

**Sistemas Operativos**

**Trabalho Prático 2**

**Jantar de Amigos (Restaurant)**

**Professor:**

Nuno Lau ([*nunolau@ua.pt*](mailto:nunolau@ua.pt))

**Realizado por:**

Diogo Falcão, 108712, P3

José Gameiro, 108840, P3

02/01/2023

Índice

[***1.*** ***Introdução*** 3](#_Toc123598390)

[***2.*** ***Material Fornecido*** 4](#_Toc123598391)

[***3.*** ***Desenvolvimento*** 8](#_Toc123598392)

[**3.1.** **Client** 8](#_Toc123598393)

[3.1.1. waitFriends () 8](#_Toc123598394)

[3.1.2. orderFood () 10](#_Toc123598395)

[3.1.3. waitFood () 11](#_Toc123598396)

[3.1.4. waitAndPay () 12](#_Toc123598397)

[**3.2.** **Waiter** 14](#_Toc123598398)

[3.2.1. waitForClientOrChef () 15](#_Toc123598399)

[3.2.2. informChef () 16](#_Toc123598400)

[3.2.3. takeFoodToTable () 16](#_Toc123598401)

[3.2.4. receivePayment () 17](#_Toc123598402)

[**3.3.** **Chef** 18](#_Toc123598403)

[3.3.1. waitForOrder () 18](#_Toc123598404)

[3.3.2. processOrder () 18](#_Toc123598405)

[***4.*** ***Resultados*** 20](#_Toc123598406)

[***5.*** ***Conclusão*** 23](#_Toc123598407)

# ***Introdução***

No âmbito da Unidade Curricular de Sistemas Operativos, foi-nos proposto realizar um trabalho prático, que consiste em simular um jantar de amigos num restaurante envolvendo três entidades: clients (clientes), waiter (empregado/a) e chef (chefe). Todas estas entidades são processos independentes, sendo que a sua sincronização e comunicação é efetuada através de vários semáforos e de memória partilhada.

A simulação começa com a chegada de todos os amigos ao restaurante. Nota-se que o primeiro amigo a chegar será o que irá fazer o pedido da comida, no entanto, só o poderá fazer quando todos os amigos tiverem chegado. O último amigo a chegar será o que irá pagar a conta e só poderá pedi-la quando todos os amigos tiverem terminado a sua refeição. De modo a evitar situações em que dois ou mais processos sejam bloqueados (visto que estão à espera de um evento que apenas pode ser despoletado por um dos processos em bloqueio, ou seja, uma situação de deadlock), usamos semáforos.

A utilização de semáforos serve essencialmente o controlo de acesso à memória partilhada, de modo a evitar choques entre as três entidades que participam na simulação. As notificações entre entidades são feitas através desses mesmos semáforos para que o programa execute sem problemas.

Com a realização deste trabalho prático, esperamos conseguir cumprir todos os pontos essenciais que são propostos no guião e alargar os nossos conhecimentos relativamente a programar com semáforos em C, visto ser um aspeto importante no que toca a controlar o acesso a determinadas regiões por parte de vários processos.

# ***Material Fornecido***

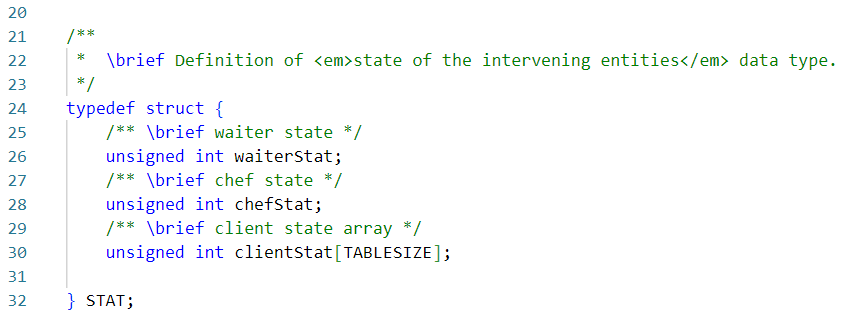
É disponibilizado um conjunto de ficheiros para a resolução do problema, em que está presente uma pasta src com ficheiros incompletos com código em C para a simulação do jantar (semSharedMemChef.c, semSharedMemClient.c e semSharedMemWaiter.c). Ao longo deste relatório iremos explicar como é que completámos as funções que se encontravam incompletas em cada um destes ficheiros.

Existem também outros ficheiros que contém dados fulcrais à simulação do jantar:

- probConst.h: este ficheiro contém variáveis que irão ser utilizadas nos 3 ficheiros incompletos. É caso de: estados dos client, estados do waiter, estados do chef e outras como o tamanho máximo da mesa e os tempos máximos para comer ou cozinhar.

Fig.1 – Constantes definidas no ficheiro probConst.h.

- probDataStruct.h: neste ficheiro estão definidas duas estruturas que são a estrutura STAT e a FULL\_STAT. A estrutura STAT tem como parâmetros 3 variáveis inteiras, que são os estados de cada entidade envolvente no jantar - e como existem 20 clientes, a variável para o estado de cada cliente será um array com o tamanho de 20 (TABLESIZE). A estrutura FULL\_STAT apresenta três tipos de parâmetros (que podemos observar na figura dois): nas linhas 43 e 45 funcionam como contadores para que sempre que um cliente chegue ao restaurante ou sempre que um cliente acaba de comer, estas variáveis incrementam; as variáveis nas linhas 48, 50, 52 e 54 irão funcionar com sinais, ou seja, o seu valor inicial é zero e quando for necessário estas variáveis serão postas a um; por fim, as variáveis tableFirst e tableLast irão guardar os ID’s do primeiro e último cliente a chegar, respetivamente.



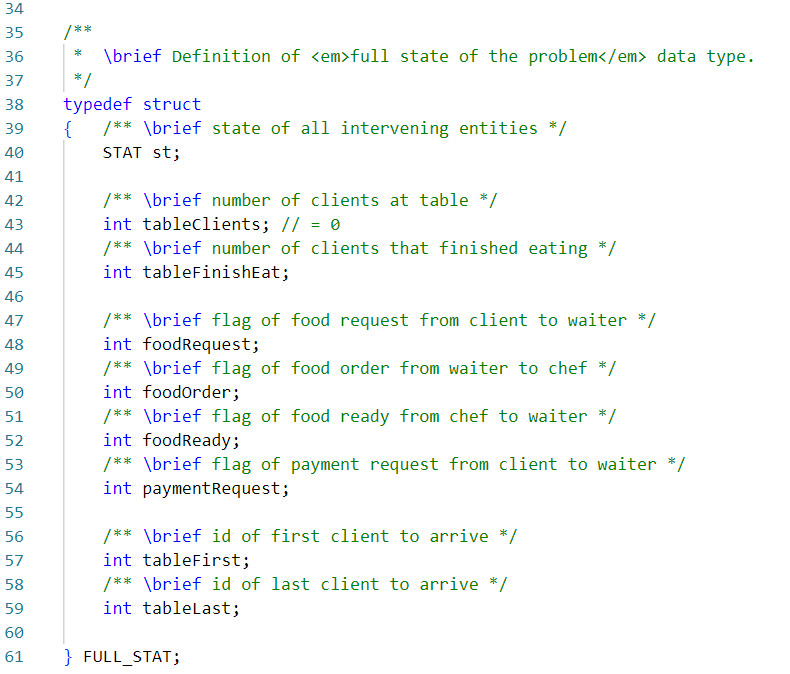


Fig.2 –Estruturas definidas no ficheiro probDataStruct.h.

- sharedDataSync.h: este ficheiro compreende a estrutura SHARED\_DATA, que inclui como parâmetros uma estrutura do tipo FULL\_STAT (fSt) e os semáforos que irão ser utilizados para evitar situações de deadlock. Estes são:

* mutex (valor inicial um) – este semáforo é usado para identificar a entrada na região crítica e a saída da mesma. Uma região crítica é uma zona de código que manipula dados partilhados e que não pode ser executada concorrentemente por mais do que um processo;
* friendsArrived (valor inicial zero) – identifica o semáforo usado pelos clientes para esperarem que os amigos cheguem;
* requestReceived (valor inicial zero) – este semáforo é usado pelos clientes para esperar pelo empregado depois de este ter feito um pedido;
* foodArrived (valor inicial zero) – identifica o semáforo usado pelos clientes para esperarem que a comida chegue;
* allFinished (valor inicial zero) – este semáforo é usado pelos clientes para esperarem que cada um termine a sua refeição;
* waiterRequest (valor inicial zero) – identifica o semáforo usado pelo empregado para esperar por um pedido vindo ou de um cliente ou do chefe;
* waitOrder (valor inicial zero) – este semáforo é usado pelo chefe para esperar por um pedido.

