

**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](#_Toc123512140)

[***2. Material Fornecido*** 4](#_Toc123512141)

[***3. Desenvolvimento*** 9](#_Toc123512142)

[**3.1. Client** 9](#_Toc123512143)

[3.1.1. waitFriends () 10](#_Toc123512144)

[3.1.2. orderFood () 11](#_Toc123512145)

[3.1.3. waitFood () 12](#_Toc123512146)

[3.1.4. waitAndPay () 13](#_Toc123512147)

# ***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 as entidades que participam nesta simulação são processos independentes, sendo que a sua sincronização e comunicação é efetuada através de vários semáforos e de memória partilhada.

A simulação é constituída por um grupo de vinte amigos, um empregado e um chefe. A simulação começa com a chegada de todos os amigos ao restaurante, em que o primeiro amigo a chegar será o que irá fazer o pedido da comida, que 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 2 ou mais processos sejam bloqueados pois estão à espera de um evento que apenas pode ser despoletado por um dos processos em bloqueio, ou seja, uma situação de deadlock, iremos usar semáforos.

A utilização de semáforos serve essencialmente para controlar o acesso à memória partilhada, evitando choques entre as 3 entidades que participam na simulação. As notificações entre entidades são feitas através de semáforos para que o programa executasse sem encontrar problemas.

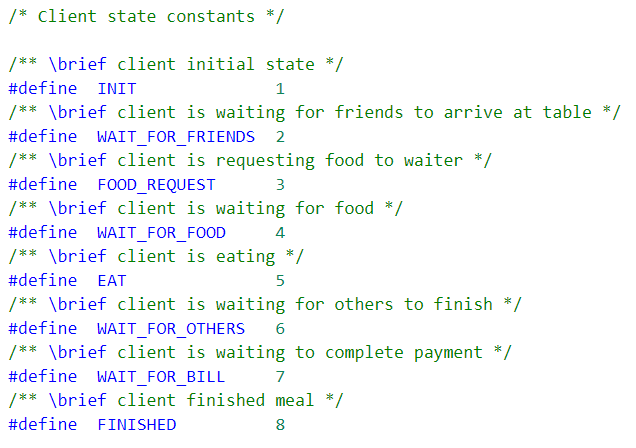
Com a realização deste trabalho prático, esperamos conseguir cumprir todos os pontos essenciais que são propostos no guião e alargar os nossos conhecimentos relativamente a programar com semáforos em C, visto ser um aspeto importante no que toca a controlar o acesso a determinadas regiões por parte de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

# ***Material Fornecido***

É disponibilizado um conjunto de ficheiros para a resolução do problema, em que está presente uma pasta src com vários ficheiros, estes contêm código em C para a simulação do jantar e, três deles, encontram-se incompletos que são semSharedMemChef.c, semSharedMemClient.c e semSharedMemWaiter.c e ao longo deste relatório iremos explicar como é que completámos as funções que se encontram incompletas em cada um dos ficheiros.

Existem também outros ficheiros que contém dados importantes para perceber como é que é simulado o jantar:

- probConst.h: este ficheiro contém variáveis que irão ser utilizadas depois nos 3 ficheiros incompletos, como os estados dos clientes, os estados do empregado, os estados do chefe, o tamanho máximo da mesa do restaurante e os tempos máximos para cozinhar a refeição e para comer.



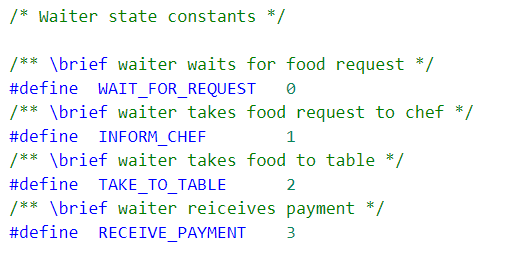
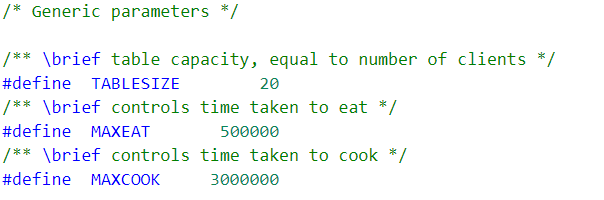
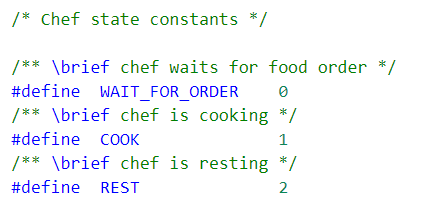
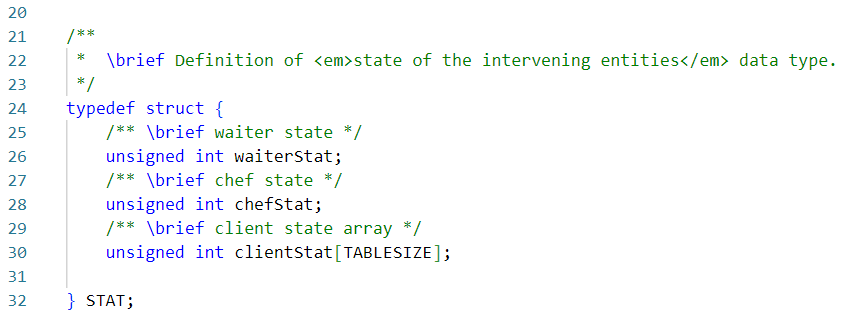


Fig.1 – Constantes definidas no ficheiro probConst.h.

