SO projeto 2 - Simulação de Jogo de Futebol

Igor Baltarejo 118832, Tiago Oliveira 118772

Dezembro 2024

Índice

1 Introdução

No âmbito da unidade curricular de Sistemas Operativos, este projeto consiste na implementação de uma simulação de um jogo de futebol. Nesta simulação existem 3 entidades - `player`, `goalie` e `referee` - que são representadas por processos independentes, sendo a sua sincronização e comunicação realizada através de semáforos e 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 restantes jogadores não entrarão no jogo. Por jogo, existe um árbitro (`referee`), que dita o início da partida, e o fim da mesma. Os semáforos são usados para garantir que o acesso à memória partilhada seja realizado sem conflitos, no caso do `MUTEX`, ou para sincronizar os processos em vários pontos da simulação.

Devido à semelhança entre as entidades `player` e `goalie` muitas vezes usamos o termo 'jogador' para nos referir-mos a qualquer um dos dois. Esta semelhança é tanta que abordámos a implementação dos dois em simultâneo.

2 Definição do problema

Antes de começarmos a escrever código foi importante modelarmos as várias partes do problema usando esquemas e diagramas. Esta metodologia permitiu-nos resolver o problema mesmo antes de pensar na implementação, o que agilizou a mesma e no geral produziu um melhor resultado. Começámos pelas estruturas de dados definidas no problema, depois os diferentes estados e transições envolvidas e por fim percebemos como é que podíamos usar os semáforos disponibilizados para reproduzir o estudado anteriormente.

2.1 A estrutura de dados

A principal estrutura de dados[1], que guarda a informação partilhada pelos processos, contém os vários ID's dos semáforos definidos e um ponteiro para a estrutura `FULL_STAT`. Esta guarda os valores das variáveis do problema como número de jogadores livres, ou o ID da próxima equipa a ser formada, etc. Para além disso guarda também um ponteiro para a estrutura `STAT` que contém os estados atuais de todas as entidades da simulação. A seguir segue-se um diagrama elucidativo:

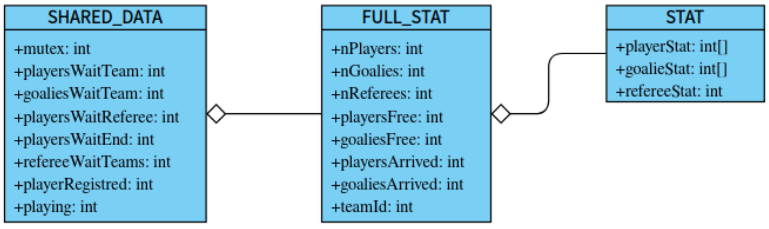

Figura 1: Estrutura de dados do problema

Quando abordarmos a implementação da solução, as estruturas de dados estãrão definidas como `sh`, `sh->fSt` e `sh->fSt.st` para `SHARED_DATA`, `FULL_STAT` e `STAT` respetivamente.

[1] Estruta definida nos *headers* probDataStruct.h e sharedDataSync.h

2.2 Estados

Os estados das entidades vão sendo alterados ao longo do programa. Esta mudança é realizada através do acesso à memória partilhada. Cada entidade (processo) apenas pode mudar o seu próprio estado.

2.2.1 `Referee`

O árbitro começa no estado `ARRIVINGR`, e após a sua chegada passa para o estado `WAITING_TEAMS`. Quando ambos os capitães confirmarem que as suas equipas estão prontas, o árbitro dá início ao jogo (estado `STARTING`) e depois de todos jogadores pertencentes às equipas serem notificados sobre o começo do jogo, passa para o estado `REFEREEING`. Passado algum tempo o árbitro termina o jogo (estado `ENDING_GAME`).

