deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

universidade de aveiro

Trabalho 2

Restaurante

Realizado por: Pedro Escaleira nº88821
Luís Fonseca nº 89066

1. Índice

2. Introdução

Este segundo trabalho da componente prática da disciplina de Sistemas Operativos foi-nos proposto pelo docente da disciplina com o objetivo de aprendermos como melhor manusear processos e *threads*. Desta forma, foi-nos apresentado um problema onde se pretende simular um restaurante com quatro entidades:

1) **Chefe**, responsável por cozinhar as refeições pedidas por cada grupo.
2) **Grupo**, responsável por ir ao restaurante, pedir uma mesa, pedir um refeição e pagar a conta.
3) **Empregado de mesa**, responsável por receber os pedidos de comida de cada grupo, levar estes até ao chefe e entregar a refeição pedida ao correspondente grupo.
4) **Rececionista**, responsável por atribuir uma mesa a cada grupo e por receber os pagamentos destes.

3. Abordagem usada para resolver o problema

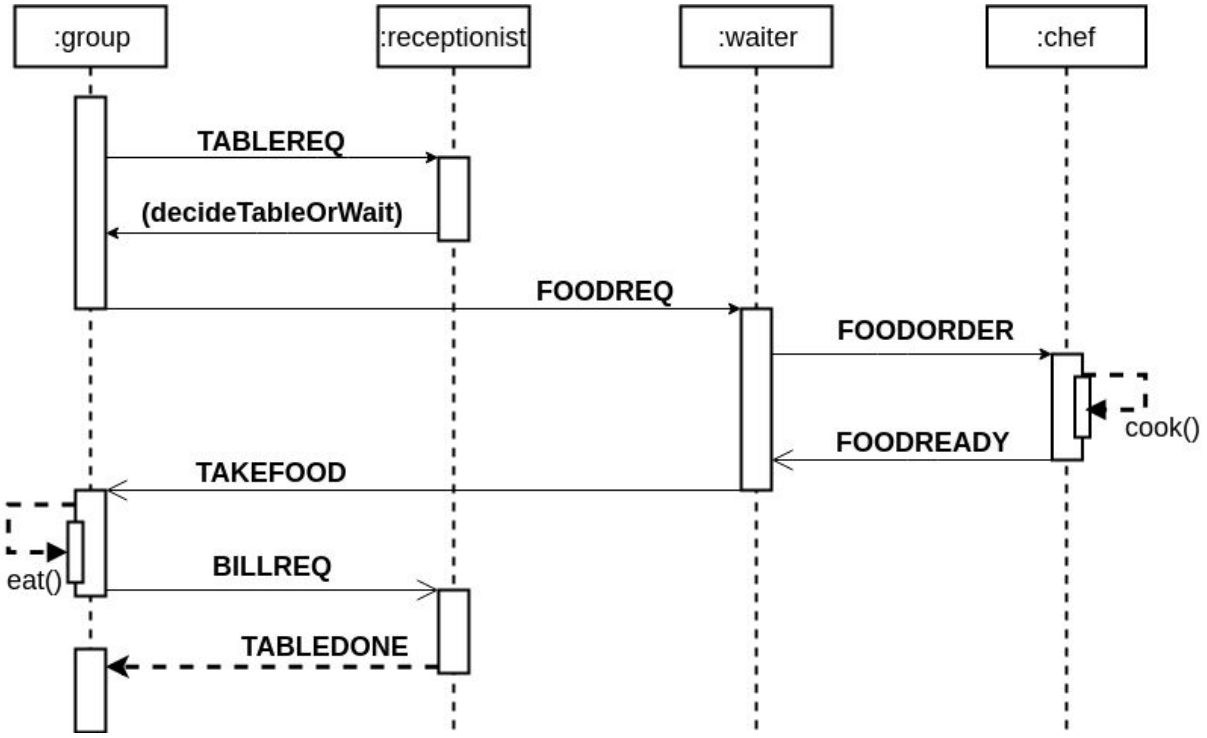
3.1. Resumo das operações a realizar sobre os semáforos

Numa primeira abordagem ao problema, decidimos começar por analisar o que o código de cada um dos ficheiros de código disponibilizados faziam quando executados. Desta forma, decidimos começar por criar uma tabela com um resumo de onde, em cada entidade, se deveriam fazer os *UPS* e *DOWNS* de cada semáforo presente no ficheiro **sharedDataSync.h**:

Semáforo	***Up***	***Down***
receptionistReq	• **grupo**, quando acaba de pedir uma mesa ou a conta ao rececionista	• **rececionista**, quando começa a espera dum pedido dum grupo
receptionistRequestPossible	• **rececionista**, quando acaba de atender um grupo	• **grupo**, antes de pedir uma mesa ou a conta ao rececionista
waiterRequest	• **grupo**, quando acaba de fazer o pedido de comida ao empregado de mesa • **chefe**, quando acaba de entregar a comida confecionada ao empregado de mesa	• **empregado de mesa**, quando começa a espera do pedido de comida (por parte dum grupo) ou de uma entrega de comida (por parte do chefe)
waiterRequestPossible	• **empregado de mesa**, quando acaba de atender um grupo ou um chefe	• **grupo**, quando chama o empregado de mesa para lhe pedir a comida • **chefe**, quando acaba de cozinhar e chama o empregado de mesa para lhe entregar a comida confecionada
waitOrder	• **empregado de mesa**, quando acaba de entregar o pedido de comida ao chefe	• **chefe**, quando começa a espera por um pedido de comida
orderReceived	• **chefe**, quando acaba de receber um pedido de comida	• **Empregado de mesa**, quando acaba de fazer um pedido de comida ao chefe

waitForTable[id]	● **rececionista**, quando acaba de atribuir uma mesa ao grupo que possui esse id	● **grupo que possui esse id**, quando acaba de fazer o pedido de mesa ao rececionista
requestReceived[table]	● **empregado de mesa**, quando acaba de receber um pedido de comida por parte do grupo que se encontra nessa mesa	● **grupo que se encontra nessa mesa**, quando acaba de fazer o pedido de comida ao empregado de mesa
foodArrived[table]	● **empregado de mesa**, quando acaba de entregar a comida a um grupo que se encontra nessa mesa	● **grupo que se encontra nessa mesa**, quando começa a espera pela comida que pediu ao empregado de mesa
tableDone[table]	● **rececionista**, quando acaba de aceitar o pagamento do grupo que se encontra nessa mesa	● **grupo que se encontra nessa mesa**, quando acaba de pedir a conta ao rececionista

Para além desta tabela foi feito também, com a ajuda cedida pelo professor da disciplina durantes as aulas práticas, o seguinte diagrama da ordem de trabalhos a ser feita por cada entidade:



Desta forma, quando se começou a alterar o código das entidades, tornou-se relativamente acessível compreender como fazê-lo.

3.2. Resumo do funcionamento de cada entidade

Após termos feito a tabela apresentada no ponto anterior, começamos a alterar o código de cada entidade de acordo com o descrito nela. Apresentamos de forma mais detalhada que alterações foram feitas em cada função de cada entidade. A ordem pela qual descrevemos cada uma delas corresponde à ordem com que de facto alteramos cada uma.

3.2.1. Chefe

3.2.1.1. Função **waitForOrder**

Esta função é responsável por manter o chefe á espera dum novo pedido do empregado de mesa. Desta forma, inicialmente é feito um *Down* no semáforo **waitOrder**.

Já dentro da região crítica, o chefe toma conhecimento de qual foi o grupo que fez o pedido, sendo que de seguida muda o seu estado para **COOK** e dá *Up* no semáforo **orderReceived**, para alertar o empregado de mesa que recebeu o pedido.

3.2.1.2. Função **processOrder**

Função responsável por simular o processo de preparação e entrega da comida ao empregado de mesa por parte do chefe. Sendo assim, após a execução da função **usleep** (que é usada com o propósito de simular o tempo que o chefe demora a cozinhar), é feito um *Down* no semáforo **waiterRequestPossible**, com o efeito de, se possível, requisitar o empregado de mesa para o chefe lhe entregar a comida feita. Deste modo, dentro da zona crítica, o chefe avisa o empregado de mesa que a comida está pronta e qual o grupo que a tinha pedido, sendo que no final é realizado um *UP* no semáforo **waiterRequest**, de forma a avisar o empregado de mesa que tem um novo pedido.