- 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, logo a variável para o estado de cada cliente será um array com o tamanho de 20 (TABLESIZE). A estrutura FULL\_STAT apresenta vários parâmetros que encontram na figura dois, sendo que 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; a variável tableFirst e tableLast irão guardar os ID’s do primeiro e último cliente a chegar, respetivamente; por fim as variáveis tableClients e tableFinishEat irão funcionar como contadores para que sempre que chegar um cliente ao restaurante ou um cliente acabar de comer estas variáveis são incrementadas.



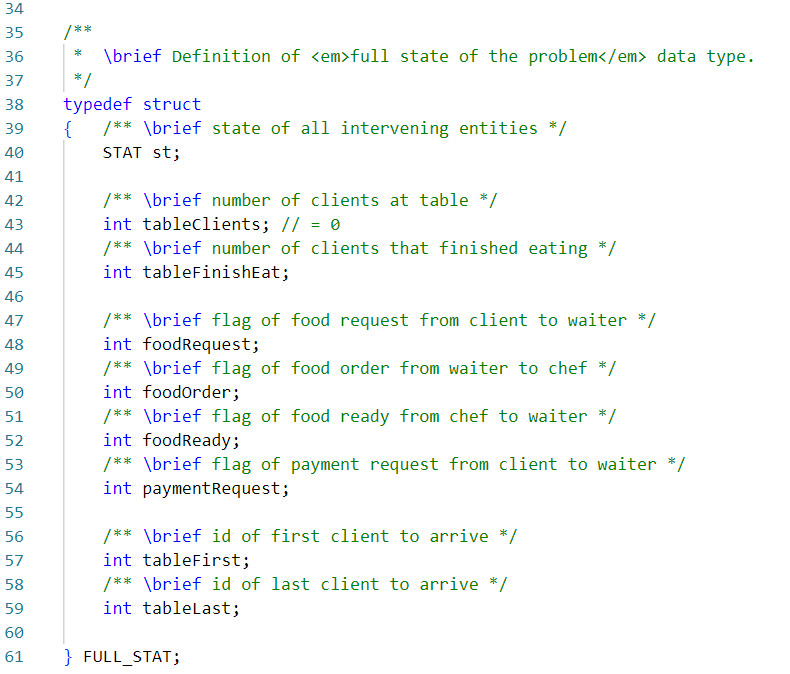


Fig.2 –Estruturas definidas no ficheiro probDataStruct.h.

