



universidade
de aveiro



Trabalho 2

Jantar de Amigos (Restaurant)

Licenciatura em Engenharia Informática
Sistemas Operativos

Docente:

Professor Nuno Lau

Alunos:

Bárbara Nóbrega Galiza – 105937

João Miguel Dias Andrade - 107969

Dezembro 2022

Conteúdo

1	Introdução	2
2	Abordagem	3
2.1	Client	3
2.1.1	waitFriends()	4
2.1.2	orderFood()	4
2.1.3	waitFood()	5
2.1.4	waitAndPay()	5
2.2	Waiter	6
2.2.1	waitForClientOrChef()	6
2.2.2	informChef()	7
2.2.3	takeFoodToTable()	7
2.2.4	receivePayment()	7
2.3	Chef	7
2.3.1	waitForOrder()	8
2.3.2	processOrder()	8
2.4	Funcionamento Geral dos Semáforos	9
3	Testes	9
3.1	Teste de <i>deadlocks</i>	10
3.2	Análise detalhada de uma das Runs	11
4	Conclusão	14
5	Referências Bibliográficas	15

1 Introdução

Esse trabalho tem como objetivo a compreensão dos mecanismos associados à execução e sincronização de processos através de uma simulação de um jantar entre amigos em um restaurante. Nesse jantar, cada indivíduo é representado por um processo, com suas ações, estados e ciclo de vida específicos. A fim de que a simulação corra sem problemas, ou seja, de que os processos sejam sincronizados corretamente, são usados 6 semáforos distintos para cada situação. Além disso, é feita a gestão da região de memória partilhada entre os processos através de um semáforo que serve como *mutex*.

Nesse relatório, será descrita a solução implementada, *i.e.* os blocos de códigos inseridos onde nos foi assinalado. Em adição, serão apresentados os testes realizados para validar a solução.

2 Abordagem

A abordagem ao problema será apresentada em três blocos, correspondentes a cada entidade do problema:

1. Client
2. Waiter
3. Chef

Conforme nos foi atribuído, cada entidade possui um ficheiro C próprio, no qual estão definidas suas funções com código a completar. Nessa secção, será explicada a lógica por trás de todo o código introduzido nas regiões assinaladas. Além disso, no fim, será apresentada uma tabela acerca do funcionamento de cada semáforo.

2.1 Client

A entidade Client representa os clientes/amigos que irão jantar no restaurante. Nesse problema, são lançados $\text{TABLESIZE} = 20$ processos Client, cada um com seu ciclo de vida independente dos outros. Cada Client possui os estados:

1. INIT
2. WAIT_FOR_FRIENDS
3. FOOD_REQUEST
4. WAIT_FOR_FOOD
5. EAT
6. WAIT_FOR_OTHERS
7. WAIT_FOR_BILL
8. FINISHED

E o ciclo de vida descrito pelas funções:

- `void travel (int id);`
- `bool waitFriends(int id);`

- `void orderFood (int id)`¹;
- `void waitFood (int id)`;
- `void eat (int id)`;
- `void waitAndPay (int id)`;

Das quais `eat()` e `travel()` são as únicas que já estavam completas e por isso não serão discutidas em detalhe.

2.1.1 `waitFriends()`

A função `waitFriends()` tem como objetivo garantir que, como descrito no problema, os clientes só possam efetuar o pedido ao Waiter após todos chegarem à mesa, sendo o tempo até a chegada definido aleatoriamente para cada Client a partir da função `travel()`. Por isso, é necessário que, ao chegar à mesa incompleta, cada Client bloqueie e fique à espera do último.

