Sistemas Operativos 2022/2023

Jantar De Amigos

Professor Nuno Lau

José Mendes, 107188
Bernardo Pinto 105926

Índice

Introdução

Este segundo trabalho prático, “Jantar de Amigos”, da unidade curricular de Sistemas Operativos, possui como objetivo a compreensão dos mecanismos associados à execução e sincronização de processos e *threads*, através do desenvolvimento de uma aplicação em C que simula o funcionamento de um restaurante, tendo como ponto de partida o código fonte disponível na página da disciplina.

O “restaurante” possui várias regras que indicam o funcionamento e a ordem de execução do mesmo, pelo que devem ser cumpridas para a realização deste trabalho prático.

Conhecendo o problema

Como dito anteriomente, o problema consiste na implementação de uma aplicação para simular o funcionamento de um restaurante em linguagem C .

No código fornecido já estão definidos os espaços que devem ser preenchidos com a implementação realizada pelo grupo, para que, dessa forma, seja obtida uma implementação completa. Além disso, as regiões críticas (zonas de acesso à memória partilhada) estão indicadas e devem ser acessadas através da utilização do semáforo *mutex.*

O restaurante é constituído por:

- Um grupo de amigos (vários *client*)
- Um empregado de mesa (*waiter*)
- Um cozinheiro (*chef*)

O trabalho prático tem regras que devem ser seguidas com o objetivo de garantir um correto funcionamento do mesmo:

1. O primeiro *client* a chegar ao restaurante realiza o pedido da comida, mas somente após todos chegarem.
2. O empregado de mesa é responsável por levar o pedido ao cozinheiro e trazer a comida para a mesa quando esta estiver pronta.
3. Ao terminarem de comer os amigos apenas devem abandonar a mesa depois de todos também terminarem.
4. O último a chegar ao restaurante tem a responsabilidade de pagar a conta.
5. O tamanho da mesa corresponde ao número de amigos.
6. Os clientes, o empregado e o cozinheiro são processos independentes.
7. Todos os processos são criados no início do programa e mantêm-se em execução a partir desta altura.
8. Assume-se que os clientes demoram algum tempo a chegar ao restaurante.
9. Os processos devem estar activos apenas quando for necessário, devendo bloquear sempre que têm de esperar por algum evento.

Ficheiro probConst.h :

Inicialmente, foi disponibilizado, através do ficheiro *probConst.h*, os parâmetros iniciais do problema, entre eles, a capacidade da mesa, o tempo máximo para comer e o tempo máximo para cozinhar.

```
/** \brief table capacity, equal to number of clients */
#define  TABLESIZE       20
/** \brief controls time taken to eat */
#define  MAXEAT       500000
/** \brief controls time taken to cook */
#define  MAXCOOK     3000000
```

Figura 1 - Estados gerais

Em seguida, ainda no mesmo ficheiro, foram declarados os estados possíveis para os 3 processos, que serão responsáveis por identificar a função que a entidade está a realizar no momento.

```
/* Client state constants */

/** \brief client initial state */
#define  INIT             1
/** \brief client is waiting for friends to arrive at table */
#define  WAIT_FOR_FRIENDS  2
/** \brief client is requesting food to waiter */
#define  FOOD_REQUEST      3
/** \brief client is waiting for food */
#define  WAIT_FOR_FOOD     4
/** \brief client is eating */
#define  EAT               5
/** \brief client is waiting for others to finish */
#define  WAIT_FOR_OTHERS   6
/** \brief client is waiting to complete payment */
#define  WAIT_FOR_BILL     7
/** \brief client finished meal */
#define  FINISHED          8
```

Figura 2 - Estados do client

```
/* Chef state constants */

/** \brief chef waits for food order */
#define  WAIT_FOR_ORDER    0
/** \brief chef is cooking */
#define  COOK              1
/** \brief chef is resting */
#define  REST              2
```

Figura 3 - Estados do chef

```c
/* Waiter state constants */

/** \brief waiter waits for food request */
#define  WAIT_FOR_REQUEST   0
/** \brief waiter takes food request to chef */
#define  INFORM_CHEF        1
/** \brief waiter takes food to table */
#define  TAKE_TO_TABLE      2
/** \brief waiter reiceives payment */
#define  RECEIVE_PAYMENT    3
```

Figura 4 - Estados do waiter

Ficheiro sharedDataSync.h:

O código referente à memória partilhada e aos índices do array de semáforos, podem ser encontrados no ficheiro *sharedDataSync.h*. Nessa estrutura são fornecidos 7 semáforos:

```c
typedef struct
        { /** \brief full state of the problem */
          FULL_STAT fSt;

          /* semaphores ids */
          /** \brief identification of critical region protection semaphore – val = 1 */
          unsigned int mutex;
          /** \brief identification of semaphore used by clients to wait for friends to arrive - val = 0 */
          unsigned int friendsArrived;
          /** \brief identification of semaphore used by client to wait for waiter after a request – val = 0 */
          unsigned int requestReceived;
          /** \brief identification of semaphore used by clients to wait for food – val = 0 */
          unsigned int foodArrived;
          /** \brief identification of semaphore used by clients to wait for friends to finish eating – val = 0  */
          unsigned int allFinished;
          /** \brief identification of semaphore used by waiter to wait for requests – val = 0  */
          unsigned int waiterRequest;
          /** \brief identification of semaphore used by chef to wait for order – val = 0  */
          unsigned int waitOrder;

        } SHARED_DATA;

/** \brief number of semaphores in the set */
#define SEM_NU              (7)

#define MUTEX                 1
#define FRIENDSARRIVED        2
#define REQUESTRECEIVED       3
#define FOODARRIVED           4
#define ALLFINISHED           5
#define WAITERREQUEST         6
#define WAITORDER             7
```