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

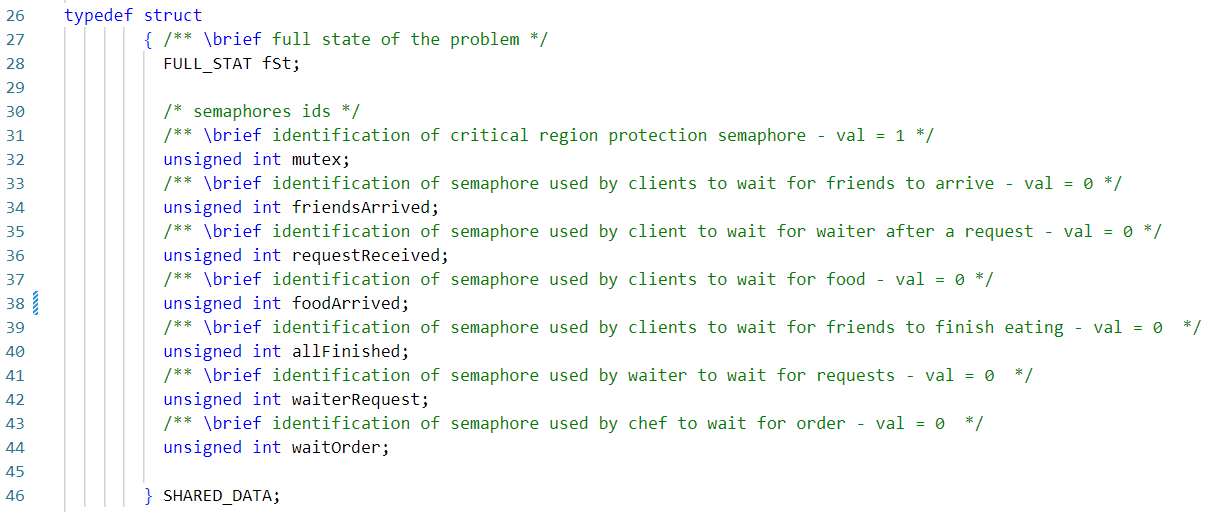
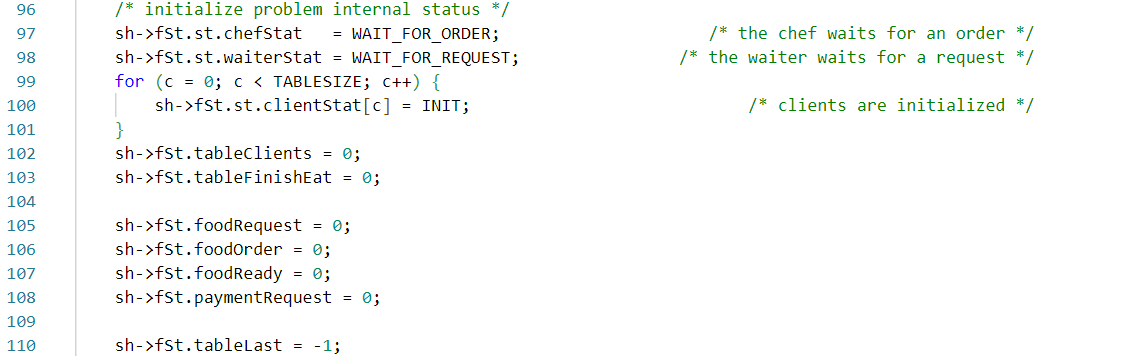
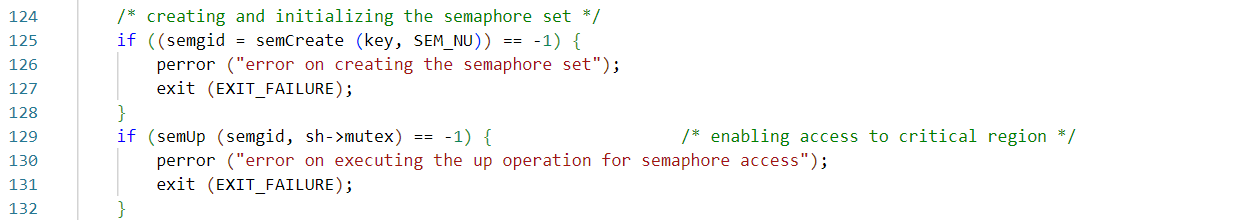
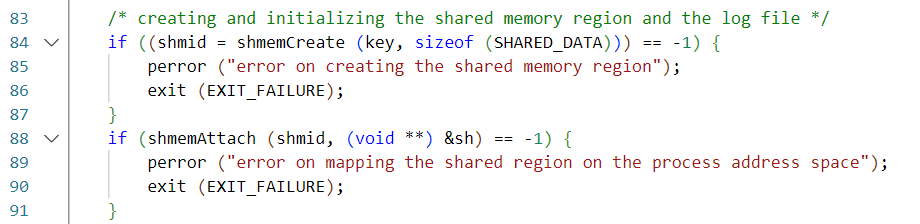
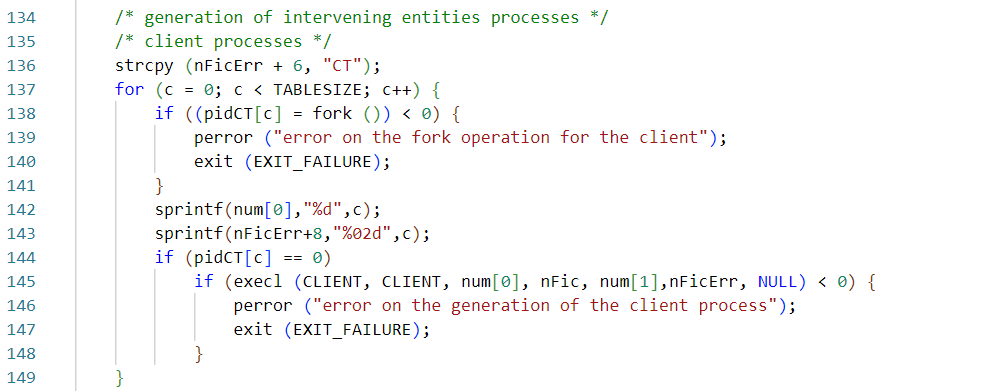
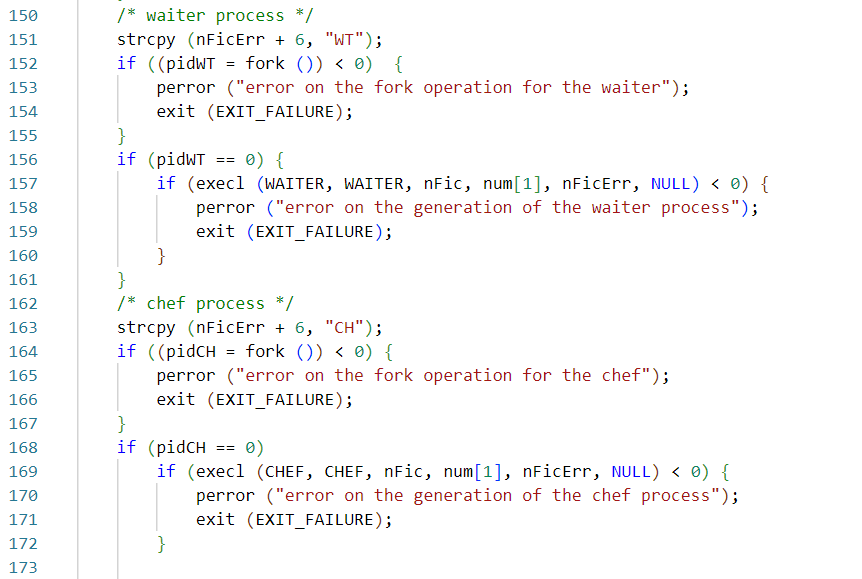


Fig.3 –Estrutura definida no ficheiro sharedDataSync.h.

- 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.





