deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Trabalho 2 - Restaurante

Trabalho de Sistemas Operativos
Professor Regente: José Nuno Panelas Nunes Lau

João Monteiro nº114547, João Pinto nº104384

Abstract

O objetivo deste trabalho consistiu no desenvolvimento de uma aplicação em C que permitisse simular o funcionamento de um restaurante, através de **semáforos** e **memória partilhada**.

O cerne do trabalho envolvia 4 processos independentes: **group**, **chef**, **receptionist** e **waiter**; assim sendo, o objetivo era que todos os grupos jantassem num restaurante que apenas contém 2 mesas.

Todos os processos deverão ser inicializados no início do programa, porém apenas devem estar em execução quando necessário. No final, todos os processos devem terminar.

Keywords: *Semáforos; memória partilhada; processos; threads; C; restaurante; funções; argumentos.*

---

1. Desenvolvimento do Restaurante (código)

Na realização deste trabalho apenas foram alterados os ficheiros **semSharedMemChef.c**, **semSharedMemWaiter.c**, **semSharedMemGroup.c** e **semSharedMemReceptionist.c**. Ao longo do relatório vamos explicar cada um destes ficheiros e as funções que implementámos.

1.1. Chef

O primeiro processo que iremos abordar é o **Chef**. A sua função fundamental neste programa é a preparação da refeição. Inicialmente, recebe um pedido do **waiter** e trata de preparar a comida. Após essa preparação, pede ao **waiter** que leve o pedido à mesa. Para este caso existem duas funções a ser tratadas: **waitForOrder()** e **processOrder()**.

Além disso, o **Chef** utiliza dois semáforos:

- **waiterRequestPossible**: utilizado pelos grupos e pelo **chef** para esperar antes de fazer um pedido ao waiter.
- **waitOrder**: utilizado pelo **chef** para esperar por um pedido do waiter.

1.1.1. waitForOrder()

Nesta função o **Chef** espera que lhe chegue um pedido do waiter. O estado inicial do chef é **WAIT_FOR_FOOD**, portanto não é necessário especificá-lo de novo. Faz-se sim, o **semDown()** do semáforo **waitOrder**, para o chef esperar por um pedido.

Posteriormente, depois do **semDown()** do semáforo **mutex**, entra-se na região crítica do programa, onde se altera o estado do chef para **COOK** e esse estado é salvo. Define-se, também a flag **foodOrder** a 0, uma vez que o chef está a processar o pedido e o chef guarda ainda na variável **lastGroup** o grupo ao qual corresponde o pedido. Isto é muito importante pois o chef quando chamar o waiter para entregar o pedido terá que dizer a que grupo corresponde como iremos ver na próxima função **processOrder()**.

Já fora da região crítica, faz-se um **semUp()** do semáforo **orderReceived**, para o waiter saber que a ordem foi recebida.

Em baixo apresentamos o código desta função. Importante referir que tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
static void waitForOrder ()
{

    //TODO insert your code here
    // semaphore used by chef to wait for order val = 0
    if (semDown (semgid, sh->waitOrder) == -1) {
        /* enter critical region */
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }

    // Fim

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


    //TODO insert your code here
    sh->fSt.st.chefStat = COOK; // Nota: so comeca a cozinhar na funcao seguinte
    saveState(nFic,&sh->fSt);

    sh->fSt.foodOrder = 0; // flag of food request from waiter to chef
    lastGroup = sh->fSt.foodGroup; // foodGroup -> group associated to food request from
    waiter to chef
    // Fim

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

    //TODO insert your code here

    // -> Received order should be acknowledged.
    if (semUp (semgid, sh->orderReceived) == -1) {
          /* enter critical region */
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }


    // Fim
}
```

1.1.2. processOrder()

Nesta função o chef cozinha durante um certo tempo (função do comando **usleep**) e, quando a comida estiver pronta, avisa o waiter, caso haja disponibilidade, que o pedido está pronto.

O **chef** espera que o waiter esteja disponível para lhe fazer um pedido, para tal efeito faz **semDown()** do semáforo **waiterRequestPossible**.

Entrando na região crítica do programa, altera-se o estado do chef para **WAIT_FOR_ORDER**. Além disso tem que indicar o tipo de pedido ao waiter, neste caso é **FOODREADY** uma vez que a comida está pronta. Tem ainda que dizer a que grupo aquele pedido corresponde e para isso utiliza a variável **lastGroup** que tinha guardado na função explicada anteriormente. O estado atual é salvo.

Por último faz-se um **semUp()** do semáforo **waiterRequest**, para notificar o waiter que o chef já terminou de cozinhar e que o pedido pode ser levado à mesa.

Em baixo apresentamos o código desta função. Importante referir que tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
static void processOrder ()
{
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0)); // so comeca a
     cozinhar aqui -> ele fica "preso" aqui

    //TODO insert your code here

    /** signals the waiter that food is ready -> "unsigned int waiterRequestPossible" ->
    identification of semaphore used by groups and chef to wait before issuing waiter
    request - val = 1 */
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
                            /* enter critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    // Fim

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

    //TODO insert your code here

    sh->fSt.st.chefStat = WAIT_FOR_ORDER;

    sh->fSt.waiterRequest.reqType = FOODREADY; // tambem pode ser "FOODREQ" -> Ver "
    semSharedMemWaiter.c"
    sh->fSt.waiterRequest.reqGroup = lastGroup;


    saveState(nFic, &sh->fSt);
    // Fim

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

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


    // Fim
}
```

Termina assim o processo do **Chef** neste programa.

1.2. Waiter

Já o **waiter** tem como função receber o pedido do grupo que está na mesa e de levar esse mesmo pedido ao **chef**, para a sua confeção. Assim que o pedido estiver pronto, o chef chama-o e ele leva o pedido para a mesa. Este processo envolve três funções: **waitForClientOrChef()**, **informChef(int n)** e **takeFoodToTable(int n)**.

Além disso, o **waiter** utiliza dois semáforos:

- **waiterRequest**: utilizado pelo **waiter** para esperar por pedidos. Pode receber pedidos de um grupo ou do chef.
- **orderReceived**: utilizado pelo **waiter** para esperar que o chef confirme a receção do pedido.

