universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Trabalho 2
Jantar de Amigos (Restaurant)

Professor:
Nuno Lau (nunolau@ua.pt)

Cristiano Nicolau, 108536, P5
Tiago Cruz, 108615, P5

Índice

1.Introdução

Este trabalho pratico surgiu no contexto da cadeira de Sistemas Operativos, onde nos foi proposto simular um jantar de amigos, envolvendo para além deles, um empregado de mesa (*waiter*) e um cozinheiro (*chef*). O trabalho prático consiste no desenvolvimento de *scripts* já fornecidos pelo professor em C mas que se encontram incompletos, como é o caso dos *scripts semSharedMemChef.c, semSharedMemClient.c* e *semSharedMemWaiter.c* .

Estes *scripts* serão feitos com base na matéria dada nas aulas práticas, onde foram abordados diversos conceitos associados à programação em C. De acordo com o guião teremos de completar partes do código que irão permitir apresentar a simulação do jantar com todas as características e condições definidas pelo guião.

Todos os processos serão criados imediatamente no início do programa e estarão a partir desse momento em execução. Importante referir também que todos os processos serão independentes, sendo que a sua sincronização irá ser realizada através de semáforos e memória partilhada.

Com este trabalho, esperamos ficar mais familiarizados com a programação em linguagem C, e em todos os processos necessários para a realização deste trabalho pratico, esperamos ainda aplicar os conteúdos abordados nas aulas teóricas da cadeira de Sistemas Operativos.

2. Guião

Um grupo de amigos combinou um jantar num restaurante que tem apenas um empregado de mesa (waiter) e um cozinheiro (chef). As regras são que o primeiro a chegar faz o pedido da comida, mas apenas depois de todos chegarem. O empregado de mesa deve levar o pedido ao cozinheiro e trazer a comida para a mesa quando esta estiver pronta. No final, os amigos apenas devem abandonar a mesa depois de todos terminarem de comer, sendo que o último a chegar ao restaurante tem a responsabilidade de pagar a conta. O tamanho da mesa está ajustado ao número de amigos.

Tomando como ponto de partida o código fonte disponível na página da disciplina. Desenvolva uma aplicação em C que simule este jantar. Os clientes, empregado e cozinheiro serão processos independentes, sendo a sua sincronização realizada através de semáforos e memória partilhada. Todos os processos são criados no início do programa e estão em execução a partir dessa altura. Assume-se que os clientes demoram algum tempo a chegar ao restaurante e que esse tempo de chegada pode ser determinado através de uma distribuição de probabilidade uniforme com tempo máximo. Os processos devem estar ativos apenas quando for necessário, devendo bloquear sempre que têm de esperar por algum evento. Apenas deve alterar as zonas assinaladas nos ficheiros com o código fonte.

Compilação e execução do trabalho

Para compilarmos o nosso programa nós tivemos por base o uso do sistema operativo Linux-Ubuntu, pelo que foi apenas necessário executar os comandos necessários para a compilação no terminal do mesmo.

Como na pasta */semaphore_restaurant/src* encontramos um *Makefile*, o único comando que necessitamos para compilar o programa é:

- *make all_bin*

Após executar este comando devemos executar mais um último comando que irá colocar em prática a simulação do jantar. O comando será o seguinte:

- *./probSemSharedMemRestaurant*

Por último verificamos que, através da execução do programa *run.sh,* podemos executar o programa que coloca em prática a simulação do jantar (*./probSemSharedMemRestaurant*) um determinado número de vezes, o que permitia verificar se existia algum *dead lock*. Para isso foram dadas permissões ao *user* para correr este programa o número de vezes que fosse necessário e só depois é que o programa *run.sh* foi executado. Para isso foram utilizados os comandos:

- *chmod u+x run.sh*
- *./run.sh*

3. Semáforos, Estados e Memória partilhada

Semáforos

Apresentamos aqui os semáforos utilizados:
friendsArrived- usado para quando o cliente tem de esperar pela chegada dos restantes.
requestReceived- quando o cliente espera pelo empregado de mesa após o ter chamado.
foodArrived- usado para quando o cliente espera pela comida.
allFinished- no momento em que o cliente termina a refeição ele espera pelos amigos terminarem também a sua.
waiterRequest- usado para quando o empregado de mesa espera pelo pedido.
waitOrder- usado pelo cozinheiro enquanto espera pelo pedido.

