Jantar de Amigos

SISTEMAS OPERATIVOS

Trabalho Realizado:

Alexandre Cotorobai - 107849

Vitalie Bologa - 107854

universidade de aveiro

2022/2023

Índice

Introdução

No âmbito da disciplina de Sistemas Operativos, foi-nos proposto como segundo trabalho prático, uma simulação de um restaurante com três entidades distintas, *Client*, *Waiter*, *Chef*.

Cada uma dessas entidades são processos independentes e a sua sincronização será realizada através de semáforos e da memória partilhada.

Durante a execução deste trabalho tivemos especial atenção ao acesso à zona crítica, impedindo que dois processos acedessem ao mesmo tempo à memória partilhada, o que poderia resultar em desconcordâncias no programa.

Este relatório tem como objetivo explicar todo o raciocínio que nos levou a formular o código necessário à boa execução e sincronização de todos os processos intervenientes.

Introdução ao Problema

Variáveis gerais

Estas variáveis são usadas ao longo do programa como constantes, pois não podem ser modificadas durante a execução do mesmo.

```
/* Generic parameters */

/** \brief table capacity, equal to number of clients */
#define  TABLESIZE        20
/** \brief controls time taken to eat */
#define  MAXEAT       500000
/** \brief controls time taken to cook */
#define  MAXCOOK     3000000
```

Estruturas de dados interna

O ficheiro *probDataStruct* contém as estruturas de dados usadas ao longo do programa.

A estrutura *STAT* vai armazenar os estados de todas as entidades do problema. Este possui três parâmetros: um inteiro para guardar o estado do *Waiter*, um inteiro para guardar o estado do *Chef* e por fim, um *array* de inteiros com tamanho do número de elementos da mesa, para guardar os estados de todos os clientes.

```
/**
 *  \brief Definition of <em>state of the intervening entities</em> data type.
 */
typedef struct {
    /** \brief waiter state */
    unsigned int waiterStat;
    /** \brief chef state */
    unsigned int chefStat;
    /** \brief client state array */
    unsigned int clientStat[TABLESIZE];

} STAT;
```

Quanto ao tipo de dados *FULL_STAT*, representa o estado completo do problema, que inclui o estado das 3 entidades, bem como das outras variáveis relacionadas à mesa, pedidos de comida e registo do primeiro e último cliente a chegar a mesa.

```
/**
 *  \brief Definition of <em>full state of the problem</em> data type.
 */
typedef struct
{   /** \brief state of all intervening entities */
    STAT st;

    /** \brief number of clients at table */
    int tableClients;
    /** \brief number of clients that finished eating */
    int tableFinishEat;

    /** \brief flag of food request from client to waiter */
    int foodRequest;
    /** \brief flag of food order from waiter to chef */
    int foodOrder;
    /** \brief flag of food ready from chef to waiter */
    int foodReady;
    /** \brief flag of payment request from client to waiter */
    int paymentRequest;

    /** \brief id of first client to arrive */
    int tableLast;
    /** \brief id of last client to arrive */
    int tableFirst;

} FULL_STAT;
```