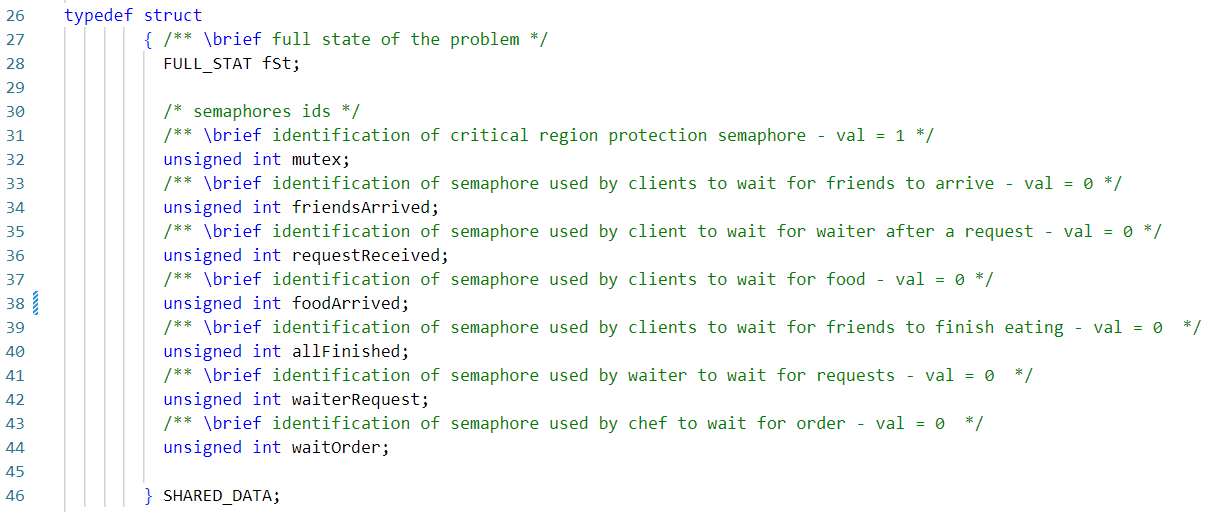
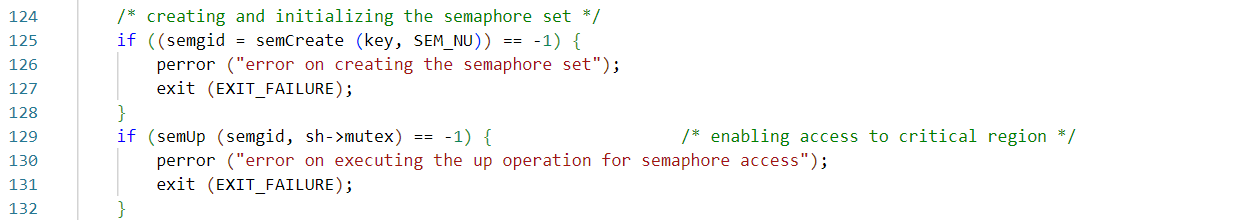
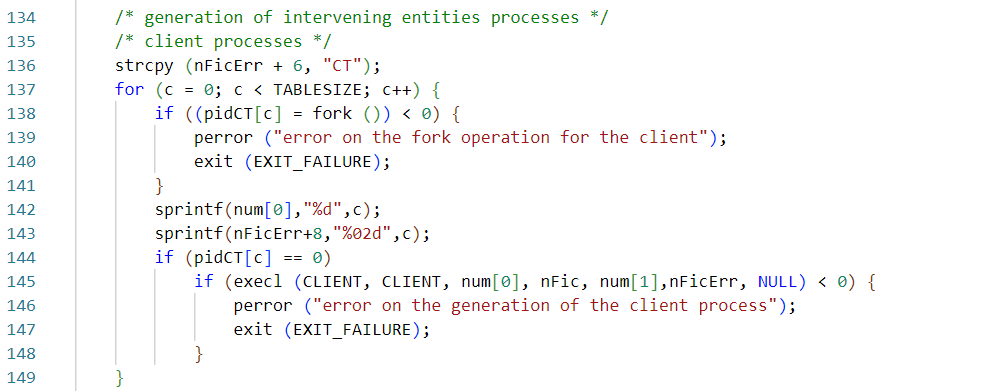
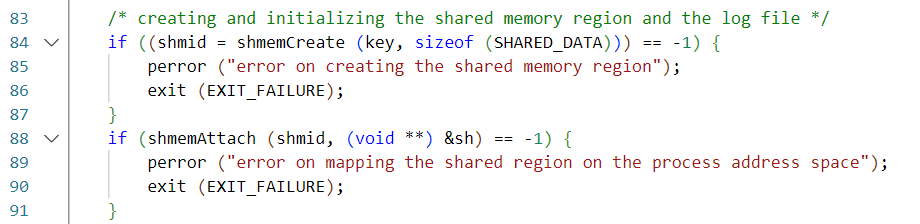
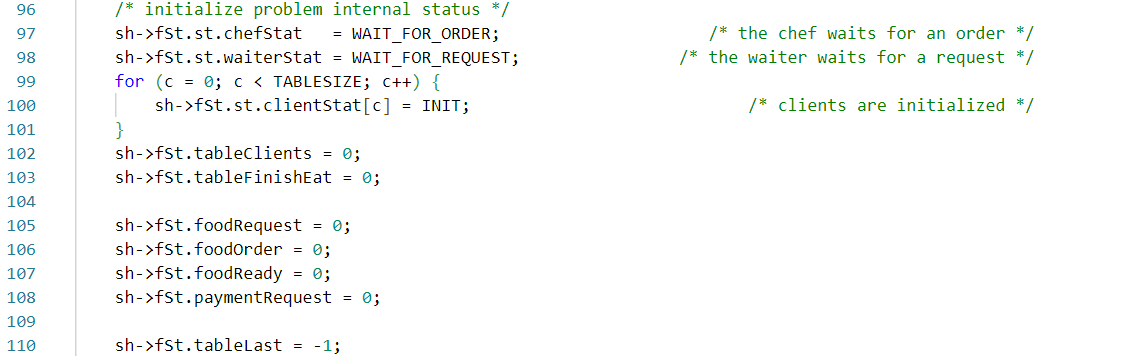


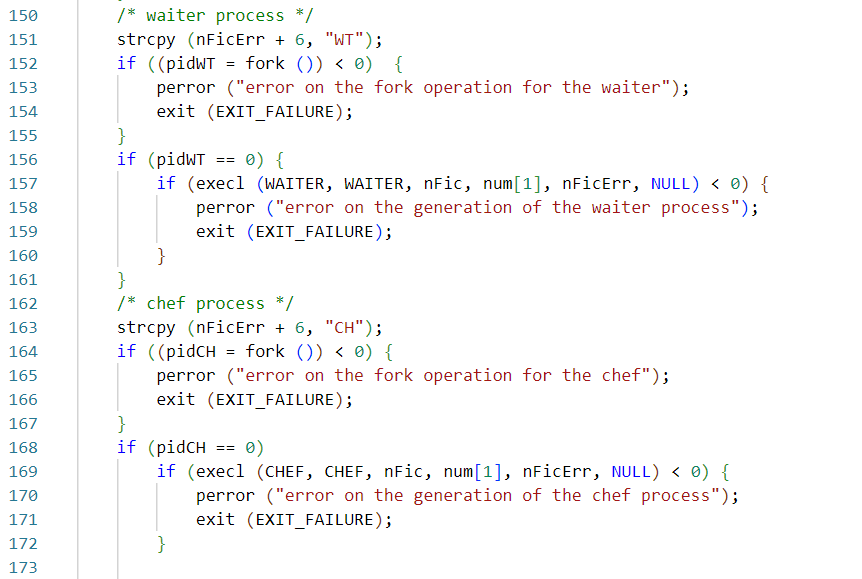
Fig.3 –Estrutura definida no ficheiro sharedDataSync.h.

- probSemSharedMemRestaurant.c: Neste ficheiro é criada a memória partilhada, são inicializados os estados de cada uma das entidades que participam na simulação, as variáveis da estrutura FULL\_STAT e os semáforos, são gerados também os processos clientes, empregado e chefe. Existem também outras operações que são realizadas neste ficheiro que irão ser importantes para a simulação. No fim de ser executado tudo, todos os semáforos e a memória partilhada são destruídos.