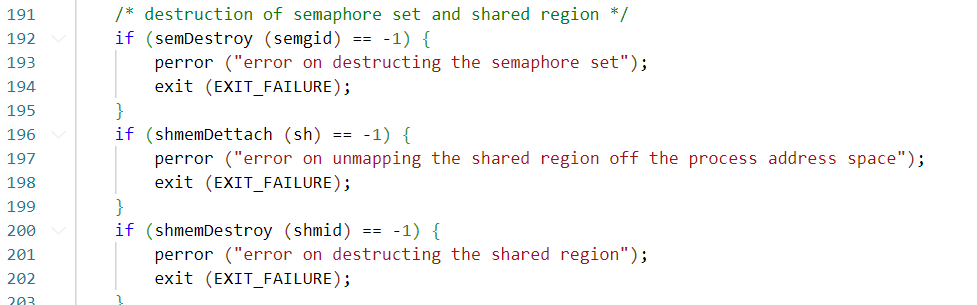


Fig.4 – Algum código presente no ficheiro probSemSharedMemRestaurant.c.

# ***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**

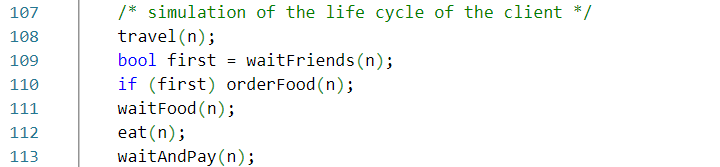
Começamos pelo client visto que a primeira ação que ocorre na simulação é a chegada de todos os clientes ao restaurante e o ciclo de vida do cliente é o maior de entre as 3 entidades envolventes na simulação, como podemos observar na figura 5.

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 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. Se todos tiverem chegado ele avança para a próxima função, orderFood (), em que irá se efetuar 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 () e, por fim, todos os clientes terão que esperar que todos terminem a sua refeição para que se possa pedir a conta e acabar o jantar, é o que acontece na função waitAndPay () e termina assim o ciclo de vida do cliente.

### 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, este terá de ser feito pelo primeiro. Será necessário guardar o estado de cada um dos clientes pois este irá alterar para todos.

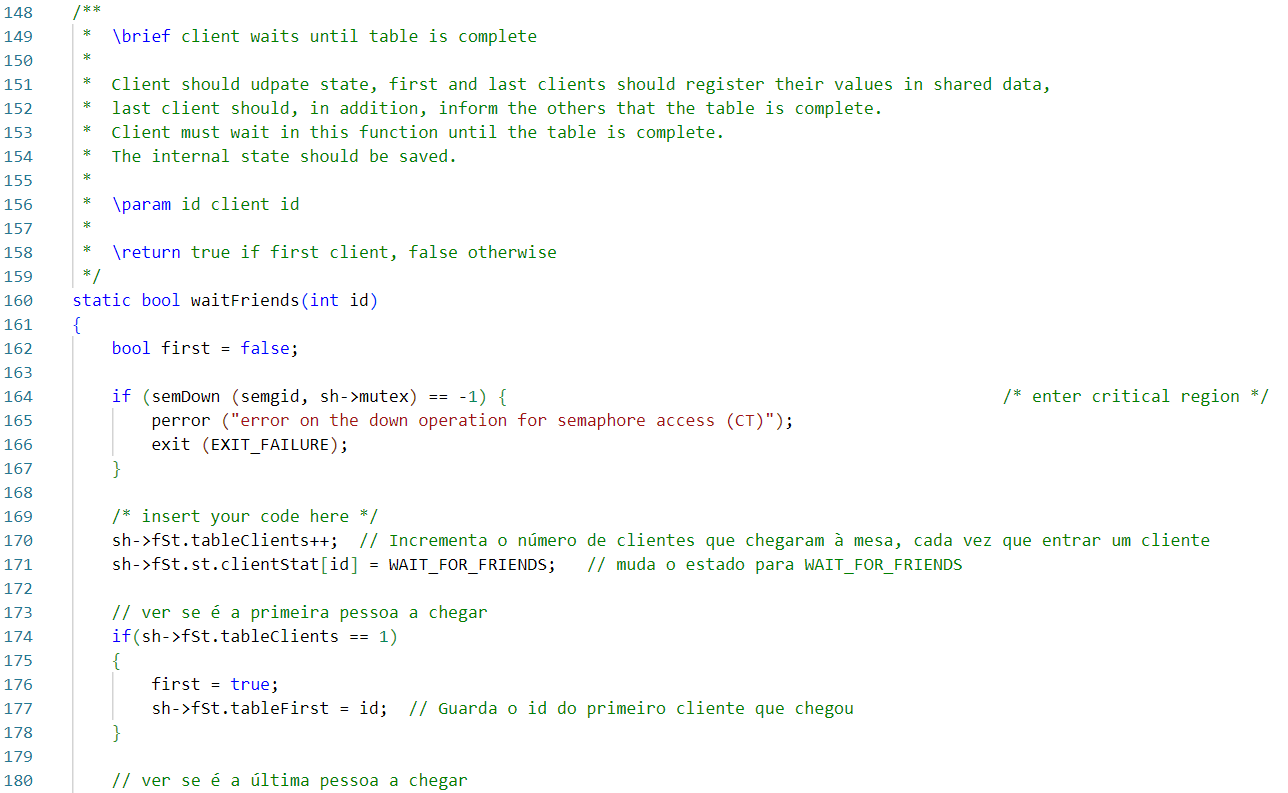


Fig.6a – Primeira parte da função waitFriends.

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 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 e é guardado o id do cliente que chegou, pois o primeiro cliente é o que efetua o pedido da comida, logo será importante guardar o ID dele.