Primeiramente, para alterar os estados e gerir o desbloqueio, dentro do *mutex* mudamos o estado de cada client, que estava em INIT, para WAIT_FOR_FRIENDS, e atualizamos o valor da variável `tableClients`, que conta quantos clientes já chegaram. A seguir, testamos se o valor de `tableClients` era igual ao tamanho da mesa, a fim de verificar se o Client em questão era o último a chegar. Caso verdade, o id atual é guardado como o id do último, e é executado um ciclo for de 1 ao tamanho da mesa. Isso servirá para desbloquear todos os clientes que ficarão à espera quando fizerem **DOWN** depois de saírem do *mutex*, ou seja, para cada cliente que estará bloqueado no semáforo FRIENDSARRIVED. Além disso, servirá para evitar que o último cliente entre em *deadlock*: é feito um **UP** a mais do que o número de clientes bloqueados, para que o último cliente não bloqueie ao chegar no **DOWN**. Ainda dentro do *mutex*, fizemos um *if* para salvar o id do primeiro a chegar na mesa na memória partilhada e para mudar o valor da variável *first* (inicializada como *false* fora da região crítica) para *true*, a fim de informar a próxima função, `orderFood()`, que o processo em questão corresponde ao primeiro cliente. Depois, salvamos o estado geral para computar as mudanças feitas. Como dito anteriormente, ao sair do *mutex*, cada cliente faz **DOWN** e fica à espera que o último cliente os desbloqueie. Quando isso acontece, cada um termina a execução da presente função.

2.1.2 `orderFood()`

A próxima função, `orderFood()`, é executada apenas pelo primeiro cliente, pois segundo as regras do problema, apenas ele faz o pedido ao Waiter. Por isso, a função anterior deverá retornar o valor *true* para que esta seja chamada.

¹Essa função só é executada pelo primeiro cliente a chegar à mesa

O objetivo dessa função é fazer a comunicação com o Waiter, e para isso utiliza os semáforos WAITERREQUEST e REQUESTRECEIVED e a *flag* foodRequest, que também serão manipulados pelo Waiter.

Primeiramente, dentro do *mutex*, alteramos o estado do primeiro cliente para FOOD_REQUEST, mudamos o valor da *flag* foodRequest para 1 e salvamos o estado. O valor da *flag* a 1 serve para informar o Waiter que o pedido em questão é um pedido de comida, pois existem três tipos de pedido distintos, como será visto na seção 2.2. Em seguida, na região de saída, é feito **UP** no WAITERREQUEST para desbloquear o Waiter, que estava bloqueado à espera de algum pedido, e **DOWN** no REQUESTRECEIVED, o que bloqueia o Client até que o Waiter atenda o seu pedido.

2.1.3 waitFood()

Em seguida, a função waitFood() simula a espera do Client pela comida. Nela, mudamos o estado do Client para WAIT_FOR_FOOD e salvamos, dentro do primeiro *mutex*. Depois, é feito **DOWN** no semáforo FOODARRIVED, para bloquear o Client até que o Waiter traga a comida à mesa (o que é representado por um **UP**). Após ser desbloqueado pelo Waiter, o Client entra novamente no *mutex*, e lá mudamos o estado para EAT e o salvamos.

2.1.4 waitAndPay()

Finalmente, na função waitAndPay(), o Client espera os outros Clients terminarem de comer, e, caso ele seja o último a chegar na mesa, pede e paga a conta. Para que os Clients sejam desbloqueados após todos terminarem de comer, é necessário que o último Client a terminar desbloqueie todos os outros. Nessa função, assim como na função waitFriends(), existe uma variável booleana (*last*) para verificar se o Client em questão é o último a chegar na mesa.

Primeiro, *last* é inicializado com o valor *false*. Depois, ao entrar no primeiro *mutex*, mudamos o estado para WAIT_FOR_OTHERS e incrementamos o campo tableFinishEat. Quando tableFinishEat for igual ao tamanho da mesa, sabemos que se trata do último Client a terminar de comer. Por isso, semelhante ao que fizemos na função waitFriends(), fazemos **UP** do semáforo ALLFINISHED 20 vezes: 19 para desbloquear todos os Clients que estavam bloqueados (vão ser bloqueados fora do *mutex*) e 1 para que o último não bloqueie no **DOWN**. Além disso verificamos se o Client em questão é o último a chegar na mesa, e atualizamos o valor da variável *last* caso seja verdade. Por fim, salvamos o estado.

Seguidamente, como mencionado, fizemos **DOWN** do semáforo ALLFINISHED em cada processo Client, que irão ser desbloqueados pelo código introduzido no *mutex* anterior.