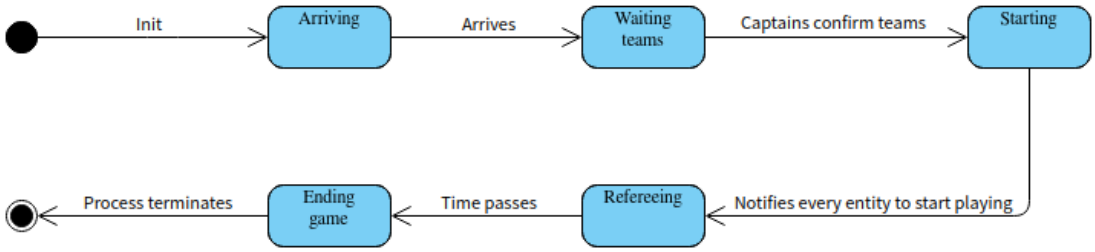

Figura 2: Diagrama de estados (`referee`)

2.2.2 `Player/Goalie`

Todos os jogadores começam no estado `ARRIVING`. Após a sua chegada, caso já tenham chegado jogadores suficientes da sua categoria para preencher duas equipas passam para o estado `LATE` e o seu processo é terminado (1). Caso ainda não haja jogadores de campo e guarda-redes suficientes para preencher uma equipa passam para o estado `WAITING_TEAM` (2), caso contrário este jogador será um capitão de equipa e passará para o estado `FORMING_TEAM`, avisando o número necessário de jogadores para formar uma equipa. Após isso, todos os jogadores desta equipa passam para o estado `WAITING_START_#TEAMID` e o capitão notifica o árbitro da formação de uma equipa (3). Quando o árbitro der início ao jogo, os jogadores de cada equipa passam para o estado `PLAYING_#TEAMID`, avisando o árbitro da mudança de estado. Quando o jogo termina os processos são terminados.

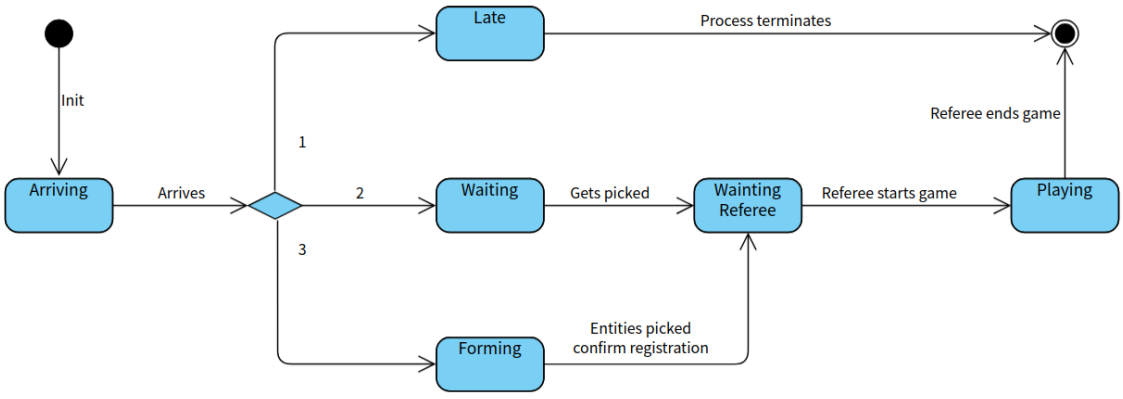

Figura 3: Diagrama de estados (`goalie/player`)

Por uma questão de clareza, no diagrama anterior os estados `WAITING_START_1` e `WAITING_START_2`, estão juntos no estado de nome 'Waiting Referee'. O mesmo se aplica aos estados `PLAYING_1` e

PLAYING_2, juntos no estado 'Playing'.

2.3 Semáforos

Como vimos existem 8 semáforos ao nosso dispor. Nesta secção exploramos como pensámos utilizar cada um deles.

O primeiro semáforo, mutex, como o próprio nome indica foi usado para definir uma zona de exclusão mútua para que os vários processos possam aceder à memória partilhada sem que hajam condições de corrida e comportamentos inesperados.

A seguir, playerWaitTeam e goaliesWaitTeam, o objetivo destes semáforos é o mesmo: garantir que execução dos processos do tipo player e goalie, repetivamente, é pausada enquanto estes esperam pelo número de jogadores necessários para se formar uma equipa (*down*). A execução seria retomada quando um jogador capitão, o ultimo necessário para a formação, quisesse formar a sua equipa (*up*). Ao formar uma equipa o capitão deve executar a operação *up* o mesmo número de vezes quanto o número de jogadores necessários à formação da equipa.

Depois temos o semáforo playerRegistered, que nos permite sincronizar as ações do jogador capitão, com as dos jogadores da sua equipa. Fazendo um *down* no processo capitão tantas vezes como os jogadores da sua equipa exceto ele, e esperando pela "confirmação" dos mesmos, o que equivale a cada um fazer um *up* no mesmo semáforo em cada membro da equipa uma vez, exceto o capitão.