1.2.1. waitForClientOrChef()

Nesta primeira função, o **waiter** altera o seu estado e espera por um pedido, quer seja do chef ou de um grupo. Por último, lê esse pedido.

Entra-se pela primeira vez na região crítica, para alterar o estado do waiter para **WAIT_FOR_REQUEST** e para salvar o estado. Posteriormente, já fora da região crítica, faz-se um **semDown()**, do semáforo **waiterRequest**, para o **waiter** esperar por pedidos.

Ao entrar de novo na região crítica, o **waiter** atribui à variável **req** o tipo e o grupo que fez o pedido. O tipo pode ser **FOODREQ** se tiver sido um pedido de comida por parte de um grupo ou pode ser **FOODREADY** se o chef tiver pedido ao **waiter** para ir buscar a comida para entregar.

Por fim, faz-se um **semUp()** do semáforo **waiterRequestPossible**, para sinalizar que está disponível para novos pedidos.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static request waitForClientOrChef()
2  {
3      // Usei 6 semaforos
4      request req;
5      if (semDown (semgid, sh->mutex) == -1)  {
           /* enter critical region */
6          perror ("error on the up operation for semaphore access (WT)");
7          exit (EXIT_FAILURE);
8      }
9
10     // TODO insert your code here
11
12     sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
13     saveState(nFic, &sh->fSt);
14
15     // Fim
16
17     if (semUp (semgid, sh->mutex) == -1)      {
        /* exit critical region */
18         perror ("error on the down operation for semaphore access (WT)");
19         exit (EXIT_FAILURE);
20     }
21
22     // TODO insert your code here
23     //  semaphore used by waiter to wait for requests val = 0
24     if (semDown(semgid, sh->waiterRequest) == -1)      {
        /* exit critical region */
25         perror ("error on the down operation for semaphore access");
26         exit (EXIT_FAILURE);
27     }
28
29     // Fim
30
31     if (semDown (semgid, sh->mutex) == -1)  {
           /* enter critical region */
32         perror ("error on the up operation for semaphore access (WT)");
33         exit (EXIT_FAILURE);
34     }
35
36     // TODO insert your code here
37
38     req.reqType = sh->fSt.waiterRequest.reqType; // pedido do chef ou de um grupo. reqType
       sera: FOODREQ (GRUPO) OU FOODREADY (chef)
39     req.reqGroup = sh->fSt.waiterRequest.reqGroup;
40
41     if (semUp (semgid, sh->mutex) == -1) {
        /* exit critical region */
42         perror ("error on the down operation for semaphore access (WT)");
43         exit (EXIT_FAILURE);
44     }
45
46     // TODO insert your code here
47
48     //  The waiter should signal that new requests are possible.
49     if (semUp (semgid, sh->waiterRequestPossible) == -1) {
                       /* exit critical region */
```

```
50          perror ("error on the up operation for semaphore access (WT)");
51          exit (EXIT_FAILURE);
52      }
53
54      // Fim
55      return req;
56
57  }
```

1.2.2. informChef (int n)

Nesta função, o **waiter** leva o pedido ao **chef** e informa o grupo que o pedido foi recebido, e espera que o chef confirme a receção do pedido.

Na região crítica do programa, o estado do waiter é alterado para **INFORM_CHEF**. O grupo que fez o pedido (argumento **n**) é guardado em **foodGroup** e a flag **foodOrder** é definida a 1 para o chef saber que tem um pedido para ser tratado. É salvo depois o estado atual.

Por último é feito o **semUp()** de dois semáforos, **requestReceived** e **waitOrder**. O primeiro serve para confirmar ao grupo que o pedido foi recebido e outro para informar o chef do pedido. O segundo semáforo, **requestReceived**, está associado a uma das duas mesas portanto é utilizado **sh->fSt.assignedTable[n]** para saber a mesa que está associada ao grupo **n**.

É feito ainda o **semDown()** do **orderReceived**, onde o **waiter** espera que o **chef** confirme a receção do pedido.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1   static void informChef (int n)
2   {
3       // 5 semaforos
4       if (semDown (semgid, sh->mutex) == -1)  {
            /* enter critical region */
5           perror ("error on the up operation for semaphore access (WT)");
6           exit (EXIT_FAILURE);
7       }
8
9       // TODO insert your code here
10      sh->fSt.st.waiterStat = INFORM_CHEF;
11      sh->fSt.foodGroup = n;
12      sh->fSt.foodOrder = 1;
13      saveState(nFic, &sh->fSt);
14
15      int assignedTable = sh->fSt.assignedTable[n];
16      // Fim
17
18      if (semUp (semgid, sh->mutex) == -1)
        /* exit critical region */
19      { perror ("error on the down operation for semaphore access (WT)");
20          exit (EXIT_FAILURE);
21      }
22
23      // TODO insert your code here
24
25      // Grupos esperam a confirmacao (used by groups to wait for waiter ackowledge)
26      if (semUp(semgid, (sh->requestReceived[assignedTable])) == -1) {
27          perror("error on the up operation for semaphore access");
28          exit(EXIT_FAILURE);
29      }
30
31      // used by chef to wait for order
32      if (semUp(semgid, sh->waitOrder) == -1) {
33          perror("error on the down operation for semaphore access");
34          exit(EXIT_FAILURE);
35      }
36
37      //  used by waiter to wait for chef
38      if (semDown(semgid, sh->orderReceived) == -1) {
39          perror("error on the down operation for semaphore access");
40          exit(EXIT_FAILURE);
41      }
```

```
42      // Fim
43  }
```

1.2.3. takeFoodToTable(int n)

Nesta função, o **waiter** atualiza o seu estado e leva o pedido à mesa para a refeição poder começar. O waiter necessita de informar que a comida já está pronta.

Ao entrar na região crítica, o estado do waiter muda para **TAKE_TO_TABLE** e esse estado é salvo.