Definição de semáforos

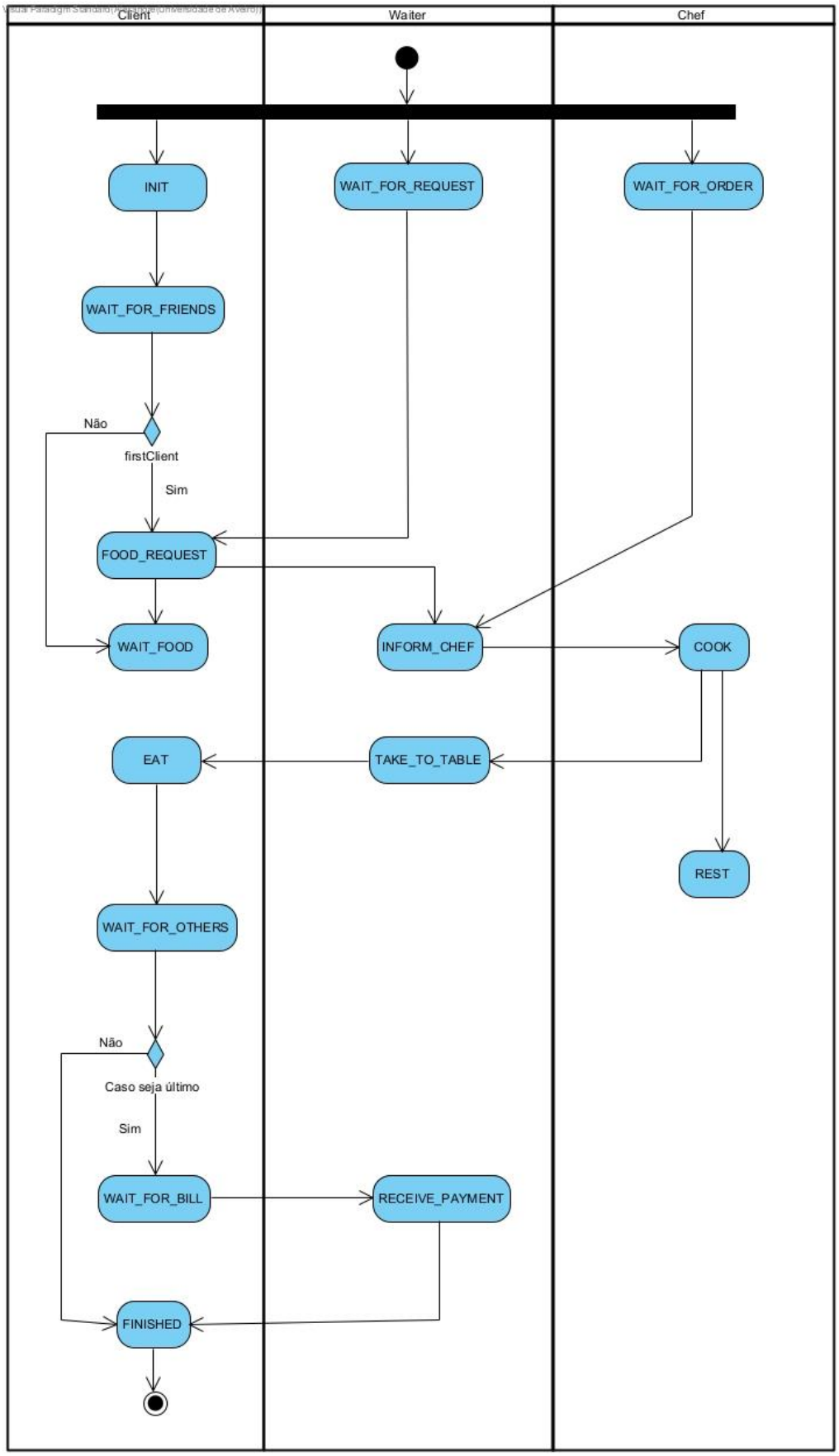
O ficheiro *sharedDataSync.h* define a estrutura *SHARED_DATA*, que representa as informações compartilhadas usadas pelos processos para comunicar e sincronizar as ações. De forma a controlar o fluxo de execução utilizamos os semáforos para manipular processos, bloqueando-os caso necessário, até certas operações serem terminadas.

```
typedef struct
        { /** \brief full state of the problem */
          FULL_STAT fSt;

          /* semaphores ids */
          /** \brief identification of critical region protection semaphore - val = 1 */
          unsigned int mutex;
          /** \brief identification of semaphore used by clients to wait for friends to arrive - val = 0 */
          unsigned int friendsArrived;
          /** \brief identification of semaphore used by client to wait for waiter after a request - val = 0 */
          unsigned int requestReceived;
          /** \brief identification of semaphore used by clients to wait for food - val = 0 */
          unsigned int foodArrived;
          /** \brief identification of semaphore used by clients to wait for friends to finish eating - val = 0  */
          unsigned int allFinished;
          /** \brief identification of semaphore used by waiter to wait for requests - val = 0  */
          unsigned int waiterRequest;
          /** \brief identification of semaphore used by chef to wait for order - val = 0  */
          unsigned int waitOrder;

        } SHARED_DATA;

/** \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

#endif /* SHAREDDATASYNC_H_ */
```

