Uma imagem com captura de ecrã, Gráficos, Tipo de letra, design gráfico

Descrição gerada automaticamente

**Projeto 2 SO**

Mecanismos associados à execução e sincronização de processos e *threads* – Simulação de uma partida de futebol

Marta Cruz nº 119572

Matilde Rodrigues nº 119714

**Resumo**

O objetivo deste trabalho consistiu em desenvolver uma estratégia eficaz e eficiente de forma para sincronizar processos independentes poe meio da utilização de semáforos e segmentos de memória partilhada. Esta sincronização vai nos permitir simular uma partida de futebol que terá como principais “entidades” : 10 *Players* (Jogadores), 3 *Goalies* (Guarda-redes) e 1 *Referee* (árbitro). A implementação desta solução foi realizada dando uso à linguagem de programação C sendo a saída do programa apresentada no terminal ou num ficheiro cujo nome é passado como argumento aquando da execução do programa. A estrutura do trabalho foi dividida em módulos e foram realizados diversos testes para garantir o funcionamento esperado da solução implementada.

**Índice**

[**1.** **Introdução** 3](#_Toc185892982)

[**2.** **Variáveis** 4](#_Toc185892983)

[2.1. Gerais 4](#_Toc185892984)

[2.2. De estado 4](#_Toc185892985)

[**3.** **Estruturas de dados** 6](#_Toc185892986)

[**4.** **Semáforos** 8](#_Toc185892987)

[**5.** **Funções** 11](#_Toc185892988)

[5.1. semSharedMemReferee.c 11](#_Toc185892989)

[5.2. semSharedMemPlayer.c 14](#_Toc185892990)

[5.3. semSharedMemGoalie.c 19](#_Toc185892991)

[**6.** **Testes** 23](#_Toc185892992)

[**7.** **Conclusão** 24](#_Toc185892993)

[**8.** **Webgrafia** 25](#_Toc185892994)

# **Introdução**

O trabalho proposto, no âmbito da disciplina de Sistemas Operativos, consiste no desenvolvimento de um programa que simule um jogo de futebol com 3 entidades diferentes: *referee*, *players* e *goalies*.

*Referee* -> é ele (o árbitro) que dá início ou fim ao jogo, após assegurar que todos os jogadores de ambas as equipas se encontram prontas para tal.

*Players* -> cada equipa tem 4 *players* (jogadores) que após a sua chegada tentam formar a equipa; caso isso não seja possível, podem existir mais do que 4 jogadores, esses são avisados que chegaram atrasados e não entram no jogo.

*Goalies* -> igual aos players, a única diferença é que é 1 *goalie* (guarda-redes) e não 4 jogadores.

Estes são os chamados processos independentes do nosso trabalho. Também é necessária a sincronização e comunicação entre os processos, que serão alcançadas através do uso de semáforos e de memória partilhada. Além disso, cada um destes processos deverá estar bloqueado enquanto espera por um determinado evento. Só deve estar ativo quando for necessário.

Este projeto, desenvolvido na linguagem de programação C, tem como principal objetivo compreender a forma como os semáforos desempenham o papel na manipulação e sincronização de múltiplos processos simultaneamente, de modo a garantir a solução pretendida.

# **Variáveis**

Primeiramente, analisámos as variáveis gerais e de estado que estão definidas no ficheiro *probConst.h*. Cada uma destas variáveis tem a sua importância (que será vista nos pontos a seguir). Elas são responsáveis por coordenar o comportamento das entidades de maneira ordenada, lógica e eficiente.

## **Gerais**

|  |  |
| --- | --- |
| Variável | Valor |
| *NUMPLAYERS* | 10 |
| *NUMGOALIES* | 3 |
| *NUMREFEREES* | 1 |
| *NUMTEAMPLAYERS* | 4 |
| *NUMTEAMGOALIES* | 1 |

Tabela 1 - Variáveis Gerais

As variáveis gerais são aquelas que definem os limites e as configurações principais do programa, tais como:

* **NUMPLAYERS**, **NUMGOALIES** e **NUMREFEREES** - determinam o número máximo de jogadores, guarda-redes e árbitros que podem participar na simulação de uma partida de futebol
* **NUMTEAMPLAYERS** e **NUMTEAMGOALIES** - especificam a composição de cada equipa, ou seja, garantem que cada equipa tem 4 jogadores e 1 guarda-redes