No final é feito o **semUp** do semáforo **foodArrived** relativo à mesa do grupo **n**, para informar o grupo que a comida chegou e que podem começar a comer.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static void takeFoodToTable (int n)
2  {
3      // 3 semaforos
4      // n corresponde ao grupo que e para entregar
5      if (semDown (semgid, sh->mutex) == -1)  {
           /* enter critical region */
6          perror ("error on the up operation for semaphore access (WT)");
7          exit (EXIT_FAILURE);
8      }
9
10     // TODO insert your code here
11     sh->fSt.st.waiterStat = TAKE_TO_TABLE;
12     saveState(nFic, &sh->fSt);
13     if (semUp(semgid, (sh->foodArrived[sh->fSt.assignedTable[n]])) == -1) {
14         perror("error on the up operation for semaphore access");
15         exit (EXIT_FAILURE);
16     }
17
18
19     // fim
20
21     if (semUp (semgid, sh->mutex) == -1)  {
         /* exit critical region */
22         perror ("error on the down operation for semaphore access (WT)");
23         exit (EXIT_FAILURE);
24     }
25 }
```

Explicámos as três funções **waitForClientOrChef()**, **informChef(int n)** e **takeFoodToTable(int n)** utilizadas pelo **waiter** e os semáforos utilizados. Em seguida iremos explicar o **Recepcionist**.

1.3. Recepcionist

Assim que um **grupo** chega ao restaurante, é ao **recepcionist** que se deve dirigir. Nessa situação, o **recepcionist** indica ao grupo se devem aguardar na sala de espera até uma mesa se encontrar disponível ou se, por sua vez, devem dirigir-se a uma das mesmas (caso haja disponibilidade para tal). No fim da refeição, é o **recepcionist** que deve receber os pagamentos. Este processo está organizado através de três funções: **waitForGroup()**, **provideTableOrWaitingRoom(int n)** e **receivePayment (int n)**.

Além disso, o **recepcionist** utiliza um semáforo:

- **receptionistReq**: utilizado pelo **recepcionist** para esperar pelos grupos.

As funções **provideTableOrWaitingRoom(int n)** e **receivePayment (int n)** dependem de **duas funções auxiliares: decideTableOrWait(int n) e decideNextGroup()**, respetivamente.

Por isso optámos por explicar primeiro estas duas funções auxiliares.

1.3.1. decideTableOrWait(int n)

Esta função auxiliar é usada na função **provideTableOrWaitingRoom(int n)** e tem a função de decidir se o **recepcionist** manda o **grupo** para uma mesa ou se manda o grupo para a sala de espera.

No código, verificámos se havia grupos à espera. Caso existam, a função devolve **-1**, que indica que o grupo deve aguardar.

Em caso negativo, o resto do código será processado. Caso não haja, então, grupos à espera, através de um ciclo **for**, percorre-se as duas mesas existentes, onde se vai verificar qual (ou quais) as mesas disponíveis. Para isso definimos uma flag, **ocupada**, com valor 0 e percorremos os vários grupos existentes. Se a mesa atribuída a esse grupo for igual à mesa que estamos a processar, a flag passa a ter valor 1 e iremos sair desse ciclo. Caso não exista nenhum grupo cuja mesa atribuída seja a mesa que estamos a processar, então iremos retornar essa mesa, de modo a informar que essa mesa está **disponível**.

Em baixo apresentamos o código desta função.

```
static int decideTableOrWait(int n)
{
     //TODO insert your code here
    if(sh->fSt.groupsWaiting == 0) { // number of groups waiting for table

        for (int tableId = 0; tableId < NUMTABLES; ++tableId) {
            int ocupada = 0; // Flag para verificar se a tabela esta ocupada
            for (int groupID = 0; groupID < sh->fSt.nGroups; ++groupID) {
                if (sh->fSt.assignedTable[groupID] == tableId) {
                    ocupada = 1; //
                    break;
                }
            }
            if (!ocupada) {
                // Encontrou uma tabela vazia, retorne o ID da tabela
                return tableId;
            }
        }
    }
    return -1;
}
```

1.3.2. decideNextGroup()

Esta função auxiliar é usada na função **receivePayment (int n)** e tem a função de indicar qual deverá ser o grupo que se deverá sentar numa mesa, no momento em que essa se torne **livre**.

Caso não exista nenhum grupo à espera, a função retorna **-1**.

Caso existam, o código através de um ciclo **for**, percorre todos os grupos e verifica se o estado desse grupo é **WAIT**. Em caso positivo, a função retorna esse grupo.

Em baixo apresentamos o código desta função.

```
static int decideNextGroup()
{
    //TODO insert your code here
    if(sh->fSt.groupsWaiting == 0) return -1;

    for (int groupID = 0; groupID < sh->fSt.nGroups; ++groupID) {
        if ( groupRecord[groupID] == WAIT ) {  // groupRecord -> receptioninst view on each
     group evolution (useful to decide table binding)
            return groupID;
        }
    }
    return -1;
}
```

1.3.3. waitForGroup()

Nesta função, o **recepcionist** deve aguardar pelo pedido de um grupo e deve lê-lo. Acabando este processo, deve avisar que está pronto para um novo pedido. No final da função retorna o pedido **ret**.

Inicialmente, entramos na região crítica do programa, onde atualizamos o estado do recepcionist para **WAIT_REQUEST**. De seguida já fora dessa região crítica, fazemos o **semDown()**, do semáforo **receptionistReq**, fazendo com que o recepcionist espere por um pedido do grupo.

Após esta fase, entramos de novo na região crítica de modo ao recepcionist ler o pedido efetuado. O **receptionist** atribui à variável **ret** o tipo e o grupo que fez o pedido. O tipo pode ser **TABLEREQ** se

tiver sido um pedido de mesa por parte de um grupo ou pode ser **BILLREQ** se o grupo tiver pedido a conta.

Por último fazemos o **semUp()**, de semáforo **recepcionistRequestPossible**, de modo a sinalizar a disponibilidade do recepcionist para um novo pedido.

Em baixo apresentamos o código desta função. Importante referir que tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static request waitForGroup()
2  {
3      request ret;
4
5      if (semDown (semgid, sh->mutex) == -1)  {
           /* enter critical region */
6          perror ("error on the up operation for semaphore access (WT)");
7          exit (EXIT_FAILURE);
8      }
9
10     // TODO insert your code here
11     // Receptionist updates state
12     sh->fSt.st.receptionistStat = WAIT_REQUEST;
13     saveState(nFic, &sh->fSt);
14
15     // FIM
16
17     if (semUp (semgid, sh->mutex) == -1)      {
        /* exit critical region */
18         perror ("error on the down operation for semaphore access (WT)");
19         exit (EXIT_FAILURE);
20     }
21
22     // TODO insert your code here
23     // waits for request from group
24     if (semDown (semgid, sh->receptionistReq) == -1) {
25         perror ("error on the down operation for semaphore access");
26         exit (EXIT_FAILURE);
27     }
28
29     // FIM
30
31     if (semDown (semgid, sh->mutex) == -1)  {
           /* enter critical region */
32         perror ("error on the up operation for semaphore access (WT)");
33         exit (EXIT_FAILURE);
34     }
35
36     // TODO insert your code here
37     // reads request
38     ret.reqType = sh->fSt.receptionistRequest.reqType; // pedido de um grupo. reqType sera:
        TABLEREQ OU BILLREQ
39     ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
40
41     // FIM
42
43     if (semUp (semgid, sh->mutex) == -1) {
        /* exit critical region */
44      perror ("error on the down operation for semaphore access (WT)");
45         exit (EXIT_FAILURE);
46     }
47
48     // TODO insert your code here
49     // signals availability for new request.
50     if (semUp (semgid, sh->receptionistRequestPossible) == -1) {
51         perror ("error on the up operation for semaphore access");
52         exit (EXIT_FAILURE);
53     }
54
55     // FIM
56
57     return ret;
58
59 }
```