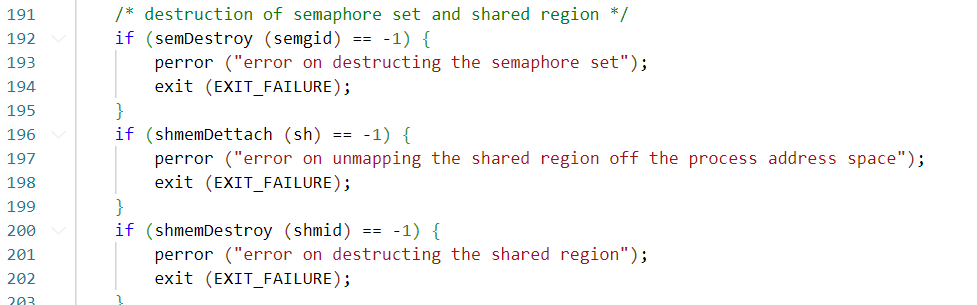


Fig.4 – Algum código presente no ficheiro probSemSharedMemRestaurant.c

# ***Desenvolvimento***

Nesta parte do relatório iremos explicar o código que implementámos nos ficheiros semSharedMemClient.c, semSharedMemWaiter.c e semSharedMemChef.c, as ideias que tivemos para evitar situações de deadlock e mostrar também os ciclos de vida de cada uma das entidades.

## **Client**

Começamos pelo client visto que a primeira ação que ocorre na simulação é a chegada de todos os clientes ao restaurante e o ciclo de vida do cliente é o maior de entre as 3 entidades envolventes na simulação, como podemos observar na figura 5.

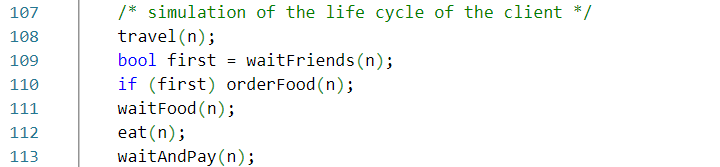


Fig.5 – Ciclo de vida da entidade client.

Ao analisar a figura 5 podemos concluir que a primeira função que o cliente executa é travel (), que consiste em fazer o client viajar de um determinado lugar até ao restaurante, esta função é a única, de entre todas, que se encontra completa.

A segunda função, waitFriends (), é usada para fazer com que o cliente avance para a próxima função apenas quando todos os clientes chegarem, através de um booleano. Se todos tiverem chegado o client avança para a próxima função, orderFood (), em que irá se efetuado o pedido da comida.

Depois de se ter feito o pedido, os clientes esperam que a comida chegue, através da função waitFood (). Ao chegar a comida, todos começam a comer, com a função eat (). Por último, todos os clientes terão de esperar que todos à mesa terminem para que se possa pedir a conta e acabar o jantar - é o que acontece na função waitAndPay (). Desta forma, termina o ciclo de vida do client.

### waitFriends ()

Tal como foi referido em cima, esta função tem como objetivo fazer com que os clientes esperem que a mesa fique completa, ou seja, que todos os clientes cheguem para depois se puder fazer o pedido da comida. O pedido terá de ser feito pelo primeiro client e, por isso, é necessário guardar o estado de cada um dos clientes pois este irá alterar para todos.

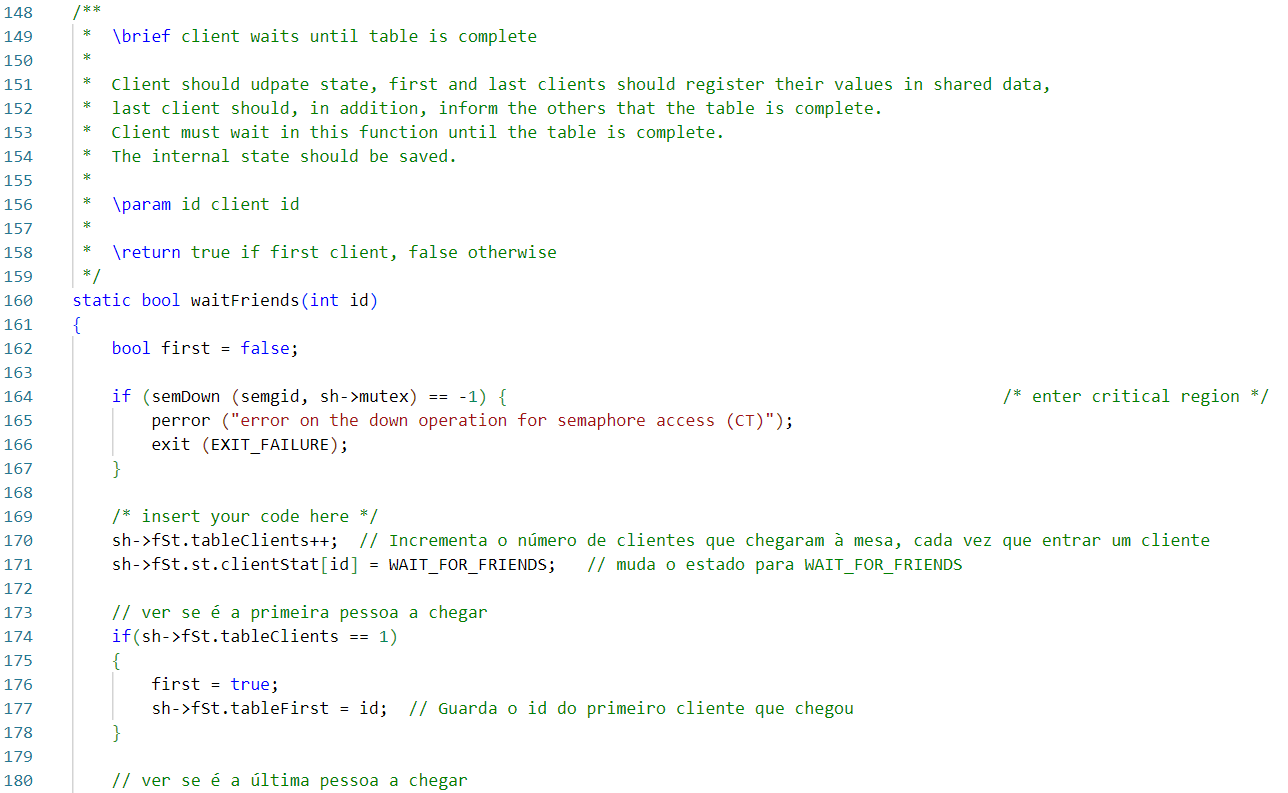


Fig.6a – Primeira parte da função waitFriends.

Em termos de código, esta função irá ser utilizada por todos os clientes e começa com a definição de um booleano designado por “first”. Ao chegar um cliente, este entra na região crítica, é incrementado o número de clientes que se encontram na mesa e o estado deste muda para WAIT\_FOR\_FRIENDS. De seguida, através da condição presente na linha 174, é verificado se o cliente que entrou na região crítica é o primeiro e, se for, a variável first é alterada para true. Por isto, é guardado o id do primeiro cliente que chegou à mesa dado que vai ser este que irá efetuar o pedido da comida.

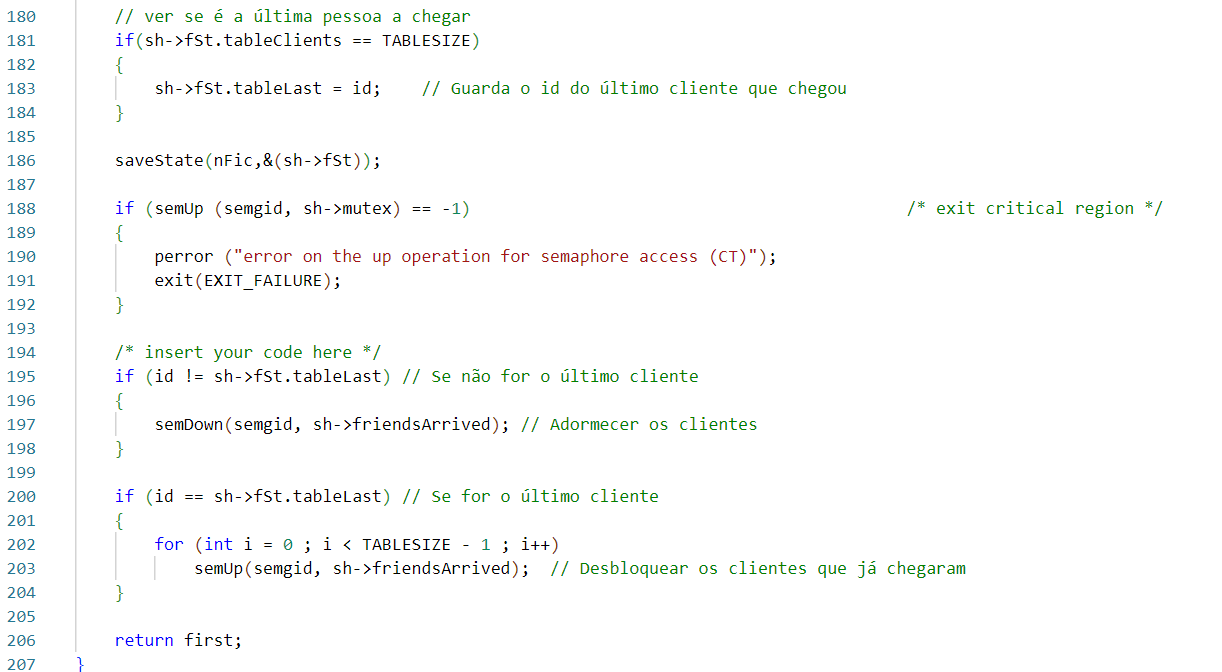


Fig.6b – Segunda parte da função waitFriends.