## **De estado**

|  |  |
| --- | --- |
| Estado | Valor |
| *ARRIVING* | 0 |
| *WAITING\_TEAM* | 1 |
| *FORMING\_TEAM* | 2 |
| *WAITING\_START\_1* | 3 |
| *WAITING\_START\_2* | 4 |
| *PLAYING\_1* | 5 |
| *PLAYING\_2* | 6 |
| *LATE* | 7 |

|  |  |
| --- | --- |
| Estado | Valor |
| *ARRIVINGR* | 0 |
| *WAITING\_TEAMS* | 1 |
| *STARTING\_GAME* | 2 |
| *REFEREEING* | 3 |
| *ENDING\_GAME* | 4 |

Tabela 3 - Variáveis de estado do Referee

Tabela 2 - Variáveis de estado dos Players/Goalies

As variáveis de estado são aquelas que acompanham a etapa em que cada entidade está, tal como:

* Estados dos *players*/*goalies*
  + **ARRIVING** - o jogador/guarda-redes (entidade) está a chegar
  + **WAITING\_TEAM** - a entidade ainda não está em nenhuma equipa e está à espera para ser colocado numa
  + ***F*ORMING\_TEAM** - a entidade está a formar uma equipa
  + **WAITING\_STAR\_1** e **WAITING\_START\_2** - a entidade está pronta e está a aguardar o início do jogo
  + **PLAYING\_1** e **PLAYING\_*2*** - a entidade está a jogar pela equipa correspondente (equipa 1 e 2 em jogo, respetivamente)
  + **LATE** - a entidade chegou atrasada e não participa no jogo
* Estados do *referee*
  + **ARRIVINGR** - o árbitro está a chegar
  + **WAITING\_TEAMS** - o árbitro está à espera de que as equipas sejam formadas
  + **STARTING\_GAME** - o árbitro está a iniciar o jogo
  + **REFEREEING** -> o árbitro está em ação (a fazer o papel de árbitro) durante o jogo
  + **ENDING\_GAME** - o árbitro está a terminar o jogo

# **Estruturas de dados**

De seguida, observámos as estruturas de dados presentes no ficheiro *probDataStruct.h*. Estas estruturas são fundamentais para organizar as informações do jogo. Neste ficheiro foram definidas duas estruturas de dados: a primeira que armazenas os estados de cada entidade, e a segunda que é a estrutura principal que combina informações gerais e de estados.

Uma imagem com texto, Tipo de letra, captura de ecrã, file

Descrição gerada automaticamente

Figura 1 - 1ª estrutura de dados

Nesta primeira estrutura (*STAT*) são rastreados os dados de cada entidade no jogo (*players*, *goalies* e *referee*). O primeiro *array* definido (*playersStat[NUMPLAYERS]*) é o que guarda o estado de cada jogador (vai buscar o número do jogador pela variável geral *NUMPLAYERS*). Já o segundo *array* (*goalieStat[NUMGOALIES]*) guarda o estado de cada guarda-redes (tal como no primeiro *array*, usa a variável geral *NUMGOALIES* com a mesma função da anterior). Por fim, a última variável (*refereeeStart*) é aquela que guarda o estado do árbitro. É uma variável e não um *array* pois só existe um árbitro, em contrário com os jogadores (10) e guarda-redes (4).

Uma imagem com texto, captura de ecrã, Tipo de letra, documento

Descrição gerada automaticamente

Figura 2 - 2ª estrutura de dados

Já nesta 2ª estrutura (*FULL\_STAT*), como é a principal, combina as informações dos estados e variáveis gerais usadas no jogo. Primeiro, esta estrutura começa por armazenar todos os dados, com uma instância de estrutura *STAT* (*STAT st*). Também conta com valores inteiros que definem o número total de jogadores, guarda-redes e árbitros (*nPlayers*, *nGoalies* e *nReferees*). De seguida, conta quantos jogadores e guarda-redes é que já chegaram (*playerArrived* e *goaliesArrived*). Estas informações vão ser úteis para determinar se o jogo pode começar ou não. Além destas contagens, também se conta quantos jogadores e guarda-redes estão disponíveis para formar as equipas (*playersFree* e *goaliesFree*). Para terminar, o *teamID* controla o *ID* da próxima equipa a ser formada (este alterna entre 1 e 2 de forma a atribuir jogadores e guarda-redes às equipas).

