deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Projeto 2: Simulação de jogo de futebol

Sistemas Operativos

Duarte Gabriel Castro Branco - 119253

2024/2025

Conteúdo

1. Introdução

O objetivo deste projeto é simular um jogo de futebol. O jogo é composto por 2 equipas, cada uma com os respetivos 4 jogadores e 1 guarda-redes. Existe também um único árbitro. Existem jogadores a mais e cabe-nos a nós dar uso aos semáforos para tratarmos da concorrência e sincronização entre os jogadores, guarda-redes e árbitro.

Para isto, foi-nos pedido que completássemos os módulos:

- **semSharedMemGoalie**
- **semSharedMemPlayer**
- **semSharedMemReferee**

Neste relatório, vou explicar o que foi feito em cada um destes módulos.

2. semSharedMemGoalie

Neste capítulo, vamos detalhar a implementação do módulo **semSharedMemGoalie**, que é responsável pelas operações realizadas pelo guarda-redes no jogo de futebol simulado. As operações/funções que nos foram pedidas para completar são:

- **arriving**: O guarda-redes leva algum tempo a chegar e atualiza o seu estado.
- **goalieConstituteTeam**: O guarda-redes constitui uma equipa, caso seja possível
- **waitReferee**: O guarda-redes espera pelo árbitro para iniciar o jogo.
- **playUntilEnd**: O guarda-redes joga até o final do jogo.

2.1 Função arrive

A função **arrive** é responsável por simular a chegada do guarda-redes ao campo. Ela atualiza o estado do guarda-redes para **ARRIVING** e salva o estado interno. A função também faz o guarda-redes esperar durante um tempo aleatório antes de prosseguir. É de notar que esta função é necessária para a implementação de todos os outros módulos.

```
static void arrive(int id)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

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

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    usleep((200.0*random())/(RAND_MAX+1.0)+60.0);
}
```

2.2 Função goalieConstituteTeam

A função **goalieConstituteTeam** verifica se há jogadores suficientes para formar uma equipa. Se houver, o guarda-redes forma a equipa e espera que os jogadores se registem. Caso contrário, o guarda-redes espera até que uma equipa possa ser formada. Existe também o caso em que o guarda-redes chega tarde e não consegue formar a equipa. A função retorna o id da equipa formada, ou 0 se o guarda-redes chegou tarde.

Comecei por incrementar o número de guarda-redes que chegaram e que estão livres, isto é necessário para verificar se há jogadores suficientes para formar equipa. Também inicializei uma nova variável booleana - *waiting* - que vai ser necessária mais à frente.

```
static int goalieConstituteTeam (int id)
{
    int ret = 0;
    // New variable
    bool waiting = false;

    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.goaliesArrived++;
    sh->fSt.goaliesFree++;
```

Depois trato dos guarda-redes que chegaram atrasados, ou seja, se o número de guarda-redes que chegaram é maior ou igual ao número total de guarda-redes, o estado do guarda-redes é atualizado para **LATE** e a função retorna 0, indicando que o guarda-redes chegou tarde.

```
if (sh->fSt.goaliesArrived >= NUMGOALIES) {
    sh->fSt.st.goalieStat[id] = LATE;
    saveState(nFic, &sh->fSt);
    ret = 0;
```

Se há jogadores e guarda-redes suficientes para formar uma equipa, o estado do guarda-redes é atualizado para **FORMING_TEAM**. O número de jogadores e guarda-redes livres é decrementado e os jogadores são acordados para se registarem na equipa. Guarda-se o estado interno e incrementa-se o id da equipa. Por fim, o árbitro é avisado de que a equipa está formada.

```
} else if (sh->fSt.playersFree >= NUMTEAMPLAYERS &&
           sh->fSt.goaliesFree >= NUMTEAMGOALIES) {
    sh->fSt.st.goalieStat[id] = FORMING_TEAM;
    saveState(nFic, &sh->fSt);

    sh->fSt.goaliesFree -= NUMTEAMGOALIES;
    sh->fSt.playersFree -= NUMTEAMPLAYERS;

    for (int i = 0; i < NUMTEAMPLAYERS; i++) {
        if (semUp (semgid, sh->playersWaitTeam) == -1) {
            perror ("error on the down operation for semaphore access (GL)");
            exit (EXIT_FAILURE);
        }

        if (semDown (semgid, sh->playerRegistered) == -1)  {
            perror ("error on the up operation for semaphore access (GL)");
            exit (EXIT_FAILURE);
        }
    }

    ret = sh->fSt.teamId;
    sh->fSt.teamId++;

    if (semUp (semgid, sh->refereeWaitTeams) == -1) {
        perror ("error on the down operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }
```