1.3.4. provideTableOrWaitingRoom(int n)

Nesta função, o recepcionist decide se um grupo poderá seguir para uma mesa ou deverá aguardar pela mesma. Para tal, entramos na região crítica onde mudamos o estado do recepcionist para **ASSIGNTABLE**.

Posteriormente, chamamos a função **decideTableOrWait(int n)**, explicada anteriormente. Caso o valor de retorno da função seja **-1**, o grupo deverá esperar e o número de **grupos à espera** é incrementado. Caso o valor de retorno seja diferente de -1, atribuímos a mesa devolvida pela função ao grupo em questão e alteramos o estado do grupo para **ATTABLE**. Por último, fazemos o **semUp()** do semáforo **waitForTable**, para o grupo em questão, indicando que o grupo já não está à espera que uma mesa fique disponível. Para terminar a função, o estado é salvo.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static void provideTableOrWaitingRoom (int n)
2  {
3      if (semDown (semgid, sh->mutex) == -1)  {
           /* enter critical region */
4          perror ("error on the up operation for semaphore access (WT)");
5          exit (EXIT_FAILURE);
6      }
7
8      // TODO insert your code here
9      // Receptionist updates state
10     sh->fSt.st.receptionistStat = ASSIGNTABLE;
11     saveState(nFic, &sh->fSt);
12
13     int ret = decideTableOrWait(n); // returns table id or -1 (in case of wait decision)
14
15
16     if( ret == -1) { // group have to wait
17         groupRecord[n] = WAIT;
18         sh->fSt.groupsWaiting++;
19     }
20     else { // returns table id
21         sh->fSt.assignedTable[n] = ret;
22         groupRecord[n] = ATTABLE;
23
24         if (semUp(semgid, sh->waitForTable[n]) == -1) {
25             perror("error on the up operation for semaphore access");
26             exit(EXIT_FAILURE);
27         }
28     }
29     // FIM
30
31     if (semUp (semgid, sh->mutex) == -1) {                                    /*
        exit critical region */
32         perror ("error on the down operation for semaphore access (WT)");
33         exit (EXIT_FAILURE);
34     }
35 }
```

1.3.5. receivePayment(int n)

Para finalizar este processo, o **recepcionist** recebe o pagamento por parte do grupo e decide se a mesa que se tornou livre deve ser ocupada por outro grupo.

Assim sendo, ao entrar na região crítica, os estados do recepcionist e do grupo serão alterados para **RECVPAY** e **DONE**, respetivamente. Estas alterações são guardadas. A mesa atribuída para o grupo inicial passa a ser **-1**, valor *default* definido para grupos sem mesa atribuída.

Posteriormente, o recepcionist terá de decidir se a mesa deverá ser ocupada por outro grupo. Para tal, chama-se a função **decideNextGroup()**, explicada anteriomente. Caso o valor de retorno dessa função seja diferente de **-1**, o valor de retorno da função é o grupo que deverá ocupar a mesa. Assim sendo, atribui-se essa mesa ao novo grupo, altera-se o estado desse grupo para **ATTABLE** e o número de grupos à espera é decrementado. Fez-se também o **semUp()** do semáforo **waitForTable**, para o grupo em questão, de modo a indicar que esse grupo já não se encontra à procura de uma mesa.

Por fim, procede-se ainda ao **semUp()**, do semáforo **tableDone**, da mesa ocupada pelo grupo em questão (sh->fSt.assignedTable[n]), para informar que essa mesa já não está a ser usada.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

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

    // TODO insert your code here
    // Receptionist updates its state
    sh->fSt.st.receptionistStat = RECVPAY;
    groupRecord[n] = DONE;
    saveState(nFic, &sh->fSt);

    int assignedTable = sh->fSt.assignedTable[n];
    sh->fSt.assignedTable[n] = -1;

    // receives payment


    // If there are waiting groups, receptionist should check if table that just became
    vacant should be occupied.
    // printf("Grupo N %d pagou e saiu da mesa %d. \n", n, sh->fSt.assignedTable[n]);
    int ret = decideNextGroup();

    if(ret != -1) { // returns group id
        // sh->fSt.assignedTable[n] -> corresponde a mesa que ficou vazia. n era o grupo
    que estava la
        sh->fSt.assignedTable[ret] = assignedTable;


        if (semUp(semgid, sh->waitForTable[ret]) == -1) {
            perror("error on the up operation for semaphore access");
            exit(EXIT_FAILURE);
        }
        sh->fSt.groupsWaiting--;
        groupRecord[ret] = ATTABLE;
    }


    // FIM

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

    // TODO insert your code here

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


    // FIM
}
```

Termina assim o processo do **recepcionist**.

1.4. Groups

A cada **grupo** cabe dirigir-se ao **recepcionist** e pedir uma mesa. Assim que obtenha uma mesa, o grupo deve fazer o pedido ao **waiter**. Após a refeição o grupo dirige-se novamente ao **recepcionist** para pagar a conta e sair.

