Uma imagem com texto

Descrição gerada automaticamente

**TRABALHO 2**

**JANTAR DE AMIGOS**

Sistemas operativos 2022

Professor Nuno Lau e Professor Guilherme Campos

Trabalho realizado por:

João Nuno da Silva Luís (107403) | 50%

Diana Raquel Rodrigues Miranda (107457) | 50%

Índice

[Introdução 2](#_Toc123137709)

[Constantes e Semáforos 3](#_Toc123137710)

[Ciclo de vida - Client 4](#_Toc123137711)

[waitFriends() 4](#_Toc123137712)

[Orderfood() 4](#_Toc123137713)

[Waitfood() 5](#_Toc123137714)

[WaitAndPay() 5](#_Toc123137715)

[Ciclo de vida - Waiter 6](#_Toc123137716)

[waitForClientorChef() 6](#_Toc123137717)

[informChef() 6](#_Toc123137718)

[takeFoodToTable() 7](#_Toc123137719)

[receivePayment() 7](#_Toc123137720)

[Ciclo de vida - Chef 8](#_Toc123137721)

[waitForOrder() 8](#_Toc123137722)

[processOrder() 9](#_Toc123137723)

[Resultados 10](#_Toc123137724)

[Conclusão 12](#_Toc123137725)

[Bibliografia 13](#_Toc123137726)

# 

INDICE DE FIGURAS

# Introdução

Este trabalho é realizado no âmbito da disciplina Sistemas Operativos do 2º ano da Licenciatura em Engenharia Informática.

O seu objetivo é ter uma melhor compreensão dos mecanismos associados à execução e sincronização de processos e threads, através do adicionamento de código ao código fonte.

O tema deste trabalho é a gestão de 3 identidades num jantar de amigos, num restaurante.

Assim, existem três tipos de entidades: **clientes, waiter** e **chef**. Com estas entidades em vista, vamos então usar semáforos para controlar o fluxo de execução e o acesso à região da memória partilhada.

# Constantes e Semáforos

Para controlar cada uma das entidades envolvidas, foram usadas algumas constantes, que vão ser redefinidas ao longo do programa, e também impressas como *output* do programa.

Uma imagem com texto, interior

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamentePara além disto, foram definidos também alguns semáforos, para controlar o acesso de cada identidade à região de memória partilhada.

Figura 1- Constantes a usar pelo chef

Figura 2- Constantes a usar pelo waiter

Figura 3- Constantes a usar pelo cliente

Também criámos uma tabela para verificar o comportamento de cada entidade, isto é, onde e quando seria necessário ocorrer **ups** e **downs** de cada semáforo presente no ficheiro ***sharedDataSync.h,*** e as funções nelas envolvidas.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Semáforo* | Entidade  down | Função  down | Número de downs | Entidade  up | | Função  up | Número de ups |
| *Mutex* |  |  |  | |  |  |  |
| *Friends* |  |  |  | |  |  |  |
| *INGREDIENTS[ID]* |  |  |  | |  |  |  |
| *WAIT2INGS[ID]* |  |  |  | |  |  |  |

*Tab.1* Tabela com o comportamento dos semáforos.

# Ciclo de vida – Client

A entidade cliente é constituída por 20 clientes (amigos do problema), cada um com id próprio, que agora se vão juntar à mesa.

Para obedecermos ao que é proposto pelo problema, o primeiro amigo a chegar à mesa será o que vai fazer o pedido da comida, depois de todos os outros amigos já terem também chegado. Já o último amigo a chegar à mesa, será o responsável por fazer o pagamento da conta ao waiter.

## waitFriends()

Nesta função, os clientes, um a um e de forma aleatória, vão atualizar o seu estado para 2, correspodente ao estado WAIT\_FOR\_FRIENDS e vão incrementar o número de clientes presentes à mesa, na variável **sh->fSt.tableClients**. Depois vai ser feito o **Down** do semáforo **friendsArrived**, para que os amigos esperem uns pelos outros, sendo depois feito o desbloquamento quando o último amigo chegar.

No caso de ser o primeiro cliente, o id do mesmo vai ser guardado na constante **sh->fSt.tableFirst**, que depois vai ser impresso na tabela, na coluna 1st.

Já o último amigo vai ter também o seu id guardado na variável **sh->fSt.tableLast**, e vai também atualizar o seu estado logo para 4 (WAIT\_FOR\_FOOD), já que, como é o último, não vai fazer o pedido. Por último, dá **Up** a cada um dos amigos que estavam à espera que todos chegassem.

Uma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamente

Figura 4- Função waitFriends()

## orderFood()

Esta função vai ser apenas utilizada pelo primeiro cliente, que é quem faz o pedido da comida.

Nesta, o mesmo vai atualizar o seu estado para 3 (FOOD\_REQUEST) e vai ser feito o **Up** no semáforo **waiterRequest**, para que o waiter possa ser desbloqueado, que se encontra sempre bloqueado à espera de um pedido, ou do cliente, ou do chef.

