
210             { /* we need 3 other players to join this team */
211                 if (semUp(semgid, sh->playersWaitTeam) == -1)
212                 {
213                     perror("error on the down operation for semaphore access (PL)");
214                     exit(EXIT_FAILURE);
215                 }
216 
217                 if (semDown(semgid, sh->playerRegistered) == -1)
218                 {
219                     perror("error on the up operation for semaphore access (PL)");
220                     exit(EXIT_FAILURE);
221                 }
222 
223                 sh->fSt.playersFree--;
224             }
225 
226             /* we also need a goalie */
227             if (semUp(semgid, sh->goaliesWaitTeam) == -1)
228             {
229                 perror("error on the down operation for semaphore access (PL)");
230                 exit(EXIT_FAILURE);
231             }
232 
233             if (semDown(semgid, sh->playerRegistered) == -1)
234             {
235                 perror("error on the up operation for semaphore access (PL)");
236                 exit(EXIT_FAILURE);
237             }
238 
239             sh->fSt.goaliesFree--;
240             ret = sh->fSt.teamId++;
241 
242             saveState(nFic, &sh->fSt);
243         }
244     }
245     else
246     { /* the player is late, he's not playing */
247         ret = 0;
248         sh->fSt.st.playerStat[id] = LATE;
249         saveState(nFic, &sh->fSt);
250     }
```

Figura 15 – Função playerConstituteTeam() na zona crítica

Caso ainda não haja jogadores livres ou guarda-redes livres suficientes, então irá ficar no estado WAITING_TEAM. Caso já tenham chegado oito jogadores(*playersArrived*), então os próximos jogadores a chegar irão passar para o estado *LATE*, e irão permanecer nesse estado até ao final do jogo.

Fora da região crítica (Fig.16), caso o jogador se encontre no estado *FORMING_TEAM*, ou seja, este está a constituir uma equipa, iremos fazer um *semUp* ao semáforo *refereeWaitTeams*, para o jogador notificar o árbitro que uma equipa foi formada.

Caso não esteja no estado *FORMIMG_TEAM*, vamos ver se está no estado *WAITING_TEAM* e caso esteja faz um *semDown* do semáforo *playersWaitTeam*, que notifica que o jogador está à espera que a equipa fique formada. De seguida, iremos à variável *ret* e colocamos o valor de *teamId*. Outro semáforo necessário para esta função é o *playerRegistered*, fazendo um *semUp*, simbolizando que houve um jogador que se registou na equipa.

```
252     if (semUp(semgid, sh->mutex) == -1)
253     { /* exit critical region */
254         perror("error on the down operation for semaphore access (PL)");
255         exit(EXIT_FAILURE);
256     }
257 
258     // TODO: insert your code here */
259     if (sh->fSt.st.playerStat[id] == FORMING_TEAM)
260     {
261         if (semUp(semgid, sh->refereeWaitTeams) == -1)
262         {
263             perror("error on the down operation for semaphore access (PL)");
264             exit(EXIT_FAILURE);
265         }
266     }
267 
268     if (sh->fSt.st.playerStat[id] == WAITING_TEAM)
269     {
270         if (semDown(semgid, sh->playersWaitTeam) == -1)
271         {
272             perror("error on the up operation for semaphore access (PL)");
273             exit(EXIT_FAILURE);
274         }
275 
276         ret = sh->fSt.teamId;
277 
278         if (semUp(semgid, sh->playerRegistered) == -1)
279         {
280             perror("error on the down operation for semaphore access (PL)");
281             exit(EXIT_FAILURE);
282         }
283     }
284 
285     return ret;
286 }
```

Figura 16 - Função playerConstituteTeam fora da zona crítica

3.3.3 - *waitReferee()*

Em relação à função *waitReferee()* (Fig.17), é dito que esta função tem de atualizar o estado do *player*, e esperar que o árbitro dê início ao jogo.

Para atualizarmos o estado temos de fazer um *if, dentro da região critica*, para saber se estamos a tratar de um *player* da *team 1* ou da team 2, caso este seja da *team 1*, o seu estado fica WAITING_TEAM_1, caso seja da *team 2*, o seu estado fica WAITING_TEAM_2.

```
297 static void waitReferee(int id, int team)
298 {
299     if (semDown(semgid, sh->mutex) == -1)
300     { /* enter critical region */
301         perror("error on the up operation for semaphore access (PL)");
302         exit(EXIT_FAILURE);
303     }
304 
305     // TODO: insert your code here
306     if (team == 1)
307     {
308         sh->fSt.st.playerStat[id] = WAITING_START_1;
309     }
310     else if (team == 2)
311     {
312         sh->fSt.st.playerStat[id] = WAITING_START_2;
313     }
314     saveState(nFic, &sh->fSt);
315 
316     if (semUp(semgid, sh->mutex) == -1)
317     { /* exit critical region */
318         perror("error on the down operation for semaphore access (PL)");
319         exit(EXIT_FAILURE);
320     }
321 
322     // TODO: insert your code here
323     if (semDown(semgid, sh->playersWaitReferee) == -1)
324     {
325         perror("error on the up operation for semaphore access(PL)");
326         exit(EXIT_FAILURE);
327     }
328 }
```

Figura 17 - Função waitReferee() do Player

Já fora da região crítica, fazemos uso do semáforo *playersWaitReferee* através da função *semDown()*, pois os *players* vão estar à espera da confirmação do árbitro para que o possam começar a jogar.