# **Semáforos**

Para este trabalho foram utilizados semáforos, e achámos que é crucial saber o que são, quais são e o que fazem. Assim, os semáforos (definidos no ficheiro *sharedDataSync.h*) são tipos de dados abstratos que permitem a sincronização de *threads*/processos (sem *busy* *waiting*). Eles permitem controlar os processos sequencialmente, bloqueando os processos caso seja necessário.

Assim, criámos esta tabela para nos ajudar durante a implementação dos projetos, de modo a compreendermos cada semáforo e o seu comportamento:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Semáforo | Entidade down | Função down | #down | Entidade up | Função up | #up |
| *playersWaitTeam* | Player | *playersConstitureTeam* | 1 | Goalie | *goalieConstituteTeam* | 4 |
| Player | *playerConstitureTeam* | 3 |
| *goaliesWaitTeam* | Goalie | *goalieConstituteTeam* | 1 | Player | *playerConstituteTeam* | 1 |
| *playersWaitReferee* | Player e Goalie | *waitReferee* | 1 | Referee | *startGame* | 10 |
| *playersWaitEnd* | Player e Goalie | *playUntilEnd* | 1 | Referee | *endGame* | 10 |
| *refereeWaitTeams* | Referee | *waitForTeams* | 2 | Goalie | *goalieConstituteTeam* | 1 |
| Player | *playerConstituteTeam* | 1 |
| *playerRegistered* | Player e Goalie | *goalieConstituteTeam* | 4 | Goalie | *goalieConstituteTeam* | 1 |
| *playerConstituteTeam* | 4 | Player | *playerConstituteTeam* | 1 |

Explicação de cada coluna da tabela:

* **Semáforo** - semáforo que está a ser analisado
* **Entidade down** - a entidade que é bloqueada pelo semáforo, ou seja, quem realiza a operação down para aguardar uma condição específica
* **Função down** - o nome da função no código que realiza a operação down (representa onde a entidade fica bloqueada)
* **#down** - quantidade de vezes que o semáforo é decrementado num determinado caso (depende do nº de entidades que precisam de ser bloqueadas)
* **Entidade up** - a entidade que é responsável por libertar o semáforo, ou seja, quem realiza a operação up
* **Função up** - o nome da função no código que realiza a operação up (indica onde a entidade é libertada)
* **#up** - quantidade de vezes que o semáforo é incrementado (depende do nº de entidades que precisam de ser libertadas)

Além desta tabela, também tivemos de compreender o propósito de cada semáforo, o que é que cada um deles representa:

Assegura que apenas uma entidade possa executar o processo. Quando entramos na região crítica do programa, uma entidade executa *down* no *mutex* para bloquear o acesso a outras entidades, protegendo assim a execução do processo atual que esta está a executar. Na saída, é executado *up* no *mutex* para libertar o acesso

mutex

(valor inicial = 1)

playersWaitTeam

(valor inicial = 0)

Bloqueia os jogadores enquanto os próprios aguardam para serem colocados numa equipa. Este semáforo sincroniza o momento em que os jogadores esperam que o capitão forme a equipa

goaliesWaitTeam

(valor inicial = 0)

Bloqueia os guarda-redes enquanto os próprios aguardam para serem colocados numa equipa. Este semáforo garante que o guarda-redes está disponível e que seja colocado corretamente numa equipa

Bloqueia todos os jogadores e guarda-redes enquanto os próprios aguardam que o árbitro inicie o jogo. Este semáforo garante que o jogo só começa quando todos os jogadores e guarda-redes estiverem prontos e as equipas formadas

playersWaitReferee

(valor inicial = 0)

Bloqueia todos os jogadores e guarda-redes enquanto os próprios aguardam que o árbitro termine o jogo. Este semáforo sincrónica o fim do jogo, garantindo assim que todos esperam que o árbitro termine o jogo

playersWaitEnd

(valor inicial = 0)

Bloqueia o árbitro enquanto as equipas estão a ser formadas. Este semáforo garante que o árbitro só começa o jogo após as duas equipas estarem formadas e completas

refereeWaitTeams

(valor inicial = 0)

Garante que todos os membros da equipa confirmar a sua alocação antes do jogo começar. Este semáforo sincrónica a conclusão da formação da equipa

playerRegistered

(valor inicial = 0)

# **Funções**

A partir daqui começámos a desenvolver o código do nosso projeto nos ficheiros indicados no guião. Estas funções são responsáveis por coordenar as ações dentro do sistema, com a utilização de semáforos e memória partilhada, de modo a garantir a sincronização e consistência.

