Projeto 2

Sistemas Operativos

Simulação de um jogo de futebol em C

Daniel Martins e Pedro Marques
115868 118895

Contents

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 caractéres 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
`semCreate()`	Cria um novo conjunto de semáforos, todos iniciados no estado "red".	`key`: chave de criação `snum`: número de semáforos no conjunto
`semConnect()`	Conecta-se a um conjunto de semáforos previamente criado.	`key`: chave de criação
`semDestroy()`	Remove um conjunto de semáforos previamente criado.	`semgid`: identificador do conjunto de semáforos
`semSignal()`	Sinaliza a inicialização das operações, permitindo o uso de estruturas partilhadas	`semgid`: identificador do conjunto de semáforos
`semDown()`	Executa a operação `down` para reduzir o valor do semáforo, bloqueando se necessário.	`semgid`: identificador do conjunto `sindex`: índice do semáforo (1 a `snum`)
`semUp()`	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 Parametrização e Definições em sharedDataSync.h

Este código define uma interface de programação para sincronização baseada em semaforos e memória compartilhada no contexto de um problema do projeto. O propósito do arquivo é especificar como os dados compartilhados e os dispositivos de sincronização são organizados para coordenar diferentes entidades envolvidas no sistema. A estrutura principal **SHARED_DATA** contém os identificadores dos semáforos. A tabela abaixo descreve os semáforos e as suas funções, indicando o papel de cada um no contexto do problema:

Semáforo	Função
MUTEX	Proteção de acesso à região crítica (controlo de exclusão mútua).
PLAYERSWAITTEAM	Jogadores aguardam a formação de equipas.
GOALIESWAITTEAM	Guarda-redes aguardam a formação de equipas.
PLAYERSWAITREFEREE	Jogadores e guarda-redes aguardam o árbitro para iniciar o jogo.
PLAYERSWAITEND	Jogadores e guarda-redes aguardam o término do jogo.
REFEREEWAITTEAMS	Árbitro aguarda a formação de equipas.
PLAYERREGISTERED	Jogadores e guarda-redes confirmam o registo na equipa.
PLAYING	Árbitro aguarda que todos estejam prontos para iniciar o jogo.

6 semSharedMemReferee.c

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

6.1 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) {

          /* enter critical region */
4         perror ("error on the up operation for
              semaphore access (RF)");
5         exit (EXIT_FAILURE);
```

```
6       }
7
8       /* TODO: insert your code here */
9       sh->fSt.st.refereeStat=ARRIVINGR;
10      saveState(nFic, &sh -> fSt);
11
12      if (semUp (semgid, sh->mutex) == -1) {

           /* leave critical region */
13          perror ("error on the up operation for
               semaphore access (RF)");
14          exit (EXIT_FAILURE);
15      }
16
17      usleep((100.0*random())/(RAND_MAX+1.0)+10.0);
18
19  }
```

6.2 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) {

           /* enter critical region */
4           perror ("error on the up operation for
               semaphore access (RF)");
5           exit (EXIT_FAILURE);
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) {
```

```
        /* leave critical region */
13          perror ("error on the up operation for
                semaphore access (RF)");
14          exit (EXIT_FAILURE);
15      }
16
17
18
19      /* TODO: insert your code here */
20
21      // Metemos 2 sem foros down enquanto esperamos que
            as equipas sejam formadas
22      if (semDown (semgid, sh->refereeWaitTeams) == -1) {
23          perror ("error on the up operation for
                semaphore access (RF)");
24          exit (EXIT_FAILURE);
25      }
26
27      if (semDown (semgid, sh->refereeWaitTeams) == -1) {
28          perror ("error on the up operation for
                semaphore access (RF)");
29          exit (EXIT_FAILURE);
30      }
31  }
```

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.