Este processo está organizado através de seis funções: **goToRestaurant (int id)**, **checkInAtReception (int id)**, **orderFood (int id)**, **waitFood (int id)**, **eat (int id)** e **checkOutAtReception (int id)**.

Além disso, os **groups** utilizam seis semáforos:

- **receptionistRequestPossible**: utilizado pelos **groups** para esperar antes de fazer um pedido ao **receptionist**.
- **waiterRequestPossible**: utilizado pelos **groups** para esperar antes de fazer um pedido ao **waiter**.
- **waitForTable[MAXGROUPS]**: utilizado pelos **groups** para esperar por mesa.
- **requestReceived[NUMTABLES]**: utilizado pelos **groups** para esperar a confirmação de receção por parte do **waiter**.
- **foodArrived[NUMTABLES]**: utilizado pelos **groups** para esperar por comida.
- **tableDone[NUMTABLES]**: utilizado pelos **groups** para esperar pelo pagamento estar concluído.

1.4.1. checkInAtReception(int id)

Assim que o **recepcionist** esteja disponível, o grupo deverá pedir uma mesa (podendo haver a possibilidade de ter de esperar pela mesma).

A função começa com um **semDown()** do semáforo **recepcionistRequestPossible**, ou seja, assim que o recepcionist esteja disponível, o grupo pede uma mesa.

Entrando na região crítica, o estado do grupo muda para **ATRECEPTION** e define-se o tipo de pedido (**reqType**) como **TABLEREQ**. Define-se ainda o grupo que fez o pedido e é salvo, então, o estado atual.

Por último, é feito um **semUp()**, do semáforo **receptionistReq** para informar o receptionist do pedido do grupo. É ainda feito um **semDown** do **waitForTable**, uma vez que o grupo espera que lhe seja atribuído mesa.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static void checkInAtReception(int id)
2  {
3      // TODO insert your code here
4      // Group should, as soon as receptionist is available, ask for a table
5      if (semDown(semgid, sh->receptionistRequestPossible) == -1) {
6          perror("error on the down operation for semaphore access");
7          exit(EXIT_FAILURE);
8      }
9
10     // fim
11
12     if (semDown (semgid, sh->mutex) == -1) {
           /* enter critical region */
13         perror ("error on the down operation for semaphore access (CT)");
14         exit (EXIT_FAILURE);
15     }
16
17     // TODO insert your code here
18     sh->fSt.st.groupStat[id] = ATRECEPTION;
19
20     // signaling receptionist of the request.
21     sh->fSt.receptionistRequest.reqType = TABLEREQ;
22     sh->fSt.receptionistRequest.reqGroup = id;
23
24     saveState(nFic, &sh->fSt);
25
26     // fim
27
28     if (semUp (semgid, sh->mutex) == -1) {
           /* exit critical region */
29         perror ("error on the up operation for semaphore access (CT)");
30         exit (EXIT_FAILURE);
31     }
32
33     // TODO insert your code here
34     if (semUp(semgid, sh->receptionistReq) == -1) {
35         perror("error on the up operation for semaphore access");
36         exit(EXIT_FAILURE);
```

```
37      }
38
39      if (semDown(semgid, sh->waitForTable[id]) == -1) {
40          perror("error on the down operation for semaphore access");
41          exit(EXIT_FAILURE);
42      }
43
44      // FIM
45
46  }
```

1.4.2. orderFood(int id)

Nesta função, o grupo deve fazer o seu pedido ao **waiter**.

Primeiro, faz-se um **semDown()** para, assim que possível, o grupo fazer o seu pedido ao **waiter**.

Na região crítica, muda-se o estado do grupo para **FOOD_REQUEST**, define-se o tipo do pedido (**reqType**) como **FOODREQ**. Define-se ainda o grupo que fez o pedido e é salvo o estado atual.

Faz-se, por último, um **semUp()**, do semáforo **waiterRequest**, para informar o waiter do pedido. É ainda feito um **semDown()** do semáforo **requestReceived**, para o grupo esperar confirmação de receção do pedido por parte do **waiter**.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1   static void orderFood (int id)
2   {
3       // TODO insert your code here
4
5       // request food to the waiter
6       if (semDown (semgid, (sh->waiterRequestPossible)) == -1) {
7           perror ("error on the down operation for semaphore access");
8           exit(EXIT_FAILURE);
9       }
10
11      // fim
12
13      if (semDown (semgid, sh->mutex) == -1) {
           /* enter critical region */
14          perror ("error on the down operation for semaphore access (CT)");
15          exit (EXIT_FAILURE);
16      }
17
18      // TODO insert your code here
19      sh->fSt.st.groupStat[id] = FOOD_REQUEST;
20
21      sh->fSt.waiterRequest.reqType = FOODREQ;
22      sh->fSt.waiterRequest.reqGroup = id;
23      saveState(nFic, &sh->fSt);
24
25      int assignedTable = sh->fSt.assignedTable[id]; // aceder a mem.partilhada dentro do
        mutex
26
27      // fim
28
29      if (semUp (semgid, sh->mutex) == -1) {
            /* exit critical region */
30          perror ("error on the up operation for semaphore access (CT)");
31          exit (EXIT_FAILURE);
32      }
33
34      // TODO insert your code here
35      // wait for the waiter to receive the request.
36
37      if (semUp (semgid, sh->waiterRequest) == -1) {
                    /* exit critical region */
38          perror ("error on the up operation for semaphore access (CT)");
39          exit (EXIT_FAILURE);
40      }
41
42      if (semDown (semgid, (sh->requestReceived[assignedTable])) == -1) {
43          perror ("error on the up operation for semaphore access");
```

```
44          exit(EXIT_FAILURE);
45      }
46  }
```

1.4.3. waitFood(int id)

Nesta função, o grupo espera que a comida seja entregue.

Na região crítica, muda-se o estado do grupo para **WAIT_FOR_FOOD** e é salvo o estado atual.

É feito um **semDown()** do semáforo **foodArrived**, para o grupo esperar que o pedido seja entregue.

