



Projeto 2

Sistemas Operativos

Simulação de um jogo de futebol em C

Daniel Martins e Pedro Marques
115868

Contents

1	Introdução e Objetivos	2
2	Parametrização e Definições em probDataStruct.h	3
3	Parametrização e Definições em probConst.h	3
4	Parametrização e Definições em semaphore.h	4
5	semSharedMemReferee.c	5
5.1	arrive()	5
5.2	waitForTeams()	6
5.3	startGame()	8
5.4	play()	9
5.5	endGame()	10
6	semSharedMemPlayer.c e semSharedMemGoalie.c	11
6.1	arrive()	11
6.2	player/goalieConstituteTeam()	12
6.3	waitReferee()	16
6.4	playUntilEnd()	18
7	Conclusão	19

1 Introdução e Objetivos

Este projeto consiste no desenvolvimento de um programa em C que utiliza memória partilhada e conceitos de concorrência para simular um jogo de futebol. O objetivo principal é explorar técnicas de programação concorrente, sincronização e coordenação entre múltiplos processos que competem por recursos do sistema.

No contexto desta simulação, diferentes atores representam os elementos do jogo: os guarda-redes (*goalies*), os jogadores (*players*) e o árbitro (*referee*). Cada ator possui um papel específico na dinâmica do jogo, e todos devem cooperar e competir para alocar recursos computacionais necessários ao início e progresso da partida.

A utilização de memória partilhada permite que esses processos interajam diretamente, enquanto mecanismos de sincronização, como semáforos e *mutexes*, são empregados para garantir a consistência e evitar condições de corrida (*race conditions*). O objetivo final é proporcionar uma representação realista e eficiente das interações entre os diversos elementos do jogo.

Deste modo, foi-nos pedido para completarmos três programas em C denominados de *semSharedMemReferee.c*, *semSharedMemGoalie.c* e *semSharedMemPlayer.c*, que tiram partido da memória partilhada e dos semáforos disponibilizados para simular as interações entre os vários atores do jogo de futebol. Estes programas implementam os ciclos de vida dos diferentes intervenientes: os jogadores (*players*), que tentam formar equipas; os guarda-redes (*goalies*), que desempenham um papel crucial na constituição das equipas e no desenrolar do jogo; e o árbitro (*referee*), que é responsável por coordenar o início, a condução e o término do jogo.

A estrutura do projeto inclui a definição de constantes para parametrizar o problema, estados que representam a dinâmica dos intervenientes e estruturas de dados para monitorizar o progresso da simulação. A coordenação entre os processos é garantida através de operações de sincronização, como espera e sinalização em semáforos, que asseguram a execução ordenada das ações e a integridade dos recursos partilhados.

2 Parametrização e Definições em probDataStruct.h

A simulação do problema *SoccerGame* utiliza estruturas de dados internas para acompanhar o estado das entidades intervenientes (jogadores, guarda-redes e árbitros) e outras variáveis relacionadas com a execução da simulação. Na tabela abaixo, são apresentadas as definições das estruturas referidas.

Atributo	Descrição
playerStat[NUMPLAYERS]	Estado dos jogadores
goalieStat[NUMGOALIES]	Estado dos guarda-redes
refereeStat	Estado do árbitro

Atributo	Descrição
st	Estado de todas as entidades intervenientes
nPlayers	Número total de jogadores
nGoalies	Número total de guarda-redes
nReferees	Número total de árbitros
playersArrived	Jogadores que já chegaram
goaliesArrived	Guarda-redes que já chegaram
playersFree	Jogadores que chegaram e estão livres (sem equipa)
goaliesFree	Guarda-redes que chegaram e estão livres (sem equipa)
teamId	ID da equipa a ser formada (inicialmente = 1)

3 Parametrização e Definições em probConst.h

Neste documento, apresentamos os estados utilizados na simulação do problema *SoccerGame*. Cada estado representa uma etapa no ciclo de vida dos intervenientes (jogadores, guarda-redes e árbitro), desde a sua chegada até o fim do jogo. Estes estados são definidos através de caracteres constantes e são descritos na tabela abaixo.