Estados

Os estados encontram-se divididos em três grupos: um para o cliente, um para o cozinheiro e um para o empregado de mesa.
Cada estado corresponde a um diferente número dentro do seu grupo e a cada um desses corresponde um significado distinto.

Estados do cliente:

- ***(1)*** **INIT –** estado inicial do cliente
- ***(2)*** **WAIT_FOR_FRIENDS –** O cliente espera que os amigos cheguem à mesa
- ***(3)*** **FOOD_REQUEST –** o cliente pede a comida ao empregado de mesa
- ***(4)*** **WAIT_FOR_FOOD –** o cliente espera pela comida
- ***(5)*** **EAT –** o cliente encontra-se a comer
- ***(6)*** **WAIT_FOR_OTHERS –** o cliente espera que os amigos terminem a refeição
- ***(7)*** **WAIT_FOR_BILL –** o cliente espera para completar o pagamento
- ***(8)*** **FINISHED –** o cliente terminou a refeição

Estados do cozinheiro:

- ***(0)*** **WAIT_FOR_ORDER –** o cozinheiro espera por um pedido
- ***(1)*** **COOK –** o cozinheiro encontra-se a cozinhar
- ***(2)*** **REST –** o cozinheiro descansar

Estados do empregado de mesa:

- ***(0)*** **WAIT_FOR_REQUEST –** o empregado de mesa espera pelo pedido
- ***(1)*** **INFORM_CHEF –** o empregado de mesa leva o pedido ao cozinheiro
- ***(2)*** **TAKE_TO_TABLE –** o empregado de mesa leva a comida à mesa
- ***(3)*** **RECEIVE_PAYMENT –** o empregado de mesa receve o pagamento

Nota: de referir que os pontos ((0), (1), (2), etc.) referem-se ao valor numérico dos estados aos quais é atribuído o significado.

Memória partilhada

Esta memória é partilhada e acessível a todas as execuções presentes no programa, contendo uma estrutura *full_stat e* id's dos semáforos.

Falando agora das estruturas *full_stat*, podemos dizer que esta estrutura apresenta variáveis essenciais para o programa, como por exemplo:

- ***tableClients* –** número de clientes à mesa
- ***tableFinishEat* –** número de clientes que terminaram de comer
- ***foodRequest* –** flag de pedido de ementa do cliente para o empregado de mesa
- ***foodOrder* –** flag de pedido de comida do empregado de mesa para o cozinheiro
- ***foodReady* –** flag de comida pronta do cozinheiro para o empregado de mesa
- ***paymentRequest* –** flag de pedido de pagamento do cliente para o empregado de mesa
- ***tableLast* –** id do primeiro cliente a chegar
- ***tableFirst* -** id do último cliente a chegar

4. Método de Resolução

Client

waitfriends()

O primeiro passo em termos de resolução é complementar o código do *script semSharedMemClient.c* . Inicialmente temos a função *waitFriends()*, que irá registar o primeiro e o último cliente a chegar. Após a chegada do último, este deve informar os outros que a mesa já se encontra completa.