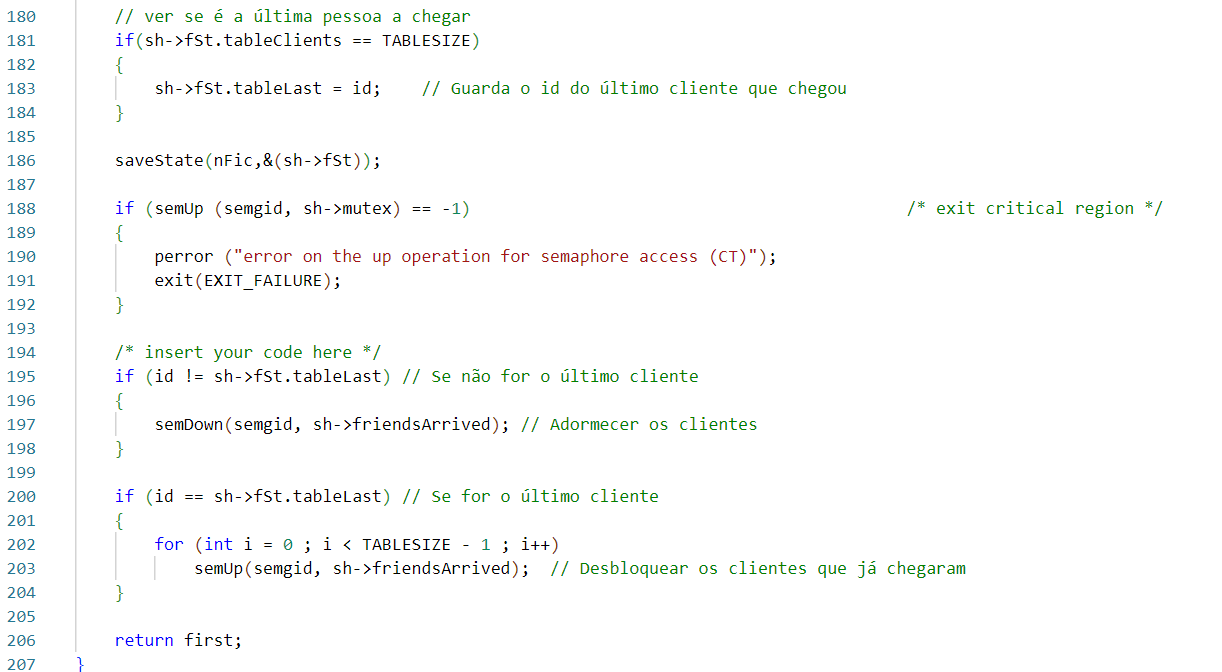


Fig.6b – Segunda parte da função waitFriends.

Se a condição da verificação do primeiro cliente não se verificar avança para a próxima condição, em que verifica se o cliente que está na região crítica é o último cliente, através da linha 181 e se esta for verdadeira é guardado o ID do cliente em questão pois o último cliente a chegar irá efetuar o pedido e o pagamento da conta. Continuando, o estado de cada cliente e as variáveis que foram alteradas serão guardadas através da função saveState (). Guardado o estado o cliente que está a percorrer a função sai da região crítica e é verificado se o seu ID não é último, se não for o cliente é adormecido com um semDown efetuado no semáforo friendsArrived, pois 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 é verificado se o ID do cliente é o último e, se for, os clientes que foram adormecidos são acordados com um semUp no semáforo friendsArrived, que é efetuado 19 vezes, pois é o número de clientes que já chegaram. Assim termina esta função e avança para a próxima.