Comportamento dos semáforos

Semáforo	Down			Up		
	Entidade	Função	Nº Downs	Entidade	Função	Nº Ups
mutex	Client	waitFriends	1	Client	waitFriends	1
		waitFood	2		waitFood	2
	Client (except lastClient)	waitAndPay	2	Client (except lastClient)	waitAndPay	2
	First Client	orderFood	1	First Client	orderFood	1
	Last Client	waitAndPay	3	Last Client	waitAndPay	3
	Chef	waitForOrder	1	Chef	waitForOrder	1
		processOrder	1		processOrder	1
	Waiter	waitForClientOrChef	6	Waiter	waitForClientOrChef	6
		informChef	1		informChef	1
		takeFoodToTable	1		takeFoodToTable	1
		receivedPayment	1		receivedPayment	1
friendsArrived	Client (except lastClient)	waitFriends	1	Last Client	waitFriends	TABLESIZE-1 (19)
requestReceived	First Client	orderFood	1	Waiter	informChef	1
	Last Client	waitAndPay	1		receivedPayment	1
foodArrived	Client	waitFood	1	Waiter	takeFoodToTable	TABLESIZE (20)
waiterRequest	Waiter	waitForClientOrChef	3	First Client	orderFood	1
				Last Client	waitAndPay	1
				Chef	processOrder	1
waitOrder	Chef	waitForOrder	1	Waiter	informChef	1
allFinished	Client	waitAndPay	1	Last Client eating	waitAndPay	TABLESIZE (20)

Diagrama de Estados



Implementação

CLIENT

Estado	Valor	Significado
INIT	1	Estado inicial do cliente quando chega na mesa.
WAIT_FOR_FRIENDS	2	Após o estado inicial, e caso não seja o último amigo a chegar, espera pelos restantes que chegarão à mesa.
FOOD_REQUEST	3	Uma vez a mesa completa, o primeiro amigo que chegou faz o pedido da comida ao *Waiter*.
WAIT_FOOD	4	No final de fazer o pedido, eles esperam pela entrega da comida.
EAT	5	Após o *Waiter* entregar a comida para toda a mesa os clientes comem.
WAIT_FOR_OTHERS	6	Cada membro da mesa vai acabando por comer e esperar que o resta dos amigos acabe de comer.
WAIT_FOR_BILL	7	Quando toda a gente acabar de comer, o último amigo que chegou vai esperar pela conta.
FINISHED	8	Após toda a mesa acabar de comer e o último pagar a conta, são declarados como *finished*.

waitFriends()

Nesta função começamos por entrar na zona crítica e cada vez que chega um amigo/*client*, é incrementado o número de *clients* na mesa na memória partilhada. No caso do primeiro cliente, será alterada a variável *boolean first* para *true*, para que o seu *ID* seja guardado na memória partilhada no *tableFirst*.

O estado de cada *client* será alterado para *WAIT_FOR_FRIENDS* e guardado.

Quando chega o último *client* o *ID* do mesmo é guardado no *tableLast*, pois mais tarde será ele a fazer o pagamento do pedido. Também é incrementado o semáforo *friends_Arrived*, permitindo que todos os processos dos outros clientes da mesa (que até então estavam pendentes) prossigam na execução.

No final da função saímos região crítica e aplicamos a operação *semDown* no semáforo *friends_Arrived*, bloqueando o próprio até que o último amigo da mesa chegue.

```c
static bool waitFriends(int id)
{
    bool first = false;

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

    /* insert your code here */

    sh->fSt.tableClients++;
    sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;

    if(sh->fSt.tableClients == 1){
        first = true;
        sh->fSt.tableFirst = id;
    }

    if(sh->fSt.tableClients == TABLESIZE){
        sh->fSt.tableLast = id;
        for(int i = 0; i < TABLESIZE - 1; i++){
            semUp(semgid, sh->friendsArrived);
        }

    }

    saveState(nFic, &(sh->fSt));


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

    /* insert your code here */
    if (sh->fSt.tableClients != TABLESIZE){
        semDown(semgid, sh->friendsArrived);
        }

    return first;
}
```

orderFood()

Nesta função, será solicitado comida, logo entramos na região crítica, definimos o *foodRequest* como "1" e avisamos o *waiter* incrementando o semáforo *waitRequest*.