Já para o pedido e pagamento da conta, introduzimos código dentro de um *if* já dado previamente, que testa o valor da variável *last*. Caso verdade, se trata do último a chegar à mesa e conseqüentemente o Client que irá efetuar o pagamento, e por isso, dentro de um

mutex atualizamos seu estado para `WAIT_FOR_BILL`, alteramos o valor da flag `paymentRequest` para 1 (para informar o Waiter), e salvamos o estado. Na região de saída, fizemos **UP** do `WAITERREQUEST`, para desbloquear o Waiter, e **DOWN** do `REQUESTRECEIVED`, para ficar bloqueado à espera do Waiter.

Por fim, já fora do *if*, entramos novamente em uma região crítica na qual mudamos o estado para `FINISHED` e salvamos. Essa porção de código é executada logo após todos terminarem de comer para os Clients que não são o último a chegar à mesa, e após o Waiter fazer **UP** do `REQUESTRECEIVED` para o último a chegar à mesa. Por isso, no resultado impresso no terminal, os Clients que não são o último já vão para o estado 8 (`FINISHED`) mesmo enquanto o último espera a conta no estado 7 (`WAIT_FOR_BILL`).

2.2 Waiter

O Waiter é a entidade responsável por receber o pedido dos clientes e levá-lo ao chefe, levar a comida à mesa e receber o pagamento da refeição. O seu ciclo de vida baseia-se em esperar pelos clientes ou pelo chefe e fazer a comunicação entre estes. O Waiter tem os seguintes estados:

1. `WAIT_FOR_REQUEST`
2. `INFORM_CHEF`
3. `TAKE_TO_TABLE`
4. `RECEIVE_PAYMENT`

Sendo o seu ciclo de vida descrito pelas funções:

- `int waitForClientOrChef();`
- `void informChef();`
- `void takeFoodToTable();`
- `void receivePayment();`

2.2.1 `waitForClientOrChef()`

A função `waitForClientOrChef()` é a função "principal" do waiter, onde é decidido o que este irá fazer. Nesta, o Waiter muda o seu estado para `WAIT_FOR_REQUEST`, onde espera ser chamado ou pelo Chef ou pelo Cliente que queira fazer o pedido ou pagar. A função é corrida 3 vezes e dependendo do que retornar, uma das outras 3 funções será chamada.

Primeiramente, alteramos o estado do Waiter para `WAIT_FOR_REQUEST` dentro do *mutex* e, seguidamente, fora do *mutex*, fazemos **DOWN** ao semáforo `WAITERREQUEST`, o que fará com que o Waiter espere que este seja posto **UP** pelo Chef ou por um dos Clients.

Quando é feito **UP** do semáforo `WAITERREQUEST`, é também mudada uma *flag* que indica ao Waiter a próxima ação que tem de tomar. Caso seja o primeiro Client a chamar o Waiter para fazer o pedido, este irá mudar a *flag* `foodRequest` para 1 e a função irá então dar reset a esta *flag* e retornar o valor `FOODREQ`, fazendo com que o Waiter vá para a função `informChef()`. No caso do Chef, este altera o valor da *flag* `foodReady`, fazendo com que o Waiter execute a função `takeFoodToTable()`. O último caso é quando o último Client chama o Waiter para pagar a conta, para isso altera a *flag* `paymentRequest` fazendo o Waiter executar a função `receivePayment()`. Toda esta parte de verificação e alteração do valor das *flags* é feita dentro do *mutex*, pois acede à região crítica.

2.2.2 `informChef()`

A função `informChef()`, como o nome diz, tem como função informar o Chef sobre o pedido que foi feito e, além disso, avisar o Client que o pedido já foi efetuado. É bastante simples pois apenas é alterado o estado do Waiter e é feito **UP** a dois semáforos.

Primeiramente, é alterado o estado do Waiter para `INFORM_CHEF` dentro do *mutex* e seguidamente, fora do *mutex*, é feito **UP** ao semáforo `REQUESTRECEIVED` que libera o Client que estava à espera que o seu pedido fosse efetuado. Seguidamente, fazemos **UP** do semáforo `WAITORDER`, o que permite ao Chef começar a cozinhar.

2.2.3 `takeFoodToTable()`

A função `takeFoodToTable()` simplesmente, dentro do *mutex*, altera o estado do Waiter para `TAKE_TO_TABLE` e dá **UP** ao semáforo `FOODARRIVED` 20 vezes, o que faz com que os Clients possam começar a comer.