Se não se verificar a condição da verificação do primeiro cliente, avança-se para a próxima condição, em que verifica se o cliente que está na região crítica é o último cliente, através da linha 181. Se for efetivamente verdade, é guardado o ID do cliente em questão, isto porque o último cliente a chegar irá efetuar o pedido da conta e o pagamento da mesma.

Simultaneamente, o estado de cada cliente e as variáveis que foram alteradas serão guardadas através da função saveState (). Guardado o estado do cliente que está a percorrer a função e saindo-se da região crítica, é verificado, desta vez fora da região crítica, se o seu ID não é último. Se não for, o cliente é adormecido com um semDown efetuado no semáforo friendsArrived. Este não pode avançar para a função seguinte até todos os outros clientes chegarem. Se a condição da linha 195 for falsa avança para a próxima condição, em que se é verificado uma última vez se o ID do cliente é o último. Caso seja, os clientes que foram adormecidos são acordados com um semUp no semáforo friendsArrived (efetuado 19 vezes, isto porque é o número de clientes que já chegaram). Assim esta função termina e retorna como verdadeiro o booleano “first”, permitindo avançar-se para a próxima função.

### orderFood ()

Esta função é utilizada apenas pelo primeiro cliente e tem como objetivo efetuar o pedido da comida. Será novamente necessário guardar o estado do primeiro cliente pois este irá ser alterado.

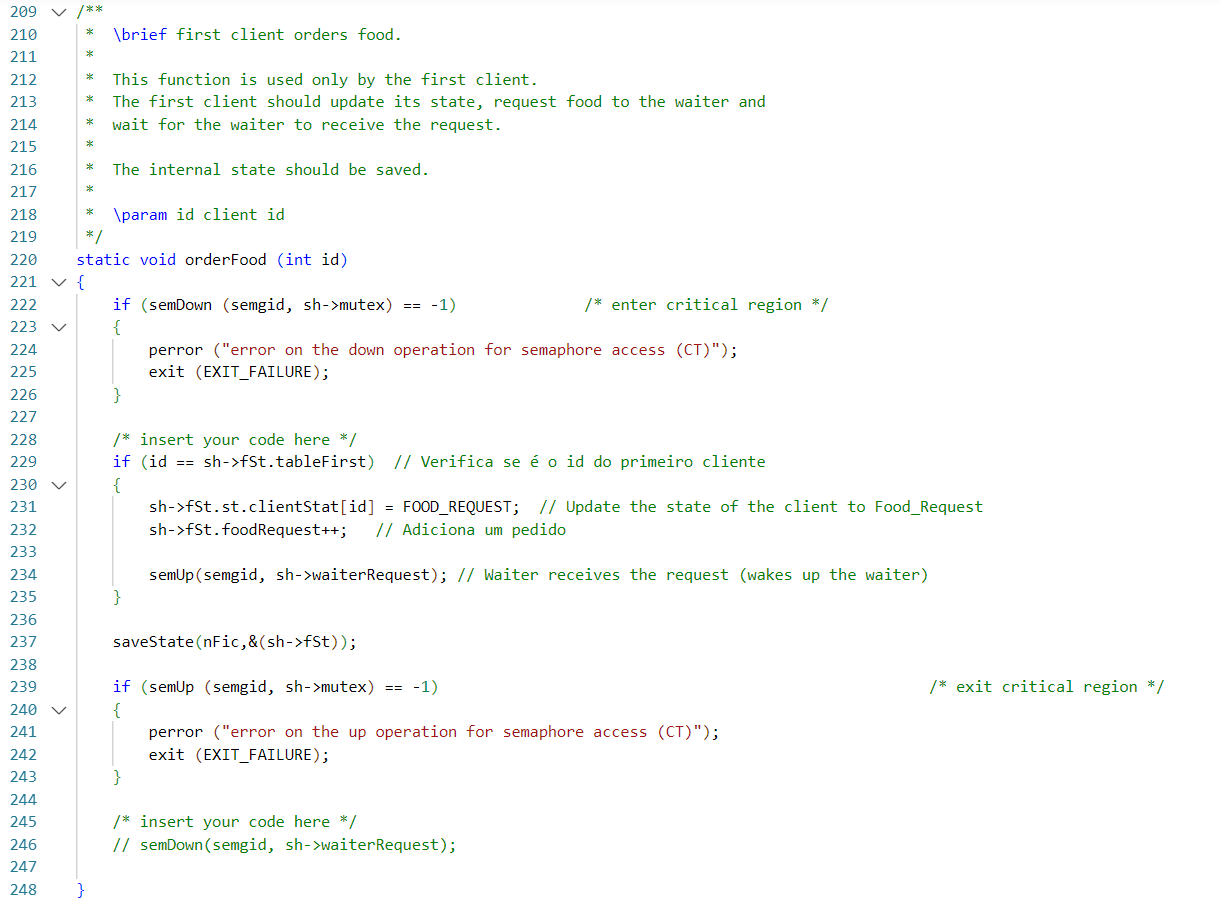


Fig.7 – Função orderFood.

Nesta função o cliente começa por entrar na região crítica, onde se é verificado se o ID do cliente é o primeiro. Nesta hipótese, o estado do cliente é alterado para FOOD\_REQUEST, é também incrementado o número de pedidos e o empregado é acordado com um semUp no semáforo waiterRequest. No final é guardado o estado interno com a função saveState.

Nas primeiras tentativas de resolução do problema, inserimos depois da região crítica um semDown no semáforo waiterRequest, no entanto, concluímos que se iria combinar uma situação de deadlock, por isso tirámos estas linhas de código.

### waitFood ()

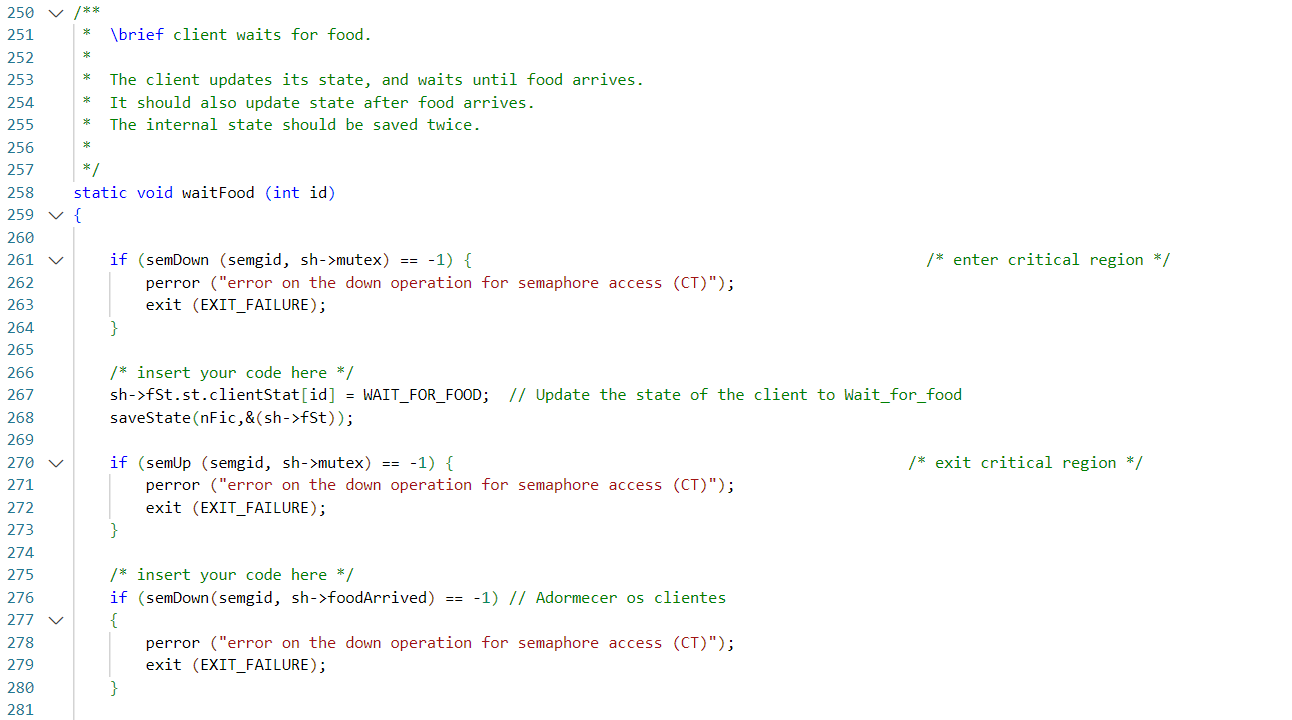
Nesta função, com o pedido efetuado, todos os clientes terão de esperar que o empregado traga a comida para a mesa para depois começarem a comer. Os estados dos clientes terão de ser guardados novamente, pois, ao contrário das funções retratadas até aqui, estes terão de ser alterados duas vezes.