O semáforo playersWaitReferee é usado para apenas permitir a execução dos jogadores à espera para jogar, quando o árbitro quiser começar o jogo propriamente dito. Para isso fazemos um *down* nos processos jogadores, e um *up* para todos os jogadores de ambas as equipas no processo do árbitro.

Por sua vez o árbitro sabe que só pode iniciar o jogo quando tiver recebido o registo de 2 equipas. Para simular esta interação fazemos 2 *downs* no semáforo refereeWaitTeams no processo árbitro, e um *up* em cada processo jogador capitão.

Para garantir que o árbitro só muda o seu estado para REFEREEING depois de todos os jogadores começarem a "jogar", fazemos no processo árbitro um *down*, no semáforo playing, por cada jogador em campo. Esta operação vai ser contraposta por um *up* em cada processo jogador. Garantindo assim a sincronia entre processos.

Por último, o semáforo playersWaitEnd para a execução dos processos jogadores (down) até que o processo árbitro faça um *up* por cada jogador em campo.

3 Implementação

Depois de definido o problema a resolução deste foi bastante simplificada, apenas tivemos de seguir os nossos próprios modelos e descrições do que achava-mos como sendo a correta solução.

A implementação consistiu em completar o código fornecido para cada entidade. Cada entidade tinha funções definidas no seu ficheiro `.c` correspondente e cabeu-nos a nós completar estas funções. Em todas as funções que necessitam de aceder à região crítica, já se encontram os incrementos e decrementos do semáforo `mutex` declarados, evidenciando quando o código deve ser preenchido dentro entre ou após estas operações com o semáforo.

No nosso código usámos as constantes definidas no ficheiro `probConst.h`, de forma a aumentar a legibilidade do código e também a fornecer alguma flexibilidade, uma vez que o código funcionará para qualquer valor inteiro positivo que seja armazenado nas constantes. Estas constantes possuem informação relativa ao número de jogadores de cada tipo por equipa e ao número de entidades simuladas por jogo.

Para manter as secções de código mais concisas e fáceis de ler utilizamos:

```
1 playersWaitReferee.up();
```

Como abstração do trecho de código:

```
1 if (semUp(semgid, sh->playersWaitReferee) == -1) {
2     perror("error on up: refereeWaitTeams (RF)");
3     exit(EXIT_FAILURE);
4 }
```

O mesmo se aplica à operação `down`, e a outros semáforos.

3.1 Referee

No ficheiro `semSharedMemReferee.c` estão definidas 5 funções cujo código apresentava-se por completar. Uma das nossa tarefas para este projeto era completar este código para para a entidade `referee` pudesse funcionar como devido.

3.1.1 arrive()

Para esta função apenas tivemos de atualizar o estado do árbitro para `ARRIVINGR`.

```
1 mutex.down();
2 sh->fSt.st.refereeStat = ARRIVINGR;
3 saveState(nFic, &sh->fSt);
4 mutex.up();
```

Trecho 1: Atualização de estado do `referee`

Esta função é apenas utilizada no início do processo que representa a entidade `referee`, daí a sua simplicidade.

3.1.2 waitForTeams()

Nesta função, o estado do `referee` é novamente atualizado, desta vez para o estado `WAITING_TEAMS`.

```
1 mutex.down();
2 sh->fSt.st.refereeStat = WAITING_TEAMS;
3 saveState(nFic, &sh->fSt);
4 mutex.up();
```

Trecho 2: Atualização de estado do `referee`

Após esta atualização de estado, o `referee` espera que as duas equipas se formem para poder depois dar o começo ao jogo. Para isso, o semáforo `refereeWaitTeams` é decrementado duas vezes. Assim cada vez que uma equipa for formada, este semáforo será incrementado, permitindo que o `referee` dê início ao jogo na função seguinte.

```
for (int i = 0; i < 2 ;i++) {
    refereeWaitTeams.down();
}
```

Trecho 3: Operações com semáforos na função `waitForTeams`

3.1.3 `startGame()`

Nesta função, o estado do `referee` é atualizado para o estado `STARTING_GAME`.

