deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Jantar de Amigos

Trabalho realizado por:

André Almeida Oliveira, nº107637

Duarte Carvalho da Cruz, nº107359

Sistemas Operativos

Prof. José Nuno Panelas Nunes Lau

Ano Letivo 2022/2023

ÍNDICE

INTRODUÇÃO

Na sequência do trabalho proposto, este relatório tem como objetivo a explicação dos raciocínios utilizados para formular o código necessário para a correta sincronização dos processos e *threads* fornecidas no programa inicial. O tema do trabalho consiste num jantar de amigos onde coexistem três entidades (uma delas pode subdividir-se em três) que correspondem a processos independentes:

- ***Client* ->** Clientes do restaurante que chegam em tempos aleatórios em que o primeiro cliente a chegar e o último têm tarefas "especiais". Este primeiro e último cliente, apesar de fazerem parte do mesmo processo mas de *threads* diferentes, iremos explicá-los separado dos clientes "normais";

- ***Waiter* ->** Recebe o pedido da comida por parte do primeiro *client* a chegar ao restaurante e, de seguida, informa o *chef* sobre o mesmo. Após o *chef* sinalizar que a comida está pronta, o *waiter* leva-a aos clientes. Por fim, quando todos os clientes acabarem de comer, o último *client* que chegou ao restaurante sinaliza o *waiter* para efetuar o pagamento;

- ***Chef* ->** Recebe o pedido de confeção por parte do *waiter* e sinaliza o mesmo quando este estiver pronto, ficando, por fim, a descansar.

O código apresentado no seguimento deste relatório, serão apenas as funções que têm código sinalizado para completar, sendo necessários os restantes ficheiros para o funcionamento total do programa.

COMPORTAMENTO DOS SEMÁFOROS

Semáforo	UP			DOWN		
	Quem?	Quando?	Quantos?	Quem?	Quando?	Quantos?
Mutex	*Client*	*waitFriends*	1	*Client*	*waitFriends*	1
		waitFood	2		*waitFood*	2
		waitAndPay	2		*waitAndPay*	2
	firstClient	*waitFriends*	1	*firstClient*	*waitFriends*	1
		orderFood	1		*orderFood*	1
		waitFood	2		*waitFood*	2
		waitAndPay	2		*waitAndPay*	2
	lastClient	*waitFriends*	1	*lastClient*	*waitFriends*	1
		waitFood	2		*waitFood*	2
		waitAndPay	3		*waitAndPay*	3
	Waiter	*waitForClientOrChef*	6	*Waiter*	*waitForClientOrChef*	6
		informChef	1		*informChef*	1
		takeFoodToTable	1		*takeFoodToTable*	1
		receivePayment	1		*receivePayment*	1
	Chef	*waitForOrder*	1	*Chef*	*waitForOrder*	1
		processOrder	1		*processOrder*	1
friendsArrived	*lastClient*	*waitFriends*	TABLESIZE	*Client*	*waitFriends*	1
				firstClient		1
				lastClient		1
requestReceived	*Waiter*	*informChef*	1	*firstClient*	*orderFood*	1
		receivePayment	1	*lastClient*	*waitAndPay*	1
foodArrived	*Waiter*	*takeFoodToTable*	TABLESIZE	*Client*	*waitFood*	1
				firstClient		1
				lastClient		1
allFinished	*lastClient*	*waitAndPay*	TABLESIZE	*Client*	*waitAndPay*	1
				firstClient		1
				lastClient		1
waiterRequest	*firstClient*	*orderFood*	1	*Waiter*	*waitForCLientOrChef*	3
	lastClient	*waitAndPay*	1			
	Chef	*processOrder*	1			
waitOrder	*Waiter*	*informChef*	1	*Chef*	*waitForOrder*	1

CHEF

O código fornecido é uma implementação para o processo do chefe de cozinha, simulando um restaurante. Este mesmo é responsável por preparar os pedidos de comida que são feitos por um processo de *Waiter*.

FUNÇÃO *WAITFORORDER()*

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

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

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

    /* fim */

    if (semDown (semgid, sh->mutex) == -1) {                          /* enter critical region */
        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);

    /* fim */

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

Figura 1 | Função waitForOrder()

Esta função é usada com o intuito do *Chef* esperar que seja efetuado um pedido de confeção pelo *Waiter*.

Inicialmente, o estado do *Chef* é atualizado para *WAIT_FOR_ORDER* e, consequentemente, é salvo. Posteriormente, decrementamos o semáforo *waitOrder* através de um *semDown()* para que o *Chef*, que estava à espera de um pedido de confeção, receba e continue a execução.