### 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 irá ser alterado.

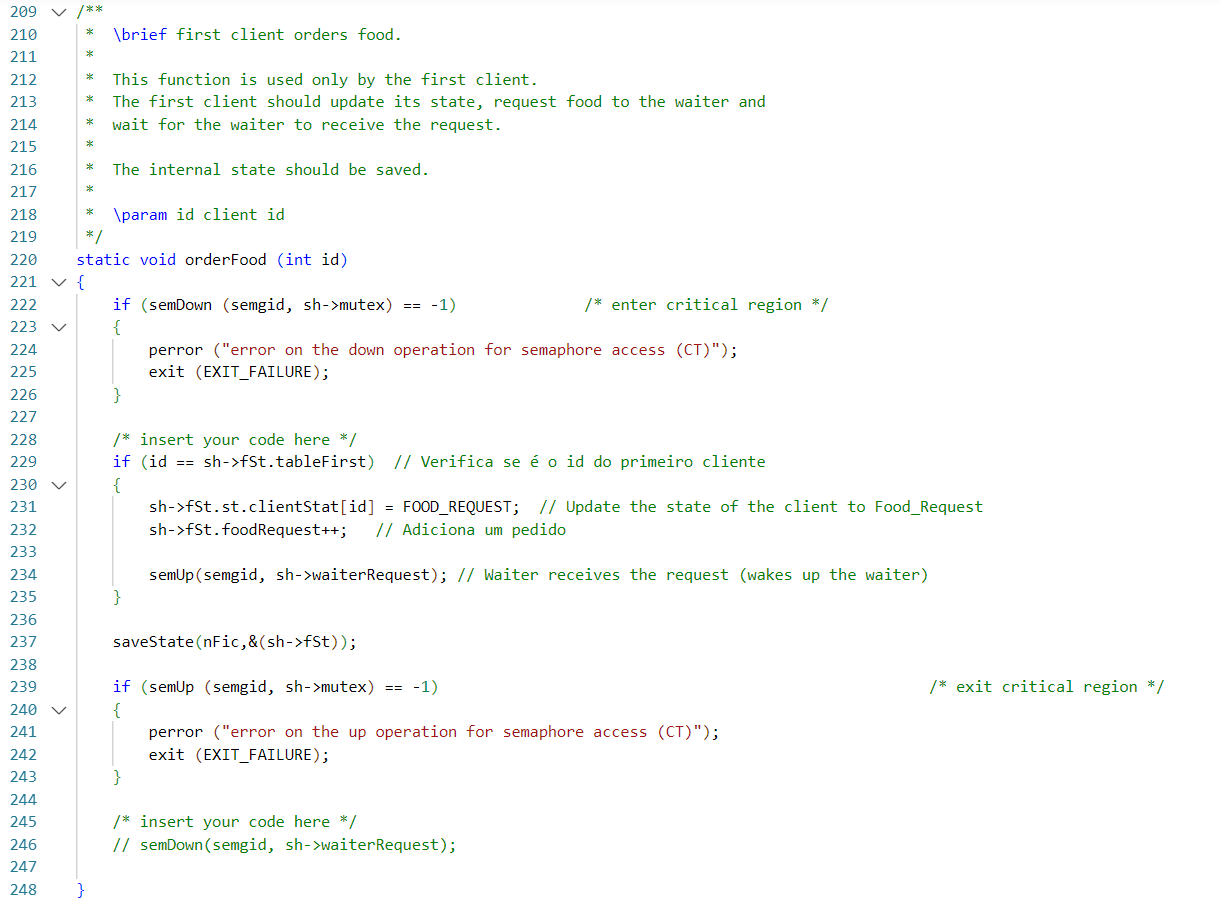


Fig.7 – Função orderFood.

Nesta função o cliente começa entrando na região crítica e verifica se o ID do cliente é o do primeiro, se for o estado do cliente é alterado para FOOD\_REQUEST, é 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. Depois da região crítica nós inserimos um semDown no semáforo waiterRequest, mas verificámos que tínhamos criado uma situação de deadlock, por isso tirámo-la.

### waitFood ()

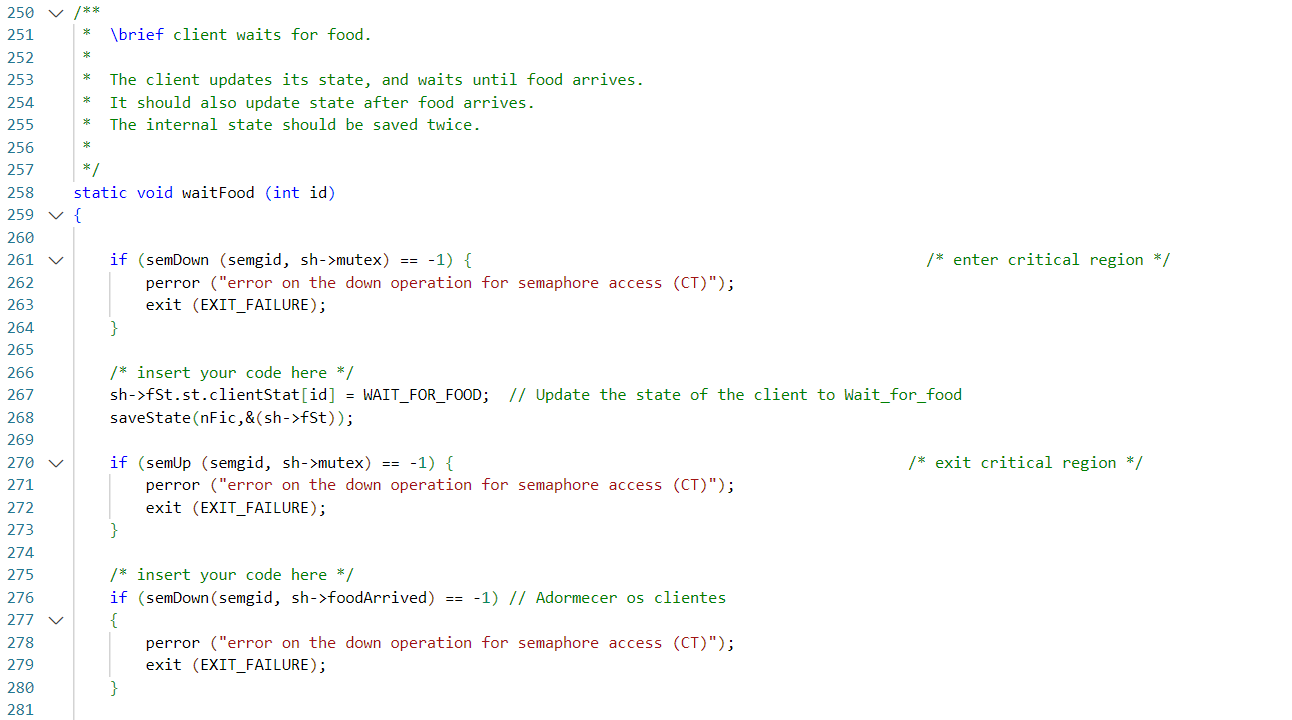
Nesta função, com o pedido efetuado, todos os clientes terão que esperar que o empregado traga a comida para a mesa e, depois de esta chegar, poderão começar a comer. O estado dos clientes terá de ser guardado novamente, pois este é alterado duas vezes.

