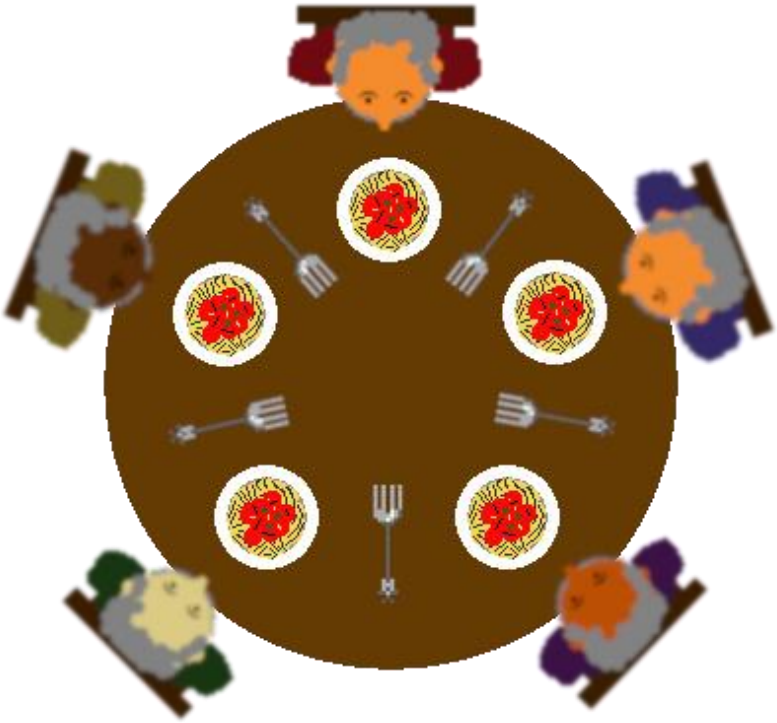
universidade de aveiro
departamento de eletrónica, telecomunicações e informática



# Jantar de Amigos (Restaurant)

## Compreensão dos mecanismos associados

## à execução e sincronização de processos e threads

**Sistemas Operativos – Professor Nuno Lau**
**Departamento de Eletrónica, Telecomunicações e Informática**
**Universidade de Aveiro**

*Rodrigo Aguiar - 108969*
*Filipe Obrist - 107471*

# INDICE

## Índice

# Introdução

Neste 2º trabalho, no âmbito da disciplina de Sistemas Operativos, pretendemos simular um jantar de amigos, em que existem três entidades: clientes, waiter e chef.

De modo a verificar as condições referidas no enunciado do problema, é necessário o uso e compreensão de semáforos, de modo a controlar o fluxo de execução e acesso de cada um dos processos envolvidos (clientes, waiter e chef) à região de memória partilhada.

Neste relatório estão explicadas as várias fases de desenvolvimento do programa, tal como o papel de cada um dos processos e funções envolvidos neste programa.

# Constantes e Semáforos

De modo a poder demonstrar os vários estados de cada uma das entidades envolvidas no problema, foram declaradas várias constantes.

```c
/* Client state constants */

/** \brief client initial state */
#define  INIT             1
/** \brief client is waiting for friends to arrive at table */
#define  WAIT_FOR_FRIENDS  2
/** \brief client is requesting food to waiter */
#define  FOOD_REQUEST      3
/** \brief client is waiting for food */
#define  WAIT_FOR_FOOD     4
/** \brief client is eating */
#define  EAT               5
/** \brief client is waiting for others to finish */
#define  WAIT_FOR_OTHERS   6
/** \brief client is waiting to complete payment */
#define  WAIT_FOR_BILL     7
/** \brief client finished meal */
#define  FINISHED          8
```

```c
/* Chef state constants */

/** \brief chef waits for food order */
#define  WAIT_FOR_ORDER    0
/** \brief chef is cooking */
#define  COOK              1
/** \brief chef is resting */
#define  REST              2
```

```c
/* Waiter state constants */

/** \brief waiter waits for food request */
#define  WAIT_FOR_REQUEST   0
/** \brief waiter takes food request to chef */
#define  INFORM_CHEF        1
/** \brief waiter takes food to table */
#define  TAKE_TO_TABLE      2
/** \brief waiter reiceives payment */
#define  RECEIVE_PAYMENT    3
```

Estas constantes são utilizadas durante o código para redefinir o estado das várias entidades e também na tabela que resulta da execução do script.