De seguida, decrementamos o semáforo *mutex* que permite entrar na região crítica do programa, atualizando o estado do *Chef* para *COOK* e colocando a *flag foodOrder* a 0, pois este acaba de receber o pedido. No fim deste processo, é guardado o seu estado e variáveis usadas e incrementado o semáforo *mutex* através de um *semUp()* para sair da região crítica.

FUNÇÃO *PROCESSORDER()*

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

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

    /* insert your code here */

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

    /* fim */

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

    /* insert your code here */

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

    /* fim */
}
```

Figura 2 | Função processOrder()

Esta função é usada com o intuito do *Chef* processar o pedido de confeção.

Inicialmente, é feita uma simulação da quantidade de tempo que a comida demora a ser confecionada. Posteriormente, entrando na região crítica do programa, procedemos à atualização do estado do *Chef* para *REST* e colocamos a *flag foodReady* a 1 indicando ao *Waiter* que a comida já está pronta, salvando o seu estado e variáveis usadas.

Por fim, saímos da região crítica do programa e, através de um *semUp()* do semáforo *waiterRequest(),* o *Waiter* é sinalizado da conclusão da confeção do pedido para o recolher.

Termina assim o ciclo de vida do *Chef.*

CLIENT

O código fornecido é uma implementação para o processo do *Client*, simulando um restaurante. Este mesmo é responsável pelo pedido da comida ao *Waiter*, pelo jantar em si e pelo pagamento do mesmo ao *Waiter.*

FUNÇÃO *WAITFRIENDS()*

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

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

    /* insert your code here */

    sh->fSt.tableClients++;

    if (sh->fSt.tableClients == TABLESIZE) {
        sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
        sh->fSt.tableLast = id;
        saveState(nFic, &sh->fSt);

        for (int i = 0; i < TABLESIZE; i++) {
            if (semUp (semgid, sh->friendsArrived) == -1) {
                perror ("error on the up operation for semaphore access (CT)");
                exit (EXIT_FAILURE);
            }
        }
    }
    else
    {
        sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;

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

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

    /* insert your code here */

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

    /* dim */

    return first;
}
```

Figura 3 | Função waitFriends()

Cada cliente tem um ID diferente, por isso, na primeira chamada da função, vai ser guardado o ID do primeiro *Client* a chegar ao restaurante (*firstClient*). Este, fica responsável por fazer o pedido de comida ao *Waiter* quando os outros chegarem.

A medida que os restantes chegam, é incrementado o número de clientes na mesa através da variável *tableClients* que servirá como termo de comparação no *for loop* seguinte para verificar se todos os amigos já se encontram na mesa. Se o mesmo se verificar, iremos mudar o estado do último a chegar para *WAIT_FOR_FOOD* e guardaremos o seu *ID* na variável *tableLast* pois, será responsável pelo pagamento da refeição. Iremos ainda percorrer outro *for loop*, com um número de interações igual ao número de amigos que se encontram sentados à mesa, de modo a incrementar o semáforo *friendsArrived* o número de vezes correspondente à quantidade de clientes na mesa (através de um *semUp),* que fará os amigos esperarem uns pelos outros para proceder ao pedido da refeição. Caso o número de clientes na mesa não corresponda ao esperado, à medida que vão chegando iremos mudando os respetivos estados para *WAIT_FOR_FRIENDS*.

Este processo descrito acima foi efetuado na região crítica do programa e as mudanças de estado das diferentes entidades e das variáveis usadas foram salvas simultaneamente.

Por fim, cada *Client* irá tentar decrementar o semáforo *friendsArrived*, o que apenas será possível quando o último chegar, como já explicado.