6.3 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) {
```

```
        /* enter critical region */
4           perror ("error on the up operation for
               semaphore access (RF)");
5           exit (EXIT_FAILURE);
6       }
7
8       /* TODO: insert your code here */
9       sh->fSt.st.refereeStat=STARTING_GAME;
10      saveState(nFic, &sh -> fSt);
11
12      if (semUp (semgid, sh->mutex) == -1) {

        /* leave critical region */
13          perror ("error on the up operation for
               semaphore access (RF)");
14          exit (EXIT_FAILURE);
15      }
16
17      /* TODO: insert your code here */
18
19      // por cada player colocamos o semaforo Up para
           eles saberem que o jogo come ou
20      for (int player = 0; player < 10; player++) {
21          if (semUp (semgid, sh->playersWaitReferee) ==
               -1) {
22              perror ("error on the up operation for
                   semaphore access (RF)");
23              exit (EXIT_FAILURE);
24          }
25      }
26      for (int player = 0; player < 10; player++) {
27          if (semDown (semgid, sh->playing) == -1) {
28              perror ("error on the up operation for
                   semaphore access (RF)");
29              exit (EXIT_FAILURE);
30          }
31      }
32
33  }
```

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.

6.4 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) {

          /* enter critical region */
4          perror ("error on the up operation for
               semaphore access (RF)");
5          exit (EXIT_FAILURE);
6     }
7
8     /* TODO: insert your code here */
9     sh->fSt.st.refereeStat=REFEREEING;
10    saveState(nFic, &sh -> fSt);
11
12    if (semUp (semgid, sh->mutex) == -1) {

          /* leave critical region */
13         perror ("error on the up operation for
               semaphore access (RF)");
14         exit (EXIT_FAILURE);
15    }
16
17    usleep((100.0*random())/(RAND_MAX+1.0)+900.0);
18 }
```

6.5 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) {

         /* enter critical region */
4         perror ("error on the up operation for
             semaphore access (RF)");
5         exit (EXIT_FAILURE);
6     }
7
8     /* TODO: insert your code here */
9     sh->fSt.st.refereeStat=ENDING_GAME;
10    saveState(nFic, &sh -> fSt);
11
12    if (semUp (semgid, sh->mutex) == -1) {

         /* leave critical region */
13        perror ("error on the up operation for
             semaphore access (RF)");
14        exit (EXIT_FAILURE);
15    }
16
17    /* TODO: insert your code here */
18
19    // por cada player colocamos o semaforo Up para
         eles saberem que o jogo acabou
20    for (int player = 0; player < 10; player++) {
21        if (semUp (semgid, sh->playersWaitEnd) == -1) {
22            perror ("error on the up operation for
                 semaphore access (RF)");
23            exit (EXIT_FAILURE);
24        }
25    }
26 }
```

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

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

7 semSharedMemPlayer.c e semSharedMemGoalie.c

As funções do *Goalie* e do *Player* partilham muitas características, sendo os seus códigos práticamente semelhantes na sua integra, 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()*.

7.1 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) {

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

```
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;
```

7.2 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

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

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

```
62      }
63
64
65      if (semUp (semgid, sh->mutex) == -1) {

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

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

7.3 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 substituíndo a palavra *player* por *goalie*.

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

```
17
18      if (semUp (semgid, sh->mutex) == -1) {

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

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          perror ("error on the up operation for
               semaphore access (PL)");
11          exit (EXIT_FAILURE);
12      }
```

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

7.4 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 práticamente idêntico, apenas substituíndo a palavra *player* por *goalie*.

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

```
            semaphore access (RF)");
25          exit (EXIT_FAILURE);
26      }
27
28      // Colocamos o semforo down para esperar que o
            rbitro  acabe o jogo
29      if (semDown (semgid, sh->playersWaitEnd) == -1) {
30          perror ("error on the up operation for
                semaphore access (PL)");
31          exit (EXIT_FAILURE);
32      }
33  }
```

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 (semUp (semgid, sh->playing) == -1) {
10          perror ("error on the up operation for
                semaphore access (RF)");
11          exit (EXIT_FAILURE);
12      }
13
14      if (semDown (semgid, sh->goaliesWaitEnd) == -1) {
15          perror ("error on the up operation for
                semaphore access (PL)");
16          exit (EXIT_FAILURE);
17      }
```

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()*.

8 Conclusão

Concluíndo, 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