Devido ao uso de uma região de memória partilhada, foi também necessário utilizar semáforos para controlar o acesso de cada uma das entidades a esta região e também controlar o seu fluxo de execução.

```c
/** \brief number of semaphores in the set */
#define SEM_NU              (7)

#define MUTEX                1
#define FRIENDSARRIVED       2
#define REQUESTRECEIVED      3
#define FOODARRIVED          4
#define ALLFINISHED          5
#define WAITERREQUEST        6
#define WAITORDER            7
```

Todos estes semáforos foram utilizados ao longo do problema para garantir que as condições referidas no enunciado se verificam e que cada entidade tem o ciclo de vida esperado.

De modo a orientar o uso de cada semáforo para controlar o ciclo de execução de cada processo, realizamos a seguinte tabela que nos ajudou a implementar os semáforos ao longo que fomos resolvendo este projeto.

| **Semáforo** | **Entidade down** | **Função down** | **#down** | **Entidade up** | **Função up** | **#up** |
|---|---|---|---|---|---|---|
| friendsArrived | Client (todos menos o último) | waitFriends() | 19 | Client (apenas o último) | waitFriends() | 19 |
| requestReceived | Client | orderFood() | 1 | Waiter | informChef() | 1 |
| | | waitAndPay() | 1 | Waiter | takeFoodToTable() | 1 |
| foodArrived | Client | waitFood() | 20 | Waiter | takeFoodToTable() | 20 |
| allFinished | Client | waitAndPay() | 19 | Client (apenas o último) | waitAndPay() | 19 |
| waiterRequest | Waiter | waitForClient Or Chef() | 3 ( no total ) | Client | orderFood() | 1 |
| | | | | | waitAndPay() | 1 |
| | | | | Chef | processOrder() | 1 |
| waitOrder | Chef | waitForOrder() | 1 | Waiter | informChef() | 1 |

# Ciclo de vida – Client

Neste problema, temos em consideração um jantar onde existem 20 amigos (clientes) numa mesa. Cada cliente é representado pelo seu respetivo id, sendo assim os id's possíveis de 0 a 19.

De acordo com o enunciado do problema, temos que o primeiro cliente a chegar ao restaurante deverá fazer o pedido da comida, mas apenas depois de todos os outros amigos chegarem e o último cliente a chegar será o responsável por pagar a conta.

## waitFriends()

Nesta primeira função, cada cliente que chega ao restaurante atualiza o seu estado para 2 (WAIT_FOR_FRIENDS). Também são guardados os id's do primeiro e último cliente a chegar ao restaurante nas variáveis *sh->fSt.tableFirst* e *sh->fSt.tableLast*.

Como neste estado cada cliente tem que esperar por todos os amigos chegarem ao restaurante, todos os clientes menos o último fazem um down (decrementa uma unidade) no semáforo *FRIENDSARRIVED*, de modo a ficarem bloqueados até que o último cliente chegue.

```
sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS; // cliente fica à espera
if (sh->fSt.tableClients == 1)
{ // se for o primeiro cliente
    first = true;
    sh->fSt.tableFirst = id; // guarda o id do primeiro cliente
}
if (sh->fSt.tableClients == TABLESIZE){
sh->fSt.tableLast = id;     // guarda o id do ultimo cliente
```

Por sua vez, o último cliente irá fazer *TABLESIZE-1* ups no semáforo, de modo a desbloquear todos os outros clientes que estão bloqueados nesta secção de código, tendo-se assim concluído com sucesso a espera por todos os amigos.