O estado de cada *client* será alterado para *FOOD_REQUEST* e dado *print* de uma linha de todos os estados no terminal.

Por fim, saímos da região crítica e a função espera que o *waiter* receba o pedido chamando a função *semDown* com o semáforo *requestReceived*.

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

    /* insert your code here */
    sh->fSt.foodRequest = 1;

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

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

    /* insert your code here */
    if (semDown(semgid, sh->requestReceived) == -1) {
    perror("error on the up operation for semaphore access(GL)");
    exit(EXIT_FAILURE);
    }
}
```

waitFood()

A função *waitFood*, tem o objetivo de esperar a entrega da comida por parte do *waiter*. Logo, entrando na região crítica o estado do cliente é atualizado para *WAIT_FOR_FOOD* e impressos os estados no terminal. Então até a comida chegar, é decrementado o semáforo *foodArrived*, com intuito do processo dar *block* até o *waiter* entregar comida à mesa inteira

Por fim, saímos da região crítica e uma vez entregue a comida pelo *Waiter* a todos os elementos da mesa, podemos atualizar o estado do *client* para *EAT* e imprimir mais uma linha de estados no terminal.

```c
static void waitFood (int id)
{

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

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

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

    /* insert your code here */
    if (semDown(semgid, sh->foodArrived) == -1) {
        perror("error on the up operation for semaphore access(CT)");
        exit(EXIT_FAILURE);
    }

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

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

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

waitAndPay()

Por fim, a última função dedicada ao *client*, tem o objetivo de esperar que todos os *clients* da mesma mesa acabem de comer e que o último *client* que chegou pague a comida.

Atendendo a esse ponto, começamos por verificar se o *ID* analisado é o último amigo a chegar à mesa, através da variável booleana *last*. O estado dos *clients* é atualizado para *WAIT_FOR_FRIENDS* e impresso uma linha de estados na consola.

Cada vez que um *client* da mesa acaba de comer, é incrementado no contador *tableFinishEat*.

Seguidamente o semáforo *allFinished* é decrementado, bloqueando o próprio até que o último amigo da mesa acabe de comer. Quando o contador chegar ao número de elementos da mesa, é incrementado o semáforo *allFinished*, permitindo que todos os processos dos outros clientes da mesa (que até então estavam pendentes) prossigam na execução.

Caso o cliente seja o último a chegar à mesa, o estado é atualizado para *WAIT_FOR_BILL* (e impressos os estados no terminal). A variável *paymentRequest* é definida a "1", para indicar que é necessário pagar a conta. Isto tudo, sempre recorrendo à região crítica para o acesso da memória partilhada.

Depois incrementamos o semáforo *waitRequest* para solicitar ao *waiter* o pagamento da conta e esperamos que o mesmo receba a solicitação, ao decrementar o semáforo *requestReceived*.

Por fim, o estado do cliente é atualizado para *FINISHED*, de forma a sinalizar que todos os clientes acabaram de comer e são impressos os estados.

```
static void waitAndPay (int id)
{
    bool last=false;

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


    /* insert your code here */

    last = (sh->fSt.tableLast == id);

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

    sh->fSt.tableFinishEat++;
    if (sh->fSt.tableFinishEat == TABLESIZE){
        for (int i = 0; i < TABLESIZE; i++){
            if (semUp (semgid, sh->allFinished) == -1) {
                perror ("error on the down operation for semaphore access (CT)");
                exit (EXIT_FAILURE);
            }
        }
    }


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

    /* insert your code here */
    if (semDown (semgid, sh->allFinished) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
```

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

    /* insert your code here */

    sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
    saveState(nFic, &sh->fSt);
    sh->fSt.paymentRequest = 1;

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

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

    /* insert your code here */

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

}

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

/* insert your code here */

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

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

WAITER

Estado	Valor	Significado
WAIT_FOR_REQUEST	0	Estado em que o *Waiter* espera que lhe seja atribuída alguma tarefa.
INFORM_CHEF	1	Caso lhe seja solicitado algum pedido de comida por parte do *client*, este leva o pedido para o *chef*
TAKE_TO_TABLE	2	Após a comida estar pronta, o *waiter* leva o pedido à mesa do *client*.
RECEIVE_PAYMENT	3	No final da mesa acabar de comer, o *Waiter* vai receber o pagamento por parte do *last client*.

waitForClientOrChef()

Na função *waitForClientOrChef* é onde o *Waiter* vai receber todos os pedidos que lhe forem atribuidos.