Figura 5 - Declaração dos semáforos

Semáforos:

	Descrição	UP	DOWN
Mutex	Define a entrada e saída da região crítica.	Usado para sair da região crítica.	Usado para entrar na região crítica.
friendsArrived	Usado pelo cliente para esperar a chegada dos amigos	Feito pelo cliente quando todos os amigos já chegaram	Feito pelo cliente quando ainda tem amigos por chegar
requestReceived	Usado pelo cliente para aguardar o waiter após um pedido	Feito pelo waiter após concluir uma solicitação	Feito pelo cliente quando fica à espera do waiter
foodArrived	Usado pelo cliente para esperar pela comida	Feito pelo cliente após realizar um pedido	Feito pelo waiter quando entrega o pedido
allFinished	Usado pelo cliente para esperar que todos acabem de comer	Feito pelo cliente quando todos já terminaram de comer	Feito pelo cliente quando ainda existem amigos comendo
waiterRequest	Usado pelo waiter para esperar por pedidos	Feito pelo cliente ou pelo chef para chamar o waiter	Feito pelo waiter para sinalizar que está disponível para receber pedidos
waitOrder	Usado pelo chef para esperar por um pedido	Feito pelo waiter quando realiza um pedido ao chef	Feito pelo chef quando se encontra disponível para fazer o pedido

A partir desses semáforos, é possível que todos os processos “comuniquem entre si” e troquem informações entre eles. Com isso os estados das entidades mantêm-se sempre em atualização.

Implementação da solução

semSharedMemChef.c

A primeira entidade a ser implementada foi o *chef*. O grupo decidiu que a melhor forma de começar seria com a entidade mais simples e uma vez que esta apenas interage com o empregado de mesa, *waiter*, foi a escolhida. Por sua vez permitiu uma melhor compreensão do funcionamento dos semáforos e do programa como um todo.

Função waitForOrder()

Esta primeira função tem como objetivo fazer com que o *chef* aguarde por um pedido de comida efetuado pelo *waiter*. Deve também alterar o estado interno do *chef*, guardando--o, e alterar a *flag* que indica que este já recebeu o pedido do *waiter*.

Na primeira zona de inserção de código, ainda fora da região crítica, é feito um **down** do semáforo **waitOrder**, uma vez que o *chef* está disponível para receber pedidos por parte do *waiter*.

```
/* insert your code here */
    if(semDown(semgid, sh->waitOrder) == -1) { // o chef encontra se disponivel para fazer o pedido
        perror("error on the up operation for semaphore access (PT)");
        exit(EXIT_FAILURE);
    }
```

Figura 6 - Primeira inserção da função waitForOrder()

Em seguida, já dentro da região crítica, o *chef* altera o seu estado interno para **COOK**, uma vez que este se encontra a preparar a comid. Guarda o seu estado e atualiza a *flag* **foodOrder** para o valor 0, pois o pedido do *waiter* já foi recebido.

```
/* insert your code here */
sh->fSt.st.chefStat = COOK; // o chef esta a cozinhar
saveState(nFic, &sh->fSt); // guarda os estados alterados
sh->fSt.foodOrder = 0; // ja recebeu o pedido do waiter
```

Figura 7 - Segunda inserção da função waitForOrder()

Função processOrder()

Esta segunda função tem como objetivo principal entregar o pedido de comida ao *waiter*. O *chef* demora algum tempo a cozinhar, tarefa realizada através de uma função previamente fornecida. De seguida, indica, através de uma *flag,* que a comida está pronta para ser recolhida pelo *waiter*, e por fim, descansa, alterando o seu estado interno e guardando-o.

Inicialmente, já dentro da região crítica, encontra-se a primeira zona de inserção, em que o *chef* deve alterar a *flag* **foodReady** para 1, indicando que a comida está pronta para recolha e alterar o seu estado interno para **REST**, uma vez que após ter cozinhado não é mais necessário, guardando em último lugar o estado.

```
/* insert your code here */
sh->fSt.foodReady = 1; // a comida está pronta
sh->fSt.st.chefStat = REST; // ja nao há pedido --> chef descança
saveState(nFic, &sh->fSt); // guarda os states
```

Figura 8 - Primeira inserção da função processOrder()

Por fim, fora de região crítica, é necessário chamar o *waiter* para que este possa realizar a entrega. Isto é feito realizando um **up** no semáforo **waiterRequest**.

```
/* insert your code here */
if (semUp (semgid, sh->waiterRequest) == -1) {
 /* exit critical region */
    perror ("error on the up operation for semaphore access (PT)");
    exit (EXIT_FAILURE);
}
```

Figura 9 - Segunda inserção da função processOrder()

semSharedMemClient.c

Após a realização do primeiro processo, o grupo obteve uma melhor compreensão do funcionamento dos semáforos e da forma como eles interagem, pelo que, foi decidido que o próximo a ser implementado seria o *client*, o processo principal.

Função waitFriends(int id)

A função *waitFriends*, que retornará um valor booleano, tem como parâmetro a variável *id,* que vai armazenar um valor inteiro responsável pela distinção entre os diferentes clientes. Primeiramente, agora já dentro do escopo da função, foi instanciada uma variável booleana **first** com valor inicial *false,* e, em seguida, o código entra na região crítica com o **down** do semáforo **mutex**.