Fig.8a – Primeira parte da função waitFood.

Esta função apresenta duas regiões críticas, visto que o estado de todos os clientes é alterado duas vezes. Na primeira região crítica, o cliente altera o seu estado para WAIT\_FOR\_FOOD e guarda-o com a função saveState. Ao sair desta região crítica, todos os clientes serão postos a dormir com a execução de um semDown no semáforo foodArrived e só serão acordados quando o empregado colocar a comida na mesa, na função takeFoodToTable () que será explicada mais à frente.

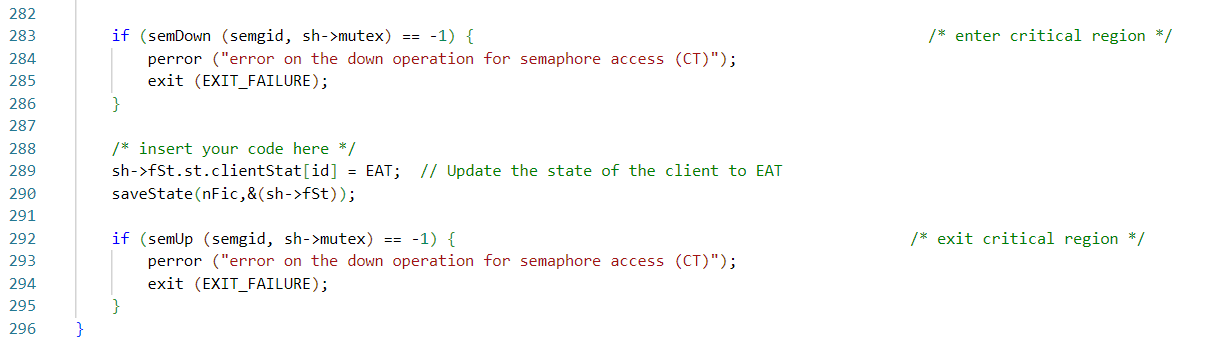


Fig.8b – Segunda parte da função waitFood.

Continuando na função, os clientes são acordados pelo empregado na função takeFoodToTable () e entram na segunda região crítica, em que o estado do cliente é alterado para EAT e, consequentemente, guardado com a função saveState.

### waitAndPay ()

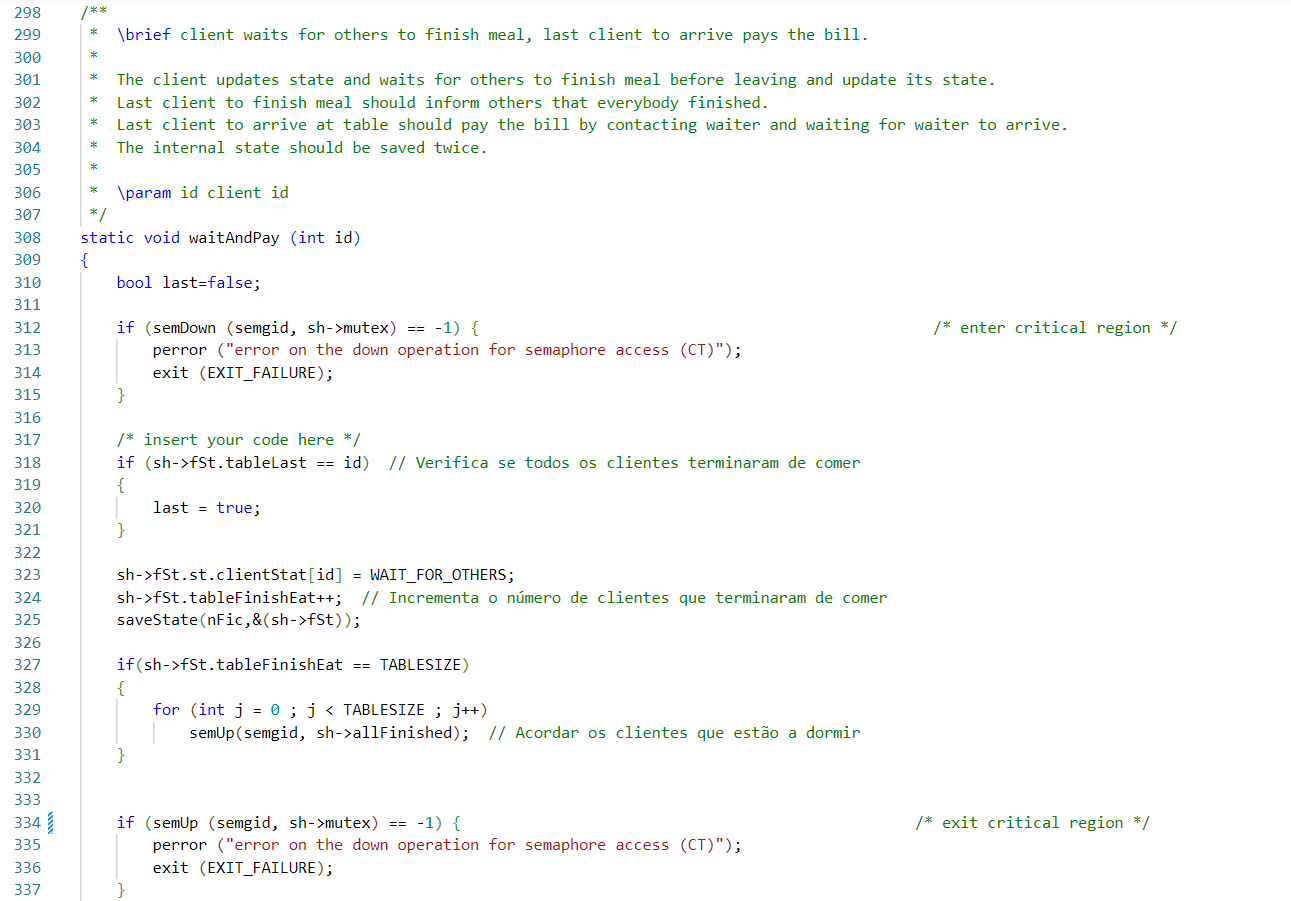
Nesta função os clientes que já acabaram de comer terão que esperar que os outros acabem também de comer, visto que só se poderá pedir a conta e efetuar o pagamento no fim de todos terminarem. Uma parte da função só será utilizada pelo último cliente que chegou ao restaurante, o mesmo que pede e efetua o pagamento da conta. Nesta função o estado de todos os clientes é alterado duas vezes à exceção do último cliente que chegou ao restaurante, que será alterado três vezes.

Fig.9a – Segunda parte da função waitAndPay.

Esta função começa com a definição de um booleano last, com valor inicial de false. Um cliente entra na primeira região crítica da função onde encontra uma condição que verifica se o ID é o do último cliente que chegou ao restaurante (isto porque assim conseguimos verificar que todos os clientes já acabaram de comer). Se for verdade, altera o booleano last para true e se não for verdade, altera o estado do cliente em questão para WAIT\_FOR\_OTHERS, incrementa o número de clientes que já terminaram a sua refeição e é guardado o estado interno de cada cliente com a função saveState.