Para isso foi alterado o estado do cliente para WAIT_FOR_FRIENDS e guardamos esse estado. Depois colocamos uma condição que implica que enquanto a mesa não estiver completa é acrescentado um cliente, até a mesa se encontrar preenchida. Quando isso acontecer o primeiro cliente irá ter o seu estado alterado para FOOD_REQUEST, enquanto que os restantes terão o estado alterado para WAIT_FOR_FOOD, sendo ambos os estados respetivamente guardados através do comando *saveState()*.

```
static bool waitFriends(int id)
{
    bool first = false;

    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] = WAIT_FOR_FRIENDS;
    saveState(nFic, &sh->fSt);
    do
    {
        sh->fSt.tableClients ++;
        break;/* code */
    } while  (fSt.tableClients != TABLESIZE);

    if (sh->fSt.tableClients==TABLESIZE){
        semUp(semgid, sh->friendsArrived);
        first=true;
        if (id==p_fSt->tableFirst){
            sh->fSt.st.clientStat[id]=FOOD_REQUEST;
            saveState(nFic, &sh->fSt);
        }else{
            sh->fSt.st.clientStat[id]=WAIT_FOR_FOOD;
            saveState(nFic, &sh->fSt);
        }
    }

    // o primeiro pede a comida de todos mas so apos todos chegarem


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

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

    return first;
}
```

Imagem 1 – Função *waitFriends()*

orderFood()

A segunda função é a função orderFood(), usada apenas pelo primeiro cliente. Nesta função o cliente muda o seu estado para FOOD_REQUEST e faz a partir daí o seu pedido ao empregado de mesa. O cliente deve esperar que o empregado de mesa receba o pedido.

```
static void orderFood (int id)
{
    if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.clientStat[id] = FOOD_REQUEST ;
    sh->fSt.foodRequest ++;
    sh->fSt.foodOrder ++;
    /* insert your code here */

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

Imagem 2 – Função *orderFood()*

waitFood()

Em seguida temos a função waitFood(), na qual o cliente espera pela comida. Para isso é mudado o seu estado para WAIT_FOR_FOOD e salva-se o estado. A partir do momento em que a comida está pronta irá novamente mudar-se o estado para EAT, salvando-se o estado. O estado é sempre salvo através do comando *saveState().*

```
static void waitFood (int id)
{

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

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

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

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

    if (semDown(semgid,sh->foodArrived)==-1){
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
    if(sh->fSt.foodReady==1){
        sh->fSt.st.clientStat[id]=FOODARRIVED;
        saveState(nFic,&sh->fSt);
    }
    sh->fSt.st.clientStat[id] = EAT;
    /* insert your code here */
    saveState(nFic, &sh->fSt);

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

Imagem 3 – Função *waitFood()*

waitAndPay()

Por último no script *semSharedMemClient.c* temos a função waitAndPay(), onde o cliente, após terminar a refeição, atualiza o estado para WAIT_FOR_OTHERS, enquanto espera pelos restantes. O último a terminar a refeição informa os restantes que toda a gente terminou e o cliente que chegou em último à mesa inicialmente procede ao pagamento. Para isso ele pede a conta ao empregado de mesa e espera pela sua chegada. Terminando todo esse processo o cliente irá atualizar o seu estado para FINISHED.

Durante as atualizações de estado o estado foi sempre salvo através do comando *saveState()*.

```
static void waitAndPay (int id)
{
    bool last=false;

    if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
    sh->fSt.paymentRequest ++;
    saveState(nFic,&sh->fSt);

    /* insert your code here */

    if (semUp (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(last) {
        if (semDown (semgid, sh->mutex) == -1) {                                   /* enter critical region */
            perror ("error on the down operation for semaphore access (CT)");
            exit (EXIT_FAILURE);
        }
        sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
        saveState(nFic,&sh->fSt);
        /* insert your code here */

        if (semUp (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 (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] = FINISHED;
    saveState(nFic,&sh->fSt);

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

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

}
```

Imagem 4 – Função *waitAndPay()*

Waiter

waitForClientOrChef()

Após completarmos o primeiro *script* avançamos para o segundo, denominado *semSharedMemWaiter.c* .