Além de todo o processo descrito, ainda é alterado estado do chef para **WAIT_FOR_ORDER**.

3.2.2. Grupo

3.2.2.1. Função **checkInAtReception**

Sendo a primeira função a ser executada por cada um dos grupos, tem como finalidade simular o pedido de mesa. Deste modo, ainda antes de se dar entrada na região crítica, é efetuado um *DOWN* no semáforo **receptionistRequestPossible** para solicitar, se possível, o rececionista. De seguida, é alterado o estado do grupo para **ATRECEPTION**, antes ainda de se avisar o rececionista do pedido, aviso este que é efetuado através a operação *UP* sobre o semáforo **receptionistReq**.

Com o objetivo de se esperar pela mesa, no final desta função é ainda realizada a operação *DOWN* ao semáforo **waitForTable**.

3.2.2.2. Função **orderFood**

É nesta função que o grupo pede a comida ao empregado de mesa. Para isso, inicialmente o grupo convoca, caso seja possível, o empregado de mesa, através dum *DOWN* no semáforo **waiterRequestPossible** e, de seguida, dentro da zona crítica, muda o seu estado para **FOOD_REQUEST**. Seguidamente, indica ao empregado de mesa o seu desejo, pelo que no final faz um *UP* a **waiterRequest**, de forma a avisar-lo de que tem um novo pedido.

Por fim, é feito um *DOWN* em **requestReceived** com o intuito do grupo saber se o pedido já foi entregue ao chefe e, consequentemente, se pode esperar pela comida.

3.2.2.3. Função **waitFood**

Após o pedido do grupo ser entregue ao chefe, é alterado, na zona crítica, o estado do grupo para **WAIT_FOR_FOOD**, indicando obviamente, que o grupo está à espera da comida. Por esse mesmo motivo, é de seguida feito um *DOWN* no semáforo **foodArrived** da mesa onde esta entidade se encontra.

Quando eventualmente a refeição chega, é atualizado o estado do grupo para **EAT**.

3.2.2.4. Função **checkOutAtReception**

Finda a tarefa de comer, chega a altura do Grupo pedir a conta. Para tal é necessário chamar rececionista quando possível, recorrendo-se então a um *DOWN* no semáforo **receptionistRequestPossible.** Assim que este esteja disponível, é alterado o estado do Grupo para **CHECKOUT**, bem como são registados os dados do pedido que o grupo faz **BILLREQ**. De seguida, é feito um *UP* a **receptionistReq,** visando avisar o rececionista que tem este novo pedido. Da mesma maneira é realizado um *DOWN* no semáforo **tableDone** da mesa onde o grupo se encontra, para que este aguarde enquanto o rececionista resolve o pedido. Assim que tudo esteja tratado, é atualizado o estado do grupo para **LEAVING**.

3.2.3. Empregado de mesa

3.2.3.1. Função **waitForClientOrChef**

Tal como o nome indica, esta função é responsável por manter o empregado à espera de um pedido, quer por parte de um grupo, quer por parte do chefe.

Deste modo, inicialmente acede-se à zona crítica para alterar o estado do empregado para **WAIT_FOR_REQUEST**. Seguidamente, já fora da zona crítica, faz-se um *DOWN* no semáforo **waiterRequest**, mantendo assim esta entidade à espera de um pedido.

Assim que recebe um pedido, é novamente efetuada uma entrada na região crítica, de forma a serem registados os dados do mesmo (identificação do grupo ao qual o pedido se refere e o tipo deste), que posteriormente são retornados pela função.

Por fim é feito um *UP* em **waiterRequestPossible**, indicando que o empregado de mesa pode então ser novamente requisitado.

3.2.3.2. Função **informChef**

Esta função é executada caso o pedido tenha sido feito por parte de um grupo.

Inicialmente, começa-se por se aceder à zona crítica, onde o estado do empregado é alterado para INFORM_CHEF, bem como é feito o pedido ao chefe (realizando-se um *UP* ao semáforo **waitOrder**) para este preparar a refeição para o grupo identificado na função **waitForClientOrChef**.

Por último, faz-se um *DOWN* em **orderReceived**, para que o empregado de mesa espere pela confirmação de que o chefe recebeu o pedido e, por fim, um *UP* em **requestReceived** do respetivo grupo, informando-o de que o pedido foi recebido e que o chefe já foi informado.