O último passo que se encontra dentro da região crítica consiste em acordar os clientes que já acabaram de comer através de um “for”, para estes poderem continuarem o ciclo de execução do programa, com um semUp no semáforo allFinished 20 vezes.

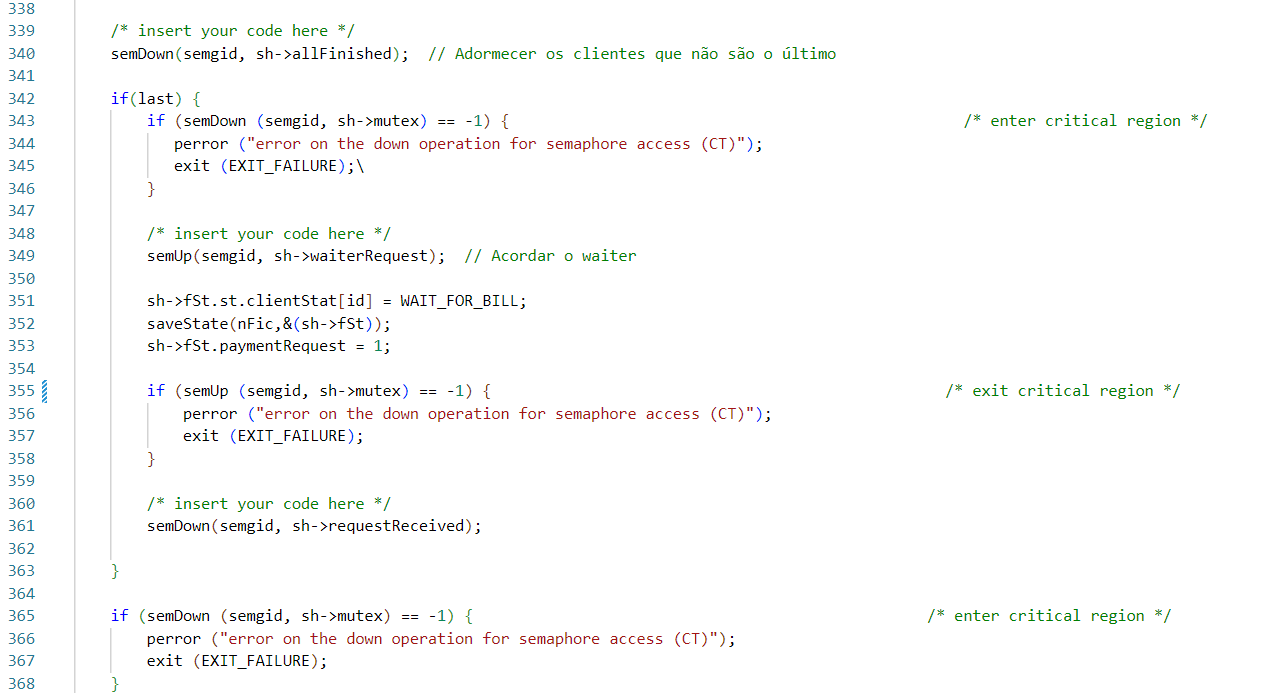


Fig.9b – Segunda parte da função waitAndPay.

Ao sair-se da região crítica e de modo a trabalhar apenas no último cliente, adormecem-se todos os clientes exceto o último com um semDown no semáforo allFinished. Desta forma, o cliente que colocou o booleano last verdadeiro, entra numa nova região crítica. Aqui, o empregado é acordado com um semUp no semáforo waiterRequest, para que este possa entregar a conta e receber o pagamento. O estado do último cliente é alterado para WAIT\_FOR\_BILL, o número de pedidos da conta é incrementado e o estado interno deste cliente é guardado com a função saveState. Ao sair da região crítica é feito um semDown no semáforo waiterRequest para adormecer o empregado pois já foi efetuado o pagamento e já não necessitamos mais dele.

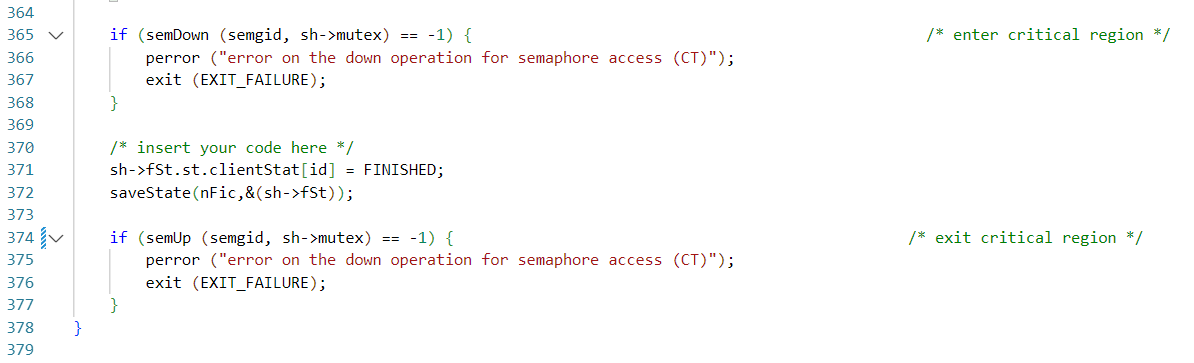


Fig.9c – Terceira parte da função waitAndPay.

Na última região crítica da função o estado de cada cliente é alterado para FINISHED, guardando-se com a função saveState. Isto conclui a simulação do jantar e o ciclo de vida do cliente.

## **Waiter**

O Waiter é a segunda mais importante entidade de três. Esta estabelece a relação entre as entidades Client e Chef. O ciclo de vida do Waiter provém de dois inteiros “req” e nReq”, variáveis que controlam um switch case que alterna entre as funções informChef (), takeFoodToTable () e receivePayment (), como é possível observar na figura 10. Isso posto, concluímos que o Waiter não tem um ciclo de vida com ordem estipulada e por isso intervém no programa maioritariamente por ordens de terceiros (Client e Chef), através da função waitForClientOrChef.

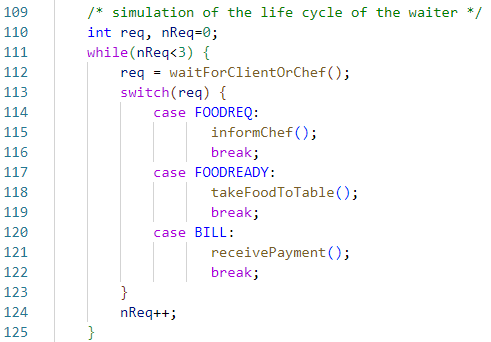


Fig.10 – Ciclo de vida da entidade Client.

### waitForClientOrChef ()

Nesta primeira função do Waiter, tem-se em conta a variável “ret” que indicará depois (através do retorno da função) a próxima função deste empregado. A variável, por deformidade, tem o valor 0.

Uma imagem com texto

Descrição gerada automaticamenteNuma primeira parte, o Waiter entra numa região crítica onde espera por ordens ou do Chef ou do Client, alterando-se o seu estado para WAIT\_FOR\_REQUEST, salvando-se depois este com a função saveState (). Depois, e já fora desta, faz-se um semDown no waiterRequest, “adormecendo-o”, fazendo-o esperar por requests.

Fig.11a – Primeira parte da função waitForClientOrChef ().

Numa segunda parte desta função, e já numa segunda região crítica, verificam-se o valor de três flags possíveis:

* foodRequest, flag do Client para o Waiter para indicar que a mesa está pronta a pedir;
* foodReady, flag do Chef para o Waiter para indicar que o pedido está confecionado;
* paymentRequest, flag do Client para o Waiter para indicar que a toda a mesa já acabou de comer e que estão prontos para pagar e sair.

Cada vez que o Waiter se encontra dentro desta região crítica, espera que uma destas flags seja levantada para poder aceitar pedidos de clientes ou do Chef. Caso isso aconteça, a variável “ret” passa a valer FOODREQ (1), FOODREADY (2) ou BILL (3) consoante a flag levantada. Desta forma no final da função (e já fora da região crítica), retorna-se esta variável permitindo dar continuação ao “switch case”.

Nota: para um correto funcionamento do “switch case” e novos requests, após o Waiter identificar a flag que foi levantada, colocámos de volta os valores dessas flags aos valores iniciais – “0”.

Uma imagem com texto

Descrição gerada automaticamente

Fig.11b – Segunda parte da função waitForClientOrChef ().