*Nota: as funções das entidades Player e Goalie são semelhantes ou até mesmo iguais. Deste modo, no ponto 5.3 terá menos explicação pois já será basicamente feita essa explicação no ponto 5.2.*

## **semSharedMemReferee.c**

Começamos pelo ficheiro da entidade *Referee* (*semSharedMemReferee.c*). As funções inseridas neste ficheiro garantem que as ações do árbitro dentro do sistema são sincronizadas, libertando os jogadores e guarda-redes nos momentos apropriados e atualizando o estado geral do jogo.

**Função arrive():**

Nesta função, o árbitro entra na área critica ( *semDown(semgid, sh->mutex)* ), o que faz com que os outros processos sejam bloqueados caso tentem aceder aos dados compartilhados durante a operação. De seguida, o estado do árbitro é atualizado para ARRIVINGR, indicando que ele iniciou a sua participação no sistema. Esse estado atualizado é salvo na estrutura compartilhada (*saveState*). Por fim, a região crítica é libertado, permitindo assim que os outros processos acedam aos dados compartilhados ( *semUp(semgid, sh->mutex)* ). Para concluir esta função, o tempo necessário para o árbitro chegar ao campo é simulado (o valor é aleatório ( *random()* ) de modo a refletir as variações no tempo de chegada).

**Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente**

Figura 3 - Função arrive() da entidade Referee

**Função waitForTeams():**

Nesta função, o árbitro volta a entrar na região crítica, utilizando outra vez o semáforo mutex, só que desta vez o seu estado é atualizado para WAITING\_TEAMS, pois este está à espera de que ambas as equipas estejam prontas para jogar. É de novo salvo na estrutura saveState e a entidade liberta a região crítica. De seguida, é usado o semáforo refereeWaitTeams. Este vai decrementar uma unidade (*semDown*), bloqueando-se a si próprio até a equipa estar formada, e só aí, depois da confirmação, é que é desbloqueado. É de realçar que este semáforo é repetido duas vezes porque existem duas equipas e o processo do árbitro só pode ser desbloqueado ao final das equipas estarem formadas.

Uma imagem com texto, captura de ecrã, Tipo de letra, documento

Descrição gerada automaticamente

Figura 4 - Função waitForTeams() da entidade Referee

**Função startGame():**

O processo de entrada e saída da região crítica é igual à das funções anteriores, mas nesta o estado do árbitro é atualizado para STARTING\_GAME. De seguida, incrementamos (semUp) o semáforo playersWaitReferee para cada elemento (jogador ou guarda-redes) de cada equipa, ou seja, é incrementado 10 vezes, libertando assim todos estes 10 elementos que estavam bloqueados, indicando aos jogadores e guarda-redes de ambas as equipas que o jogo começou.

Uma imagem com texto, captura de ecrã, Tipo de letra, documento

Descrição gerada automaticamente

Figura 5 - Função startGame() da entidade Referee

**Função play():**

Nesta função, é feito de novo o processo de entrada e saída da região crítica, atualizando o estado do árbitro para REFEREEING, ou seja, neste momento o jogo está a decorrer e esta função simula o que o árbitro faz durante o jogo (que é arbitrar).

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente

Figura 6 - Função play() da entidade Referee

**Função endGame():**

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamentePor fim, a última função desta entidade é o encerramento do jogo. Assim, o processo de entrada e saída da região crítica é executado e o estado do árbitro é atualizado para ENDING\_GAME. Em relação a semáforos, incrementa-se (*semUp*) o semáforo *playersWaitEnd* para cada elemento de cada equipa, ou seja, incrementa-se 10 vezes. Os seus processos estavam bloqueados pelo semáforo enquanto o jogo decorria , e portanto eles estavam à espera que o árbitro desse por terminado o jogo.

Figura 7 - Função endGame() da entidade Referee

## **semSharedMemPlayer.c**

De seguida, abordamos a entidade Player (semSharedMemPlayer.c). As funções desta entidade permitem sincronizar as suas ações com os capitães, aguarda o início do jogo e responde às libertações feitas pelo árbitro e capitães.