O ultimo cliente também atualiza diretamente o seu estado para 4 ( *WAIT_FOR_FOOD* ).

```
if (id != sh->fSt.tableLast){
    semDown(semgid, sh->friendsArrived); // todos os clientes menos o ultimo ficam bloqueados em espera

}
if (sh->fSt.tableLast == id){    // se for o último cliente
    sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
    for (int i=0 ; i<TABLESIZE-1 ; i++){
    semUp(semgid, sh->friendsArrived); // desbloquear todos os outros clientes
}
}
```

Tendo-se concluído a espera por todos os amigos, o primeiro cliente a chegar ao restaurante poderá agora fazer o pedido da comida para todos.

## Orderfood()

Esta função destina-se apenas a ser utilizada pelo primeiro cliente, sendo que este é quem está responsável pelo pedido da comida.

O primeiro cliente atualiza o seu estado para 3 (*FOOD_REQUEST)* e incrementa a variável

*sh->fSt.foodRequest*, sinalizando assim ao waiter que pretende realizar um pedido de comida.

De modo que o waiter possa responder a este pedido, é necessário fazer um Up no semáforo *waiterRequest*, sendo que o waiter se encontra sempre bloqueado em espera de um pedido de um cliente ou do chef.

```
// if its the first client, update state to request food
if (sh->fSt.tableFirst == id){                // if its the first client
    sh->fSt.st.clientStat[id] = FOOD_REQUEST; // update state to request food
    sh->fSt.foodRequest++;                      // +1 pedido de comida
    semUp(semgid, sh->waiterRequest);          // waiter receives a request from the 1st client
    saveState(nFic, &(sh->fSt));               // save state
}
```

Após isto, o cliente que faz o pedido fica bloqueado até que o seu pedido seja respondido pelo waiter, ou seja, é feito um Down no semáforo *requestReceived*.

```
// wait for waiter to receive request , when he does it should up the semaphore
semDown(semgid,sh->requestReceived);
```

## waitFood()

Nesta função, todos os clientes esperam que o pedido da comida feito pelo 1º cliente chegue à mesa, pelo que atualizam o seu estado para 4 ( *WAIT_FOR_FOOD* ). Após isso, ficam bloqueados até que a comida chegue, através de um down no semáforo *foodArrived*, sendo o sinal que a comida chegou dado pelo waiter.

Após a comida chegar, os clientes atualizam o seu estado para 5 ( *EAT* ).

```
sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD; // update state to waiting for food
saveState(nFic, &(sh->fSt));
```

```
if (semDown(semgid, sh->foodArrived) == -1) // wait for food to arrive
{
    perror("error on the up operation for semaphore access (CT)");
    exit(EXIT_FAILURE);
}
```

```
// update state to eating
sh->fSt.st.clientStat[id] = EAT; // update state to eating
saveState(nFic, &(sh->fSt));
```

## waitAndPay()

Esta é a função final executada pelos clientes, que tem o objetivo de atualizar o seu estado quando acabam a sua refeição.
Também é responsável por realizar o pagamento da conta, através do último cliente.

Quando um cliente termina a sua refeição, altera o seu estado para 6 ( *WAIT_FOR_OTHERS* ) e espera neste estado até que todos os seus amigos terminem a refeição. É incrementado também o valor da variável *sh->fSt.tableFinishEat*, para representar que mais um cliente acabou a sua refeição.

Quando o ultimo cliente termina a sua refeição, desbloqueia todos os outros clientes, sinalizando-lhes que todos já acabaram a refeição, após o qual alteram o estado para 8 ( *FINISHED* ). Este

altera o seu estado para 7 ( *WAIT_FOR_BILL* ) e incrementa a flag relativa a um pedido de pagamento, sinalizando assim ao waiter um pedido de pagamento.