No final da função, é feito o **Down** do semáforo **requestReceived**, e assim, o cliente que faz o pedido vai ficar bloqueado até receber uma resposta.

Uma imagem com texto

Descrição gerada automaticamente

Figura 5- Função orderFood()

## Uma imagem com texto Descrição gerada automaticamentewaitFood()

Nesta função, todos os clientes vão esperar que o pedido da comida seja feito pelo 1º amigo, atualizando o seu estado para 4 (WAIT\_FOR\_FOOD ). Após isso, ficam bloqueados até que a comida chegue, através de um **down** no semáforo **foodArrived**.

Quando a comida chegar após o aviso do waiter, os clientes vão então atualizar o seu estado para 5 (EAT).

Fig.3 Função Waitfood()

Figura 6- Função waitFood()

## waitAndPay()

Esta função começa por atualizar o estado dos clientes que, depois de terem comido, vão atualizar para 6 (WAIT\_FOR\_OTHERS), e incrementa o número de clientes que terminaram de comer, número que vai ser impresso no terminal na coluna FIE.

É também feita a atualização da variável **last**, e feito **Down** do semáforo **allFinished**, que faz com que os amigos esperem pelos restantes. Quando todos chegarem ao estado 6, vai ser feito o **Up** do semáforo **allFinished** para cada cliente, desbloqueando-os.

Uma imagem com texto

Descrição gerada automaticamente

Figura 7-Função waitAndPay()

No caso de o cliente ser o último, cuja *flag* foi atualizada anteriormente, este vai atualizar o seu estado para 7 (WAIT\_FOR\_BILL), e ativar a flag **sh->fSt.paymentRequest**, que vai ser utilizada na função *WaitForClientOrChef*() do waiter, para permitir a atualização da variável **ret**, que depois vai levar à chamada da função *receivePayment()* , também do waiter. Por fim, vai ser feito o **Up** do semáforo **waiterRequest**, que vai sinalizar o waiter para um pedido de pagamento. Após o waiter receber esse pedido, o último cliente vai também ser desbloqueado.

Uma imagem com texto

Descrição gerada automaticamente

Figura 8- Função waitAndPay()

Por fim, todos os clientes atualizam uma última vez o seu estado para 8 (FINISHED), concluindo-se assim o ciclo de vida do client, tendo todos os amigos acabado a sua refeição e tendo sido feito o pagamento

Uma imagem com texto

Descrição gerada automaticamente

Figura 9- Função waitAndPay()

# 

# Ciclo de vida – Waiter

A entidade waiter vai ser responsável por receber os pedidos de comida feitos pelo prmeiro cliente, levar esses pedidos ao chefe, trazer a comida pelo chefe cozinhada, e no fim, receber o pagamento feito pelo último cliente.

## waitForClientOrChef()

Esta função está dentro de um *while loop*, que a vai assim chamar 3 vezes, e serve para o waiter esperar por um pedido, ou do client ou do chef,

Assim, sendo o waiter comça por ter o seu estado inicializado a 0 (WAIT\_FOR\_REQUEST). De seguida, o semáforo **waiterRequest** vai dar **Down**, para bloquear o waiter, enquanto não houverem pedidos.

De seguida, são feitos 3 *if’s*, para que, havendo um pedido, o waiter possa saber para quem o encaminhar, ou seja, decidir que função chama a seguir. As *flags* usadas nos *if’s* vão sendo atualizadas ao longo dos ciclos de vida do cliente ou chef, e, para não haver problemas em posteriores chamadas da função, estas *flags* são colocadas novamente a 0.

Uma imagem com texto

Descrição gerada automaticamente

## 

Uma imagem com texto

Descrição gerada automaticamente

Figura 10- Função waitForClientOrChef

## Uma imagem com texto Descrição gerada automaticamenteinformChef()

Esta função vai ser executada pelo waiter quando recebe um pedido do cliente.

Assim, o waiter atualiza o seu estado para 1 (INFORM\_CHEF).

Para além disso e através do uso dos semáforos, é feito o desbloqueio do cliente que fez o pedido e o bloqueio do waiter, que espera um pedido vindo do waiter.

Agora, o chef irá proceder à preparação da comida, depois de ser chamada novamente a função *waitForClientOrChef()*, que vai bloquear novamente o waiter

Figura 11- Função informChef()

## takeFoodToTable()

Uma imagem com texto

Descrição gerada automaticamenteA função começa por desbloquear os clientes que estavam à espera que a comida chegasse, já que esta função é executada após o chef cozinhar a comida e desbloquear o próprio waiter.

Figura 12- Função takeFoodToTable

Como o waiter vai entregar a comida à mesa, vai também atualizar o seu estado para 2 (TAKE\_TO\_TABLE).

## receivePayment()

A última função do waiter recebe o pagamento da conta feito pelo último cliente a chegar ao restaurante, depois de este acabar a sua refeição. O waiter sinaliza o cliente que recebeu o pedido com sucesso, desbloqueando-o, através do **Up** do semáforo **requestReceived.**

Assim, é terminado o ciclo de vida do waiter

Uma imagem com texto

Descrição gerada automaticamente

Figura 13- Função receivePayment()

# Ciclo de vida - Chef

A entidade chef vai ser responsável por cozinhar o pedido do último cliente entregue pelo waiter.