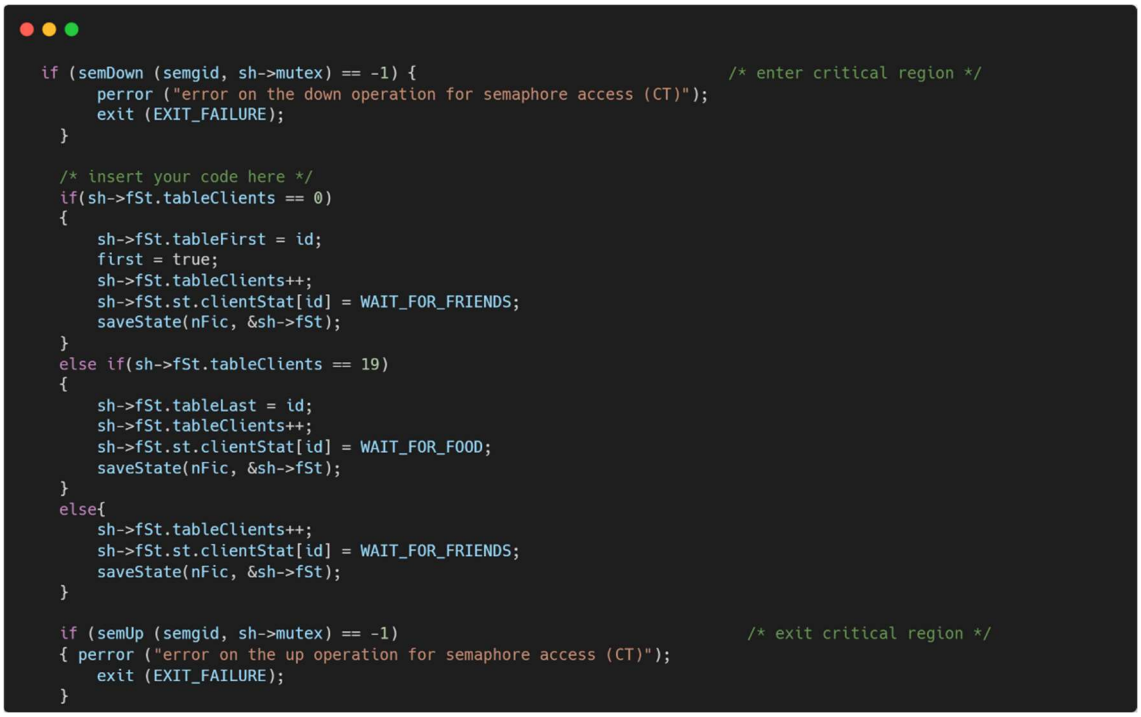
Uma vez dentro da região crítica, foi implementado um bloco condicional que será responsável por verificar se o *client* (i.e., *id* atual) é o primeiro ou o último a chegar e por atualizar o estado do *client*. Consequentemente, foram desenvolvidas as seguintes estruturas:

- O *client* é o primeiro a chegar *(i.e., sh->fSt.tableClients == 0):*
    - *Id* é armazenado na variável **tableFirst**
    - Variável *first* recebe o valor *true*
    - Número de clientes presentes na mesa é incrementado em uma unidade
    - Id adota o estado **WAIT_FOR_FRIENDS**

- O *client* é o último a chegar *(i.e., sh->fSt.tableClients == 19)*
    - *Id* é armazenado na variável **tableLast**
    - Número de clientes presentes na mesa é incrementado em uma unidade
    - Id adota o estado **WAIT_FOR_FOOD**

- O *client* não é o primeiro nem o último:
    - Número de clientes presentes na mesa é incrementado em uma unidade
    - *Id* adota o estado ***WAIT_FOR_FRIENDS***

Vale ressaltar que no final de todas as opções do bloco condicional o estado interno é guardado, e, em seguida, é efetuada a saída da zona crítica, através do **up** do semáforo **mutex**.



```
if (semDown (semgid, sh->mutex) == -1) {                              /* enter critical region */
      perror ("error on the down operation for semaphore access (CT)");
      exit (EXIT_FAILURE);
  }

  /* insert your code here */
  if(sh->fSt.tableClients == 0)
  {
      sh->fSt.tableFirst = id;
      first = true;
      sh->fSt.tableClients++;
      sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;
      saveState(nFic, &sh->fSt);
  }
  else if(sh->fSt.tableClients == 19)
  {
      sh->fSt.tableLast = id;
      sh->fSt.tableClients++;
      sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
      saveState(nFic, &sh->fSt);
  }
  else{
      sh->fSt.tableClients++;
      sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;
      saveState(nFic, &sh->fSt);
  }

  if (semUp (semgid, sh->mutex) == -1)                                 /* exit critical region */
  { perror ("error on the up operation for semaphore access (CT)");
      exit (EXIT_FAILURE);
  }
```

Figura 10 - Primeira inserção da função waitFriends()

Em sequência, foi implementado um loop *while* que realiza continuamente o **down** do semáforo **friendsArrived** enquanto a mesa ainda não estiver completa. Quando, por fim, a mesa se encontra completa, fora do ciclo é realizado o **up** desse semáforo.

Finalmente, é realizado o *return* do valor da variável booleana *first*, que será utilizada como condição para a execução da função *orderFood.*

```
/* insert your code here */
    while(sh->fSt.tableClients != 20){
        if (semDown (semgid, sh->friendsArrived) == -1)
        { perror ("error on the down operation for semaphore access (CT)");
            exit (EXIT_FAILURE);
        }
    }
    if (semUp (semgid, sh->friendsArrived) == -1)
    { perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    return first;
```

Figura 11 - Segunda inserção da função waitFriends()

Função orderFood(int id)

Inicialmente, é válido ressaltar que, considerando que uma das regras do restaurante é o primeiro a chegar realizar o pedido da comida, a função **orderFood** é executada apenas para esse primeiro cliente.