### informChef ()

Uma imagem com texto

Descrição gerada automaticamenteA função informChef () serve para levar o pedido da mesa ao chef. Esta função conta apenas com uma região crítica que coloca o estado do Waiter para INFORM\_CHEF e salva-o com a função saveState (). Depois desta região de código, é levantado o semáforo waitOrder (através de um semUp) para o Waiter acordar o Chef e levar-lhe o pedido.

Fig.12 – Função informChef ().

### takeFoodToTable ()

A função takeFoodToTable () consiste também em apenas uma região crítica e está desenvolvida para levar os pratos já confecionados desde o Chef até à mesa. Dentro desta região, passa-se o estado do Waiter para TAKE\_TO\_TABLE e salva-se com a função saveState (). Após isto, leva a comida cliente a cliente, acordando-os desta forma com o seu pedido (como é possível observar nas linhas 244 e 245 da imagem com o código da função). Ulteriormente, é levantado o semáforo requestReceived do cliente de modo a dar como terminado esta fase do programa onde a mesa dos amigos Uma imagem com texto

Descrição gerada automaticamenteespera pelos pedidos.

Fig.13 – Função takeFoodToTable.

### receivePayment ()

Uma imagem com texto

Descrição gerada automaticamentePor último, temos a função receivePayment (). Tal como as duas funções anteriores, apenas possui uma região crítica, responsável pela receção do pagamento. Mais uma vez dentro da região crítica mudamos o estado o Waiter para RECEIVE\_PAYMENT e salvamos esta operação através da função saveState (). Finalmente, fazemos semUp do semáforo requestReceived que permite notificar as clientes que o Waiter deu entrada no pedido.

Fig.14 – Função receivePayment ().

## **Chef**

O chef é a última de 3 entidades que falta referir. Esta entidade apenas “conversa” com o Waiter, ou seja, não tem comunicação direta com a entidade Client. O seu ciclo de vida no programa é relativamente simples, como é possível observar na imagem 15. Possui apenas 2 funções: waitForOrder e processOrder. Ao contrário do que se assume, estas funções não fazem necessariamente o que indicam no seu nome, como iremos explicar mais adiante.

### waitForOrder ()

O nome desta função pode induzir ao erro já que “wait for order” traduz-se para “esperar por pedido”. De facto, o Chef já possui como estado inicial o “WAIT\_FOR\_ORDER” (0) e como o estado seguinte possível é “COOK”, esta função irá tocar apenas nestes (estados). Por isto, o único código em falta antes da região crítica da função é a realização de um semDown do semáforo waitOrder, levantado pelo Waiter na função informChef ().

Já dentro da região crítica desta função colocamos o estado do chef para COOK, ou seja, é aqui que o Chef começa o processo de confeção dos pedidos dos clientes. Na linha seguinte 131, salva-se este novo estado com a função saveState ().

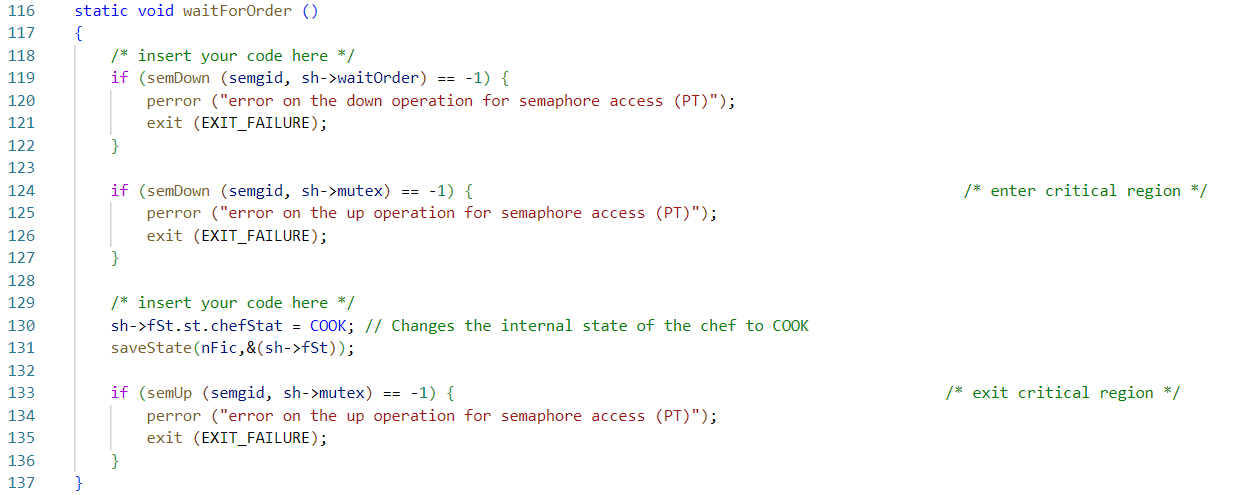


Fig.15 – Função waitForOrder ().

### processOrder ()

Na função processOrder, o chef cozinha e, aqui, através de um sleep correspondente ao tempo que o Chef demora a cozinhar o pedido, demora-se um certo tempo até a comida estar pronta. Quando a comida ficar pronta, o chef irá entrar numa região crítica na qual o seu estado muda para REST e é levantada uma flag que indica que a comida está pronta. Salva-se posteriormente este estado. Ao sair da região crítica, o Chef vai acordar o Waiter com um semUp do waiterRequest para este levar a comida até à mesa, como é possível verificar na imagem 16.