Tipo	Valor
Jogador/Guarda-redes	'A' (ARRIVING)
Jogador/Guarda-redes	'W' (WAITING_TEAM)
Jogador/Guarda-redes	'F' (FORMING_TEAM)
Jogador/Guarda-redes	's' (WAITING_START_1)
Jogador/Guarda-redes	'S' (WAITING_START_2)
Jogador/Guarda-redes	'p' (PLAYING_1)
Jogador/Guarda-redes	'P' (PLAYING_2)
Jogador/Guarda-redes	'L' (LATE)
Árbitro	'A' (ARRIVINGR)
Árbitro	'W' (WAITING_TEAMS)
Árbitro	'S' (STARTING_GAME)
Árbitro	'R' (REFEREEING)
Árbitro	'E' (ENDING_GAME)

No âmbito da simulação do problema *SoccerGame*, certos parâmetros genéricos são utilizados para definir o número de jogadores, guarda-redes e árbitros, bem como a composição das equipas. Estes valores são apresentados na tabela abaixo para referência.

Parâmetro	Valor
Número total de jogadores	10
Número total de guarda-redes	3
Número total de árbitros	1
Número de jogadores por equipa	4
Número de guarda-redes por equipa	1

4 Parametrização e Definições em semaphore.h

O ficheiro `semaphore.h` define a interface para a gestão de semáforos no contexto de programação concorrente. Este módulo fornece um conjunto de operações fundamentais para criar, manipular e destruir conjuntos de semáforos. Essas operações são essenciais para sincronizar processos e controlar o acesso a recursos partilhados.

Abaixo está apresentada uma visão geral das principais funções disponíveis, incluindo suas descrições, parâmetros, e valores de retorno:

Função	Descrição	Parâmetros
<code>semCreate()</code>	Cria um novo conjunto de semáforos, todos iniciados no estado "red".	key: chave de criação snum: número de semáforos no conjunto
<code>semConnect()</code>	Conecta-se a um conjunto de semáforos previamente criado.	key: chave de criação
<code>semDestroy()</code>	Remove um conjunto de semáforos previamente criado.	semgid: identificador do conjunto de semáforos
<code>semSignal()</code>	Sinaliza a inicialização das operações, permitindo o uso de estruturas partilhadas	semgid: identificador do conjunto de semáforos
<code>semDown()</code>	Executa a operação down para reduzir o valor do semáforo, bloqueando se necessário.	semgid: identificador do conjunto sindex: índice do semáforo (1 a snum)
<code>semUp()</code>	Executa a operação up , aumentando o valor do semáforo e desbloqueando processos, se aplicável.	semgid: identificador do conjunto sindex: índice do semáforo (1 a snum)

5 semSharedMemReferee.c

O ficheiro *semSharedMemReferee.c* é dividido nas seguintes funções: *arrive()*, *waitForTeams()*, *startGame()*, *play()* e por fim *endgame()*.

5.1 arrive()

A função *arrive()* tem como principal objetivo definir o estado inicial do *referee* como **ARRIVING** e guarda esse estado com recurso ao comando *saveState*. Convém realçar que id

```
1 static void arrive ()
```

```

2 {
3     if (semDown (semgid, sh->mutex) == -1) {
4
5         /* enter critical region */
6         perror ("error on the up operation for
7             semaphore access (RF)");
8         exit (EXIT_FAILURE);
9     }
10
11     /* TODO: insert your code here */
12     sh->fSt.st.refereeStat=ARRIVINGR;
13     saveState(nFic, &sh -> fSt);
14
15     if (semUp (semgid, sh->mutex) == -1) {
16
17         /* leave critical region */
18         perror ("error on the up operation for
19             semaphore access (RF)");
20         exit (EXIT_FAILURE);
21     }
22
23     usleep((100.0*random())/(RAND_MAX+1.0)+10.0);
24 }