Este *script* apresenta também diversas funções que foram completadas por nós, sendo uma delas a função waitForClientOrChef(). Nesta função o empregado de mesa espera pelo pedido do cliente ou do cozinheiro e depois lê esse mesmo pedido. Os tipos de pedido podem ser FOODREQ, FOODREADY ou PAYREQ.

```
static int waitForClientOrChef()
{
    int ret=0;
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    ret= FOODREQ;

    /* insert your code here */

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

    /* insert your code here */

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

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

    return ret;

}
```

Imagem 5 – Função *waitForClientOrChef()*

informChef()

A próxima função, com nome informChef(), tem como único propósito levar o pedido do cliente ao cozinheiro. Durante esse processo o empregado de mesa atualiza o seu estado para INFORM_CHEF. Todo o processo é visível pelo código presente na imagem 6.

```
static void informChef ()
{
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.waiterStat = INFORM_CHEF;
    sh->fSt.foodRequest --;
    saveState(nFic, &sh->fSt);
    /* insert your code here */

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

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

Imagem 6 – Função *informChef()*

takeFoodToTable()

Em adição temos também a função takeFoodToTable(), onde o empregado de mesa simplesmente o seu estado para TAKE_TO_TABLE e guarda o estado, sendo este salvo através do comando *saveState()*. Desta forma o empregado de mesa leva a comida para a mesa e permite o começo da refeição.

```
static void takeFoodToTable ()
{
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &sh->fSt);
    /* insert your code here */

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

Imagem 7 – Função *takeFoodToTable()*

receivePayment()

Por último para este *script* temos a função receivePayment(), em que, como o nome indica, o empregado de mesa recebe o pagamento vindo do último cliente, mudando o seu estado para RECEIVE_PAYMENT. O estado é salvo logo em seguida através do comando *saveState()*, como se pode ver na imagem 8.

```
static void receivePayment ()
{
    if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
    saveState(nFic, &sh->fSt);
    /* insert your code here */

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

Imagem 8 – Função *receivePayment()*

Chef

waitForOrder()

Por último temos também o *script semSharedMemChef.c* . Este *script* também se encontrava incompleto, pelo que o nosso trabalho passou pelo desenvolvimento das 2 funções que necessitavam de código complementar.

Uma dessas funções era a função waitForOrder(), em que basicamente o cozinheiro espera pelo pedido que será entregue pelo empregado de mesa. Para isso ele atualiza o seu estado para WAIT_FOR_ORDER e salva através do comando *saveState().*

```
static void waitForOrder ()
{
    /* insert your code here */

    if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.chefStat   = WAIT_FOR_ORDER;
    saveState(nFic, &sh->fSt);
    /* insert your code here */

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

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

Imagem 9 – Função *waitForOrder()*

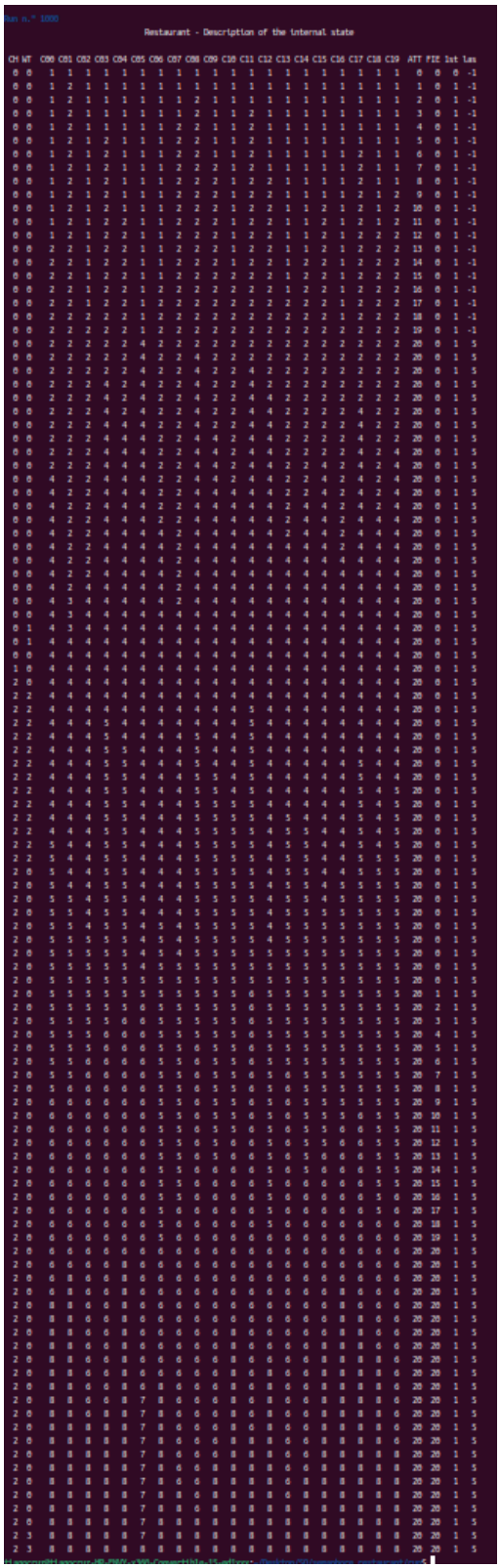
processOrder()