Caso ainda não haja jogadores suficientes para formar a equipa, o estado do guarda-redes é atualizado para **WAITING_TEAM** e a variável *waiting* é definida como *true*.

```
} else {
    sh->fSt.st.goalieStat[id] = WAITING_TEAM;
    saveState(nFic, &sh->fSt);
    waiting = true;
}
```

Depois destas verificações, a função sai da região crítica (código do professor) e, logo de seguida, verifica-se se a variável *waiting* é *true*. Se for, o guarda-redes espera até que uma equipa possa ser formada. Quando a equipa é formada, o id da equipa é retornado.

```
if (waiting) {
    if (semDown (semgid, sh->goaliesWaitTeam) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    ret = sh->fSt.teamId;

    if (semUp (semgid, sh->playerRegistered) == -1) {
        perror ("error on the down operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }
}
return ret;
```

2.3 Função waitReferee

A função **waitReferee** faz o guarda-redes esperar pelo árbitro para iniciar o jogo. O estado do guarda-redes é atualizado para **WAITING_START** e o estado interno é salvo. Os últimos 2 semáforos são usados para sincronizar o guarda-redes com o árbitro.

```
static void waitReferee (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

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

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->playersWaitReferee) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    if (semUp (semgid, sh->playing) == -1) {
        perror ("error on the down operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }
}
```

2.4 Função playUntilEnd

A função **playUntilEnd** faz o guarda-redes jogar até o final do jogo. O estado do guarda-redes é atualizado para **PLAYING** e o estado interno é salvo. O último semáforo é para esperar pelo fim do jogo.

```
static void playUntilEnd (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.goalieStat[id] = (team == 1) ? PLAYING_1 : PLAYING_2;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->playersWaitEnd) == -1)  {
        perror ("error on the up operation for semaphore access (GL)");
        exit (EXIT_FAILURE);
    }
}
```

3. semSharedMemPlayer

Neste capítulo, vamos detalhar a implementação do módulo **semSharedMemPlayer**, que é responsável pelas operações realizadas pelos jogadores no jogo de futebol simulado. As operações principais incluem:

- **arrive**: O jogador leva algum tempo a chegar e atualiza seu estado.
- **playerConstituteTeam**: O jogador constitui uma equipa, caso seja possível
- **waitReferee**: O jogador espera pelo árbitro para iniciar o jogo.
- **playUntilEnd**: O jogador joga até o final do jogo.

3.1 Função arrive

A função **arrive** é responsável por simular a chegada do jogador ao campo. Ela atualiza o estado do jogador para **ARRIVING** e salva o estado interno, tal como no módulo **semSharedMemGoalie**.

```
static void arrive(int id)
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

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

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    usleep((100.0*random())/(RAND_MAX+1.0)+10.0);
}
```

3.2 Função playerConstituteTeam

A função **playerConstituteTeam** verifica se há jogadores suficientes para formar uma equipa. Segue a mesma forma que a função **goalieConstituteTeam** do módulo anterior, embora com algumas diferenças.

Comecei por incrementar o número de jogadores que chegaram e que estão livres e inicializei uma nova variável, tal como fiz anteriormente.

```
static int playerConstituteTeam (int id)
{
    int ret = 0;
    // New variable
    bool waiting = false;

    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.playersArrived++;
    sh->fSt.playersFree++;
```

Se o número de jogadores que chegaram é maior ou igual ao número total de jogadores possíveis (2 equipas * 4 jogadores = 8), o estado do jogador é atualizado para **LATE** e a função retorna 0, indicando que o jogador chegou tarde.

```
if (sh->fSt.playersArrived > (2 * NUMTEAMPLAYERS)) {
    sh->fSt.st.playerStat[id] = LATE;
    saveState(nFic, &sh->fSt);
    ret = 0;
```

Se há jogadores e guarda-redes suficientes para formar uma equipa, o estado do jogador é atualizado para **FORMING_TEAM**. O número de jogadores e guarda-redes livres é decrementado e os jogadores são acordados para se registarem na equipa. Os novos semáforos são utilizados para acordar os guarda-redes que estão à espera de formar uma equipa e para esperar que os jogadores confirmem o seu registo na equipa, sendo este processo repetido para todos os jogadores necessários. Guarda-se o estado interno e incrementa-se o id da equipa. Por fim, o árbitro é avisado de que a equipa está formada.

```
} else if (sh->fSt.playersFree >= NUMTEAMPLAYERS &&
           sh->fSt.goaliesFree >= NUMTEAMGOALIES) {
    sh->fSt.st.playerStat[id] = FORMING_TEAM;
    saveState(nFic, &sh->fSt);

    sh->fSt.playersFree -= NUMTEAMPLAYERS;
    sh->fSt.goaliesFree -= NUMTEAMGOALIES;

    if (semUp (semgid, sh->goaliesWaitTeam) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }
    if (semDown (semgid, sh->playerRegistered) == -1)  {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }
    for (int i = 0; i < (NUMTEAMPLAYERS - 1); i++) {
        // Wake up players
        if (semUp (semgid, sh->playersWaitTeam) == -1) {
            perror ("error on the down operation for semaphore access (PL)");
            exit (EXIT_FAILURE);
        }
        // Wait for players to acknowledge registration
        if (semDown (semgid, sh->playerRegistered) == -1)  {
            perror ("error on the up operation for semaphore access (PL)");
            exit (EXIT_FAILURE);
        }
    }

    ret = sh->fSt.teamId;
    sh->fSt.teamId++;

    if (semUp (semgid, sh->refereeWaitTeams) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }
```