Quando a função é chamada o *Waiter* passa (caso o *mutex* o permita) ao estado *WAIT_FOR_REQUEST*, estado esse que informa que o mesmo está livre para receber uma tarefa.

Após a alteração de estado, o *Waiter* ficará à espera de receber algum pedido novo, aplicando a operação *semDown* ao semáforo *waiterRequest*.

Assim que ele receba algum pedido vai voltar a entrar na zona critica e analisar que tipo de pedido é que recebeu. Existem três possíveis: *FOODREQ* (*client* faz pedido da comida), *FOODREADY* (quando o chefe acaba de preparar a comida) e *BILL* (quando o cliente vai fazer pagamento).

No final esta função retorna um dos três tipos de *request*.

```c
static int waitForClientOrChef()
{
    int ret=0;
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

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


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

    /* insert your code here */
    if (semDown (semgid, sh->waiterRequest) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

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

    /* insert your code here */

    if (sh->fSt.foodRequest == 1) {
        ret = FOODREQ;
        sh->fSt.foodRequest = 0;
    } else if (sh->fSt.foodReady == 1) {
        sh->fSt.foodReady = 0;
        ret = FOODREADY;
    } else if (sh->fSt.paymentRequest == 1) {
        sh->fSt.paymentRequest = 0;
        ret = BILL;
    }

    if (ret == '\0'){
        perror ("waitForClientOrChef: error in return value (WT)");
        exit (EXIT_FAILURE);
    }

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

    return ret;

}
```

InformChef()

Esta função limita-se a informar o chefe sobre um pedido.

Aqui a memória partilhada é acedida, recorrendo à região crítica, assinala que se trata de um pedido de comida (*foodOrder* = 1) e passa o estado do *Waiter* a *INFORM_CHEF* e também é impressa uma nova linha de estados.

Após isso, aplicamos a operação *semUp* ao semáforo *requestReceived*, para o *Waiter* informar o Cliente que este recebeu o pedido, e *semUp* ao semáforo *waitOrder*, para o *Chef* começar a confecionar o pedido.

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

    /* insert your code here */

    sh->fSt.foodOrder = 1;
    sh->fSt.st.waiterStat = INFORM_CHEF;
    saveState(nFic, &(sh->fSt));

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

    /* insert your code here */

    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);
    }
}
```

takeFoodToTable()

Esta função é um processo do *waiter* para levar comida à mesa.

Começamos por atualizar o estado do *waiter* como *TAKE_TO_TABLE* (dentro da região critica). De forma a indicar que a comida chegou a todos os clientes, iteramos o *tableClients* e por cada cliente incrementamos o semáforo *foodArrived* para que os clientes possam começar a comer.

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

    /* insert your code here */
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &(sh->fSt));
    for (int i = 0; i < sh->fSt.tableClients; i++) {
        if (semUp (semgid, sh->foodArrived) == -1) {
            perror ("error on the up operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
    }
    if (semUp (semgid, sh->mutex) == -1)  {
     perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
}
```

receivePayment()

Aqui iremos novamente entrar na região crítica para alterar estado do *Waiter* para *RECEIVE_PAYMENT* e impresso no terminal uma linha com os estados.

Depois aplicamos a operação *semUp* ao semáforo *requestReceived*, para avisar o *Client* que está pronto a receber o pagamento.

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

    /* insert your code here */

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

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

CHEF

Estado	Valor	Significado
WAIT_FOR_ORDER	0	Inicialmente o *chef* encontra-se preparado para fazer a comida, logo fica a espera pelo pedido que será entregue pelo *waiter*.
COOK	1	Após receber o pedido, começa a cozinhar o pedido para uma determinada mesa.
REST	2	No fim de fazer a comida ele descansa.

waitForOrder()