```

5.2 waitForTeams()

Quando o *referee* chega, ele tem de esperar que as equipas sejam formadas. Assim temos de atualizar o seu estado para **WAITING_TEAMS**.

```

1     static void waitForTeams ()
2 {
3     if (semDown (semgid, sh->mutex) == -1) {
4
5         /* enter critical region */
6         perror ("error on the up operation for
7             semaphore access (RF)");
8         exit (EXIT_FAILURE);
9     }

```

```

6      }
7
8      /* TODO: insert your code here */
9      sh->fSt.st.refereeStat = WAITING_TEAMS;
10     saveState(nFic,&sh->fSt);
11
12     if (semUp (semgid, sh->mutex) == -1) {
13
14         /* leave critical region */
15         perror ("error on the up operation for
16                 semaphore access (RF)");
17         exit (EXIT_FAILURE);
18     }
19
20     /* TODO: insert your code here */
21
22     // Metemos 2 sem foros down enquanto esperamos que
23     as equipas sejam formadas
24     if (semDown (semgid, sh->refereeWaitTeams) == -1) {
25         perror ("error on the up operation for
26                 semaphore access (RF)");
27         exit (EXIT_FAILURE);
28     }
29
30     if (semDown (semgid, sh->refereeWaitTeams) == -1) {
31         perror ("error on the up operation for
32                 semaphore access (RF)");
33         exit (EXIT_FAILURE);
34     }
35 }

```

Aqui, fazemos uso do semáforo *refereeWaitTeams*, o qual fazemos down 2 vezes com o intuito de o bloquear até que ambas as equipas estejam formadas, cada uma fazendo up, para assim desbloquear o semáforo.

5.3 startGame()

Aqui, as equipas já se encontram formadas e o árbitro pode inicializar o jogo. Deste modo, mudamos o estado atual do árbitro para **STARTING_GAME**.

```
1  static void startGame ()
2  {
3      if (semDown (semgid, sh->mutex) == -1) {
4
5          /* enter critical region */
6          perror ("error on the up operation for
7              semaphore access (RF)");
8          exit (EXIT_FAILURE);
9      }
10
11      /* TODO: insert your code here */
12      sh->fSt.st.refereeStat=STARTING_GAME;
13      saveState(nFic, &sh -> fSt);
14
15      if (semUp (semgid, sh->mutex) == -1) {
16
17          /* leave critical region */
18          perror ("error on the up operation for
19              semaphore access (RF)");
20          exit (EXIT_FAILURE);
21      }
22
23      /* TODO: insert your code here */
24
25      // por cada player colocamos o semaforo Up para
26      eles saberem que o jogo come ou
27      for (int player = 0; player < 10; player++) {
28          if (semUp (semgid, sh->playersWaitReferee) ==
29              -1) {
30              perror ("error on the up operation for
31                  semaphore access (RF)");
32              exit (EXIT_FAILURE);
33          }
34      }
35  }
```

Desta vez, usamos o semáforo *playersWaitReferee*, ao qual fazemos up 10 vezes. Este semáforo estava a bloquear todos os jogadores que estavam à espera do início do jogo (10 jogadores), assim ao fazermos semUp 10 vezes desbloqueamos todos os jogadores.