2.2.4 `receivePayment()`

Finalmente, a função `receivePayment()` acaba o ciclo de vida do Waiter. Nela, este recebe o pagamento e permite que os Clients se vão embora. Inicialmente, dentro da zona crítica, atualiza o seu estado para `RECEIVE_PAYMENT` e faz **UP** do `REQUESTRECEIVED` para desbloquear o Client que estava à espera para pagar.

2.3 Chef

O Chef é a entidade mais simples, pois apenas tem duas funções. Este é responsável por receber o pedido do Waiter e cozinhar a comida, e tem os seguintes estados:

1. WAIT_FOR_ORDER
2. COOK
3. REST

Sendo o seu ciclo de vida descrito pelas funções:

- `void waitForOrder();`
- `void processOrder();`

2.3.1 `waitForOrder()`

A função `waitForOrder()` é a primeira a ser executada e é responsável por fazer o Chef esperar até o Waiter lhe trazer o pedido. Inicialmente, é feito um **DOWN** no semáforo WAITORDER que é responsável por fazer o Chef esperar. Quando o Waiter entrega o pedido ao Chef, na função `informChef()`, este faz **UP** ao semáforo WAITORDER o que desbloqueia o Chef. Após ser desbloqueado e dentro do *mutex*, o Chef altera o seu estado para COOK.

2.3.2 `processOrder()`

Depois da função `waitForOrder()`, é executada a função `processOrder()`, onde o Chef cozinha e entrega a comida ao Waiter. Primeiramente, é feito um `sleep`, que representa o tempo que o Chef demora a cozinhar. O tempo de `sleep` é calculado fazendo $(MAXCOOK * random()) / RAND_MAX + 100.0$, sendo $MAXCOOK = 3000000$, `random()` uma função que retorna um `long` com valor entre 0 e $2^{31}-1$ e `RAND_MAX` uma constante que representa o maior valor que a função `random()` pode devolver ($2^{31}-1 = 2147483647$). Esta equação devolve sempre um valor entre 0 e 1. Depois de a comida estar preparada, a função entra na zona crítica, onde trocamos o estado do Chef para REST e alteramos a *flag* `foodReady` para 1, o que sinaliza ao Waiter que a comida está pronta. Para além disso, fora do *mutex*, também damos **UP** ao semáforo WAITERREQUEST que desbloqueia o Waiter e permite que ele venha buscar a comida para levar aos Clients.

2.4 Funcionamento Geral dos Semáforos

Semáforo	Quem espera?	Função	Quem faz UP?	Função	N UPs
FRIENDSARRIVED	Client	waitFriends()	Last Client	waitFriends()	20
REQUESTRECEIVED	Client	orderFood() waitAndPay()	Waiter	informChef receivePayment()	2
FOODARRIVED	Client	waitFood()	Waiter	takeFoodToTable()	20
ALLFINISHED	Client	waitAndPay()	Client (último cliente a terminar de comer)	waitAndPay()	20
WAITERREQUEST	Waiter	waitForClient-OrChef()	First Client, Chef e Last Client	processOrder(), orderFood() e waitAndPay()	3
WAITORDER	Chef	waitForOrder()	Waiter	informChef()	1