Esta função inicia aguardando pelo pedido do *Waiter* através do decremento do semáforo *waitOrder*. Após o *chef* receber o pedido atualiza o semáforo *foodOrder* para "0" de forma a informar que recebeu-o. E por fim, o estado é atualizado para *COOK* e impresso, simbolizando que começou a fazer a comida.

```c
static void waitForOrder ()
{
    /* insert your code here */

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

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

    /* insert your code here */
    sh->fSt.foodOrder = 0;
    sh->fSt.st.chefStat = COOK;
    saveState(nFic, &sh->fSt);

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

processOrder()

Entramos na região crítica e alterado na memória partilhada o *foodReady* para 1, mostrando que a comida está pronta a ser entregue, e o estado do *chef* para *REST*, pois sempre que o *chef* acaba de cozinhar ele descansa.

E terminamos a chamar a função *semUp* para o semaforo *waiterRequest*, de modo a chamar o *Waiter* para ir entregar a comida.

```c
static void processOrder ()
{
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));

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

    /* insert your code here */

    sh->fSt.foodReady = 1;
    sh->fSt.st.chefStat = REST;
    saveState(nFic, &(sh->fSt));


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

    /* insert your code here */
    if (semUp(semgid, sh->waiterRequest) == -1) {
        perror("error on the up operation for semaphore access (PT)");
        exit(EXIT_FAILURE);
    }