5.4 play()

Tal como na função *arrive()*, apenas alteramos o estado atual do referee, mas desta vez ele passa a **REFEREEING**.

```
1  static void play ()
2  {
3      if (semDown (semgid, sh->mutex) == -1) {
4
5          /* enter critical region */
6          perror ("error on the up operation for
7              semaphore access (RF)");
8          exit (EXIT_FAILURE);
9      }
10
11     /* TODO: insert your code here */
12     sh->fSt.st.refereeStat=REFEREEING;
13     saveState(nFic, &sh -> fSt);
14
15     if (semUp (semgid, sh->mutex) == -1) {
16
17         /* leave critical region */
18         perror ("error on the up operation for
19             semaphore access (RF)");
20         exit (EXIT_FAILURE);
21     }
22
23     usleep((100.0*random())/(RAND_MAX+1.0)+900.0);
24 }
```

5.5 endGame()

O jogo termina e o árbitro passa para o seu último estado: **ENDING_GAME**.

```
1  static void endGame ()
2  {
3      if (semDown (semgid, sh->mutex) == -1) {
4
5          /* enter critical region */
6          perror ("error on the up operation for
7              semaphore access (RF)");
8          exit (EXIT_FAILURE);
9      }
10
11      /* TODO: insert your code here */
12      sh->fSt.st.refereeStat=ENDING_GAME;
13      saveState(nFic, &sh -> fSt);
14
15      if (semUp (semgid, sh->mutex) == -1) {
16
17          /* leave critical region */
18          perror ("error on the up operation for
19              semaphore access (RF)");
20          exit (EXIT_FAILURE);
21      }
22
23      /* TODO: insert your code here */
24
25      // por cada player colocamos o semaforo Up para
26      // eles saberem que o jogo acabou
27      for (int player = 0; player < 10; player++) {
28          if (semUp (semgid, sh->playersWaitEnd) == -1) {
29              perror ("error on the up operation for
30                  semaphore access (RF)");
31              exit (EXIT_FAILURE);
32          }
33      }
34  }
```

Por fim repetimos a instrução *semUp*(do semáforo *playersWaitEnd*) exatamente

10 vezes para indicar aos jogadores que o jogo chegou ao fim.

6 semSharedMemPlayer.c e semSharedMemGoalie.c

As funções do *Goalie* e do *Player* partilham muitas características, sendo os seus códigos praticamente semelhantes na sua íntegra, apenas com algumas pequenas diferenciações. Portanto, para não tornar o processo de leitura exaustivo, decidimos condensar a informação relativa a estes dois agentes numa só secção, realçando as diferenças relevantes quando necessário.

Cada um dos ficheiros *semSharedMemPlayer.c* e *semSharedMemGoalie.c* é dividido nas seguintes funções: *arrive()*, *player/goalieConstituteTeam()*, *waitReferee()* e por fim *playUntilEnd()*.

6.1 arrive()

A função *arrive()* tem como principal objetivo definir o estado inicial do *player/goalie* como **ARRIVING** e guarda esse estado com recurso ao comando *saveState*. Nota-se que, neste caso, como existem muitos jogadores/guarda-redes, é necessário parametrizar o estado com o *id* do agente passado como argumento á função. A baixo segue-se o código relativo ao *player*. O código relativo ao *goalie* é quase idêntico, apenas havendo a diferença na linha da declaração de estado.

```
1 static void arrive (int id){
2     if (semDown (semgid, sh->mutex) == -1) {
3
4         /* enter critical region */
5         perror ("error on the up operation for
6             semaphore access (RF)");
7         exit (EXIT_FAILURE);
8     }
9     /* TODO: insert your code here */
    sh->fSt.st.playerStat[id]=ARRIVING;
    saveState(nFic, &sh -> fSt);
```

```

10     if (semUp (semgid, sh->mutex) == -1) {
        /* leave critical region */
11         perror ("error on the up operation for
            semaphore access (RF)");
12         exit (EXIT_FAILURE);
13     }
14     usleep((100.0*random())/(RAND_MAX+1.0)+10.0);
15
16 }

```

No caso do *goalie* seria:

```

1 sh->fSt.st.goalieStat[id]=ARRIVING;

```

6.2 player/goalieConstituteTeam()