A primeira implementação dentro da função é um **down** no semáforo **mutex**, para que seja realizada a entrada na região crítica, e, em sequência, o estado do *client* representado pelo *id* passa para **FOOD_REQUEST** e a *flag* **foodRequest** recebe o valor 1, que indica que o *client* realizou um pedido ao *waiter*. Então, após as alterações serem salvas, é feita a saída da região crítica utilizando o **up** do **mutex**.

```
if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.st.clientStat[id] = FOOD_REQUEST; // esta a pedir comida
    sh->fSt.foodRequest = 1;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1)                                /* exit critical region */
    { perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
```

Figura 12 - Primeira inserção da função orderFood()

Em seguida, fora da região crítica, foi desenvolvido um ciclo *while* que é percorrido até a execução de um *break*. Dentro deste, foi implementado um outro ciclo, desta vez um *for,* responsável por verificar se todos os clientes, com exceção do último a chegar, já estão no estado **WAIT_FOR_FOOD**, e, em caso positivo, é feito o *break* do ciclo.

```
    /* insert your code here */

    int i, count;
    while(1){
        count = 0;
        for(i = 0; i < TABLESIZE; i++){
            if(sh->fSt.st.clientStat[i] == WAIT_FOR_FOOD)
                count++;
        }
        if(count == 19)
            break;
    }
```

Figura 13 - Segunda inserção da função orderFood()

Posteriormente, é feito o **up** do semáforo **waiterRequest**, que representa o momento que o *client* chama o *waiter*, e o **down** do semáforo **requestReceived**, que representa o momento que o *client* passa a ficar à espera do *waiter*.

```
if (semUp (semgid, sh->waiterRequest) == -1)
   { perror ("error on the up operation for semaphore access (CT)");
       exit (EXIT_FAILURE);
   }

if (semDown (semgid, sh->requestReceived) == -1)
   { perror ("error on the down operation for semaphore access (CT)");
       exit (EXIT_FAILURE);
   }
```

Figura 14 - terceira inserção da função orderFood()

Por fim, foi construído um ciclo *while* que fica em execução até que a *flag* **foodOrder** receba o valor 0, que representa o momento em que o *chef* começou a cozinhar. Então, após isso, o estado do *client* representado pelo *id,* passado como parâmetro da função (i.e *first),* é posto em **WAIT_FOR_FOOD**, tal como os restantes.

```
while(1){
        if(sh->fSt.foodOrder == 0)
            break;
    }

    sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
    saveState(nFic, &sh->fSt);

}
```

Figura 15 - Quarta inserção da função orderFood()

Função waitFood()

Nesta função, o *client* espera que a comida chegue, atualizando assim o seu estado. Após a chegada da comida, este deve, novamente, atualizar o seu estado. O estado interno deve, portanto, ser guardado duas vezes.

Na primeira zona de inserção de código, dentro da região crítica, é necessário atualizar o estado interno do *client* para **WAIT_FOR_FOOD**, com exceção do primeiro *client* a chegar à mesa, **tableFirst**, pois este tem de pedir a comida, e do último a chegar á mesa, **tableLast**, pois este já é colocado neste estado anteriormente. Deve, também, ser guardado o estado interno.

```
/* insert your code here */
if(id != sh->fSt.tableLast && id != sh->fSt.tableFirst){
  sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
  saveState(nFic, &sh->fSt);
}
```

Figura 16 - Primeira inserção da função waitFood()

Posteriormente, é necessário que o *client* espere que todos, incluíndo o que faz o pedido da comida, **tableFirst**, estejam com o estado interno **WAIT_FOR_FOOD**, para poder realizar um **up** no semáforo **foodArrived**. Em último lugar, é necessário esperar que a comida chegue à mesa para poder avançar.

```
/* insert your code here */
while(1){
    int count = 0;
    for(int i = 0; i < TABLESIZE; i++){
        if(sh->fSt.st.clientStat[i] == WAIT_FOR_FOOD)
            count++;
    }
    if(count == TABLESIZE)
        break;
}

if (semUp (semgid, sh->foodArrived) == -1) {
    perror ("error on the down operation for semaphore access (CT)");
    exit (EXIT_FAILURE);
}

while(1)
    if(sh->fSt.st.waiterStat == TAKE_TO_TABLE)
        break;
```

Figura 17 - Segunda inserção da função waitFood()

Por fim, na última zona de inserção de código desta função, o estado do *client* é alterado para **EAT**, indicando que este já está a comer, e o seu estado é guardado pela segunda vez nesta função.

```
/* insert your code here */
sh->fSt.st.clientStat[id] = EAT;
saveState(nFic, &sh->fSt);
```

Figura 18 - Terceira inserção da função waitFood()

Função waitAndPay()

Nesta função, o *client* atualiza o seu estado e espera que todos os outros acabem de comer antes de sair do restaurante. Por sua vez, o último *client* a acabar a refeição deve avisar os outros que todos já acabaram de comer. O último *client* a chegar à mesa, **tableLast**, deve, como anteriormente referido, pagar a conta, chamando o *waiter* e esperando que este chegue. O estado interno deve ser guardado duas vezes.