Após esta fase, entramos de novo na região crítica. Muda-se o estado do grupo para **EAT** e é salvo o estado atual.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static void waitFood (int id)
2  {
3      if (semDown (semgid, sh->mutex) == -1) {
          /* enter critical region */
4          perror ("error on the down operation for semaphore access (CT)");
5          exit (EXIT_FAILURE);
6      }
7
8      // TODO insert your code here
9      sh->fSt.st.groupStat[id] = WAIT_FOR_FOOD;
10     saveState(nFic, &sh->fSt);
11
12     int assignedTable = sh->fSt.assignedTable[id]; // aceder a mem.partilhada dentro do
       mutex
13
14     // fim
15
16     if (semUp (semgid, sh->mutex) == -1) {
        /* enter critical region */
17         perror ("error on the down operation for semaphore access (CT)");
18         exit (EXIT_FAILURE);
19     }
20
21     // TODO insert your code here
22
23     if (semDown(semgid, (sh->foodArrived[assignedTable])) == -1) {
24         perror("error on the down operation for semaphore access");
25         exit(EXIT_FAILURE);
26     }
27
28     // fim
29
30     if (semDown (semgid, sh->mutex) == -1) {
          /* enter critical region */
31         perror ("error on the down operation for semaphore access (CT)");
32         exit (EXIT_FAILURE);
33     }
34
35     // TODO insert your code here
36
37     sh->fSt.st.groupStat[id] = EAT;
38     saveState(nFic, &sh->fSt);
39
40     // fim
41
42     if (semUp (semgid, sh->mutex) == -1) {
        /* enter critical region */
43         perror ("error on the down operation for semaphore access (CT)");
44         exit (EXIT_FAILURE);
45     }
46 }
```

1.4.4. checkOutAtReception (int id)

Nesta função, o **group** espera que o **receptionist** esteja disponível para lhe fazer um pedido, para tal efeito faz **semDown()** do semáforo **receptionistRequestPossible**.

Entra-se pela primeira vez na região crítica, para alterar o estado do **group** para **CHECKOUT**. Define-se o tipo do pedido (**reqType**) como **BILLREQ** pois o grupo pretende pagar. Define-se ainda o grupo que fez o pedido e é salvo o estado atual.

Depois de sair da região crítica faz-se um **semUp()** do semáforo **receptionistReq**, para informar o receptionist do pedido.

Em seguida faz-se **semDown()** do semáforo **tableDone** onde o **group** espera que o pagamento seja concluído.

Após esta fase, entramos de novo na região crítica. Muda-se o estado do grupo para **LEAVING** e é salvo o estado atual.

Em baixo apresentamos o código desta função. Mais uma vez tivemos o cuidado de mexer na memória partilhada apenas dentro do **mutex**.

```
1  static void checkOutAtReception (int id)
2  {
3      // TODO insert your code here
4      if (semDown(semgid, sh->receptionistRequestPossible) == -1) {
5          perror("error on the down operation for semaphore access");
6          exit(EXIT_FAILURE);
7      }
8
9
10     // FIM
11
12     if (semDown (semgid, sh->mutex) == -1) {
           /* enter critical region */
13         perror ("error on the down operation for semaphore access (CT)");
14         exit (EXIT_FAILURE);
15     }
16
17     // TODO insert your code here
18     sh->fSt.st.groupStat[id] = CHECKOUT;
19
20     // signaling receptionist of the request.
21     sh->fSt.receptionistRequest.reqType = BILLREQ;
22     sh->fSt.receptionistRequest.reqGroup = id;
23     saveState(nFic, &sh->fSt);
24
25     int assignedTable = sh->fSt.assignedTable[id]; // aceder a mem.partilhada dentro do
       mutex
26
27     // FIM
28     if (semUp (semgid, sh->mutex) == -1) {
        /* enter critical region */
29         perror ("error on the down operation for semaphore access (CT)");
30         exit (EXIT_FAILURE);
31     }
32
33     // TODO insert your code here
34     if (semUp (semgid, sh->receptionistReq) == -1) {
35         perror ("error on the down operation for semaphore access");
36         exit (EXIT_FAILURE);
37     }
38
39     if (semDown (semgid, (sh->tableDone[assignedTable])) == -1) {
40         perror ("error on the down operation for semaphore access");
41         exit(EXIT_FAILURE);
42     }
43
44
45     // FIM
46     if (semDown (semgid, sh->mutex) == -1) {
           /* enter critical region */
47         perror ("error on the down operation for semaphore access (CT)");
48         exit (EXIT_FAILURE);
49     }
50
```

```
    // TODO insert your code here
    sh->fSt.st.groupStat[id] = LEAVING;
    saveState(nFic, &sh->fSt);

    // FIM

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

}
```

2. Testes ao programa

Para testar a nossa solução, criámos um ficheiro **test.sh**, que nos permitia testar cada parte da nossa solução (chef, waiter, recepcionist e group) **de forma individual** com o restante da solução fornecida pelos docentes. As nossas soluções foram testadas ao longo da criação da aplicação pela ordem que apresentámos no relatório. O script **test.sh** permite testar o nosso programa na íntegra, com todos os seus 4 elementos criados por nós.

De seguida, mostramos a nossa implementação para esse script, que executava o **makefile** e estava dependente dos argumentos passados, já definidos.

```
#!/bin/bash

# comando para remover todos os segmentos de memoria compartilhada no Unix-like systems
# ./clean
ipcrm -a

# Navega para o diretorio src
cd ../src

# Verifica se ha argumentos
if [ $# -eq 0 ]; then
    # Se nao houver argumentos, executa make all_bin
    make all_bin
else
    # Se houver argumentos, executa make com o primeiro argumento
    make "$1"
fi

# Navega para o diretorio run
cd ../run

# Executa o programa probSemSharedMemRestaurant
./probSemSharedMemRestaurant
```

Por exemplo para testarmos o código desenvolvido em **semSharedMemChef.c** fizemos **./test.sh ch** e obtivemos o seguinte resultado, o que nos permitiu aferir o bom funcionamento do nosso código. Como se pode observar todos os grupos conseguiram comer e no final fizeram todos o CHECKOUT. Decidimos não colocar a tabela completa com os resultados para o relatório não ficar muito extenso.