Caso ainda não haja jogadores suficientes para formar a equipa, o estado do jogador é atualizado para **WAITING_TEAM** e a variável *waiting* é definida como *true*.

```
} else {
    sh->fSt.st.playerStat[id] = WAITING_TEAM;
    saveState(nFic, &sh->fSt);
    waiting = true;
}
```

Depois destas verificações, a função sai da região crítica (código do professor) e, logo

em seguida, verifica-se se a variável *waiting* é *true*. Se for, o jogador espera até que uma equipa possa ser formada. Quando a equipa é formada, o id da equipa é retornado.

```
if (waiting) {
    if (semDown (semgid, sh->playersWaitTeam) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    ret = sh->fSt.teamId;

    if (semUp (semgid, sh->playerRegistered) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }
}
return ret;
```

3.3 Função waitReferee

A função **waitReferee** faz o jogador esperar pelo árbitro para iniciar o jogo. O estado do jogador é atualizado para **WAITING_START** e o estado interno é salvo, pelo que não muda comparativamente com o módulo anterior.

```
static void waitReferee (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

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

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->playersWaitReferee) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (semUp (semgid, sh->playing) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }
}
```

3.4 Função playUntilEnd

A função **playUntilEnd** faz o jogador jogar até o final do jogo. O estado do jogador é atualizado para **PLAYING** e o estado interno é salvo e, tal como a função anterior, mantém-se fiel à implementação anterior.

```
static void playUntilEnd (int id, int team)
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.playerStat[id] = (team == 1) ? PLAYING_1 : PLAYING_2;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->playersWaitEnd) == -1) {
        perror ("error on the up operation for semaphore access (PL)");
        exit (EXIT_FAILURE);
    }
}
```

4. semSharedMemReferee

Por fim, vou detalhar a implementação do módulo **semSharedMemReferee**, que é responsável pelas operações realizadas pelo árbitro no jogo de futebol. As operações/funções principais incluem:

- **arrive**: O árbitro leva algum tempo a chegar e atualiza o seu estado.
- **waitForTeams**: O árbitro espera que as equipas sejam formadas.
- **startGame**: O árbitro inicia o jogo.
- **play**: O árbitro permite que o jogo continue por um tempo.
- **endGame**: O árbitro termina o jogo.

4.1 Função arrive

A função **arrive** é responsável por simular a chegada do árbitro ao campo. Ela atualiza o estado do árbitro para **ARRIVING** e salva o estado interno, pelo que permanece praticamente igual aqui como nos outros módulos.

```
static void arrive ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.refereeStat = ARRIVING;
    saveState(nFic , &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    usleep((100.0*random())/(RAND_MAX+1.0)+10.0);
}
```

4.2 Função waitForTeams

A função **waitForTeams** faz o árbitro esperar até que as duas equipas estejam completamente formadas. O estado do árbitro é atualizado para **WAITING_TEAMS** e o estado interno é salvo.

```
static void waitForTeams ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.refereeStat = WAITING_TEAMS;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    // For each team
    for (int i = 1; i <= 2; i++) {
        if (semDown (semgid, sh->refereeWaitTeams) == -1) {
            perror ("error on the up operation for semaphore access (RF)");
            exit (EXIT_FAILURE);
        }
    }
}
```

4.3 Função startGame

A função **startGame** faz o árbitro iniciar o jogo. O estado do árbitro é atualizado para **STARTING_GAME** e o estado interno é salvo. Em seguida, o árbitro sinaliza para os jogadores e guarda-redes começarem a jogar.

```
static void startGame ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

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

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    // Signal teams to start playing (10 players including the late ones)
    for (int i = 0; i < NUMPLAYERS; i++) {
        if (semUp (semgid, sh->playersWaitReferee) == -1) {
            perror ("error on the up operation for semaphore access (RF)");
            exit (EXIT_FAILURE);
        }
    }

    for (int i = 0; i < NUMPLAYERS; i++) {
        if (semDown (semgid, sh->playing) == -1) {
            perror ("error on the down operation for semaphore access (RF)");
            exit (EXIT_FAILURE);
        }
    }
}
```