Após o waiter receber o pedido de pagamento, o ultimo cliente é desbloqueado e altera também o seu estado para 8 ( *FINISHED* ).

```
if (sh->fSt.tableLast == id){
    last = true;
    }
sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
sh->fSt.tableFinishEat++;
saveState(nFic, &(sh->fSt));
if (sh->fSt.tableFinishEat == TABLESIZE){
    for (int i = 0 ; i < TABLESIZE ; i++){
    semUp(semgid, sh->allFinished); // unblock all others
    }
}
```

```
semDown(semgid, sh->allFinished); // wait for all others to finish eating
/* insert your code here */

if (last)
{
    if (semDown(semgid, sh->mutex) == -1)
    { /* enter critical region */
        perror("error on the down operation for semaphore access (CT)");
        exit(EXIT_FAILURE);
    }

    sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
    saveState(nFic, &(sh->fSt));
    sh->fSt.paymentRequest++; // pedido de pagamento
    semUp(semgid, sh->waiterRequest); // waiter receives a payment request from the last client

    /* insert your code here */

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

   semDown(semgid,sh->requestReceived); // o ultimo cliente espera que o waiter receba a request
```

```
sh->fSt.st.clientStat[id] = FINISHED;
saveState(nFic, &(sh->fSt));
/* insert your code here */
```

Com isto conclui-se o ciclo de vida de um cliente, sendo que todos os clientes acabaram a refeição e o pagamento foi realizado.

# Ciclo de vida – Waiter

Neste problema, o waiter é responsável por atender os pedidos de comida e levá-los ao chefe, levar a comida à mesa e, no fim do jantar, receber o pagamento da conta do último cliente.

## waitForClientOrChef()

Esta função é utilizada pelo waiter para esperar por um pedido, quer seja de um cliente ou do chef.

Para começar, o waiter atualiza o seu estado para 0 ( *WAIT_FOR_REQUEST* ).

Após isso, o waiter fica bloqueado através de um Down no semáforo *waiterRequest*, sendo este o semáforo utilizado para bloquear o fluxo de execução do waiter sempre que não há nenhum pedido de clientes ou do chef.

```
// update state to waiting
sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
// save state
saveState(nFic, &sh->fSt);
```

```
if (semDown(semgid, sh->waiterRequest) == -1)
{ // make waiter wait for a request
    perror("error on the up operation for semaphore access (WT)");
    exit(EXIT_FAILURE);
}
```

Como esta função é utilizada pelo waiter para esperar por um pedido, também é necessário, quando é realizado um pedido, saber que pedido é em concreto, pelo que esta função apresenta um valor de retorno ( *ret* ), indicativo de qual foi o pedido feito ao waiter.

```
if (sh->fSt.foodRequest)
    ret = FOODREQ;
if (sh->fSt.foodReady)
    ret = FOODREADY;
if (sh->fSt.paymentRequest)
    ret = BILL;
// after reading the request reset the flags
sh->fSt.paymentRequest = 0;
sh->fSt.foodReady= 0;
sh->fSt.foodRequest = 0;
```

Estas três flags são utilizadas pelos clientes e pelo chef quando pretendem fazer um pedido, de modo a manter o ciclo de execução do waiter como esperado, sendo que este valor de retorno determina a próxima função a ser executada.

De modo a não haver conflitos, após a leitura de qual é o pedido, todas as flags são colocadas novamente a 0.

## informChef()

Esta função deverá ser executada pelo waiter quando receber um pedido de comida de um cliente, desbloqueando-o da função *waitForClientOrChef*.

O waiter começa por atualizar o seu estado para 1 ( *INFORM_CHEF* ).

```
sh->fSt.st.waiterStat = INFORM_CHEF;
saveState(nFic, &(sh->fSt));
```

Após isso, o waiter realiza um Up no semáforo waitOrder, que é utilizado pelo chef para esperar por um pedido do waiter, encontrando-se bloqueado até tal não acontecer.

O waiter também sinaliza ao primeiro cliente ( que fez o pedido ), que o pedido foi realizado com sucesso, desbloqueando-o através de um Up no semáforo *requestReceived.*

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

Sendo assim, o waiter entregou com sucesso o pedido ao chef, que vai proceder a cozinhar a comida.

Após executar esta função, o waiter voltará novamente à função waitForClientOrChef, onde irá estar bloqueado novamente até ter um pedido de um cliente ou do chef.

## takeFoodToTable()

Após o waiter entregar o pedido ao chef ficará à espera de que este o cozinhe.