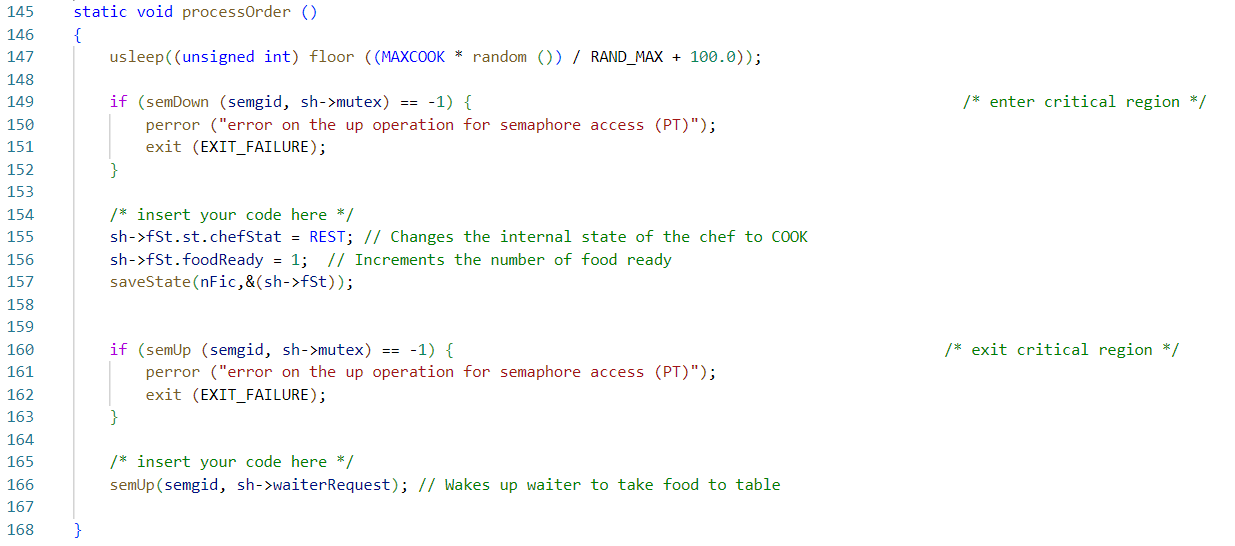


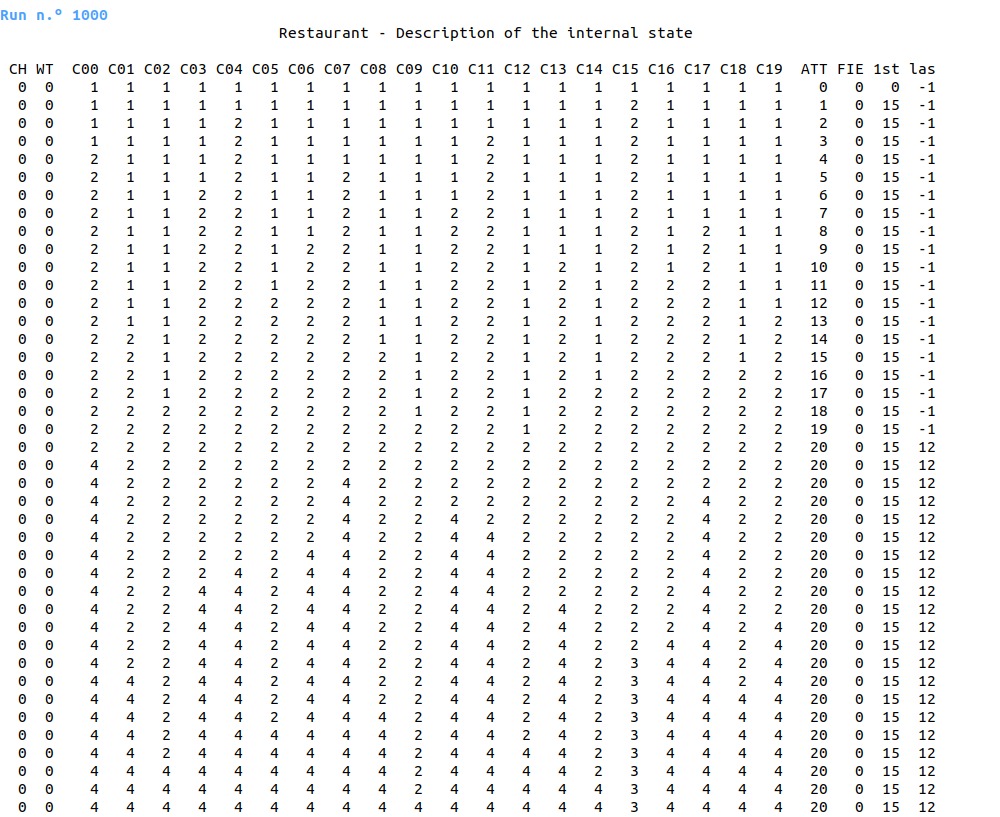
Fig.16 – Função processOrder ().

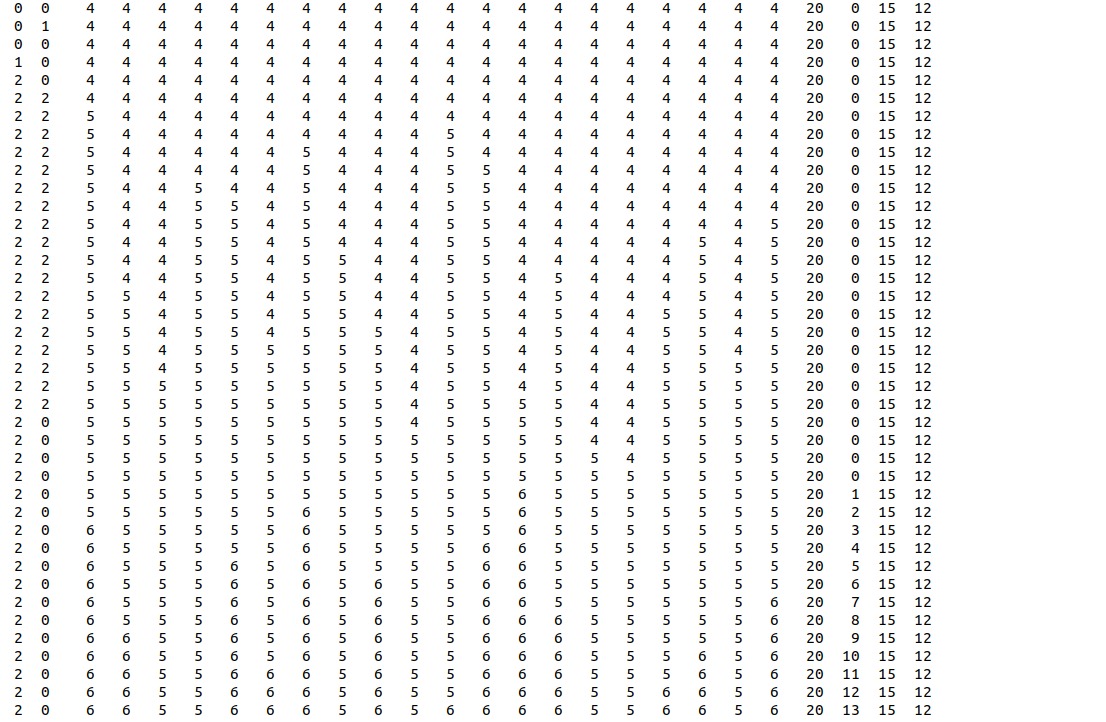
# ***Resultados***

Durante a implementação da nossa solução, fizemos compilámos, com o comando make, e corremos o executável probSemSharedMemRestaurant, que resultou da compilação do ficheiro proSemSharedMemRestaurant.c. Compilámos a solução que o professor disponibilizou usando o comando make\_allbin e corremos o executável para podermos comparar os nossos resultados com os do professor.

Neste ponto do relatório, irão ser avaliados os resultados obtidos, para isso, corremos o script run.sh que simula 1000 jantares. O objetivo de correr este script é para sabermos se existem situações de deadlock na nossa implementação.

Escolhemos a milésima e última vez que o programa correu para analisarmos e reparamos que não existem situações de deadlock e todas as regras descritas no enunciado do problema são cumpridas, logo podemos concluir que a nossa implementação funciona corretamente.





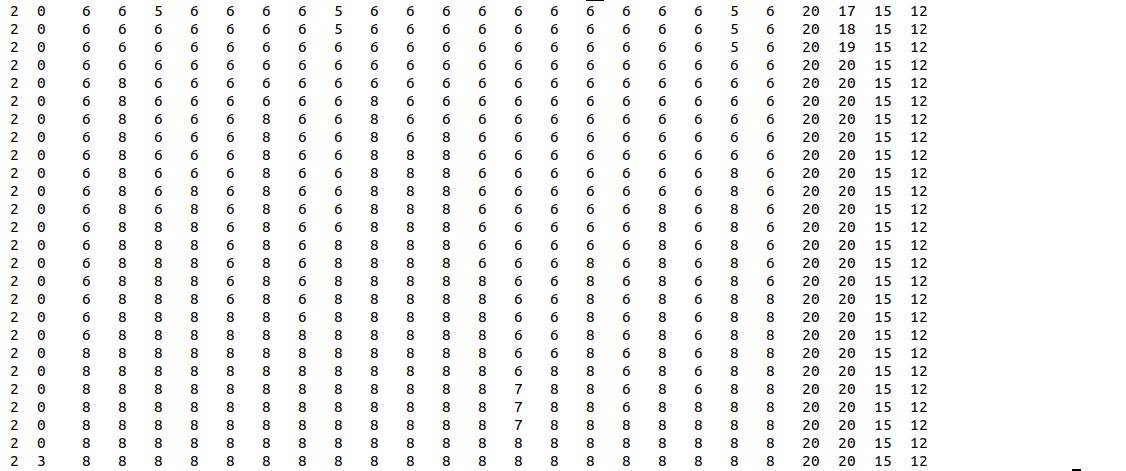


Fig.17 – Output da solução implementada por nós.

Para melhor compreender e verificar a existência de erros contruímos a seguinte tabela com informações relativas aos semáforos, como qual a entidade que faz up ou down no semáforo, o número de down’s e up’s feitos e a função onde se faz up ou down num semáforo.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Semáforo | Entidade que faz down | Função onde se faz down | Número de down’s | Entidade que faz up | Função onde se faz up | Número de up’s |
| friendsArrived | Clientes | waitFriends () | 19 | LC | waitFriends () | 19 |
| requestReceived | LC; | waitAndPay () | 1 | Empregado | takeFoodToTable ()  receivePayment () | 1  1 |
| foodArrived | Clientes | waitFood () | 20 | Empregado | takeFoodToTable () | 20 |
| allFinished | Clientes | waitAndPay () | 20 | Clientes | waitAndPay () | 20 |
| waiterRequest | Empregado | waitForClientOrChef () | 1 | FC;  LC;  Chefe | orderFood ()  waitAndPay ()  processOrder () | 1  1  1 |
| waitOrder | Chefe | waitForOrder () | 1 | Empregado | informChef () | 1 |

**Legenda:**

* LC - Last Client (último cliente a chegar ao restaurante);
* FC - First Client (primeiro cliente a chegar ao restaurante).

# ***Conclusão***

Em conclusão, apresentámos através deste relatório uma noção de semáforos e regiões críticas e relacionámos estes conteúdos com o segundo projeto da cadeira de Sistemas Operativos. Completámos o código fornecido incidindo principalmente nas entidades Client, Waiter e Chef, processos independentes cuja sincronização e comunicação é feita através de vários semáforos e de memória partilhada. Conseguimos também ultrapassar situações de deadlock (onde dois ou mais processos – que sejam da mesma entidade ou não – estejam bloqueados e, por fim, adquirimos conhecimentos sobre programação de semáforos em C.