}
```

Resultados

Durante a implementação do código foram feitos testes, tanto com a nossa implementação, como com o código pré-compilado fornecido pelo Professor, assim conseguimos perceber se os resultados que nós obtivemos estavam dentro do esperado.

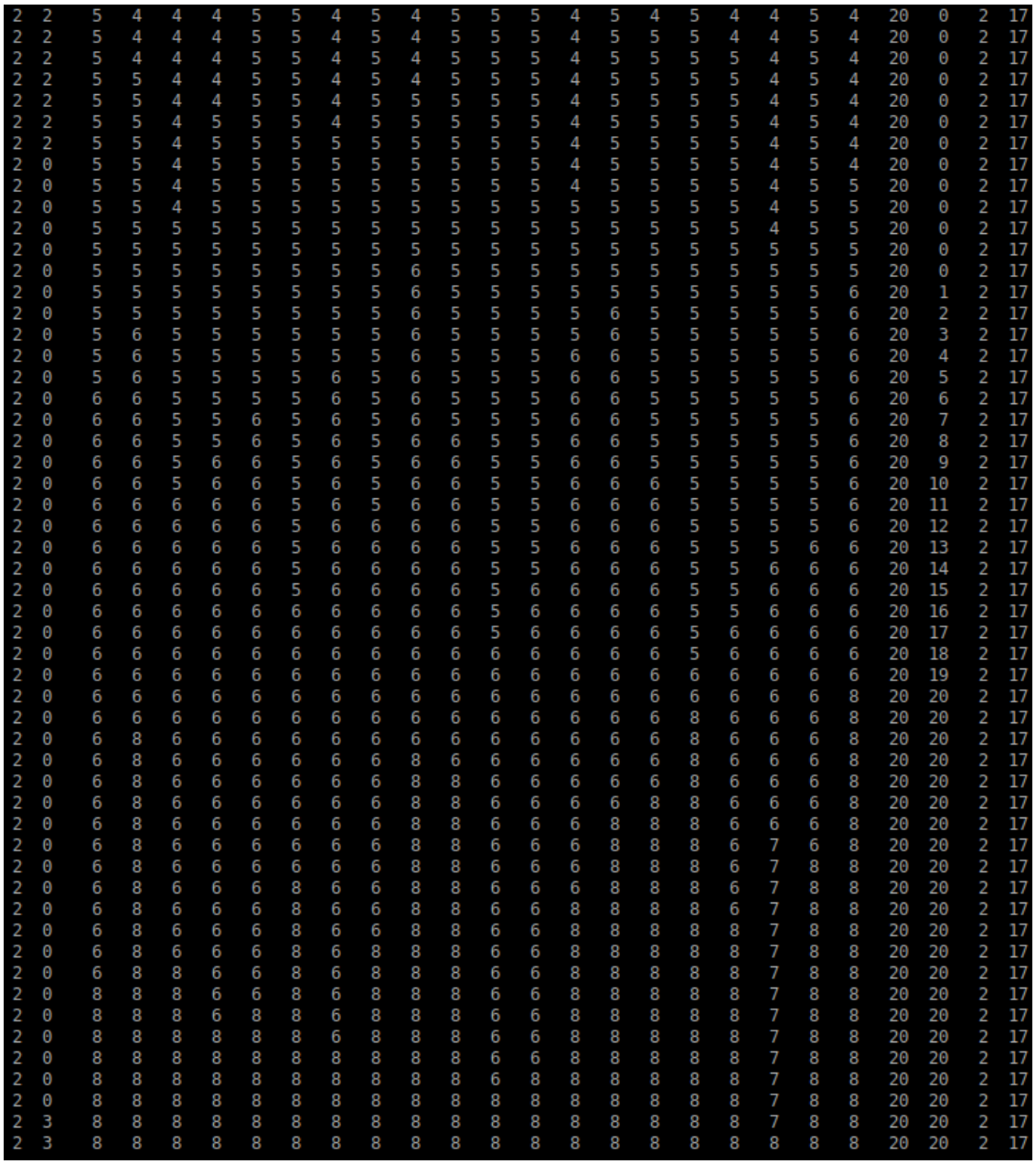
Corremos o programa várias vezes, nunca apresentou qualquer problema ou falha, os resultados foram sempre satisfatórios.

Aqui temos o Run nº 10 como exemplo. O resto dos resultados estão disponíveis para consulta numa pasta junto com o código.

Run n.º 10

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	1	0	0	0	-1
0	0	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	2	-1
0	0	1	1	2	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	2	0	2	-1
0	0	1	1	2	1	1	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	3	0	2	-1
0	0	1	1	2	1	2	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	4	0	2	-1
0	0	1	1	2	1	2	1	1	1	1	2	1	2	1	1	1	2	1	1	1	1	5	0	2	-1
0	0	1	1	2	1	2	1	1	1	1	2	1	2	1	1	1	2	1	1	2	1	6	0	2	-1
0	0	2	1	2	1	2	1	1	1	1	2	1	2	1	1	1	2	1	1	2	1	7	0	2	-1
0	0	2	1	2	1	2	1	1	2	1	2	1	2	1	1	1	2	1	1	2	1	8	0	2	-1
0	0	2	1	2	1	2	1	1	2	1	2	1	2	1	1	2	2	1	1	2	1	9	0	2	-1
0	0	2	1	2	1	2	1	1	2	1	2	2	2	1	1	2	2	1	1	2	1	10	0	2	-1
0	0	2	1	2	1	2	1	2	2	1	2	2	2	1	1	2	2	1	1	2	1	11	0	2	-1
0	0	2	1	2	1	2	1	2	2	1	2	2	2	1	2	2	2	1	1	2	1	12	0	2	-1
0	0	2	1	2	1	2	1	2	2	1	2	2	2	1	2	2	2	2	1	2	1	13	0	2	-1
0	0	2	1	2	1	2	2	2	2	1	2	2	2	1	2	2	2	2	1	2	1	14	0	2	-1
0	0	2	1	2	1	2	2	2	2	1	2	2	2	2	2	2	2	2	1	2	1	15	0	2	-1
0	0	2	1	2	1	2	2	2	2	1	2	2	2	2	2	2	2	2	1	2	2	16	0	2	-1
0	0	2	2	2	1	2	2	2	2	1	2	2	2	2	2	2	2	2	1	2	2	17	0	2	-1
0	0	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	18	0	2	-1
0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	19	0	2	-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	2	17
0	0	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	2	2	2	2	20	0	2	17
0	0	4	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	2	2	2	2	4	2	2	2	2	2	2	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	2	2	2	2	4	2	2	2	2	2	4	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	2	2	2	2	4	2	4	2	2	2	4	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	2	2	2	2	4	2	4	2	2	4	4	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	2	2	4	2	4	2	4	2	2	4	4	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	4	2	4	2	4	2	4	2	2	4	4	2	2	2	2	20	0	2	17
0	0	4	2	2	2	4	4	2	4	2	4	2	4	2	2	4	4	2	2	4	2	20	0	2	17
0	0	4	2	2	2	4	4	2	4	2	4	4	4	2	2	4	4	2	2	4	2	20	0	2	17
0	0	4	2	2	2	4	4	4	4	2	4	4	4	2	2	4	4	2	2	4	2	20	0	2	17
0	0	4	2	2	2	4	4	4	4	2	4	4	4	2	4	4	4	2	2	4	2	20	0	2	17
0	0	4	4	2	2	4	4	4	4	2	4	4	4	2	4	4	4	2	2	4	2	20	0	2	17
0	0	4	4	3	2	4	4	4	4	2	4	4	4	2	4	4	4	2	2	4	2	20	0	2	17
0	0	4	4	3	2	4	4	4	4	4	4	4	4	2	4	4	4	2	2	4	2	20	0	2	17
0	0	4	4	3	2	4	4	4	4	4	4	4	4	2	4	4	4	4	2	4	2	20	0	2	17
0	0	4	4	3	4	4	4	4	4	4	4	4	4	2	4	4	4	4	2	4	2	20	0	2	17
0	0	4	4	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	4	2	20	0	2	17
0	0	4	4	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	4	4	20	0	2	17
0	0	4	4	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
0	1	4	4	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
0	0	4	4	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
1	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
2	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
2	2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
2	2	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	2	17
2	2	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	5	4	4	4	4	20	0	2	17
2	2	5	4	4	4	4	4	4	4	4	4	4	4	4	4	5	4	5	4	4	4	20	0	2	17
2	2	5	4	4	4	4	4	4	5	4	4	4	4	4	4	5	4	5	4	4	4	20	0	2	17
2	2	5	4	4	4	4	4	4	5	4	5	4	4	4	5	4	5	4	4	4	4	20	0	2	17
2	2	5	4	4	4	4	4	4	5	4	5	4	4	4	5	4	5	4	4	5	4	20	0	2	17
2	2	5	4	4	4	5	4	4	5	4	5	4	4	4	5	4	5	4	4	5	4	20	0	2	17
2	2	5	4	4	4	5	5	4	5	4	5	4	4	4	5	4	5	4	4	5	4	20	0	2	17
2	2	5	4	4	4	5	5	4	5	4	5	4	5	4	5	4	5	4	4	5	4	20	0	2	17



Rodando vezes suficientes é possível observar que não existem *deadlocks* neste programa pois caso contrário o mesmo travaria a sua execução a meio.