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

Trecho 4: Atualização do estado do `referee`

Depois desta atualização de estado, todos os jogadores são avisados para começar a jogar. Isto é feito através do incremento do semáforo `playersWaitReferee`, uma vez que os jogadores já não precisam de esperar pelo `referee`, e do posterior decremento do semáforo `playing`, para os jogadores começarem a jogar. Cada operação com os semáforos é realizada um número de vezes igual ao número de jogadores presentes nas duas equipas, representado pela variável `totalPlayers`.

```
int totalPlayers = 2 * (NUMTEAMGOALIES + NUMTEAMPLAYERS);

/* Notifies all waiting players to start playing */
for (int i = 0; i < totalPlayers ; i++) {
    playersWaitReferee.up();
}

/* Waits for all players to start playing */
for (int j = 0; j < totalPlayers ; j++) {
    playing.down();
}
```

Trecho 5: Operações com semáforos na função `startGame()`

3.1.4 `play()`

Esta função é muito simples, tal como a função `arrive()`, consiste apenas na mudança de estado do `referee` e nada mais. O estado do `referee` passa então a ser `REFEEREING`.

```
mutex.down();
sh->fSt.st.refereeStat = REFEEREING;
saveState(nFic, &sh->fSt);
mutex.up();
```

Trecho 6: Atualização do estado do `referee`

3.1.5 `endGame()`

Nesta função o estado do `referee` é atualizado pela última vez para `ENDING_GAME`.

```
mutex.down();
sh->fSt.st.refereeStat = ENDING_GAME;
saveState(nFic, &sh->fSt);
mutex.up();
```

Trecho 7: Atualização do estado do `referee`

Após a atualização de estado, todos os jogadores são avisados para parar de jogar através do incremento do semáforo `playersWaitEnd`.

```
int totalPlayers = 2 * (NUMTEAMGOALIES + NUMTEAMPLAYERS);

/* Notifies all playing players to stop playing */
for (int i = 0; i < totalPlayers ; i++) {
    playersWaitEnd.up();
}
```

Trecho 8: Operações com semáforos na função `endGame()`

Após isto o processo que representa a entidade `referee` chega ao fim e é então terminado.

3.2 Player/Goalie

As entidades `player` e `goalie` possuem muitas semelhanças e as funções que nos foram encarregadas de completar também. Então devido à similaridade do código entre as funções nos ficheiros `semSharedMemPlayer.c` e `semSharedMemGoalie.c`, no relatório iremos abordar a implementação das duas entidades em simultâneo, de modo a evitar que o relatorio fique desnecessáriamente longo.

3.2.1 arrive(int id)

Estas funções são muito semelhantes à função `arrive()` do `referee`, que apenas consiste mudar o estado da entidade para `ARRIVING`.

```
mutex.down();

sh->fSt.st.playerStat[id] = ARRIVING;
saveState(nFic, &sh->fSt);

mutex.up();
```

Trecho 9: Atualização do estado do `player/goalie`

3.2.2 playerConstituteTeam(int id) - goalieConstituteTeam(int id)

Nesta função simula a criação de equipas após a chegada de cada jogador.

Caso já tenham chegados jogadores suficientes para criar duas equipas, o estado do jogador é atualizado para `LATE` e o processo acaba, então o valor de retorno é zero.

Caso ainda não estejam jogadores presentes suficientes para formar uma equipa, o estado do jogador passa para `WAITING_TEAMS`, o número de jogadores disponíveis é incrementado e o valor de retorno da função é o ID da equipa à qual se irá juntar.

No último caso, no qual existem jogadores suficientes para se formar um equipa, o número de jogadores disponíveis são decrementados devidamente, o estado do jogador passa a ser `FORMING_TEAM` e o valor de retorno da função será o ID da equipa que foi formada, que é incrementado para servir de ID para a próxima equipa que se formar.