3.3.4 - *playUntilEnd()*

Relativamente à função *playUntilEnd()* (Fig.18), o que fizemos foi: atualizar o estado o *player* e esperar que o árbitro termine o jogo.

Dentro da região crítica, vamos atualizar o estado do *players*, a atualização é feita com um *if*, para distinguir a team a que o player pertence, para a *team* 1, o estado com que o *goalie* fica é PLAYING_1, para a *team* 2, o estado com que o *player* fica é PLAYING_2. Ao fim usamos a função *saveState()*, para que o estado seja atualizado.

```
339 static void playUntilEnd(int id, int team)
340 {
341     if (semDown(semgid, sh->mutex) == -1)
342     { /* enter critical region */
343         perror("error on the up operation for semaphore access (PL)");
344         exit(EXIT_FAILURE);
345     }
346 
347     // TODO: insert your code here
348     if (team == 1)
349     {
350         sh->fSt.st.playerStat[id] = PLAYING_1;
351     }
352     else if (team == 2)
353     {
354         sh->fSt.st.playerStat[id] = PLAYING_2;
355     }
356     saveState(nFic, &sh->fSt);
357 
358     if (semUp(semgid, sh->mutex) == -1)
359     { /* exit critical region */
360         perror("error on the down operation for semaphore access (PL)");
361         exit(EXIT_FAILURE);
362     }
363 
364     // TODO: insert your code here
365     if (semDown(semgid, sh->playersWaitEnd) == -1)
366     {
367         perror("error on the up operation for semaphore access (PL)");
368         exit(EXIT_FAILURE);
369     }
370 }
```

Figura 18 - Função playUntilEnd()

Já fora da região crítica, fazemos uso do semáforo *playersWaitEnd,* com recurso à função *semDown(),* fazemos um Down no semáforo, pois desta forma os *players* vão estar à espera que o jogo acabe.

4 - Resultados

Durante a implementação do código foram feitos testes (*make pl*, *make gl*, *make rf*), com a nossa implementação e com o código pré-compilado fornecido pelo Professor, para termos sempre uma noção se estávamos a ir no caminho certo para encontrar a solução.

Neste ponto do relatório, vão ser avaliados os resultados obtidos, para isso, inicialmente iremos correr o *script run.sh*, *script* esse que simula 1000 jogos, ao correr-mos o *run.sh* ficamos a saber se existiam *deadlocks* na nossa implementação.

Mais tarde, o que fizemos foi escolher arbitrariamente um desses *outputs*, e analisar detalhadamente todas as transições de estados de todos os indivíduos envolvidos.

4.1 - Confirmação dos resultados

4.1.1 - Avaliação de *deadlocks*

Ora, nesta fase decidimos correr o nosso programa 1000 vezes (Fig.19), o que é um número considerável de vezes para analisar a existência de *deadlocks*, para isso usamos o script *run.sh*, e como correr todo até ao fim, podemos concluir que não existem quaisquer *deadlocks* na nossa implementação.

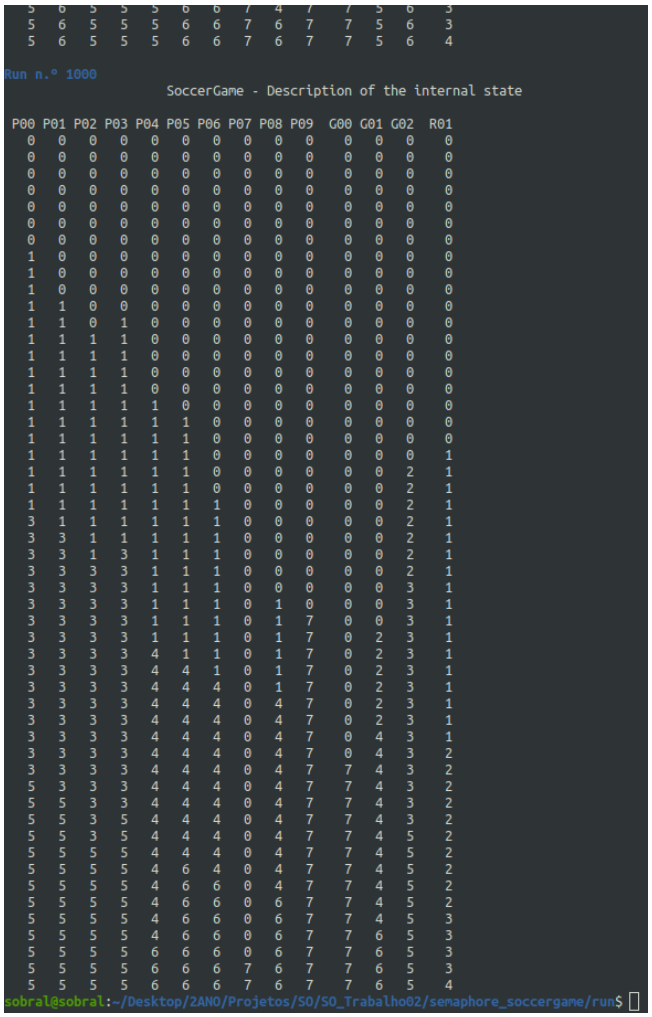

Figura 19 – Averiguação de deadlocks

Não haver *deadlock*, é uma condição necessária, mas não suficiente para um resultado correto, para isso temos de analisar um jogo e ver se todas as fases do mesmo se encontram da maneira certa.