```
joao@joaoh:~/Documents/Universidade/2ano/1S/SO/SO_Project2/semaphore_restaurant/run$ ./test
    .sh ch
cp ../run/group_bin_64 ../run/group
cp ../run/waiter_bin_64 ../run/waiter
gcc -Wall   -c -o semSharedMemChef.o semSharedMemChef.c
gcc -Wall   -c -o sharedMemory.o sharedMemory.c
gcc -Wall   -c -o semaphore.o semaphore.c
gcc -Wall   -c -o logging.o logging.c
gcc -o ../run/chef semSharedMemChef.o sharedMemory.o semaphore.o logging.o -lm
cp ../run/receptionist_bin_64 ../run/receptionist
gcc -Wall   -c -o probSemSharedMemRestaurant.o probSemSharedMemRestaurant.c
gcc -o ../run/probSemSharedMemRestaurant probSemSharedMemRestaurant.o sharedMemory.o
    semaphore.o logging.o -lm
rm -f *.o
                              Restaurant - Description of the internal state

 CH WT RC  G00 G01 G02 G03 G04  gWT T00 T01 T02 T03 T04
  0  0  0    1   1   1   1   1    0   .   .   .   .   .
```

```
  0  0  0    1   1   1   1   1    0   .   .   .   .   .
  0  0  0    1   1   1   1   1    0   .   .   .   .   .
  0  0  0    1   2   1   1   1    0   .   .   .   .   .
  0  0  1    1   2   1   1   1    0   .   .   .   .   .
  0  0  0    1   2   1   1   1    0   .   0   .   .   .
  0  0  0    1   3   1   1   1    0   .   0   .   .   .

                         ...

  0  2  0    7   5   7   7   6    0   .   0   .   .   1
  0  2  2    7   5   7   7   6    0   .   0   .   .   1
  0  2  0    7   5   7   7   6    0   .   0   .   .   .
  0  2  0    7   5   7   7   7    0   .   0   .   .   .
  0  2  0    7   6   7   7   7    0   .   0   .   .   .
  0  2  2    7   6   7   7   7    0   .   0   .   .   .
  0  2  2    7   7   7   7   7    0   .   .   .   .   .
```

Para testar a validade dos restantes ficheiros desenvolvidos: **semSharedMemWaiter.c**, **semSharedMemGroup.c** e **semSharedMemReceptionist.c** fizemos **./test.sh wt**, **./test.sh gr** e **./test.sh rt**, respetivamente. Processo idêntico ao demonstrado em cima para o **chef**.

Depois de garantirmos que todos os participantes (chef, waiter, recepcionist e group) estavam bem feitos, testámos os 4 em conjunto. Para isso fizemos **./test.sh all**. Em baixo demonstramos o resultado de um dos testes feitos ao nosso programa. Todos os grupos conseguiram comer e no final fizeram todos o CHECKOUT.

```
joao@joaoh:~/Documents/Universidade/2ano/1S/SO/SO_Project2/semaphore_restaurant/run$ ./test.sh all
gcc -Wall   -c -o semSharedMemGroup.o semSharedMemGroup.c
gcc -Wall   -c -o sharedMemory.o sharedMemory.c
gcc -Wall   -c -o semaphore.o semaphore.c
gcc -Wall   -c -o logging.o logging.c
gcc -o ../run/group semSharedMemGroup.o sharedMemory.o semaphore.o logging.o -lm
gcc -Wall   -c -o semSharedMemWaiter.o semSharedMemWaiter.c
gcc -o ../run/waiter semSharedMemWaiter.o sharedMemory.o semaphore.o logging.o
gcc -Wall   -c -o semSharedMemChef.o semSharedMemChef.c
gcc -o ../run/chef semSharedMemChef.o sharedMemory.o semaphore.o logging.o -lm
gcc -Wall   -c -o semSharedMemReceptionist.o semSharedMemReceptionist.c
gcc -o ../run/receptionist semSharedMemReceptionist.o sharedMemory.o semaphore.o logging.o -lm
gcc -Wall   -c -o probSemSharedMemRestaurant.o probSemSharedMemRestaurant.c
gcc -o ../run/probSemSharedMemRestaurant probSemSharedMemRestaurant.o sharedMemory.o semaphore.o logging.o -lm
rm -f *.o
                                Restaurant - Description of the internal state

 CH WT RC  G00 G01 G02 G03 G04  gWT T00 T01 T02 T03 T04
  0  0  0    1   1   1   1   1    0   .   .   .   .   .
  0  0  0    1   1   1   1   1    0   .   .   .   .   .
  0  0  0    1   1   1   1   1    0   .   .   .   .   .
  0  0  0    1   2   1   1   1    0   .   .   .   .   .
  0  0  1    1   2   1   1   1    0   .   .   .   .   .
  0  0  0    1   2   1   1   1    0   .   0   .   .   .
  0  0  0    1   3   1   1   1    0   .   0   .   .   .
  0  1  0    1   3   1   1   1    0   .   0   .   .   .
  0  1  0    1   4   1   1   1    0   .   0   .   .   .
  1  1  0    1   4   1   1   1    0   .   0   .   .   .
  1  0  0    1   4   1   1   1    0   .   0   .   .   .
  0  0  0    1   4   1   1   1    0   .   0   .   .   .
  0  2  0    1   4   1   1   1    0   .   0   .   .   .
  0  0  0    1   4   1   1   1    0   .   0   .   .   .
  0  0  0    1   5   1   1   1    0   .   0   .   .   .
  0  0  0    1   5   2   1   1    0   .   0   .   .   .
  0  0  1    1   5   2   1   1    0   .   0   .   .   .
  0  0  0    1   5   2   1   1    0   .   0   1   .   .
  0  0  0    1   5   3   1   1    0   .   0   1   .   .
  0  1  0    1   5   3   1   1    0   .   0   1   .   .
  0  1  0    1   5   4   1   1    0   .   0   1   .   .
  1  1  0    1   5   4   1   1    0   .   0   1   .   .
  1  0  0    1   5   4   1   1    0   .   0   1   .   .
  0  0  0    1   5   4   1   1    0   .   0   1   .   .
  0  2  0    1   5   4   1   1    0   .   0   1   .   .
  0  0  0    1   5   4   1   1    0   .   0   1   .   .
  0  0  0    1   5   5   1   1    0   .   0   1   .   .
  0  0  0    1   5   5   2   1    0   .   0   1   .   .
```