Tudo isto é feito na região crítica da memória partilhada logo é necessário decrementar o semáforo `mutex` e incrementá-lo de volta após a realização das operações.

```
int ret = 0;
```

```
mutex.down();

sh->fSt.playersArrived++;

if (sh->fSt.playersArrived > 2 * NUMTEAMPLAYERS) {
    sh->fSt.st.playerStat[id] = LATE;
}

else if ((sh->fSt.goaliesFree < NUMTEAMGOALIES)
    | (sh->fSt.playersFree < NUMTEAMPLAYERS - 1)) {
    sh->fSt.st.playerStat[id] = WAITING_TEAM;
    sh->fSt.playersFree++;
}

else {
    sh->fSt.st.playerStat[id] = FORMING_TEAM;
    sh->fSt.playersFree -= NUMTEAMPLAYERS - 1;
    sh->fSt.goaliesFree -= NUMTEAMGOALIES;
}

saveState(nFic, &sh->fSt);

mutex.up();
```

No seguinte trecho de código utilizamos um `switch-case` para realizar as operações necessárias com semáforos dependendo do estado do jogador.

```
switch (sh->fSt.st.playerStat[id])
```

Caso o jogador esteja à espera de uma equipa, o semáforo `playersWaitTeam` é decrementado e quando eventualmente o semáforo voltar a ser incrementado e o ID da equipa na qual jogador é colocado passa a ser o valor de retorno da função. Depois o jogador é registado na equipa através da incrementação do semáforo `playerRegistered`.

```
case WAITING_TEAM:
{
    /* Wait for captain */
    playersWaitTeam.down();

    ret = sh->fSt.teamId;

    /* Confirm registration */
    playerRegistered.up();

    break;
}
```

Caso o jogador esteja a formar uma equipa, ele chamará todos os jogadores que precisar para formar uma equipa. Isto é feito através da incrementação dos semáforos `playersWaitTeam` e `goaliesWaitTeam`. Depois o jogador irá avisar aos jogadores que escolheu para se registarem na equipa através da decrementação do semáforo `playerRegistered`. Quando todos se registarem, a valor de retorno será o ID da equipa que se formou e este ID será posteriormente incrementado para estar pronto para a equipa seguinte o usar. Por fim, o jogador irá avisar o `referee` que a equipa já está pronta para jogar, através do incremento do semáforo `refereeWaitTeam`.

```
case FORMING_TEAM:
{
    /* Only 1 entity should create a team at a given time */
    mutex.down();

    /* Pick players */
    for (int i = 0; i < NUMTEAMPLAYERS - 1; i++)
    {
        playersWaitTeam.up();
    }

    /* Pick goalies */
    for (int j = 0; j < NUMTEAMGOALIES; j++)
    {
        goaliesWaitTeam.up();
    }

    /* Wait for registration acknowledgement */
    for (int h = 0; h < NUMTEAMPLAYERS + NUMTEAMGOALIES - 1; h++)
    {
        playerRegistered.down();
    }

    ret = sh->fSt.teamId++;

    mutex.up();

    /* Notify referee of team creation */
    refereeWaitTeam.up();
    break;

}
```

```
return ret;
```

3.2.3 waitReferee(int id, int team)

Nesta função o estado do jogador é atualizado para `WAITING_START_1` ou `WAITING_START_2`, dependendo da equipa à qual pertence. Após isso, o jogador fica à espera que o aŕbitro comece o jogo, através do decremento do semáforo `playersWaitReferee`.

```
mutex.up();

sh->fSt.st.playerStat[id] = (team == 1) ? WAITING_START_1 : WAITING_START_2;
saveState(nFic, &sh->fSt);

mutex.up();

playersWaitReferee.down();
```

3.2.4 playUntilEnd(int id, int team)

Nesta função, o estado do jogador é atualizado para `PLAYING_1` ou `PLAYING_2`, dependendo do equipa à qual pertence. Após isso o jogador começa a jogar e espera até o jogo acabar. Isto é feito através do incremento semáforo `playing` e do decremento do semáforo `playersWaitEnd`.

```
1  mutex.down();
2
3  sh->fSt.st.playerStat[id] = (team == 1) ? PLAYING_1 : PLAYING_2;
4  saveState(nFic, &sh->fSt);
5
6  mutex.up();
7
8  /* Notify referee */
9  playing.up();
10
11 /* Block until referee ends game */
12 playersWaitEnd.down();
```

4 Conclusão

Damos assim por concluído este projeto. Ao longo do mesmo fomos capazes de aplicar os conceitos bases da multi-programação, nomeadamente, perceber como podemos estabelecer a comunicação entre processos e como aceder à memória partilhada pelos mesmos, sem que hajam condições de corrida.

O problema à primeira vista parece mais complexo do que realmente é. A nossa análise e modelação permitiu-nos chegar rapidamente a uma solução, que consideramos robusta e com alguma flexibilidade extra, ainda que esta fosse fora do escopo do projeto.