FUNÇÃO *ORDERFOOD()*

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

    /* insert your code here */

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

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

    /* fim */

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

    /* fim */
}
```

Figura 4 | Função orderFood()

Esta função, apenas será usada pelo primeiro *Client* que chegou pois, será aqui onde será efetuado o pedido da comida ao *Waiter*.

Primeiramente, entraremos na região crítica do programa e, de seguida, atualizaremos o estado do *Client* em questão para *FOOD_REQUEST* e, também utilizamos a *flag foodRequest* incrementada a 1 para indicar que o pedido a efetuar é referente à comida e não ao pagamento. Antes da saída da região crítica, são atualizados os respetivos estados e variáveis e através de um *semUp* do semáforo *waiterRequest,* sinalizaremos o *Waiter* que o *Client* pretende ser atendido.

Por fim, sairemos da região crítica e o *Client* ficará à espera de que o *Waiter* receba o pedido e entregue ao *Chef* através do *semDown* do semáforo *requestReceived.*

FUNÇÃO *WAITFOOD()*

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

    if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
        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);

    /* fim */

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

    /* insert your code here */

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

    /* fim */

    if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
        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);

    /* fim */

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

Figura 5 | Função waitFood()

Esta função, é responsável por fazer os clientes esperarem até que a comida lhes seja entregue e possam começar a comer.

Num primeiro passo, entraremos na região crítica do programa para atualizar o estado de cada cliente para *WAIT_FOR_FOOD* e, de seguida, salvando-o.

Já fora da região crítica, teremos o *semDown* do semáforo *foodArrived* responsável pela espera dos clientes começarem a comer até que a comida chegue.

A partir do momento em que a comida chega, entraremos novamente na região crítica do programa e atualizamos os estados de cada cliente para *EAT*, guardando-os.

FUNÇÃO *WAITANDPAY()*

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

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

    /* insert your code here */

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

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

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

    /* fim */

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

    /* insert your code here */

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

    /* fim */

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

        /* insert your code here */

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

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

        /* fim */

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

        /* insert your code here */

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

        /* fim */
    }

    if (semDown (semgid, sh->mutex) == -1) {                                        /* enter critical region */
        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));

    /* fim */

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

Figura 6 | Função waitAndPay()

Esta função, é responsável pela espera dos clientes uns pelos outros até que todos acabem de comer para que último *Client* que chegou ao restaurante pague a refeição.

Primeiramente, entraremos na região crítica do programa e, a cada cliente que passe nesta região, incrementaremos o número de clientes que acabaram de comer através da variável *tableFinishEat* atualizando o seu estado para *WAIT_FOR_OTHERS.* Como expectável, iremos guardar as mudanças de estado e a variável usada. Ainda na mesma região, verificaremos se o *Client* em questão é o *lastClient* que fará o pagamento da refeição. Quando todos tiverem acabado de comer, através de um *for loop* incrementaremos o semáforo *allFinished* o número de vezes correspondente à quantidade de clientes na mesa. Por fim, procedemos á saída da região crítica do programa.

Após o processo anterior, cada *Client* fará *semDown* do semáforo *allFinished* que apenas será possível quando todos acabarem de comer, como já explicado.

De seguida, se o *Client* em questão for o *lastClient*, entrará na região crítica do programa e tanto incrementa a *flag paymentRequest* para indicar ao *Waiter* que é um pedido de pagamento, como atualiza o seu estado para *WAIT_FOR_BILL*. Antes de sair da região crítica, incrementaremos o semáforo *waiterRequest* para o *Waiter* atender ao seu pedido. Sairemos assim da região crítica da função, guardando o respetivo estado e variável usada e, através do *semDown* do semáforo *requestReceived, o Client* em questão esperará que o *Waiter* receba o pagamento.

Por fim, todos os clientes entrarão na região crítica do programa e atualizarão o seu estado para *FINISHED*, guardando-o.

Termina assim o ciclo de vida do *Client.*

WAITER

O código fornecido é uma implementação para o processo do *Waiter*, simulando um restaurante. Este mesmo é responsável por receber o pedido de comida por parte do *firstClient*. De seguida, infroma o *Chef* sobre o mesmo e quando a comida estiver pronta, trá-la para a mesa. Por fim, recebe o pagamento da refeição por parte do *lastClient.*

FUNÇÃO WAITFORCLIENTORCHEF()

```
static int waitForClientOrChef()
{
    int ret=0;
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        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);

    /* fim */


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

    /* insert your code here */

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

    /* fim */

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

    /* insert your code here */

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

    /* fim */

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

    return ret;
}
```

Figura 7 | Função waitForClientOrChef()

Anteriormente, já num *loop* infinito, o Waiter espera por uma solicitação de um *Client* ou *Chef* usando a função esta função.

Primeiramente, nesta função, entraremos na região crítica da função para atualizar o estado do *Waiter* para *WAIT_FOR_REQUEST* e, de seguida, guardamo-lo e saímos da região crítica.

Num passo seguinte, faremos *semDown* do semáforo *waiterRequest* para o *Waiter* antender ao pedido do *Chef* ou do *Client* que tenha efetuado *semUp* do mesmo semáforo anteriormente.

Entrando novamente na região crítica do programa, através das *flags* sinalizadas previamente pelas entidades já abordadas, o *Waiter* atenderá a diferentes tipos de solicitações:

- Se a solicitação for um pedido de comida, o programa informa o cozinheiro sobre o pedido usando a função *informChef*;

- Se a solicitação for a indicação de que a comida está pronta, o *Waiter* leva a comida até a mesa do dos clientes usando a função *takeFoodToTable*;

- Se a solicitação for o pedido da conta, o programa recebe o pagamento do cliente usando a função *receivePayment*.

Dependendo do tipo de solicitação, a varíavel *ret* irá tomar diferentes valores e no *loop* infinito referido inicialmente, o *Waiter* será encaminhado para diferentes funções. Sairemos assim da região crítica do programa após salvar as respetivas variáveis.

FUNÇÃO *INFORMCHEF()*

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

    /* insert your code here */

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

    /* fim */

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

    /* fim */
}
```