Nota: A figura (esta aqui ao lado), mostra o resultado da última execução, porém existe um ficheiro com as execuções todas que pode ser consultado aqui (Por razões de tamanho é necessário fazer download do ficheiro, o GitHub não o consegue abrir).

4.1.2 - Confirmação dos resultados

De acordo com todas as condições que foram por nós abordadas no ponto 2.2 – O problema, o resultado obtido (Fig.20) mostra que todas essas condições são abordadas e tidas em atenção pelo nosso código C, dessa forma obtivemos os seguintes resultados satisfatórios, a equipa 1 tem 5 elementos (P01, P02, P03, P04, G01), a equipa 2 tem 5 elementos (P01, P05, P06, P06, G02), e um arbitro (R01), sendo que há 3 jogadores que chegaram atrasados e não jogaram (P08, P09, G00).

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	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	1	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0	0	0	0	0	0	0
1	1	1	1	1	0	1	0	0	0	0	0	0	0
1	1	1	1	1	0	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	0	0	0	2	0	0
1	1	1	1	1	1	1	1	7	0	0	2	0	0
1	1	1	1	1	1	1	1	7	0	0	2	2	0
1	1	1	1	1	1	1	1	7	0	7	2	2	0
1	1	1	1	1	1	1	1	7	0	7	2	2	1
1	3	1	1	1	1	1	1	7	0	7	2	2	1
1	3	1	3	1	1	1	1	7	0	7	2	2	1
1	3	3	3	1	1	1	1	7	0	7	2	2	1
1	3	3	3	3	1	1	1	7	0	7	2	2	1
1	3	3	3	3	1	1	1	7	7	7	2	2	1
1	3	3	3	3	1	1	1	7	7	7	3	2	1
1	3	3	3	3	1	1	4	7	7	7	3	2	1
4	3	3	3	3	1	1	4	7	7	7	3	2	1
4	3	3	3	3	4	1	4	7	7	7	3	2	1
4	3	3	3	3	4	4	4	7	7	7	3	2	1
4	3	3	3	3	4	4	4	7	7	7	3	4	1
4	3	3	3	3	4	4	4	7	7	7	3	4	2
4	3	5	3	3	4	4	4	7	7	7	3	4	2
4	3	5	3	3	4	4	6	7	7	7	3	4	2
6	3	5	3	3	4	4	6	7	7	7	3	4	2
6	3	5	3	3	4	4	6	7	7	7	5	4	2
6	5	5	3	3	4	4	6	7	7	7	5	4	2
6	5	5	3	3	4	6	6	7	7	7	5	4	2
6	5	5	3	5	4	6	6	7	7	7	5	4	2
6	5	5	5	5	4	6	6	7	7	7	5	4	2
6	5	5	5	5	4	6	6	7	7	7	5	4	3
6	5	5	5	5	4	6	6	7	7	7	5	6	3
6	5	5	5	5	6	6	6	7	7	7	5	6	3
6	5	5	5	5	6	6	6	7	7	7	5	6	4

Figura 20 – Analise dos resultados

5- Conclusão

Algo muito importante a reter deste trabalho prático é o conhecimento adquirido com utilização de semáforos e de memória partilhada, ficando mais claros certos pormenores sobre estas temáticas.

No início da realização deste mesmo trabalho, sentimos alguma dificuldade em analisar todo o código fornecido pelo Professor, pois não sabíamos ainda bem o que cada ficheiro continha e fazia, porém à medida que fomos trabalhando sobre os mesmos, essas dificuldades tornaram-se escassas.

Neste ponto, e em tom conclusivo, achamos que conseguimos alcançar todas as metas que o guião do trabalho prático propôs. Os resultados obtidos são nos aspetos essenciais semelhantes aos que o Professor forneceu, e nesse sentido estamos em condições de concluir que conseguimos alcançar uma solução possível e certa.

6 - Bibliografia

- Para a realização deste trabalho prático consultamos os slides disponibilizados pelo Professor na página do *e-learning* da unidade curricular Sistemas Operativos.

Os sites foram consultados entre os dias 20/01/2021 e 23/01/2021:

https://pt.stackoverflow.com/

https://www.geeksforgeeks.org/use-posix-semaphores-c/

https://riptutorial.com/c/example/31715/semaphores