Sendo assim, quando o chef acaba de cozinhar o pedido desbloqueia o waiter, fazendo um Up no semáforo *waiterRequest* e sinalizando que a comida está pronta através de uma flag.

Após isto, o waiter poderá prosseguir e levar a comida cozinhada para os clientes na mesa.

O waiter começa por atualizar o seu estado para 2 ( *TAKE_TO_TABLE* ).

Como a comida já se encontra na mesa, o waiter desbloqueia todos os clientes que se encontravam à espera de que a comida chegasse, que poderão agora prosseguir com o seu ciclo de vida

```
sh->fSt.st.waiterStat = TAKE_TO_TABLE;
// save waiter state
saveState(nFic, &(sh->fSt));
for(int i=0; i< TABLESIZE; i++){
semUp(semgid, sh->foodArrived);     // unblock all clients waiting for food
}
```

### receivePayment()

Nesta última função, o waiter recebe o pagamento da conta pelo último cliente a chegar ao restaurante quando este acaba a sua refeição, sinalizando ao cliente que recebeu o pedido com sucesso, desbloqueando-o.

```
// update waiter state
sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
saveState(nFic, &(sh->fSt));
// save waiter state
semUp(semgid, sh->requestReceived); // up request received semaphore
```

## Ciclo de vida – Chef

Neste problema, o chef é responsável por cozinhar o pedido feito pelo último cliente, sendo que o faz apenas uma vez.

### waitForOrder()

Tal como o nome da função sugere, nesta função o chef espera que receba um pedido de comida do waiter para cozinhar, encontrando-se bloqueado até tal não acontecer.

```
static void waitForOrder()
{
    if (semDown(semgid, sh->waitOrder) == -1) // chef waits for a request to cook
    {
        perror("error on the up operation for semaphore access (PT)");
        exit(EXIT_FAILURE);
    }
```

Quando o waiter informa o chef de um pedido para cozinhar, faz um Up deste semáforo ( *waitOrder* ).

Após isto, o chef irá cozinhar o pedido, alterando o seu estado para 1 ( *COOK* ) .

```
sh->fSt.st.chefStat = COOK;
saveState(nFic, &sh->fSt);
```

## processOrder()

Nesta função, o chef cozinha o pedido que lhe foi dado e sinaliza ao waiter que tem um pedido para levar para a mesa.

Primeiramente o chef começa por cozinhar o pedido, sendo o tempo que este demora a faze-lo simulado através da função *usleep*.

```
usleep((unsigned int)floor((MAXCOOK * random()) / RAND_MAX + 100.0));
```

Quando o chef termina de cozinhar o pedido, altera o seu estado para 2 ( *REST* ), altera o valor da flag *sh->fSt.foodReady* para 1 e, por fim, desbloqueia o waiter que se encontrava à espera de um pedido.

```
    if (semDown(semgid, sh->mutex) == -1)
    { /* enter critical region */
        perror("error on the up operation for semaphore access (PT)");
        exit(EXIT_FAILURE);
    }

    sh->fSt.st.chefStat = REST; // chef has finished cooking, he can rest
    sh->fSt.foodReady = 1; // food is ready
    saveState(nFic, &sh->fSt);

    /* insert your code here */

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

    semUp(semgid,sh->waiterRequest); // call waiter to deliver food
```

Com isto, conclui-se com sucesso o ciclo de vida do chef, que recebeu o pedido, cozinhou-o e sinalizou ao waiter que podia levar o pedido para a mesa.