**Função arrive(int id):**

Nesta função também é feito o processo de entrada e saída da região crítica, atualizando o estado do jogador para ARRIVING (indica que ele está a chegar). Além disto, depois de sair da região crítica, é simulado ( *usleep(…)* )o tempo de chegada do jogador ao campo, com um valor aleatório ( *random()* ).

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente

Figura 8 - Função arrive(int id) da entidade Player

**Função playerConstituteTeam(int id):**

Esta função tem como objetivo organizar a formação das equipas, ou seja, associa uma equipa a cada player. (equipa 1 - identificador 1; equipa 2 - identificador 2).

Primeiro, o jogador entra na região crítica (*semDown*), bloqueando assim o acesso simultâneo de outros projetos.

De seguida, vamos incrementar dois contadores: *playersArrived*, que é o número total de jogadores que chegaram ao campo (este valor vai ser usado para verificar se o jogador chegou atrasado); *playersFree*, que é o número de jogadores livres para formar uma equipa.



Figura 9 - Contadores

Depois, efetuamos vários verificações.

A primeira é o jogador chegar atrasado: caso o jogador seja o nono ou posterior ( *sh->fSt.playersArrived > 8* ), ele é considerado LATE e não participa no erro. Assim, o estado do jogador é atualizado e sai da região crítica (*semUp*).

Uma imagem com texto, captura de ecrã, Tipo de letra

Descrição gerada automaticamente

Figura 10 - Verificação de jogador atrasado

A segunda é a disponibilidade do jogador para formar uma equipa, onde é verificado se há jogadores e guarda-redes suficientes para formar uma equipa (equipa constituída por 4 *players* e 1 *goalie*). O número de jogadores disponíveis (playersFree) deve ser maior ou igual ao número de jogadores necessários (4), e o número de guarda-redes disponíveis (goaliesFree) deve ser maior ou igual ao número de guarda-redes necessários (1).

Figura 11 - Verificação da disponibilidade para formar uma equipa

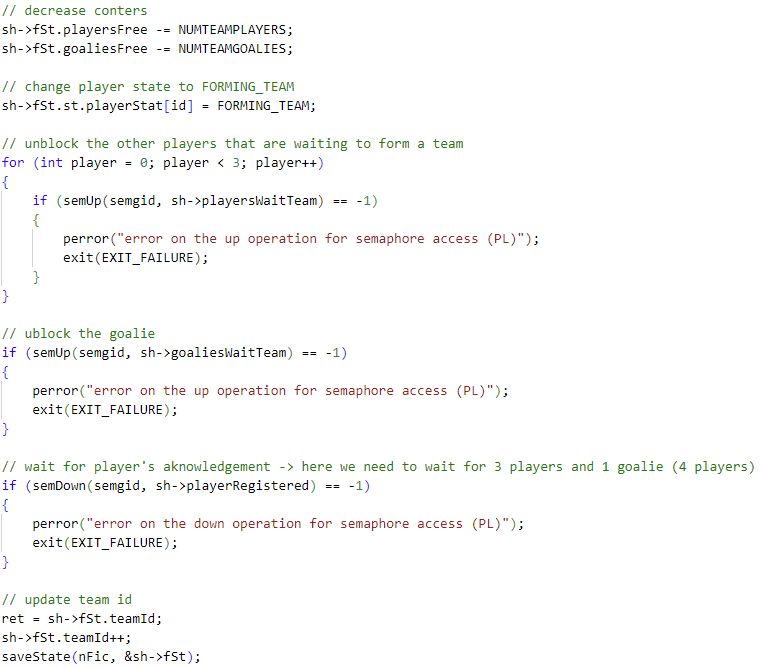
Caso esta última verificação aconteça, o jogador atual torna-se capitão (ou seja, o quinto elemento a chegar é o capitão, neste caso estamos a considerar que é um *player*) e forma uma equipa. Também temos de atualizar os contadores, reduz-se o número de jogadores e guarda-redes livres, e atualiza-se o estado para FORMING\_TEAM. De seguida, liberta-se (*semUp*) os 3 jogadores do semáforo *playersWaitTeam* e 1 guarda-redes do semáforo *goaliesWaitTeam*. Por fim, é feito o registo dos membros da equipa (semDown 4 vezes) no semáforo *playerRegistered* e incrementa-se o teamID e salva-se o estado.