Primeiramente, na primeira zona de inserção de código, é necessário incrementar a variável **tableFinishEat**, que indica quantas pessoas das que estão na mesa já terminaram a sua refeição. Por sua vez, o estado do *client*, é atualizado para **WAIT_FOR_OTHERS**, indicando que está à espera que o resto dos clientes acabem de come. Guardando, por fim, o seu estado interno.

```
/* insert your code here */
sh->fSt.tableFinishEat++;
sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
saveState(nFic, &sh->fSt);
```

Figura 19 - Primeira inserção da função waitAndPay()

Na segunda zona de inserção de código, fora da região crítica, o *client* deve esperar que todos os outros terminem as suas refeições para poder avançar. Tal é feito dando um **down** ao semáforo **tableFinishedEat** enquanto ainda há clientes a comer, indicando que ainda não pode avançar, e um **up** a este mesmo semáforo quando todos os clientes acabaram de comer.

```
/* insert your code here */
while(sh->fSt.tableFinishEat != 20){
    if (semDown (semgid, sh->allFinished) == -1)

    { perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}
last = true;
if (semUp (semgid, sh->allFinished) == -1)
{ perror ("error on the up operation for semaphore access (CT)");
    exit (EXIT_FAILURE);
}
```

Figura 20 - Segunda inserção da função waitAndPay()

Seguidamente, dentro da região crítica, caso o *client* seja o que tem de efetuar o pagamento, **tableLast**, este atualiza o seu estado para **WAIT_FOR_BILL**, indicando que está à espera do *waiter* chegar para poder pagar a conta, caso contrário altera o seu estado para **FINISHED**.

```
/* insert your code here */
if(id == sh->fSt.tableLast)
{
    sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
    saveState(nFic, &sh->fSt);
}
else{
    sh->fSt.st.clientStat[id] = FINISHED;
    saveState(nFic, &sh->fSt);
}
```

Figura 21 - Terceira inserção da função waitAndPay()

Posteriormente, na próxima inserção de código, o *client* espera que todos menos o que tem de efetuar o pagamento terminem (estado **FINISHED**) para que finalmente o pagamento possa ser feito. Quando isto acontece, é realizado um **up** no semáforo **waiterRequest** com a variável **paymentRequest** com o valor 1, significando que este quer pagar a conta. Após o pagamento, ou seja, quando o *waiter* tem o estado **RECEIVE_PAYMENT**, é possível avançar.

```
/* insert your code here */
while(1)
{
    int count = 0;
    for(int i = 0; i < TABLESIZE; i++){
        if(sh->fSt.st.clientStat[i] == FINISHED)
            count++;
    }
    if(count == TABLESIZE-1){
        break;
    }
}
sh->fSt.paymentRequest = 1;
if (semUp (semgid, sh->waiterRequest) == -1) {       // chama o waiter

    perror ("error on the up operation for semaphore access (CT)");
    exit (EXIT_FAILURE);
    }
}
while(1)
{
    if(sh->fSt.st.waiterStat == RECEIVE_PAYMENT)
        break;
}
```

Figura 22 - Quarta inserção da função waitAndPay()

Em último lugar, após tudo isto ter terminado, é necessário que o estado do *client* que efetuou o pagamento, **tableLast**, seja atualizado para **FINISHED** tal como o dos outros clientes, sendo por fim guardado o seu estado.

```
/* insert your code here */
if(id == sh->fSt.tableLast)
{
    sh->fSt.st.clientStat[id] = FINISHED;
    saveState(nFic, &sh->fSt);
}
```

Figura 23 - Quinta inserção da função waitAndPay()

semSharedMemWaiter.c

O *waiter* foi a última entidade a ser realizada pelo grupo, uma vez que, apesar de possuir uma mais fácil implementação comparando com o *client*, este comunica com os outros dois processsos. O *client* e o *chef* apenas comunicam com o *waiter*. Logo, concluímos que seria melhor deixar este para último.

Função waitForClientOrChef():

A função **waitForClientOrChef** inicia com a entrada na zona crítica, através do **down** do semáforo **mutex**, para que o estado do waiter seja atualizado para **WAIT_FOR_REQUEST**, que irá representar que o *waiter* está à espera de um pedido de comida, e, portanto, é realizada a saída da zona crítica do programa.

```
if (semDown (semgid, sh->mutex) == -1)  {                              /* enter critical region */
       perror ("error on the up operation for semaphore access (WT)");
       exit (EXIT_FAILURE);
   }

   /* insert your code here */
   sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
   saveState(nFic, &sh->fSt);

   if (semUp (semgid, sh->mutex) == -1)      {                            /* exit critical region */
       perror ("error on the down operation for semaphore access (WT)");
       exit (EXIT_FAILURE);
   }
```

Figura 24 - Primeira inserção da função waitForClientOrChef()

Em seguida, foi implementado o **down** do semáforo **waiterRequest** para que, dessa forma, o *waiter* sinalize que está disponível para receber solicitações.

```
/* insert your code here */
   if (semDown (semgid, sh->waiterRequest) == -1) {
       perror ("error on the down operation for semaphore access (WT)");
       exit (EXIT_FAILURE);
   }
```

Figura 25 - Segunda inserção da função waitForClientOrChef()

Posteriormente, novamente dentro da região crítica, foi desenvolvido um bloco condicional que testa se 3 flags estão ativas ou não, que são:

- fSt.foodRequest == 1:
    - **foodRequest** passa a 0
    - **ret** recebe **FOODREQ**
- fSt.foodReady == 1:
    - **foodReady** passa a 0
    - **ret** recebe **FOODREADY**
- paymentRequest == 1:
    - **paymentRequest** passa a 0
    - **ret** recebe **BILL**

Por fim, a função *waitForClientOrChef* retorna o valor da variável inteira **ret**.

```
if (semDown (semgid, sh->mutex) == -1)  {                          /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

    if(sh->fSt.foodRequest == 1){
        sh->fSt.foodRequest = 0;
        ret = FOODREQ;
    }
    if(sh->fSt.foodReady == 1){
        sh->fSt.foodReady = 0;
        ret = FOODREADY;
    }
    if(sh->fSt.paymentRequest == 1){
        sh->fSt.paymentRequest = 0;
        ret = BILL;
    }

    if (semUp (semgid, sh->mutex) == -1) {                         /* exit critical region */
     perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    return ret;
```

Figura 26 - Terceira inserção da função waitForClientOrChef()