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, os clientes são 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.

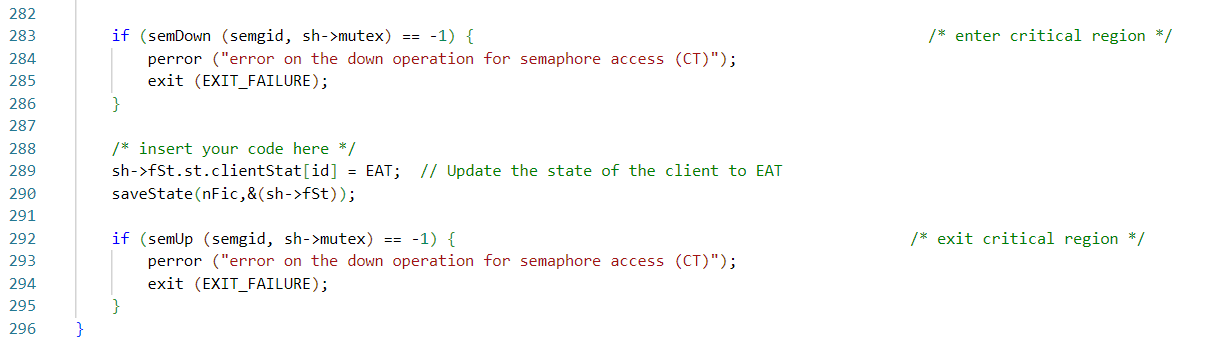


Fig.8b – Segunda parte da função waitFood.

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

### waitAndPay ()

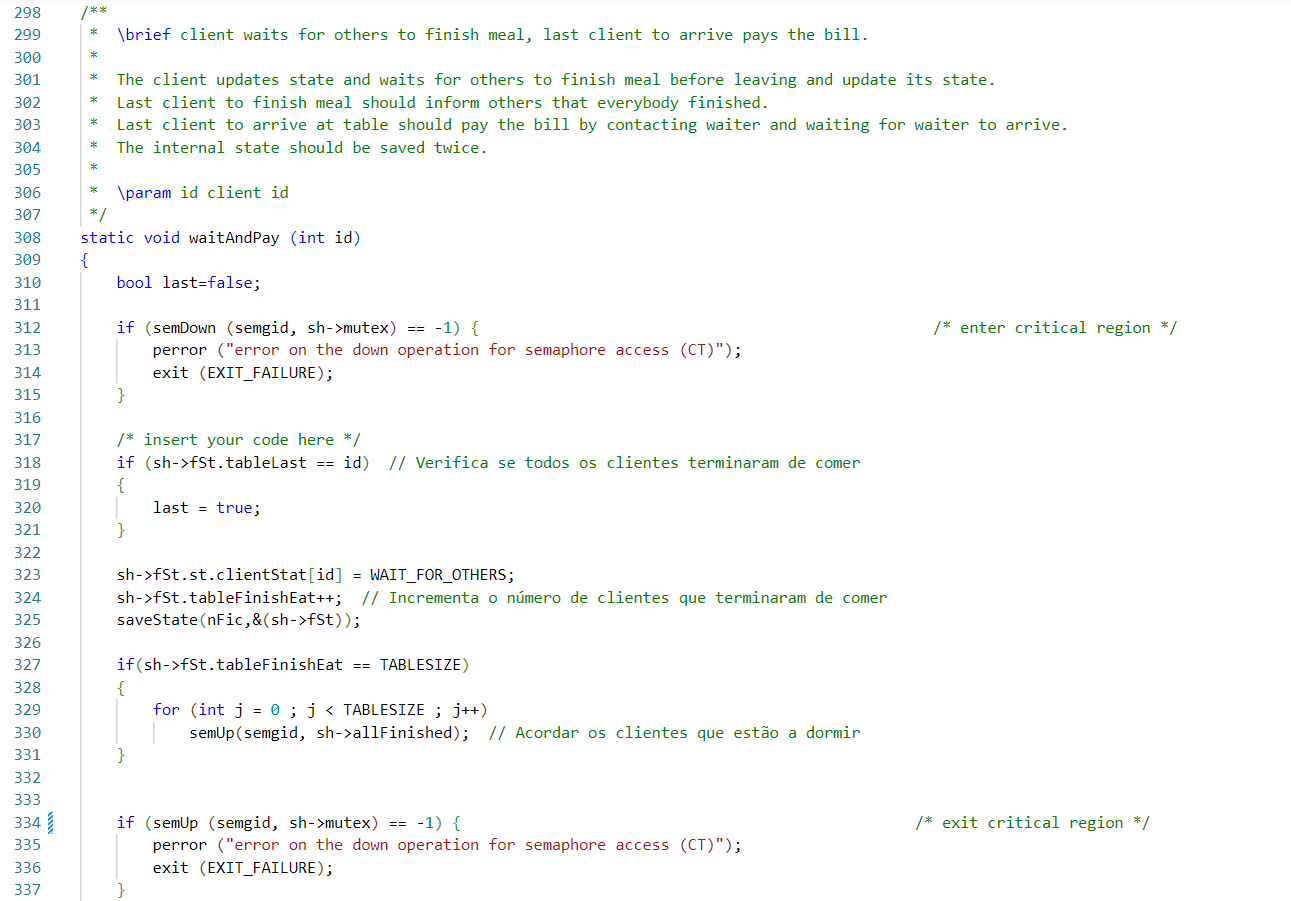
Nesta função os clientes que já acabaram de comer terão que esperar que os outros acabem de comer, visto que só se poderá pedir a conta e efetuar o pagamento no fim de todos terminarem. Uma parte da função só será utilizada pelo último cliente que chegou ao restaurante pois este irá pedir a conta e efetuar o pagamento da mesma. Nesta função o estado de todos os clientes é alterado duas vezes exceto o 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 e se for verdade, altera o booleano last para true, se não 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, para estes poderem continuar a executar o código da função, e é o que acontece quando se realiza um semUp no semáforo allFinished 20 vezes.