Figura 12 - Formação e registo da equipa pelo capitão

No caso de a verificação não acontecer, o jogador não é capitão, logo entra no estado WAITING\_TEAM e salva-se o estado.

Uma imagem com texto, Tipo de letra, branco, file

Descrição gerada automaticamente

Figura 13 - Jogador a aguardar pela equipa

Se o jogador não for capitão, ele tem de esperar para ser libertado do semáforo *playersWaitTeam*. Após isso acontecer, ele regista a sua equipa (*semUp*) no semáforo *playerRegistered*. Isto só acontece caso um outro jogador for capitão e passar por todos os passos acima mencionados.

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente

Figura 14 - Alocação numa equipa por um capitão

Por fim, o capitão informa o árbitro que a sua equipa está formada, realizando *semUp* no semáforo *refereeWaitTeams*.

Uma imagem com texto, captura de ecrã, Tipo de letra, file

Descrição gerada automaticamente

Figura 15 - Comunicação com o árbitro

**Função waitReferee (int id, int team):**

O jogador realiza a entrada na região crítica (semDown) e o seu estado é atualizado para WAITING\_START\_1 ou WAITING\_START\_2 (dependendo do identificador da equipa em que está). Depois sai da região crítica (semUp).

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamenteO jogador volta a fazer semDown, no entanto é num semáforo diferente, *playersWaitReferee*, para aguardar que o árbitro o liberte, indicando o início do jogo.

Figura 16 - Função waitReferee (int id, int team) da entidade Player

**Função playUntilEnd (int id, int team):**

Na última função, os jogadores já se encontram durante o jogo. Assim, o jogador entra na região crítica para atualizar o seu estado para PLAYING\_1 ou PLAYING\_2. Em relação aos semáforos, o jogador fica à espera de que o árbitro termine o jogo, logo o semáforo *playersWaitEnd* vai ser decrementado (*semDown*) e o processo fica bloqueado até o árbitro o desbloquear na função *endGame().*

Uma imagem com texto, captura de ecrã, Tipo de letra, documento

Descrição gerada automaticamente

Figura 17 - Função playUntilEnd (int id, int team) da entidade Player

## **semSharedMemGoalie.c**

Por fim, desenvolvemos a entidade Goalie (semSharedMemGoalie.c). As funções desta entidade garantem que está inserida corretamente numa equipa, sincroniza a sua participação com os jogadores e o árbitro, de modo a manter o fluxo do jogo de forma organizada.

**Função arrive (int id):**

Nesta função também é feito o processo de entrada e saída da região crítica, atualizando o estado do guarda-redes para ARRIVING (indica que ele está a chegar). Além disto, depois de sair da região crítica, é simulado ( *usleep(…)* )o tempo de chegada do guarda-redes ao campo, com um valor aleatório ( *random()* ).

Uma imagem com texto, captura de ecrã, Tipo de letra

Descrição gerada automaticamente

Figura 18 - Função arrive (int id) da entidade Goalie

**Função goalieConstituteTeam (int id):**

Esta função é bastante semelhante à função playerConstituteTeam() definida no ficheiro semSharedMemPlayer.c, tendo apenas poucas exceções (por se tratar do guarda-redes).

Então, as exceções são as seguintes:

* caso o guarda-redes seja o último a chegar, ele será o capitão, e não um jogador
* os semáforos *playersWaitTeam* e *playersRegistered* serão repetidas 4 vezes, um para cada jogador, ficando o primeiro semáforo desbloqueado (uma vez que depois dos jogadores formarem uma equipa, deixam de esperar por ela) e o segundo bloqueado

**Função waitReferee (int id, int team):**

Esta função faz a mesma coisa que a função *waitReferee()* da entidade *player*. O guarda-redes entra e saiu da região crítica atualizando o seu estado para WAITING\_START\_1 ou WAITING\_START\_2.

De seguida volta a fazer *semDown* no semáforo *playersWaitReferee*, para aguardar que o árbitro o liberte, indicando o início do jogo.