Esta função tem como objetivo constituir 2 equipas. Conforme os *players* e *goalies* vão chegando eles incrementam as variáveis **playersArrived** ou **goaliesArrived** e **playersFree** ou **GoaliesFree** respetivamente. Caso passe a existir gente suficiente para formar uma equipa quando o *goalie* ou o *player* chegam, esse jogador é nomeado de "capitão" e começa a formar equipa. Senão, o estado deles é atualizado para **WAITING_TEAM** e damos *semDown* no semáforo *playersWaitTeam* ou no semáforo *goaliesWaitTeam* respetivamente para blquearmos os mesmos. Por outro lado, caso já existam 8 *players*, os proximos *players* que chegarem são considerados como atrasados e o seu estado é atualizado para **LATE**. Relativamente aos *goalies*, isto acontece quando já existem 2.

Caso o *player* seja capitão, ele passa para o estado **FORMING_TEAM** e decrementamos a variável **playersFree** 4 vezes e a variável **goaliesFree** 1 vez.

Fazemos também *semUp* 3 vezes no semaforo *playersWaitingTeam* e uma no semáforo *goaliesWaitTeam* ou caso o *goalie* seja o capitão, fazemos *semUp* 4 vezes no semaforo *playersWaitingTeam*, para assim passarmos ao registo dos *players* e *goalies* nas equipa.

Para registarmos os *players* nas equipas fazemos *semDown* 4 vezes no semáforo *playerRegistered*, não importando agora se o jogador é *goalie* ou *player*.

Por fim atribuímos o *teamID* atual à equipa e incrementamos para a criação da segunda equipa e fazemos uma vez *semUp* no semáforo *refereeWaitTeams* desbloqueando assim o árbitro quando a segunda equipa acabar de ser formada e

registrada. Abaixo segue-se a solução obtida para o módulo relativo ao *player*:

```
1  static int playerConstituteTeam (int id)
2  {
3      int ret = 0;
4
5      if (semDown (semgid, sh->mutex) == -1) {
6
7          /* enter critical region */
8          perror ("error on the up operation for
9              semaphore access (PL)");
10         exit (EXIT_FAILURE);
11     }
12
13     /* TODO: insert your code here */
14     // Aumentamos o numero de players que chegaram e
15     // que est o livres
16     sh->fSt.playersArrived++;
17     sh->fSt.playersFree++;
18
19     if (sh->fSt.playersArrived <= 8){
20         // Verifica o se o player for
21         // capit o (Ultimo jogador necess rio para
22         // formar equipa)
23         if (sh->fSt.playersFree >= NUMTEAMPLAYERS &&
24             sh->fSt.goaliesFree >= NUMTEAMGOALIES){
25             sh->fSt.st.playerStat[id] = FORMING_TEAM;
26
27             sh->fSt.playersFree -= NUMTEAMPLAYERS;
28             sh->fSt.goaliesFree -= NUMTEAMGOALIES;
29
30             // colocamos o semaforo Up para todos os
31             // players menos o capit o
32             for (int player = 0; player < 3; player++){
33                 if (semUp (semgid, sh->playersWaitTeam)
34                     == -1) {
35                     perror ("error on the up operation
36                         for semaphore access (PL)");
37                     exit (EXIT_FAILURE);
38                 }
39             }
40         }
41     }
```

```

29         }
30     }
31
32     // colocamos o semaforo Up para o
33     // guarda-redes
34     if (semUp (semgid, sh->goaliesWaitTeam) ==
35         -1) {
36         perror ("error on the up operation for
37             semaphore access (GL)");
38         exit (EXIT_FAILURE);
39     }
40
41     // colocamos o semaforo down para registrar
42     // os jogadores na equipa menos o capit o
43     for (int player = 0; player < 4; player++){
44         if (semDown (semgid,
45             sh->playerRegistered) == -1) {
46             perror ("error on the down
47                 operation for semaphore access
48                 (PL)");
49             exit (EXIT_FAILURE);
50         }
51     }
52
53     // associamos a equipa a um id
54     ret = sh->fSt.teamId;
55     sh->fSt.teamId++;
56     saveState(nFic, &sh->fSt);
57
58 } else {
59     // Caso este player n o seja capit o
60     sh->fSt.st.playerStat[id] = WAITING_TEAM;
61     saveState(nFic, &sh->fSt);
62 }
63
64 } else {
65     // Caso este player chegue atrasado
66     sh->fSt.st.playerStat[id] = LATE;
67     saveState(nFic, &sh->fSt);
68     sh->fSt.playersFree -= 1;