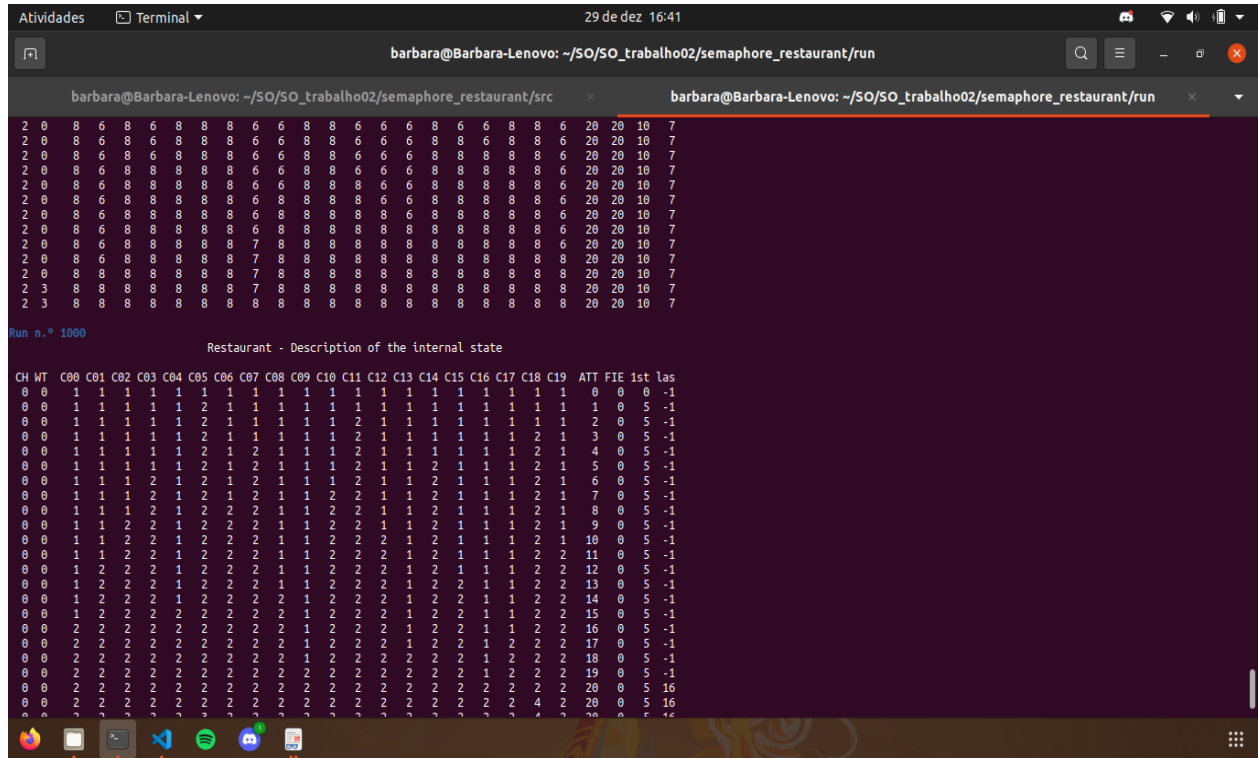
3 Testes

A fim de validar nossa solução, realizamos dois procedimentos:

1. Corremos o script `run.sh` sem argumentos, ou seja, deixamos correr 1000 vezes o programa, para assim assegurar de que nenhuma vez ocorria *deadlocks*. Como o script correu as 1000 vezes e terminou normalmente, concluímos que não há *deadlocks* de fato.
2. Analisamos a saída, linha por linha, de 10 "runs" distintas para verificar que os estados estavam sendo atualizados conforme as regras do problema. Na seção **3.2** é analisada uma dessas "runs" em detalhe. Além disso, comparamos essas saídas com as saídas da solução nos dada através dos ficheiros "_bin" (`make all_bin`), o que mostrou que ambas as soluções produziam resultados semelhantes e nos permitiu concluir que nossa solução está correta.