3.2.3.3. Função **takeFoodToTable**

Já esta função é executada caso o pedido tenha sido feito por parte do chefe, significando isto que a refeição está pronta a ser servida. Desta forma, dentro da zona crítica, é alterado o estado do empregado de mesa para **TAKE_TO_TABLE** e é feito um *UP* no semáforo **foodArrived** do dado grupo, com o intuito de informar este de que a comida está pronta a ser servida.

3.2.4. Rececionista

3.2.4.1. Função **waitForGroup**

Função encarregada de colocar o rececionista em espera por um novo pedido por parte dum grupo.

Inicialmente, é feito um acesso à região crítica, de forma a alterar o estado desta entidade para **WAIT_FOR_REQUEST**. De seguida, já fora da zona crítica, é feito um *DOWN* no semáforo **receptionistReq**, de forma a que, efetivamente, o rececionista fique à espera de um grupo.

Assim sendo, quando o rececionista é requisitado por um grupo, volta a aceder à zona crítica, de forma a registar os dados do pedido deste, para que mais tarde seja retornado da função. Ainda antes de se sair da região crítica, é feito um *UP* em **receptionistRequestPossible**, indicando que o rececionista pode voltar a ser solicitado por um novo grupo.

3.2.4.2. Função **provideTableOrWaitingRoom**

Caso o pedido feito pelo grupo seja do tipo **TABLEREQ**, isto é, um pedido de mesa, esta é a função que é chamada.

Primeiramente, é alterado o estado do rececionista para **ASSIGNTABLE** dentro da região crítica**,** indicando que este está a tomar a seguinte decisão decisão: se fornece uma mesa ao grupo que fez o pedido, ou se o deixa em espera. Esta decisão está a cargo de uma outra função, **decideTableOrWait**, que usa um algoritmo relativamente simples para verificar se ainda há mesas disponíveis, de forma a que uma delas possa ser ocupada por um novo grupo.

Caso seja possível fornecer uma mesa ao grupo, é feito um *UP* no semáforo **waitForTable** do respetivo grupo, caso contrário o grupo é mantido em espera.

3.2.4.3. Função **receivePayment**

Função chamada no caso do pedido feito pelo grupo ser do tipo **BILLREQ**, ou seja, um pedido para pedir a conta.

Após se entrar na zona crítica, é atualizado o estado do rececionista para **RECVPAY,** indicando que este está a receber o pagamento da refeição. De seguida, dá-se um *UP* no semáforo **tableDone** da mesa onde o grupo se encontrava, permitindo informar este de que o pagamento foi recebido com sucesso e pode-se ir embora.

Não obstante, é ainda necessário verificar, no caso de ainda haverem grupos à espera, se a mesa que acabou de ser desocupada o pode ser novamente. Para isso, é mais uma vez alterado o estado do Rececionista para **ASSIGNTABLE** e, de seguida, faz-se uso da função **decideNextGroup**, de forma a decidir, caso seja possível, qual o próximo grupo a ocupar a dada mesa. Assim, no caso de ser

possível, dá-se *UP* no semáforo **waitForTable** do respetivo grupo, permitindo-lhe assim dirigir-se à mesa.

4. Testes realizados para validar a solução

Como é evidente, os primeiros testes que efetuamos para validar a nossa solução foram com o intuito de verificar as possíveis existências de *deadlocks*, sendo que para isso executamos, com o auxílio do script **run.sh** cedido pelo professor da disciplina, 100.000 vezes o código feito para 3 e 10 grupos, sendo que não obtivemos, em nenhum dos casos, qualquer *deadlock* durante estas execuções.

Numa segunda fase, precisamos de verificar se os *outputs* obtidos tinham lógica de acordo com o problema que nos foi proposto. Desta forma, para algumas execuções, analisamos "linha a linha" o resultado obtido, sendo que não verificamos qualquer anomalia. Contudo, chegamos à conclusão que poderiam existir certas situações em os resultados obtidos não iriam coincidir com aquilo que esperávamos, situações estas relativamente invulgares e que, sendo assim, seria praticamente impossível detectá-las "a olho" na ínfima quantidade de execuções que verificamos. Desta forma, decidimos implementar um mecanismo de verificação automática de erros, que será mais à frente explicado.