Função informChef():

Inicialmente, é realizada a entrada na região crítica do programa para que, assim, o estado do *waiter* seja alterado para **INFORM_CHEF**, que representa o momento que o *waiter* informa o *chef* do pedido que recebeu. Em seguida a *flag* **foodOrder** recebe o valor 1 e é feita a saída da região crítica.

```
 if (semDown (semgid, sh->mutex) == -1)  {                          /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.st.waiterStat = INFORM_CHEF;
    sh->fSt.foodOrder = 1;
    saveState(nFic, &sh->fSt);



if (semUp (semgid, sh->mutex) == -1)                               /* exit critical region */
    { perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
```

Figura 27 - Primeira inserção da função informChef()

Por fim, é realizado o **up** tanto do semáforo **waitOrder** como do **requestReceived** que representam, respetivamente, que o *waiter* realizou um pedido ao *chef* e que o *waiter* concluiu uma solicitação.

```
    /* insert your code here */

    if (semUp (semgid, sh->waitOrder) == -1)
    { perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    if (semUp (semgid, sh->requestReceived) == -1)
    { perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
```

Figura 28 - Segunda inserção da função informChef()

Função takeFoodToTable ():

O desenvolvimento da função *takeFoodToTable* inicia com a atualização do estado do *waiter* para **TAKE_TO_TABLE**, que representa o momento no qual a comida é levada à mesa. Após as informações serem salvas, é feito o **down** do semáforo **foodArrived**, que significa que o pedido foi entregue. Vale ressaltar que toda a implementação dessa função é realizada dentro da região crítica do programa.

```
 if (semDown (semgid, sh->mutex) == -1)  {                          /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &sh->fSt);

    if (semDown (semgid, sh->foodArrived) == -1)  {
     perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    if (semUp (semgid, sh->mutex) == -1)  {                          /* exit critical region */
     perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
```

Figura 29 - Primeira inserção da função takeFoodTable()

Função receivePayment():

A função receivePayment, que também foi alterada apenas dentro da região crítica, começa por alterar o estado do waiter para **RECEIVE_PAYMENT** e, em seguida, a *flag* **paymentRequest** recebe o valor 0, para que assim represente o momento no qual o empregado de mesa recebeu o pagamento.

```
if (semDown (semgid, sh->mutex) == -1)  {                          /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
    sh->fSt.paymentRequest = 0;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1)  {                        /* exit critical region */
     perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
```

Figura 30 - Primeira inserção da função receivePayment()

.