# Resultados

```
Run n.º 1
                                Restaurant - Description of the internal state

 CH WT  C00 C01 C02 C03 C04 C05 C06 C07 C08 C09 C10 C11 C12 C13 C14 C15 C16 C17 C18 C19  ATT FIE 1st las
  0  0    1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   2    1   0  19  -1
  0  0    1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   2    1   0  19  -1
  0  0    1   1   1   1   1   1   1   1   2   1   1   1   1   1   1   1   1   1   1   2    2   0  19  -1
  0  0    1   1   1   1   2   1   1   1   2   1   1   1   1   1   1   1   1   1   1   2    3   0  19  -1
  0  0    1   1   1   1   2   1   1   1   2   1   1   2   1   1   1   1   1   1   1   2    4   0  19  -1
  0  0    2   1   1   1   2   1   1   1   2   1   1   2   1   1   1   1   1   1   1   2    5   0  19  -1
  0  0    2   1   1   1   2   1   1   1   2   1   1   2   1   1   2   1   1   1   1   2    6   0  19  -1
  0  0    2   1   1   1   2   1   1   1   2   1   1   2   1   1   2   1   1   2   1   2    7   0  19  -1
  0  0    2   1   1   1   2   1   1   1   2   1   1   2   1   2   2   1   1   2   1   2    8   0  19  -1
  0  0    2   1   1   1   2   1   1   1   2   2   1   2   1   2   2   1   1   2   1   2    9   0  19  -1
  0  0    2   1   2   1   2   1   1   1   2   2   1   2   1   2   2   1   1   2   1   2   10   0  19  -1
  0  0    2   1   2   1   2   1   1   1   2   2   1   2   2   2   2   1   1   2   1   2   11   0  19  -1
  0  0    2   2   2   1   2   1   1   1   2   2   1   2   2   2   2   1   1   2   1   2   12   0  19  -1
  0  0    2   2   2   1   2   1   1   1   2   2   1   2   2   2   2   2   1   2   1   2   13   0  19  -1
  0  0    2   2   2   1   2   1   1   1   2   2   1   2   2   2   2   2   1   2   2   2   14   0  19  -1
  0  0    2   2   2   1   2   1   1   2   2   2   1   2   2   2   2   2   1   2   2   2   15   0  19  -1
  0  0    2   2   2   2   2   1   1   2   2   2   1   2   2   2   2   2   1   2   2   2   16   0  19  -1
  0  0    2   2   2   2   2   1   1   2   2   2   2   2   2   2   2   2   1   2   2   2   17   0  19  -1
  0  0    2   2   2   2   2   1   2   2   2   2   2   2   2   2   2   2   1   2   2   2   18   0  19  -1
  0  0    2   2   2   2   2   1   2   2   2   2   2   2   2   2   2   2   2   2   2   2   19   0  19  -1
  0  0    2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   20   0  19   5
  0  0    2   2   2   2   2   4   2   2   2   2   2   2   2   2   2   2   2   2   2   3   20   0  19   5
  0  0    2   2   2   2   2   4   2   2   4   2   2   2   2   2   2   2   2   2   2   3   20   0  19   5
  0  0    2   2   2   2   2   4   2   2   4   2   2   4   2   2   2   2   2   2   2   3   20   0  19   5
  0  0    2   2   2   2   4   4   2   2   4   2   2   4   2   2   2   2   2   2   2   3   20   0  19   5
  0  0    2   2   2   2   4   4   2   2   4   2   2   4   2   2   4   2   2   2   2   3   20   0  19   5
  0  0    2   2   2   2   4   4   2   2   4   2   2   4   2   2   4   2   2   4   2   3   20   0  19   5
  0  0    4   2   2   2   4   4   2   2   4   2   2   4   2   2   4   2   2   4   2   3   20   0  19   5
  0  0    4   2   2   2   4   4   2   2   4   2   2   4   2   4   4   2   2   4   2   3   20   0  19   5