A outra função presente é a função processOrder(), em que o cozinheiro cozinha a comida. Para isso acontecer usamos uma condição *while* em que enquanto o número de pratos for diferente do número de pessoas na mesa ele cozinha mais uma refeição. Enquanto esta condição estiver a ser usada o cozinheiro terá como estado COOK. Quando o número de refeições equivaler ao número de clientes na mesa o cozinheiro muda o seu estado para REST, que significa que terminou o seu trabalho e pode descansar. Para terminar o estado é sempre salvo através do comando *saveState()*.

```
static void processOrder ()
{
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));

    if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
    while (sh->fSt.foodReady != TABLESIZE)
    {
        sh->fSt.st.chefStat = COOK;
        sh->fSt.foodReady ++;
        saveState(nFic, &sh->fSt);
    }

    /* insert your code here */

    if (semUp (semgid, sh->mutex) == -1) {                                      /* exit critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.st.chefStat   = REST;
    saveState(nFic, &sh->fSt);



    /* insert your code here */
}
```

Imagem 10 – Função process*Order()*

5. Resultados



Para termos uma melhor noção da evolução do nosso trabalho foram feitos diversos testes com o que íamos fazendo através de compilações como *make client_bin, waiter_bin ou chef_bin.*

No entanto, na parte final foi realizado o teste em que verificávamos os resultados obtidos na globalidade. Para isso recorremos ao *script run.sh*, que irá simular 1000 vezes a nossa simulação, verificando assim a existência de *deadlocks*.

Como durante esta execução o programa foi executado por inteiro podemos confirmar que não existiam deadlocks

Nota: devido à extensão das tabelas apresentamos apenas a última, que comprova a execução total de *run.sh.*

Imagem 11 – *./run.sh*

Por último realizamos a execução de uma simulação, através do comando *./probSemSharedMemRestaurant*, para verificar se esta foi bem sucedida. Como podemos ver pela Imagem 12, temos 20 clientes presentes, no qual o primeiro a chegar foi o cliente 3 e o último foi o cliente 14, verificamos a chegada dos clientes à mesa e também o número de pessoas que vai terminando a refeição, tudo em valores de acordo com o que era expectável.