Uma imagem com texto, captura de ecrã, documento, Tipo de letra

Descrição gerada automaticamente

Figura 19 - Função waitReferee (int id, int team) da entidade Goalie

**Função playUntilEnd (int id, int team):**

A última função é igual à função *playUntilEnd()* da entidade *player*. O guarda-redes já se encontra durante o jogo, por isso entra e saiu da zona crítica atualizando o seu estado para PLAYING\_1 ou PLAYING\_2. O semáforo *playersWaitEnd* é decrementado e o processo fica bloqueado até o árbitro o desbloquear na função *endGame().*

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente

Figura 20 - Função playUntilEnd (int id, int team) da entidade Goalie

# **Testes**

A secção de testes é crucial para a garantia do funcionamento correto do nossa solução, uma vez que assegura que nenhuma das entidades fica presa em alguma das fases que pode atingir durante a execução do programa. Nesta secção apresentamos alguns testes que corremos para ter essa garantia verificando a falta de *Deadlocks*, que evitamos durante o desenvolvimento do projeto. Durante a implementação tivemos o cuidado de atribuir corretamente os estados a cada entidade indicados no enunciado do trabalho de forma a podermos ter um *output* coeso e coerente com o que era pretendido. Na *Figura 21* são apresentados os estados para cada entidade.

A screenshot of a computer program

Description automatically generated

Figure 21 – Estados das entidades (probConst.h)

Para a realização dos testes utilizamos os scripts já fornecidos de partida na diretoria *run* e os ficheiros binários disponibilizados pelos docentes. Deste modo pudemos fazer testes unitários a cada uma das entidades ao compilar apenas os ficheiros que nos interessavam testar. Abaixo apresentamos os testes unitários realizados às entidades desenvolvidas: *Player, Referee* e *Goalie*.

IMAGEM TESTE UNITÁRIO COM MAKE PL – corre com o script ./run.sh 5000

De forma a não sobrecarregar o relatório, embora tenhamos feitos muitos mais testes, o teste apresentado na *Figura 22* apresenta a última iteração do *output* ao executar o *script* **./run.sh 5000**, o que irá executar o programa 5000 vezes para garantir que não existem nenhum tipo de *Deadlocks* presentes. Podemos então concluir que a entidade *Player* tem o comportamento esperado, caso chegue depois de já estarem presentes 8 outros jogadores muda o seu estado automaticamente para LATE, quando pode formar uma equipa, forma-a e fica corretamente em espera nos momentos indicados, finalmente mudando o seu estado para PLAYING quando o árbitro inicia a partida.

IMAGEM TESTE UNITÁRIO COM MAKE REF (ns o comando específico)

O mesmo teste unitário foi feito com a entidade *Referee*. Podemos então concluir que o resultado é o esperado uma vez que o árbitro fica corretamente à espera de que as equipas sejam formadas antes de mudar o seu estado para STARTING\_GAME e, mais tarde para REFEREEING. Depois de algum tempo muda o seu estado para o estado final atingível: ENDING\_GAME.

IMAGEM TESTE UNITÁRIO COM MAKE GOALIE (ns o comando específico)

Mais uma vez fizemos o mesmo teste unitário com a nossa implementação da entidade *Goalie*. Podemos concluir que o guarda-redes tem o comportamento esperado, mudando o seu estado para LATE caso seja o 3º a chegar, espera corretamente se não tiver elementos suficientes para ser ele a formar a equipa ou forma a equipa caso contrário e que muda o seu estado para PLAYING quando ambas as equipas estão formadas.

# **Conclusão**

Em suma, acreditamos que cumprimos com todos os requisitos propostos e adquirimos bases sólidas sobre a utilização de semáforos e segmentos de memória partilhada. Este projeto foi uma excelente oportunidade para explorar conceitos complexos e associá-los a conceitos mencionados no dia-a-dia através de uma analogia muito bem conseguida, consolidando assim os nossos conhecimentos. Durante o desenvolvimento, enfrentamos e resolvemos diversos problemas que resultaram em situações de deadlock, o que aprofundou o nosso entendimento sobre programação concorrente e os cuidados necessários em sistemas que usam este tipo de estrutura. Além disso, a analogia do jogo de futebol demonstrou ser uma abordagem eficaz para compreender e aplicar conceitos abstratos, o que nos permitiu conectar a teoria ao mundo real de forma interessante, desenvolvendo assim o nosso esquema mental sobre esta matéria. Por fim, acreditamos que esta experiência nos preparou melhor para projetos futuros que envolvam programação concorrente e comunicação entre processos.

# **Webgrafia**