```

```
  0  0    4   2   2   2   4   4   2   2   4   4   2   4   2   4   4   2   2   4   2   3   20   0  19   5
  0  0    4   2   2   2   4   4   2   2   4   4   2   4   4   4   4   2   2   4   2   3   20   0  19   5
  0  0    4   4   2   2   4   4   2   2   4   4   2   4   4   4   4   2   2   4   2   3   20   0  19   5
  0  0    4   4   4   2   4   4   2   2   4   4   2   4   4   4   4   2   2   4   2   3   20   0  19   5
  0  0    4   4   4   2   4   4   2   2   4   4   2   4   4   4   4   4   2   4   2   3   20   0  19   5
  0  0    4   4   4   2   4   4   2   2   4   4   2   4   4   4   4   4   2   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   2   2   4   4   2   4   4   4   4   4   2   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   2   4   4   4   2   4   4   4   4   4   2   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   2   4   4   4   4   4   4   4   4   4   2   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   2   4   4   4   4   4   4   4   4   4   2   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   2   4   4   3   20   0  19   5
  0  1    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   3   20   0  19   5
  0  0    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   20   0  19   5
  1  0    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   20   0  19   5
  2  0    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   20   0  19   5
  2  2    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   20   0  19   5
  2  2    4   4   4   4   4   4   4   4   5   4   4   4   4   4   4   4   4   4   4   4   20   0  19   5
  2  2    4   4   4   4   4   4   4   4   5   4   4   5   4   4   4   4   4   4   4   4   20   0  19   5
  2  2    4   4   4   4   5   4   4   4   5   4   4   5   4   4   4   4   4   4   4   4   20   0  19   5
```

```
2 2 4 4 4 4 5 4 4 4 5 4 4 5 4 4 5 4 4 4 4 4 20 0 19 5
2 2 4 4 4 4 5 4 4 4 5 4 4 5 4 4 5 4 4 5 4 4 20 0 19 5
2 2 5 4 4 4 5 4 4 4 5 4 4 5 4 4 5 4 4 5 4 4 20 0 19 5
2 2 5 4 4 4 5 4 4 4 5 4 4 5 4 5 5 4 4 5 4 4 20 0 19 5
2 2 5 4 4 4 5 4 4 4 5 5 4 5 4 5 5 4 4 5 4 4 20 0 19 5
2 2 5 4 4 4 5 4 4 4 5 5 4 5 5 5 5 4 4 5 4 4 20 0 19 5
2 2 5 5 4 4 5 4 4 4 5 5 4 5 5 5 5 4 4 5 4 4 20 0 19 5
2 2 5 5 5 4 5 4 4 4 5 5 4 5 5 5 5 4 4 5 4 4 20 0 19 5
2 2 5 5 5 4 5 4 4 4 5 5 4 5 5 5 5 5 4 5 4 4 20 0 19 5
2 2 5 5 5 4 5 4 4 4 5 5 4 5 5 5 5 5 4 5 5 4 20 0 19 5
2 2 5 5 5 5 5 4 4 4 5 5 4 5 5 5 5 5 4 5 5 4 20 0 19 5
2 2 5 5 5 5 5 4 4 5 5 5 4 5 5 5 5 5 4 5 5 4 20 0 19 5
2 2 5 5 5 5 5 4 4 5 5 5 5 5 5 5 5 5 4 5 5 4 20 0 19 5
2 2 5 5 5 5 5 4 5 5 5 5 5 5 5 5 5 5 4 5 5 4 20 0 19 5
2 2 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 5 5 4 20 0 19 5
2 2 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 20 0 19 5
2 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 20 0 19 5
2 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 20 0 19 5
2 0 5 6 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 20 1 19 5
2 0 5 6 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 5 5 5 20 2 19 5
2 0 6 6 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 5 5 5 20 3 19 5
2 0 6 6 5 5 5 6 6 5 5 5 5 5 5 5 5 5 5 5 5 5 20 4 19 5