Com o intuito de tornar os testes feitos ao programa mais fluidos e eficientes para o programador, e tendo em conta que sempre que se quer compilar e correr um programa é necessário alternar entre os diretórios *run* e *src*, foi criado um *script* em *bash* capaz de fazer isso de forma automática. Este é intitulado de **autoCompile.sh** e encontra-se na diretoria *run* do projeto.

Assim, para o executar é necessário passar como argumentos:

- **um número inteiro**, que representa o número de *run*s que se deseja fazer
- **as entidades que se pretendem compilar**. Destas, podem-se escolher, embora de forma exclusiva:
    - **all_bin**, que indica que se pretende compilar todas as entidades fornecidas no pré-executável fornecido pelo docente*;*
    - **all**, que compila todas as entidades a ser alteradas pelos elementos do grupo;
    - **gr**, **wt**, **ch** ou **rt**, que compilam a respetiva entidade, juntamente com os pré-executáveis das 3 restantes.
- **a opção -e**, que indica que se pretendem realizar as condições de teste. Para isso, foram criados dois novos documentos, **logging2.c** (que contém a totalidade do código do documento **logging.c**, mas com os as verificações realizadas de forma automática e que cria o documento **errors.txt**, onde são apresentados todos os erros obtidos durante a execução) e **probDataStruct2.c** (que possui o código do documento **probDataStruct.c**, mas com a adição de variáveis de controlo de erros), sendo que quando esta opção é selecionada, estes dois documentos são renomeados para, respetivamente, **logging.c** e **probDataStruct.c**

É ainda possível, de todo o conjunto de entidades, escolher quais as várias que se querem compilar. Neste caso devem-se usar como argumentos *group*, *waiter*, *chef* e *receptionist*, podendo estes ser colocados, portanto, de maneira não exclusiva.

Por fim é importante notar que, seguindo as regras explicitadas acimas, é possível colocar os argumentos em qualquer ordem.

4.1. Breve explicação da verificação automática de erros e respetivos resultados

No já atrás referido documento **logging2.c** foram adicionadas várias verificações a serem realizadas cada vez que uma entidade atualiza o seu estado:

1. **Verificação dos intervalos dos valores obtidos**, onde é verificado se os valores dos estados de cada entidade, do número de grupos em espera e do índice da mesa atribuída a cada grupo se encontram dentro dos valores que podem possuir
2. **Verificação da continuidade dos valores dos estados dos grupos e do número de grupos em espera**, onde se verifica se os valores dos estados dos grupos e do número de grupos em espera não "dão saltos abruptos" em relação aos estados ou número de grupos em espera anteriores, isto é, se seguem uma certa continuidade.
3. **Verificação de se uma mesa foi atribuída a mais do que um grupo ao mesmo tempo**
4. **Verificação da lógica das interações entre os grupos e as restantes entidades**, onde se verifica a relação lógica e temporal que deve existir entre os estados dos grupos e das restantes entidades.

Todas estas verificações, ao serem executadas, não alertaram para a existência de qualquer erro, à exceção da de a que empregado de mesa levava a comida (estado **TAKE_TO_TABLE**) enquanto o grupo se encontrava à espera que esta chegasse (estado **WAIT_FOR_FOOD**). Por fim, chegámos á conclusão que este erro de lógica não é devido à má implementação do nosso código, mas à forma como o problema está construído. Tal conclusão é suportada pela observação deste mesmo "erro" nos pré-executáveis fornecidos pelo docente. Para tal observação, foi usada a seguinte linha de código de *bash*, que verifica que por vezes, o empregado de mesa leva a comida a um grupo sem este ainda se encontrar no estado **WAIT_FOR_FOOD**:

```
./autoCompile.sh 1000 all_bin | awk '{printf ("%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t\n", $2,
$4, $5, $6, $7, $8, $9, $10, $11, $12, $13)}' | grep -v '[4-7]' | awk '{print $1}' | grep 2
```

5. Considerações finais

De forma a evitar a “poluição do código” feito com verificações constantes de se cada semáforo está ou não inicializado, foi criado o documento **handle_semaphore_access.h**, onde foi introduzida a função **verifySemError**, que tem como intuito fazer a referida verificação para o semáforo passado por argumento.