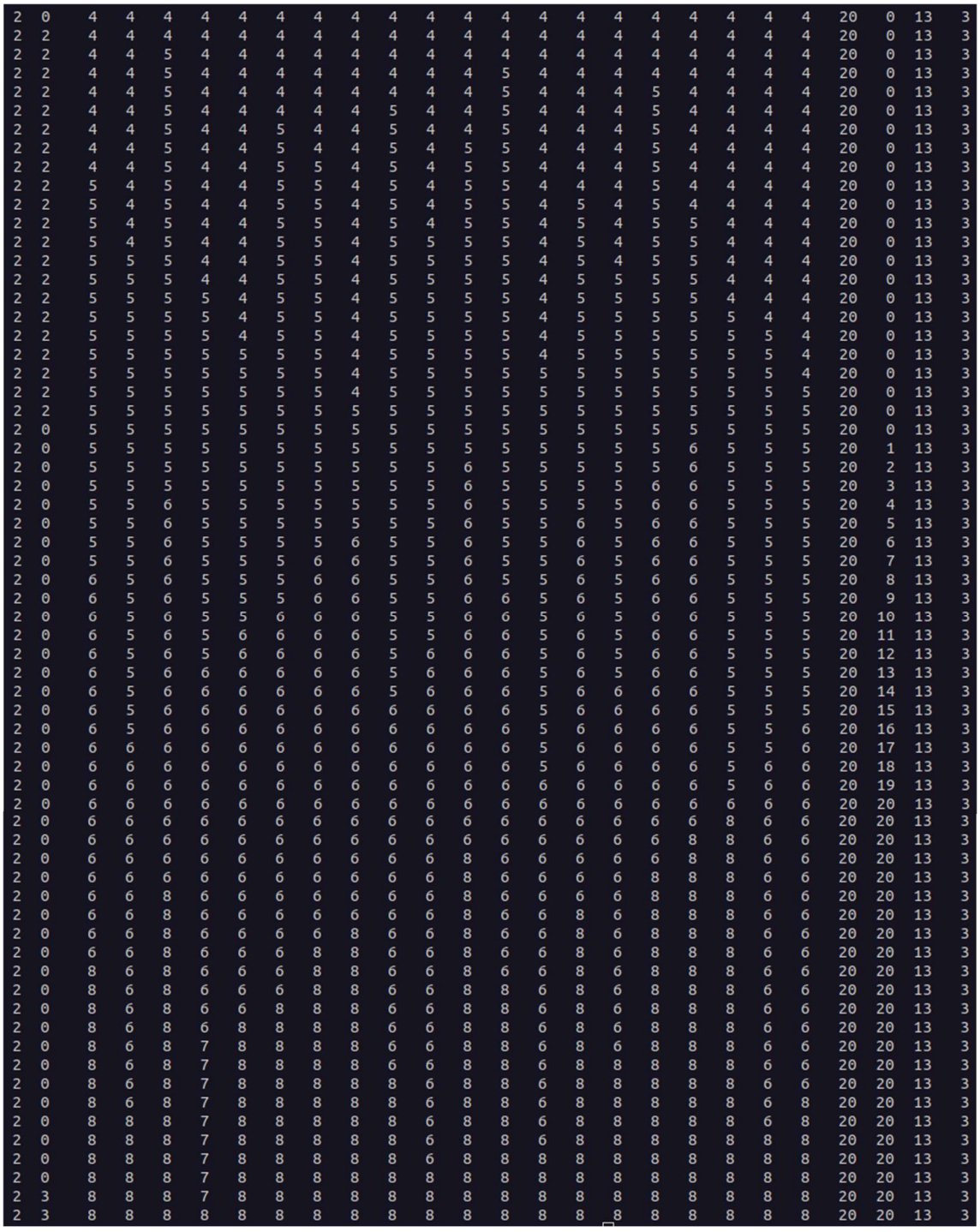
Resultados obtidos

Para ser possível verificar que esta implementação era a correta, foi testada várias vezes ao longo do seu desenvolvimento. Estes testes foram realizados utilizando o programa *makefile* (make all) previamente fornecido que se encontra no diretório *src* para obter o ficheiro com a solução, sendo os testes realizados com o programa *probSemSharedMemRestaurant* (./probSemSharedMemRestaurant) que se encontra no diretório *run*. Uma vez que o resultado obtido é sempre diferente, pois a ordem pela qual os clientes interagem nunca é a mesma, o grupo concluiu que seria necessário realizar vários testes. Para verificar que o programa não possui *deadlocks*, este também foi executado usando o comando *./run* que executa a solução mil vezes.

```
endes@mendesze:~/Desktop/SO/Projeto_2/semaphore_restaurant/run$ ./probSemSharedMemRestaurant
                        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   1    0   0   0  -1
 0  0    1   1   1   1   1   1   1   1   1   1   1   1   1   2   1   1   1   1   1   1    1   0  13  -1
 0  0    1   1   2   1   1   1   1   1   1   1   1   1   1   2   1   1   1   1   1   1    2   0  13  -1
 0  0    1   1   2   1   1   1   1   1   1   1   1   1   1   2   1   1   2   1   1   1    3   0  13  -1
 0  0    1   1   2   1   1   2   1   1   1   1   1   1   1   2   1   1   2   1   1   1    4   0  13  -1
 0  0    1   1   2   1   1   2   1   1   1   1   1   1   1   2   1   1   2   1   1   2    5   0  13  -1
 0  0    1   2   2   1   1   2   1   1   1   1   1   1   1   2   1   1   2   1   1   2    6   0  13  -1
 0  0    1   2   2   1   1   2   1   1   2   1   1   1   1   2   1   1   2   1   1   2    7   0  13  -1
 0  0    1   2   2   1   1   2   1   1   2   1   1   1   1   2   1   2   2   1   1   2    8   0  13  -1
 0  0    1   2   2   1   2   2   1   1   2   1   1   1   1   2   1   2   2   1   1   2    9   0  13  -1
 0  0    2   2   2   1   2   2   1   1   2   1   1   1   1   2   1   2   2   1   1   2   10   0  13  -1
 0  0    2   2   2   1   2   2   1   1   2   1   2   1   1   2   1   2   2   1   1   2   11   0  13  -1
 0  0    2   2   2   1   2   2   1   1   2   1   2   1   1   2   1   2   2   2   1   2   12   0  13  -1
 0  0    2   2   2   1   2   2   2   1   2   1   2   1   1   2   1   2   2   2   1   2   13   0  13  -1
 0  0    2   2   2   1   2   2   2   1   2   2   2   1   1   2   1   2   2   2   1   2   14   0  13  -1
 0  0    2   2   2   1   2   2   2   1   2   2   2   1   2   2   1   2   2   2   1   2   15   0  13  -1
 0  0    2   2   2   1   2   2   2   1   2   2   2   2   2   2   1   2   2   2   1   2   16   0  13  -1
 0  0    2   2   2   1   2   2   2   1   2   2   2   2   2   2   1   2   2   2   2   2   17   0  13  -1
 0  0    2   2   2   1   2   2   2   2   2   2   2   2   2   2   1   2   2   2   2   2   18   0  13  -1
 0  0    2   2   2   1   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   19   0  13  -1
 0  0    2   2   2   4   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   20   0  13   3
 0  0    2   2   2   4   2   2   2   2   2   2   2   2   2   3   2   2   2   2   2   2   20   0  13   3
 0  0    2   2   4   4   2   2   2   2   2   2   2   2   2   3   2   2   2   2   2   2   20   0  13   3
 0  0    2   2   4   4   2   2   2   2   2   2   2   2   2   3   2   2   4   2   2   2   20   0  13   3
 0  0    2   2   4   4   2   4   2   2   2   2   2   2   2   3   2   2   4   2   2   2   20   0  13   3
 0  0    2   2   4   4   2   4   2   2   2   2   2   2   2   3   2   2   4   2   2   4   20   0  13   3
 0  0    2   4   4   4   2   4   2   2   2   2   2   2   2   3   2   2   4   2   2   4   20   0  13   3
 0  0    2   4   4   4   2   4   2   2   4   2   2   2   2   3   2   2   4   2   2   4   20   0  13   3
 0  0    2   4   4   4   2   4   2   2   4   2   2   2   2   3   2   4   4   2   2   4   20   0  13   3
 0  0    2   4   4   4   4   4   2   2   4   2   2   2   2   3   2   4   4   2   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   2   2   4   2   2   2   2   3   2   4   4   2   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   2   2   4   2   4   2   2   3   2   4   4   2   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   2   2   4   2   4   2   2   3   2   4   4   4   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   2   4   2   4   2   2   3   2   4   4   4   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   2   4   4   4   2   2   3   2   4   4   4   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   2   4   4   4   2   4   3   2   4   4   4   2   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   2   4   4   4   4   4   3   2   4   4   4   4   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   4   4   4   4   4   4   3   2   4   4   4   4   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   4   4   4   4   4   4   3   4   4   4   4   4   4   20   0  13   3
 0  1    4   4   4   4   4   4   4   4   4   4   4   4   4   3   4   4   4   4   4   4   20   0  13   3
 0  0    4   4   4   4   4   4   4   4   4   4   4   4   4   3   4   4   4   4   4   4   20   0  13   3
 1  0    4   4   4   4   4   4   4   4   4   4   4   4   4   3   4   4   4   4   4   4   20   0  13   3
 1  0    4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   20   0  13   3
```