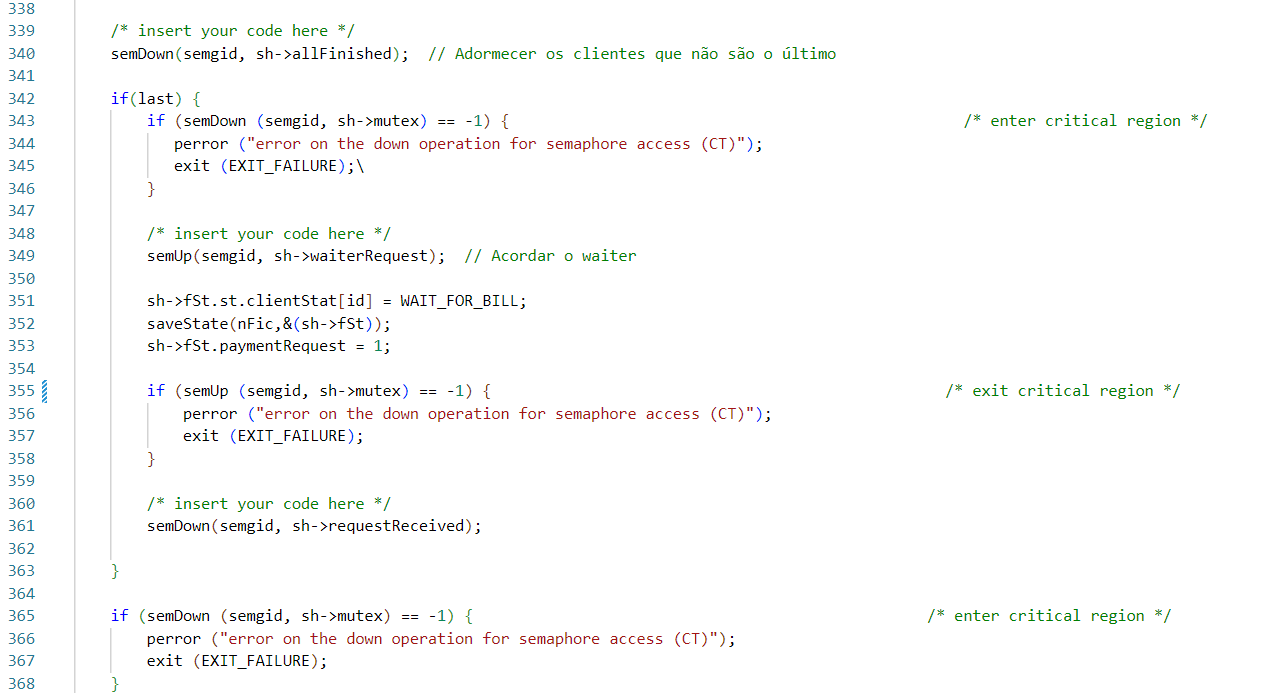


Fig.9b – Segunda parte da função waitAndPay.

Ao sair da região crítica os clientes são postos a dormir com um semDown no semáforo allFinished, menos o cliente com o ID do último cliente que chegou ao restaurante, pois este avança para a segunda região crítica da função, que será apenas percorrida por esse cliente. Nesta região crítica, o empregado é acordado com um semUp no semáforo waiterRequest para que este possa entregar a conta e receber o pagamento. O estado do 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.

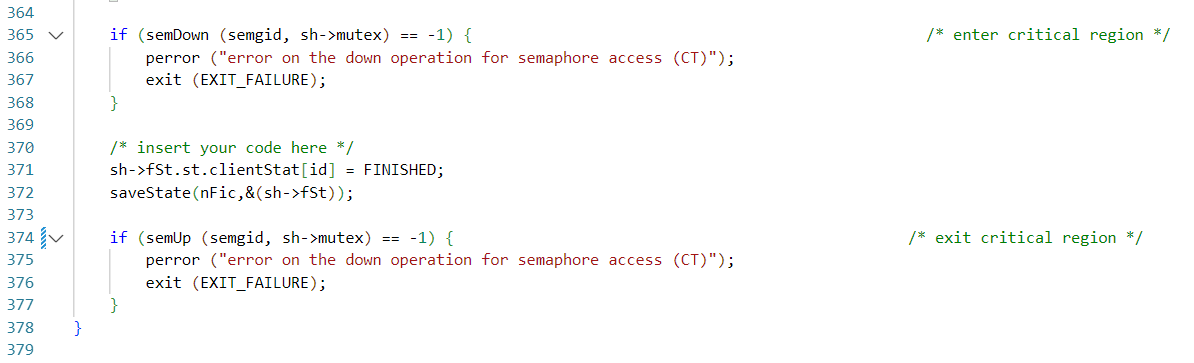


Fig.9c – Terceira parte da função waitAndPay.

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