Figura 8 | Função informChef()

Nesta função, o *Waiter* fica responsável de informar o *Chef* à cerca do pedido efetuado pelo *firstClient.*

Inicialmente, entramos na região crítica da função mudando o estado do *Waiter* para *INFORM_CHEF* e ao mesmo tempo incrementamos a *flag foodOrder* para indicar ao *Chef* que tem um pedido para ser preparado. De seguida, são guardados os respetivos estados e variáveis usadas e saímos da região crítica do programa.

Seguidamente, através de um *semUp* do semáforo *requestReceived,* o *Waiter* indica ao *firstClient* que o pedido foi recebido pelo *Chef.*

Por fim, com o *semUp* do semáforo *waitOrder,* o *Chef* é sinalizado que pode começar a confecionar o pedido.

FUNÇÃO *TAKEFOODTOTABLE()*

```c
static void takeFoodToTable ()
{
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        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 < TABLESIZE; i++) {
        if (semUp(semgid, sh->foodArrived) == -1)  {                                    /* enter critical region */
            perror ("error on the up operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
    }

    /* fim */

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

Figura 9 | Função takeFoodToTable()

Com esta função, o *Waiter* fica responsável de levar a comida pronta pelo *Chef* aos clientes.

Primeiro, entramos na região crítica da função para alterar o estado do *Waiter* para *TAKE_TO_TABLE*, guardando-o. Logo de seguida, através de um *for loop* o semáforo *foodArrived* sofre *semUp* correspondente ao número de clientes da mesa, pois todos precisam de indicação que a comida chegou á mesa e que podem começar a comer.

Por fim, saímos apenas da região crítica do programa.

FUNÇÃO *RECEIVEPAYMENT()*

```c
static void receivePayment ()
{
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        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);

    /* fim */

    if (semUp (semgid, sh->mutex) == -1)  {                                    /* exit critical region */
     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);
    }
}
```

Figura 10 | Função receivePayment()

Com esta função, o *Waiter* fica responsável de recolher o pagamento por parte do *firstClient.*

Primeiramente, entramos na região crítica da função apenas para alterar o estado do *Waiter* para *RECEIVE_PAYMENT,* guardando-a e voltando a sair da região logo de seguida.

Por último, é efetuado um *semUp* do semáforo *requestReceived* indicando ao *lastClient* que o pagamento foi recebido.

Termina assim o ciclo de vida do *Waiter.*

Validação de Resultados

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

Figura 11 | Exemplo de execução do programa

A figura anterior é o resultado de um teste realizado ao restaurante implementado pelas funções apresentadas anteriormente. *CH* e *WT* correspondem aos estados do *Chef* e do *Waiter*, respetivamente.

Os diferentes *C*'s correspondem ao tanto de amigos que foram ao restaurante. O *ATT* corresponde ao número de amigos que chegaram ao restaurante, o *FIE* corresponde ao número de amigos que acabaram de comer, *1st* corresponde ao primeiro a chegar ao restaurante e *las* ao último.

Após o procedimento a alguns testes do código, de soluções aleatórias, verificámos que este foi um sucesso.

```
/* 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

/* 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
```

```
/* 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

/* 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

#endif /* PROBCONST_H_ */
```

Figura 12 | Constantes dos diferentes estados

Sendo que cada número do lado direito corresponde a um estado específico da entidade comentada em cabeçalho podemos observar no resultado obtido que todas as entidades são atualizadas para os estados seguintes na altura correta, mediante aquilo que foi descrito nas funções.

CONCLUSÃO

Este trabalho possibilitou-nos desenvolver conhecimentos sobre os mecanismos associados à execução e sincronização de processos e *threads*. Posto isto, o trabalho foi desenvolvido de forma adequada e de acordo com todos os requisitos propostos, visto que os objetivos propostos foram alcançados.

De um modo geral, a maior dificuldade foi entender a cronologia das funções dadas para os ciclos de vida das diferentes entidades, visto que todo o código teria de ter uma estrutura e ordem exigente para o seu correto funcionamento.

Após múltiplos testes, foi-se aperfeiçoando erros existentes e por fim obtivemos um código bem formulado.