## waitForOrder()

Uma imagem com texto

Descrição gerada automaticamenteEsta função faz o chef esperar por um pedido, encontrando-se bloquado.

Figura 14- Função waitForOrder()

Quando for o momento de cozinhar a *flag* **sh->fSt.foodOrder** vai ter o valor 1, proveniente do waiter, o que provoca o atualizar do estado do chefe para 1 (COOK).

## processOrder()

A última função do chef vai cozinhar o pedido e depois sinalizar o waiter que o mesmo está pronto e que o pode levar para a mesa.

O chef cozinha o pedido durante um tempo aleatório, calculado na função *usleep* (linha 147).

Uma imagem com texto

Descrição gerada automaticamenteDepois de cozinhar o pedido, o chef altera o valor da flag **sh->fSt.foodReady** para 1 e altera o seu estado para 2 (REST). Por fim, desbloqueia o waiter que se encontrava à espera de um pedido, fazendo **Up** do semáforo **waiterRequest.** Assim, é terminado o ciclo de vida do chef.

Figura 15- Função processOrder()

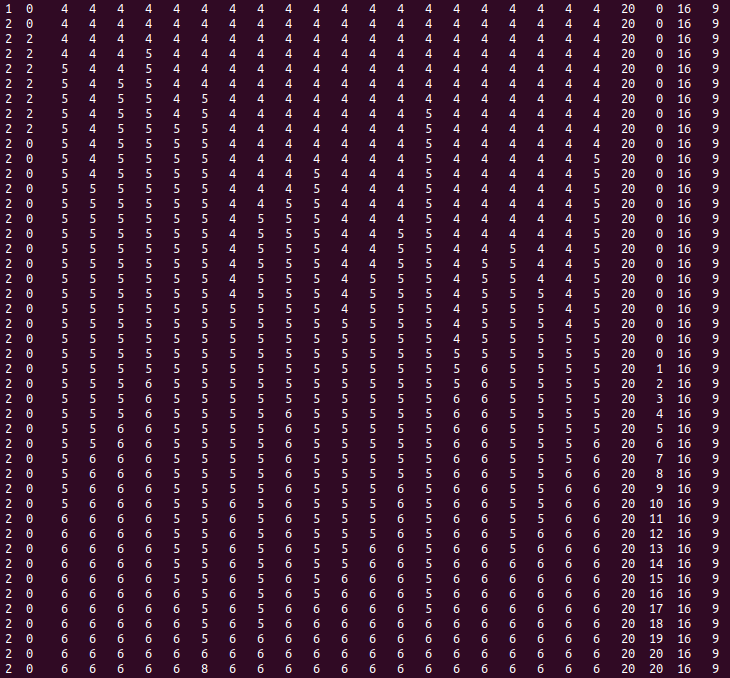
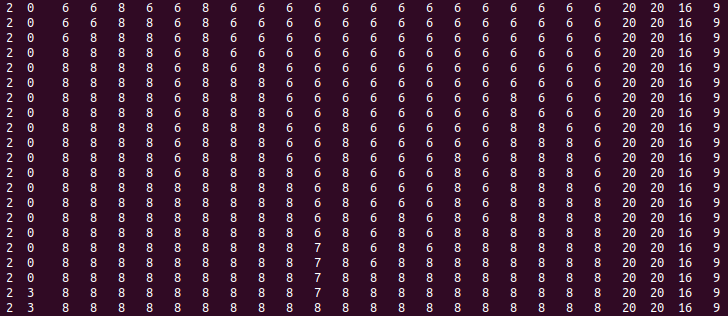
# Resultados

Para verificarmos o *output* do programa e se este estava correto, utilizámos partes de código pré-compliado fornecido pelo professor, através de comandos como *make ct* e *make ct\_wt*. Para verificarmos se a nossa solução acompanhava o expectável, fizemos também *make all\_bin*, para testar exclusivamente a solução fornecida.

Depois de serem resolvidos todos os problemas, fizemos *make all* e obtivemos o seguinte:

Uma imagem com texto, portão

Descrição gerada automaticamente



# Conclusão

Com este trabalho, pudemos aprofundar os nossos conhecimentos sobre o uso de semáforos e *flags* associadas. Percebemos melhor como usá-los e qual a sua relação com a área crítica.

As nossas maiores dificuldades foram perceber todas as funções, semáforos e *flags* necessárias, visto que todo o código teria que ter uma estrutura e um funcionamento correto. Para além disso, algumas vezes tivemos dificulades em perceber quando bloquear ou desativar certas identidades com os semáforos, mas com a criação da Tabela acima descrita foi mais fácil desenvolver o trabalho.

Assim, concluimos que chegámos ao objetivo do trabalho, cumprindo as orientações do problema fornecido com sucesso.

# Bibliografia

* Slides teóricos
* Aula prática nº 10