3.1 Teste de *deadlocks*

O script termina normalmente, o que implica que não houve *deadlock* em nenhuma das "runs":



```
barbara@Barbara-Lenovo: ~/SO/SO_trabalho02/semaphore_restaurant/run
barbara@Barbara-Lenovo: ~/SO/SO_trabalho02/semaphore_restaurant/src
2 0 8 6 8 6 8 8 8 6 6 6 8 8 6 6 6 8 6 6 8 8 6 20 20 10 7
2 0 8 6 8 6 8 8 8 6 6 6 8 8 6 6 6 8 8 6 8 8 6 20 20 10 7
2 0 8 6 8 6 8 8 8 6 6 6 8 8 6 6 6 8 8 8 8 8 6 20 20 10 7
2 0 8 6 8 8 8 8 8 6 6 6 8 8 6 6 6 8 8 8 8 8 6 20 20 10 7
2 0 8 6 8 8 8 8 8 6 6 6 8 8 8 6 6 8 8 8 8 8 6 20 20 10 7
2 0 8 6 8 8 8 8 8 6 6 8 8 8 8 6 6 8 8 8 8 8 6 20 20 10 7
2 0 8 6 8 8 8 8 8 6 6 8 8 8 8 8 6 8 8 8 8 8 6 20 20 10 7
2 0 8 6 8 8 8 8 8 6 6 8 8 8 8 8 8 8 8 8 8 8 6 20 20 10 7
2 0 8 6 8 8 8 8 8 6 6 8 8 8 8 8 8 8 8 8 8 8 6 20 20 10 7
2 0 8 8 8 8 8 8 8 7 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 10 7
2 0 8 8 8 8 8 8 8 7 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 10 7
2 3 8 8 8 8 8 8 8 7 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 10 7
2 3 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 10 7
Run n.º 1000
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 1 1 1 1 1 1 1 1 1 1 0 0 0 -1
0 0 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 5 -1
0 0 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 0 5 -1
0 0 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 3 0 5 -1
0 0 1 1 1 1 1 2 1 2 1 1 1 2 1 1 1 1 1 1 1 4 0 5 -1
0 0 1 1 1 1 1 2 1 2 1 1 1 2 1 1 1 1 1 2 1 5 0 5 -1
0 0 1 1 1 2 1 2 1 2 1 1 1 2 1 1 1 2 1 1 2 6 0 5 -1
0 0 1 1 1 2 1 2 1 2 1 1 1 2 2 1 1 2 1 1 2 7 0 5 -1
0 0 1 1 1 2 1 2 2 2 1 1 2 2 1 1 2 1 1 1 2 8 0 5 -1
0 0 1 1 2 2 1 2 2 2 1 1 2 2 1 1 2 1 1 1 2 9 0 5 -1
0 0 1 1 2 2 1 2 2 2 1 1 2 2 2 1 2 1 1 1 2 10 0 5 -1
0 0 1 1 2 2 1 2 2 2 1 1 2 2 2 1 2 1 1 1 2 11 0 5 -1
0 0 1 2 2 2 1 2 2 2 1 1 2 2 2 1 2 1 1 1 2 12 0 5 -1
0 0 1 2 2 2 1 2 2 2 1 1 2 2 2 1 2 2 1 1 2 13 0 5 -1
0 0 1 2 2 2 1 2 2 2 1 1 2 2 2 1 2 2 1 1 2 14 0 5 -1
0 0 1 2 2 2 2 2 2 2 1 2 2 2 1 2 2 1 1 2 15 0 5 -1
0 0 2 2 2 2 2 2 2 2 1 2 2 2 1 2 2 1 1 2 16 0 5 -1
0 0 2 2 2 2 2 2 2 2 1 2 2 2 1 2 2 1 2 2 17 0 5 -1
0 0 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 1 2 2 18 0 5 -1
0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 19 0 5 -1
0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 20 0 5 16
0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 20 0 5 16
0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 20 0 5 16
```

Figura 1: Captura de tela que mostra que o script chegou até a Run 1000.