2 0 6 6 5 5 5 6 6 5 5 5 6 5 5 5 5 5 5 5 5 5 20 5 19 5
2 0 6 6 5 5 6 6 6 5 5 5 6 5 5 5 5 5 5 5 5 5 20 6 19 5
2 0 6 6 5 5 6 6 6 5 5 5 6 5 5 5 5 6 5 5 5 5 20 7 19 5
2 0 6 6 5 5 6 6 6 5 5 6 6 5 5 5 5 6 5 5 5 5 20 8 19 5
2 0 6 6 5 6 6 6 6 5 5 6 6 5 5 5 5 6 5 5 5 5 20 9 19 5
2 0 6 6 5 6 6 6 6 5 5 6 6 5 5 5 5 6 5 5 5 6 20 10 19 5
2 0 6 6 5 6 6 6 6 5 5 6 6 5 5 6 5 6 5 5 5 6 20 11 19 5
2 0 6 6 5 6 6 6 6 6 5 6 6 5 5 6 5 6 5 5 5 6 20 12 19 5
2 0 6 6 5 6 6 6 6 6 5 6 6 5 5 6 5 6 5 5 6 6 20 13 19 5
```

```
2 0 6 6 5 6 6 6 6 6 5 6 6 5 6 6 5 6 5 5 6 6 20 14 19 5
2 0 6 6 5 6 6 6 6 6 5 6 6 5 6 6 5 6 5 6 6 6 20 15 19 5
2 0 6 6 5 6 6 6 6 6 5 6 6 6 6 6 5 6 5 6 6 6 20 16 19 5
2 0 6 6 5 6 6 6 6 6 5 6 6 6 6 6 5 6 6 6 6 6 20 17 19 5
2 0 6 6 5 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 20 18 19 5
2 0 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 20 19 19 5
2 0 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 19 5
2 0 6 8 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 19 5
2 0 8 8 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 19 5
2 0 8 8 6 6 6 6 8 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 19 5
2 0 8 8 6 6 6 7 8 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 19 5
2 0 8 8 6 6 6 7 8 6 6 6 8 6 6 6 6 6 6 6 6 6 20 20 19 5
2 0 8 8 6 6 6 7 8 6 6 6 8 6 6 6 6 8 6 6 6 6 20 20 19 5
2 0 8 8 6 6 8 7 8 6 6 6 8 6 6 6 6 8 6 6 6 6 20 20 19 5
2 0 8 8 6 6 8 7 8 6 6 8 8 6 6 6 6 8 6 6 6 6 20 20 19 5
2 0 8 8 6 8 8 7 8 6 6 8 8 6 6 6 6 8 6 6 6 6 20 20 19 5
2 0 8 8 6 8 8 7 8 6 6 8 8 6 6 6 6 8 6 6 6 8 20 20 19 5
2 0 8 8 6 8 8 7 8 6 6 8 8 6 6 8 6 8 6 6 6 8 20 20 19 5
2 0 8 8 6 8 8 7 8 8 6 8 8 6 6 8 6 8 6 6 6 8 20 20 19 5
2 0 8 8 6 8 8 7 8 8 6 8 8 6 6 8 6 8 6 6 8 8 20 20 19 5
2 0 8 8 6 8 8 7 8 8 6 8 8 6 6 8 6 8 6 8 8 8 20 20 19 5
2 0 8 8 6 8 8 7 8 8 6 8 8 6 8 8 6 8 6 8 8 8 20 20 19 5
2 0 8 8 6 8 8 7 8 8 6 8 8 8 8 8 6 8 8 8 8 8 20 20 19 5
2 0 8 8 6 8 8 7 8 8 6 8 8 8 8 8 8 8 8 8 8 8 20 20 19 5
2 0 8 8 8 8 8 7 8 8 6 8 8 8 8 8 8 8 8 8 8 8 20 20 19 5
2 0 8 8 8 8 8 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 19 5
2 0 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 19 5
2 3 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 19 5
```

Devido ao tamanho da descrição do problema, para uma melhor visualização, será também fornecido um ficheiro na pasta com os resultados obtidos.

Para testar a existência de deadlocks, foi utilizado o programa run.sh para executar o script 1000 vezes, conseguindo completar as 1000 execuções com sucesso, pelo que se pode concluir que não há possibilidade de ocorrer deadlocks no script.

# Conclusão

De acordo com os resultados obtidos, podemos concluir que o problema foi resolvido com sucesso.
O uso de semáforos e flags permitiu a sincronização dos vários processos envolvidos e garantiu que o ciclo de vida e de execução de cada processo foi cumprido com sucesso, tendo cada processo os recursos que precisava para executar corretamente.

Este trabalho permitiu melhorar o nosso conhecimento sobre semáforos e a sincronização de vários processos numa mesma região de memória partilhada, sendo que era este o objetivo do mesmo.

# Bibliografia

- Slides das aulas teóricas

- Aula prática nº 10