Apenas fizemos 10 execuções porque achamos um número razoável, mas mesmo para um maior número não apresenta qualquer problema.

De seguida temos outra execução, agora aplicando o filtro, e assim como anteriormente está tudo dentro do expectável.

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
.	.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	.	0	0	-1
.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	2	0	2	-1
.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3	0	2	-1
.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	5	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	6	0	2	-1
.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	7	0	2	-1
.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	8	0	2	-1
.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	9	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	10	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	11	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	12	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	13	0	2	-1
.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	14	0	2	-1
.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	15	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	16	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	17	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	18	0	2	-1
.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	19	0	2	-1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	20	0	2	13
.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	0	2	13
.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	0	2	13
.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	0	2	13
.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	0	2	13
.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4	.	.	.	.	.	.	0	2	13
.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13

,

2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	0	2	13
.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	0	2	13
.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	0	2	13
.	0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	0	2	13
.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	6	.	1	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	2	13
.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	4	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	5	2	13
.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	7	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	8	2	13
.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	9	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	10	2	13
.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	11	2	13
.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	12	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	13	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	14	2	13
.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	15	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	16	2	13
.	.	.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	17	2	13
.	.	.	.	.	.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	18	2	13
.	.	.	.	.	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	19	2	13
.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	7	.	.	.	.	.	.	.	20	2	13
.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	20	2	13
.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	20	2	13
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	8	.	.	.	.	.	.	.	20	2	13

Conclusão

Este trabalho foi extremamente importante para percebermos a utilidade e aplicabilidade da utilização de semáforos e da memória partilhada na sincronização de diversos processos, uma vez que sem este recurso o programa ficaria super condicionado e sujeito a falhas.

A maior dificuldade passou por se ter sempre em mente a visão geral do programa ("*big picture*") pois o mesmo usa muitos processos em paralelo o que exige extrema coordenação entre todas as partes para o seu correto funcionamento.

Dito isto, obtemos os resultados esperados, atingindo assim todos os objetivos propostos e estamos confiantes no seu bom desempenho.