```
  0  0  1    1   5   5   2   1    0   .   0   1   .   .
  0  0  0    1   5   5   2   1    1   .   0   1   .   .
  0  0  0    1   5   5   2   2    1   .   0   1   .   .
  0  0  1    1   5   5   2   2    1   .   0   1   .   .
  0  0  0    1   5   5   2   2    2   .   0   1   .   .
  0  0  0    2   5   5   2   2    2   .   0   1   .   .
  0  0  1    2   5   5   2   2    2   .   0   1   .   .
  0  0  0    2   5   5   2   2    3   .   0   1   .   .
  0  0  0    2   5   6   2   2    3   .   0   1   .   .
  0  0  2    2   5   6   2   2    3   .   0   1   .   .
  0  0  0    2   5   6   2   2    2   1   0   .   .   .
  0  0  0    2   5   7   2   2    2   1   0   .   .   .
  0  0  0    3   5   7   2   2    2   1   0   .   .   .
  0  1  0    3   5   7   2   2    2   1   0   .   .   .
  0  1  0    4   5   7   2   2    2   1   0   .   .   .
  1  1  0    4   5   7   2   2    2   1   0   .   .   .
  1  0  0    4   5   7   2   2    2   1   0   .   .   .
  0  0  0    4   5   7   2   2    2   1   0   .   .   .
  0  2  0    4   5   7   2   2    2   1   0   .   .   .
  0  0  0    4   5   7   2   2    2   1   0   .   .   .
  0  0  0    5   5   7   2   2    2   1   0   .   .   .
  0  0  0    6   5   7   2   2    2   1   0   .   .   .
  0  0  2    6   5   7   2   2    2   1   0   .   .   .
  0  0  0    6   5   7   2   2    1   .   0   .   1   .
  0  0  0    6   5   7   3   2    1   .   0   .   1   .
  0  0  0    7   5   7   3   2    1   .   0   .   1   .
  0  1  0    7   5   7   3   2    1   .   0   .   1   .
  0  1  0    7   5   7   4   2    1   .   0   .   1   .
  1  1  0    7   5   7   4   2    1   .   0   .   1   .
  1  0  0    7   5   7   4   2    1   .   0   .   1   .
  0  0  0    7   5   7   4   2    1   .   0   .   1   .
  0  2  0    7   5   7   4   2    1   .   0   .   1   .
  0  0  0    7   5   7   4   2    1   .   0   .   1   .
  0  0  0    7   5   7   5   2    1   .   0   .   1   .
  0  0  0    7   5   7   6   2    1   .   0   .   1   .
  0  0  2    7   5   7   6   2    1   .   0   .   1   .
  0  0  0    7   5   7   6   2    0   .   0   .   .   1
  0  0  0    7   5   7   7   2    0   .   0   .   .   1
  0  0  0    7   5   7   7   3    0   .   0   .   .   1
  0  1  0    7   5   7   7   3    0   .   0   .   .   1
  1  1  0    7   5   7   7   3    0   .   0   .   .   1
  1  0  0    7   5   7   7   3    0   .   0   .   .   1
  1  0  0    7   5   7   7   4    0   .   0   .   .   1
  0  0  0    7   5   7   7   4    0   .   0   .   .   1
  0  2  0    7   5   7   7   4    0   .   0   .   .   1
  0  2  0    7   5   7   7   5    0   .   0   .   .   1
  0  2  0    7   5   7   7   6    0   .   0   .   .   1
  0  2  2    7   5   7   7   6    0   .   0   .   .   1
  0  2  0    7   5   7   7   6    0   .   0   .   .   .
  0  2  0    7   5   7   7   7    0   .   0   .   .   .
  0  2  0    7   6   7   7   7    0   .   0   .   .   .
  0  2  2    7   6   7   7   7    0   .   0   .   .   .
  0  2  2    7   7   7   7   7    0   .   .   .   .   .
```

Testámos, também, a nossa solução com o script **run.sh** dado pelos docentes, para garantir que o programa funcionava corretamente independentemente do número de vezes que fosse executado.

Por exemplo, executámos a nossa solução 10000 vezes, **./run.sh 10000**, tendo funcionado corretamente todas as vezes. Em baixo apresentamos o resultado da iteração 10000.

```

 Run n 10000
                                  Restaurant - Description of the internal state

CH WT RC  G00 G01 G02 G03 G04  gWT T00 T01 T02 T03 T04
 0  0  0    1   1   1   1   1    0   .   .   .   .   .
 0  0  0    1   1   1   1   1    0   .   .   .   .   .
 0  0  0    1   1   1   1   1    0   .   .   .   .   .
 0  0  0    1   2   1   1   1    0   .   .   .   .   .
 0  0  1    1   2   1   1   1    0   .   .   .   .   .
 0  0  0    1   2   1   1   1    0   .   0   .   .   .
                          ...

 0  2  2    7   5   7   7   6    0   .   0   .   .   1
 0  2  0    7   5   7   7   6    0   .   0   .   .   .
```

```
16  0  2  0    7  5  7  7  7    0  .  0  .  .  .
17  0  2  0    7  6  7  7  7    0  .  0  .  .  .
18  0  2  2    7  6  7  7  7    0  .  0  .  .  .
19  0  2  2    7  7  7  7  7    0  .  .  .  .  .
```

3. Conclusões

Este trabalho teve como objetivo o desenvolvimento de uma aplicação que permitisse simular um restaurante. Foi nos bastante útil, porque ajudou a desenvolver algumas competências em C e também porque melhorou o nosso trabalho em equipa. No entanto, o mais enriquecedor deste trabalho, foi o desenvolvimento de capacidades para programar com semáforos e memória partilhada, para garantir a sincronização de processos.

References

[1] How to use POSIX semaphores in C language. Site consultado a 23/12/2023 https://www.geeksforgeeks.org/use-posix-semaphores-c/

[2] Semaphores in Process Synchronization. Site consultado a 24/12/2023: https://www.geeksforgeeks.org/semaphores-in-process-synchronization/.