4.4 Função play

A função **play** faz com que o árbitro permita que o jogo continue por um tempo. O estado do árbitro é atualizado para **REFEREEING** e o estado interno é salvo. A função também faz o árbitro esperar durante um tempo aleatório antes de prosseguir.

```
static void play ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.refereeStat = REFEREEING;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }

    usleep((100.0*random())/(RAND_MAX+1.0)+900.0);
}
```

4.5 Função endGame

A função **endGame** faz com que o árbitro termine o jogo. O estado do árbitro é atualizado para **ENDING_GAME** e o estado interno é salvo. Em seguida, o árbitro sinaliza para os jogadores e guarda-redes que o jogo terminou.

```
static void endGame ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.refereeStat = ENDING_GAME;
    saveState(nFic , &sh->fSt);
    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (RF)");
        exit (EXIT_FAILURE);
    }
    // Signal teams that the game has ended
    for (int i = 0; i < NUMPLAYERS; i++) {
        if (semUp (semgid, sh->playersWaitEnd) == -1) {
            perror ("error on the up operation for semaphore access (RF)");
            exit (EXIT_FAILURE);
        }
    }
}
```

5. Testes

Para testar o projeto, é preciso compilar usando "*make all*"dentro da pasta *src*, depois, vamos para a pasta *run* e podemos correr o programa resultante **probSemSharedMemSoccerGame**. Existem também mais 2 *scripts* que ajudam nos testes, estes sendo: *filter.sh* e *run.sh*. O *run.sh* permite correr o programa resultante várias vezes, o que ajuda a detetar certos erros e notar o comportamento "comum"do projeto. O *filter.sh* permite uma melhor visualização do programa principal, sinalizando quando há mudanças de estados, sendo este o programa que mais usei no desenvolvimento deste projeto. É de notar que, na etapa de compilação, podemos também testar apenas um dos nossos módulos, usando "*make pl*, *gl* ou *rf*", o que foi importante no desenvolvimento deste projeto.

Executei o *filter.sh* e obtive:

SoccerGame - Description of the internal state

P00	P01	P02	P03	P04	P05	P06	P07	P08	P09	G00	G01	G02	R01
A	A	A	A	A	A	A	A	A	A	A	A	A	A
.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	W	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	W	.
.	.	.	.	.	.	.	.	.	.	.	L	.	.
.	.	.	.	.	.	.	W	.	.	.	.	.	.
.	W	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	W	.	.	.	.	.	.	.	.
.	.	.	.	F	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	s	.	.	.
.	.	.	.	.	.	.	s	.	.	.	.	.	.
.	s	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	s	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	W	.	.	.	.
.	.	.	.	s	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	W	.	.	.	.	.
.	.	.	W	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	F	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	S	.
.	.	.	.	.	.	.	.	.	S	.	.	.	.
.	.	.	.	.	.	.	.	S	.	.	.	.	.
.	.	.	S	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	S	.	.	.	.	.	.	.
.	.	L	.	.	.	.	.	.	.	.	.	.	.
L	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	W
.	.	.	.	.	.	.	.	.	.	.	.	.	S
.	.	.	.	.	.	.	p	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	p	.	.	.
.	p	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	p	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	P	.
.	.	.	.	p	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	P	.	.	.	.
.	.	.	P	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	P	.	.	.	.	.
.	.	.	.	.	.	P	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	R
.	.	.	.	.	.	.	.	.	.	.	.	.	E

Os resultados obtidos foram os esperados. O goalie1 e os players0 e 2, como chegam em último, não conseguem formar equipa e ficam LATE. A cada 4 jogadores que chegam, começam a formar equipa (F - FORMING_TEAM) e, assim que a equipa está formada, esperam pelo arbitro para começar o jogo (s - WAITING_START_1; S - WAITING_START_2). O árbitro chega e espera pelas equipas (W - WAITING_TEAMS) e a seguir começa o jogo (S - STARTING_GAME). Os jogadores e guarda-redes jogam (P - PLAYING_1; P - PLAYING_2) e o árbitro arbitra (R - REFEREEING). No final, o árbitro termina o jogo (E - ENDING_GAME).

Também corri o *script* run.sh, que correu o programa **probSemSharedMemSoccerGame** 1000 vezes. Fiz isto não para ver o resultado de cada iteração do *script*, mas para verificar que o resultado final do projeto permite rodar uma grande quantidade de testes e que não existe a possibilidade de existir um conflito mais raro entre os semáforos que impossibilite o término do *script* - *deadlock*. Felizmente, o *script* concluiu sem qualquer problema.

Os testes realizados demonstraram que a implementação dos módulos **semSharedMemGoalie**, **semSharedMemPlayer** e **semSharedMemReferee** está correta e funcional. A sincronização baseada em semáforos e memória partilhada foi eficaz na coordenação das ações entre jogadores, guarda-redes e árbitro.