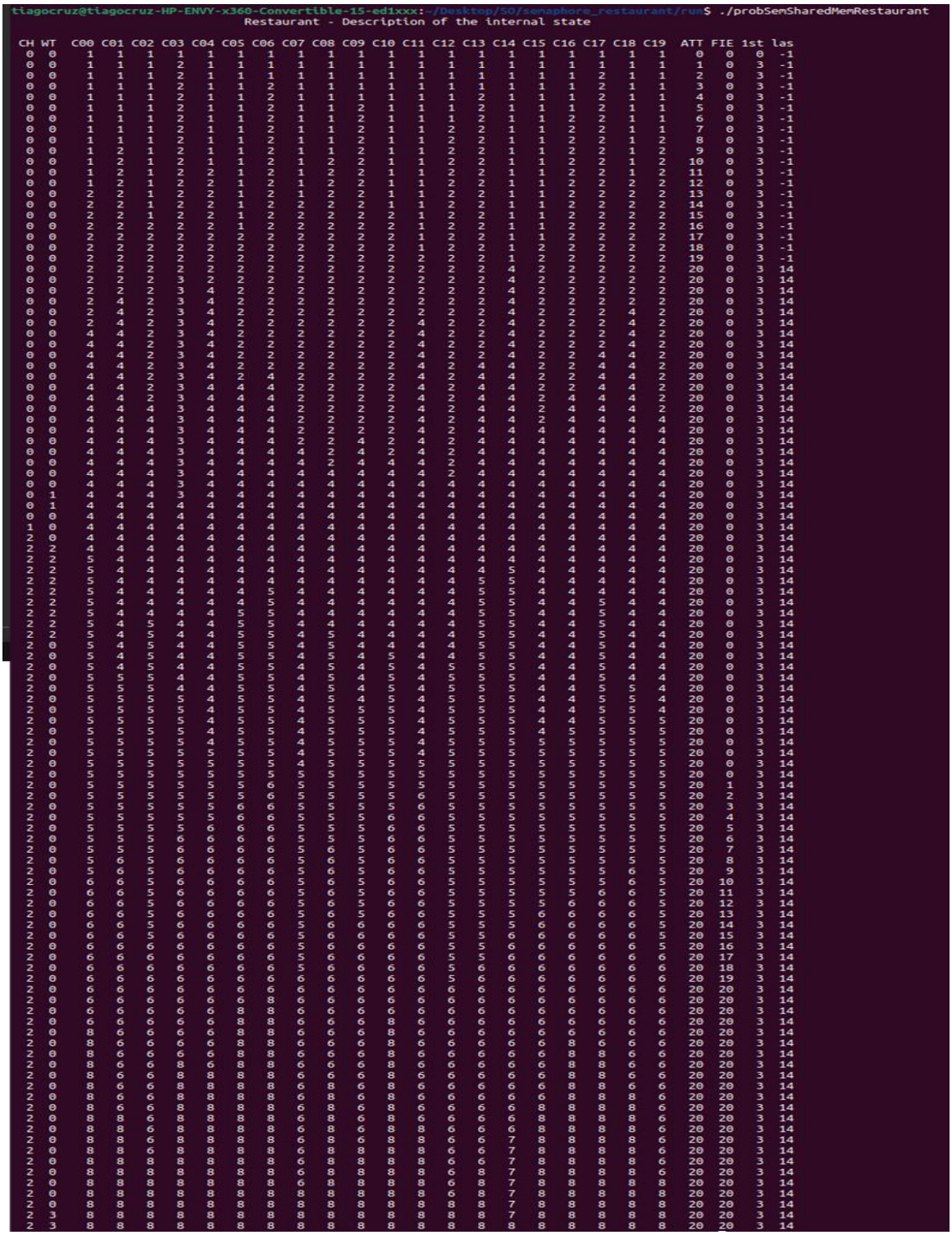


Imagem 12 – *./probSemSharedMemRestaurant*

6. Conclusão

O trabalho desenvolvido permitiu obter uma simulação de um jantar num restaurante envolvendo cliente, empregado de mesa e cozinheiro.

Com a realização do projeto foi-nos possível compreender melhor como utilizar semáforos e memória partilhada, e consideramos que depois de realizar este trabalho estamos mais elucidados em relação a certos detalhes destes temas.

Inicialmente a preocupação passou por tentar perceber o código fornecido pelo professor. Ao início não foi fácil, no entanto, conforme fomos trabalhando com os *scripts,* ficamos cada vez mais familiarizados com o código e tornou mais simples a implementação do código que desenvolvemos.

Em suma, consideramos que conseguimos alcançar os objetivos propostos pelo guião. Todo o trabalho permitiu pormos em prática tanto o conhecimentos adquirido nas aulas teóricas como a experiência adquirida através dos guiões realizados nas aulas práticas.

7. Bibliografia

Uma das ferramentas mais utilizadas para a realização deste trabalho foram os PowerPoints fornecidos e utilizados pelo professor durante as aulas teóricas, que se encontram disponíveis na página do *e-learning* da respetiva unidade curricular.

Alguns dos sites que consultamos foram os seguintes:

- https://pt.stackoverflow.com/
-