```

```

62     }
63
64
65     if (semUp (semgid, sh->mutex) == -1) {
66
67         /* exit critical region */
68         perror ("error on the down operation for
69             semaphore access (PL)");
70         exit (EXIT_FAILURE);
71     }
72
73     /* TODO: insert your code here */
74
75     if (sh->fSt.st.playerStat[id] == WAITING_TEAM){
76         // Quando o player n o capit o
77         if (semDown (semgid, sh->playersWaitTeam) ==
78             -1) {
79             perror ("error on the down operation for
80                 semaphore access (PL)");
81             exit (EXIT_FAILURE);
82         }
83
84         // Associamos a equipa a um id
85         ret = sh->fSt.teamId;
86
87         // Colocamos o sem foro up para registar o
88         // player
89         if (semUp (semgid, sh->playerRegistered) == -1)
90         {
91             perror ("error on the up operation for
92                 semaphore access (PL)");
93             exit (EXIT_FAILURE);
94         }
95
96     } else if (sh->fSt.st.playerStat[id] ==
97         FORMING_TEAM){
98         // Caso o player seja capit o
99         // Colocamos o semaforo up para registar a
100         // equipa e informar o rbitro
101         if (semUp (semgid, sh->refereeWaitTeams) == -1)

```



```

92         {
93             perror ("error on the up operation for
94                 semaphore access (RF)");
95             exit (EXIT_FAILURE);
96         }
97     }
98     return ret;
99 }

```

6.3 waitReferee()

Nesta etapa, as equipas já se estão formadas e encontram-se à espera que o árbitro dê início ao jogo. Assim, é necessário atualizar o estado das mesmas para **WAITING_START_1** e **WAITING_START_2** conforme os *id* de cada equipa. A baixo segue-se o código relativo ao *player*. O código do *goalie* segue exatamente a mesma estrutura, apenas substituindo a palavra *player* por *goalie*.

```

1  static void waitReferee (int id, int team)
2  {
3      if (semDown (semgid, sh->mutex) == -1) {
4
5          /* enter critical region */
6          perror ("error on the up operation for
7              semaphore access (PL)");
8          exit (EXIT_FAILURE);
9      }
10
11     /* TODO: insert your code here */
12
13     // Verificar em que equipa est o player e
14     // atribuir-lhe o estado s ou S ('equipa 1 espera o
15     // come o' ou 'equipa 2 espera o come o')
16     if(team == 1){
17         sh->fSt.st.playerStat[id] = WAITING_START_1;
18     }else{
19         sh->fSt.st.playerStat[id] = WAITING_START_2;
20     }
21     saveState(nFic, &sh->fSt);

```

```

17
18     if (semUp (semgid, sh->mutex) == -1) {
19
20         /* exit critical region */
21         perror ("error on the down operation for
22             semaphore access (PL)");
23         exit (EXIT_FAILURE);
24     }
25
26     /* TODO: insert your code here */
27
28     // Colocamos o semaforo down enquanto esperamos que
29     // o rbitro comece o jogo
30     if (semDown (semgid, sh->playersWaitReferee) == -1)
31     {
32         perror ("error on the up operation for
33             semaphore access (PL)");
34         exit (EXIT_FAILURE);
35     }
36 }

```

No caso do *goalie* seria:

```

1     if(team == 1){
2         sh->fSt.st.goalieStat[id] = WAITING_START_1;
3     }else{
4         sh->fSt.st.goalieStat[id] = WAITING_START_2;
5     }
6     .
7     .
8     .
9     if (semDown (semgid, sh->goaliesWaitReferee) == -1)
10    {
11        perror ("error on the up operation for
12            semaphore access (PL)");
13        exit (EXIT_FAILURE);
14    }

```

Cada equipa faz semDown, ficando assim ambas bloqueadas até que o árbitro comece o jogo e os desbloqueie na função *startGame()*.