Figura 31 - Output da solução (1/2)



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

Figura 32 - Output da solução (2/2)

Ao executar o programa, a solução aparece em forma de tabela, em que cada coluna representa algo diferente:

- **Coluna 1**: Corresponde ao estado em que o *chef* se encontra;
- **Coluna 2**: Corresponde ao estado em que o *waiter* se encontra;
- **Coluna 3-22**: Corresponde ao estado em que cada *client*, pertencente ao grupo de amigos, se encontra;
- **Coluna 23**: Corresponde ao número de clientes que se encontra na mesa;
- **Coluna 24**: Corresponde ao número de clientes que já terminou de comer e está à espera que os outros também terminam
- **Coluna 25**: Corresponde ao id do primeiro *cliente* que chega à mesa
- **Coluna 26**: Corresponde ao id do último *cliente* que chega à mesa

Através desta tabela e da variação de estados que a mesma apresenta, é possível perceber se os resultados obtidos estão corretos ou não. Analisando os resultados, verifica-se que:

Inicialmente, o *chef* e o *waiter* encontram se no estado 0 e os clientes no estado 1, indicando que os dois primeiros estão à espera de receber ordens para poderem agir e que os clientes se encontram no seu estado inicial. Em seguida, os clientes vão se sentando na mesa, um de cada vez, alterando o seu estado para 2, **WAIT_FOR_FRIEND**, indicando que esse *client* está à espera dos outros. À medida a que isto acontece, a coluna número 3 vai sendo incrementada. O primeiro *client* a alterar o seu estado para dois é, portanto, guardado na coluna número 25 através do seu id. Quando todos os clientes se encontram no estado 2 com a exceção de um, este último altera o seu estado diretamente para o estado 4, **WAIT_FOR_FOOD**, pois uma vez que todos os clientes já chegaram, este agora espera pela chegada da comida. Este também tem o seu id guardado na coluna número 26, sinalizando que é o último *client* a chegar à mesa.

Seguidamente, o primeiro *client* a chegar à mesa altera o seu estado para 3, **FOOD_REQUEST**, uma vez que, como visto anteriormente, este tem de efetuar o pedido de comida ao *waiter*. Os restantes clientes vão alterando o seu estado para 4, **WAIT_FOR_FOOD**, indicando que estão à espera de que a comida chegue. Quando todos, com a exceção do primeiro, se encontram no estado 4, vai ser efetuado o pedido de comida. O *waiter* atualiza o seu estado para 1, **INFORM_CHEF**, uma vez que tem de avisar o *chef* para este começar a preparar a comida, e logo de seguida altera novamente para 0, **WAIT_FOR_REQUEST**, pois, para já, não é necessário.

Posteriormente, o *chef* altera o seu estado para 1, **COOK**, pois encontra-se a cozinhar, e seguidamente, o primeiro *client* altera o seu estado para 4. Após algum tempo, o tempo que o *chef* necessita para cozinhar, este atualiza o seu estado para 2, **REST**, o que significa que a comida já está pronta e este pode descansar. Consequentemente, o *waiter* altera o seu estado para 2, **TAKE_TO_TABLE**, levando assim a comida até aos clientes. Aos poucos, um de cada vez, os clientes vão atualizando o seu estado para 5, **EAT**, pois encontram-se a comer. Entretanto, o *waiter* altera o seu estado novamente para 0, **WAIT_FOR_REQUEST**.

Após todos acabarem de comer, quando todos tiverem o seu estado em 5, mais uma vez, aos poucos, vão alterando o seu estado para 6, **WAIT_FOR_OTHERS**, sinalizando que esse *client* está à espera de que os restantes terminem a sua refeição. Em seguida quando todos se encontrarem no estado 6, estes vão começando a atualizar o seu estado para 8, **FINISHED**, uma vez que já terminaram a sua refeição, com a exceção do *client* que tinha chegado em último lugar à mesa, pois, como visto anteriormente, este tem de realizar o pagamento. Este altera o seu estado para 7, **WAIT_FOR_BILL**. Quando todos se encontram no estado 8, exceto o último a chegar à mesa que se deve encontrar no estado 7, este último chama o *waiter* para poder, portanto, realizar o pagamento. O *waiter*, por sua vez altera o estado para 3, **RECEIVE_PAYMENT**, indicando que está a receber o pagamento.

Por fim, após o *waiter* receber o pagamento, o que tinha sido o último *client* a chegar à mesa, altera o seu estado também para 8, **FINISHED**, indicando que também terminou. Por fim, o programa termina.

Conclusão

A realização deste segundo trabalho prático permitiu aprofundar os nossos conhecimentos sobre os mecanismos associados à execução e sincronização de processos e *threads*.

O maior desafio foi perceber as funções que cada semáforo possuí, uma vez que para realizar corretamente o trabalho todo o código teria de ter uma estrutura e um funcionamento correto. No entanto as dificuldades foram superadas através de conhecimento obtido previamente nas aulas teóricas e práticas.

Em suma, é possível verificar que todas as metas propostas no guião foram alcançadas pelos membros do grupo, uma vez que todos os testes foram realizados com sucesso.