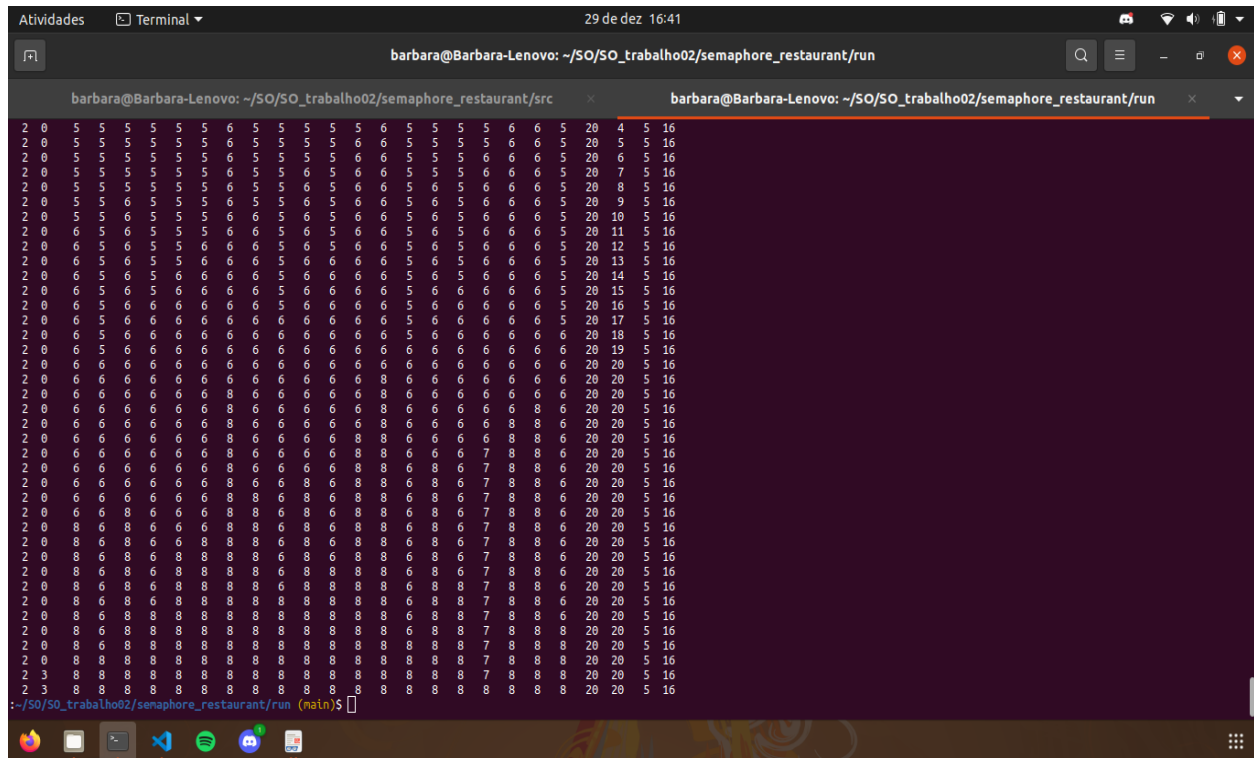


Figura 2: Captura de tela que mostra que o script terminou normalmente após a run 1000.

3.2 Análise detalhada de uma das Runs

O programa arranca com os estados iniciais corretos, 0 para o Waiter e Chef e 1 para todos os 20 Clients. A medida que os Clients vão chegando, seus estados mudam para 2 (WAIT_FOR_FRIENDS), e o campo ATT, que computa o número de Clients na mesa é incrementado de um em um. Além disso, nota-se que quando chega o primeiro Client (o 11 no caso da "run" mostrada na figura 3), o campo 1st é atualizado para 11. Assim que o último cliente chega à mesa, o campo ATT fica com 20 e o campo "las" fica com o último cliente a chegar à mesa, nesse caso o 15.

Então, na linha imediatamente abaixo desta, o primeiro Client 11 muda, e somente ele, para o estado 3 (FOOD_REQUEST), como era esperado. Os outros Clients, na medida que vão sendo desbloqueados, mudam para o estado 4 (WAIT_FOR_FOOD).

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

Figura 3: Captura de tela que mostra a parte inicial da saída do programa, descrita acima.

A seguir, uma (nesse caso em específico, pois podem ser mais de uma) linha depois de todos os Clients fora o 1st mudarem para o estado 4 (WAIT_FOR_FOOD), o Waiter muda seu estado para 1 (INFORM_CHEF), como deve ser. Poucas linhas depois, o 1st muda também para o estado 4, após ser desbloqueado pelo Waiter. Em seguida, o Chef muda para 1 (COOK) e o programa faz uma pequena pausa, que corresponde ao sleep realizado pelo Chef a fim de simular o tempo que este leva para cozinhar. Logo após cozinhar, o Chef muda seu estado para 2 (REST), no qual permanece até o fim do programa, e, quase de imediato, o Waiter muda para 2 (TAKE_TO_TABLE) e permanece nesse estado durante algum tempo. Nesse período, os Clients vão mudando os seus estados para 5 (EAT), e depois de algum tempo para 6 (WAIT_FOR_OTHERS), e na medida que mudam para 6, o campo FIE que indica quantos clientes já terminaram de comer vai sendo incrementado um a um.

5 Referências Bibliográficas

CAMPOS, Guilherme - Guiões práticos para a disciplina Sistemas Operativos, 2022-2023

LAU, Nuno - Guiões teóricos para a disciplina Sistemas Operativos, 2022-2023