6.4 playUntilEnd()

Nesta função, o jogadores começam a jogar. Assim é necessário atualizar o seu estado para **PLAYING_1** caso pertençam à equipa de *id* 1 ou **PLAYING_2** caso pertençam à equipa de *id* 2. Abaixo encontra-se o código relativamente ao *player*. O código do *goalie* é praticamente idêntico, apenas substituindo a palavra *player* por *goalie*.

```
1 static void playUntilEnd (int id, int team){
2     if (semDown (semgid, sh->mutex) == -1) {
3
4         /* enter critical region */
5         perror ("error on the up operation for
6             semaphore access (PL)");
7         exit (EXIT_FAILURE);
8     }
9
10    /* TODO: insert your code here */
11
12    // Verificar em que equipa est o player e
13    // atribuir-lhe o estado p ou P ('A jogar na equipa
14    // 1' ou 'A jogar na equipa 2')
15    if(team == 1){
16        sh->fSt.st.playerStat[id] = PLAYING_1;
17    }else{
18        sh->fSt.st.playerStat[id] = PLAYING_2;
19    }
20    saveState(nFic, &sh->fSt);
21
22    if (semUp (semgid, sh->mutex) == -1) {
23
24        /* exit critical region */
25        perror ("error on the down operation for
26            semaphore access (PL)");
27        exit (EXIT_FAILURE);
28    }
29
30    /* TODO: insert your code here */
31    // Colocamos o sem foro down para esperar que o
32    // rbitro acabe o jogo
33    if (semDown (semgid, sh->playersWaitEnd) == -1) {
```

```

24     perror ("error on the up operation for
25             semaphore access (PL)");
26     exit (EXIT_FAILURE);
27 }

```

No caso do *goalie* seria:

```

1     if(team == 1){
2         sh->fSt.st.goalieStat[id] = PLAYING_1;
3     }else{
4         sh->fSt.st.goalieStat[id] = PLAYING_2;
5     }
6     .
7     .
8     .
9     if (semDown (semgid, sh->goaliesWaitEnd) == -1) {
10         perror ("error on the up operation for
11                 semaphore access (PL)");
12         exit (EXIT_FAILURE);
13     }

```

Recorremos ao comando *semDown* do semáforo *playersWaitEnd*, para bloquearmos cada jogador, com o objetivo de os fazer esperar que o árbitro indique o final da partida, desbloqueando assim todos os jogadores dentro da função *endGame()*.

7 Conclusão

Concluindo, o projeto desenvolvido destaca a importância da utilização de conceitos de programação concorrente, como memória partilhada e sincronização, para a criação de uma simulação eficiente e realista de um jogo de futebol. Ao empregar técnicas de sincronização, como semáforos e *mutexes*, a aplicação permite que múltiplos processos, representando jogadores, guarda-redes e um árbitro, interajam e compartilhem recursos de forma ordenada, evitando condições de corrida e garantindo a integridade do sistema. A estrutura do projeto, com o uso de constantes para parametrizar os estados e a dinâmica do jogo, bem como o controle de progresso através de semáforos, contribui para a implementação de uma simulação

robusta e coordenada. Em última análise, o projeto oferece uma oportunidade valiosa para explorar e aplicar técnicas de programação concorrente dentro de um contexto prático, fornecendo uma base sólida para o desenvolvimento de sistemas mais complexos que exigem a interação e coordenação entre múltiplos processos.

Fim do Relatório

Daniel Martins e Pedro Marques

Janeiro 2025