

Licenciatura em Engenharia Informática

Sistemas Operativos

“Restaurante”

Aveiro, 11 de janeiro de 2019



universidade
de aveiro

Ana Sofia Fernandes, nmec 88739

Carina Neves, nmec 90451

Índice

Índice	1
Introdução	2
Introdução ao problema	3
Implementação da solução	7
semSharedMemChef.c	10
Função waitForOrder()	10
Função processOrder()	12
semSharedMemGroup.c	13
Função checkInAtReception(int id)	14
Função orderFood(int id)	15
Função waitFood(int id)	17
Função checkOutAtReception(int id)	18
semSharedMemReceptionist.c	20
Função decideTableOrWait(int n)	20
Função decideNextGroup()	21
Função waitForGroup()	22
Função provideTableOrWaitingRoom(int n)	24
Função receivePayment(int n)	25
semSharedMemWaiter.c	27
Função waitForClientOrChef()	27
Função informChef(int n)	29
Função takeFoodToTable(int n)	31
Resultados obtidos	32
Conclusão	36

Introdução

Este segundo trabalho prático, da unidade curricular de *Sistemas Operativos*, que possui como objetivo a compreensão de mecanismos associados à execução e sincronização de processos e *threads*, surge da solicitação para o desenvolvimento de uma aplicação que simula o funcionamento de um restaurante, tomando como ponto de partida o código fonte disponível na página da disciplina.

O trabalho foi realizado por grupos de dois alunos, e a data de entrega é o dia 11 de Janeiro de 2019.

Introdução ao problema

Como dito anteriormente, o problema consiste na implementação de uma aplicação, na linguagem de programação C, que simula o funcionamento de um restaurante. No código fornecido, já se encontram definidos os lugares destinados à inserção de novo código, de forma a obter a implementação final, bem como as regiões críticas (zonas de acesso à memória partilhada através da utilização do semáforo *mutex*).

O restaurante possui, na sua constituição:

- Duas mesas;
- Um rececionista (*receptionist*);
- Um empregado de mesa (*waiter*);
- Um cozinheiro (*chef*).

Para que tudo corra como desejado, existem algumas regras:

1. Cada grupo (*client*) deverá dirigir-se, primeiramente, ao rececionista, que lhe indicará a mesa a ocupar ou que deve esperar.
2. Após ocupar a mesa respetiva, o grupo deve pedir comida ao empregado de mesa e aguardar para, depois, começar a comer, assim que este lhe entrega o pedido. Ou seja, o *waiter* deve levar o pedido ao cozinheiro (que apenas recebe pedidos do *waiter* e prepara a comida), e entregar o prato assim que este se encontre pronto.
3. Assim que acabar de comer, o grupo volta a contactar com o rececionista, de forma a pedir a conta, pagar e sair.

Neste restaurante, as quatro entidades/processos independentes - *receptionist*, *waiter*, *chef* e *client* - devem estar sincronizadas, para que consigam rastrear o estado umas das outras. Esta sincronização será feita através de semáforos e memórias partilhadas. No entanto, cada entidade apenas altera o seu próprio estado - não pode interferir com o estado dos outros processos.

Desta forma, cada entidade apresenta vários estados (que identificam a função que se encontram a realizar), definidos no ficheiro `probConst.h` (que podemos encontrar no diretório `/src`):

```
/** \brief id of table request (group->receptionist) */
#define TABLEREQ 1
/** \brief id of bill request (group->receptionist) */
#define BILLREQ 2
/** \brief id of food request (group->waiter) */
#define FOODREQ 3
/** \brief id of food ready (chef->waiter) */
#define FOODREADY 4
```

Figura 1 - Estados gerais

```
/* Client state constants */

/** \brief group initial state */
#define GTOREST 1
/** \brief client is waiting at reception or waiting for table */
#define ATRECEPTION 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 checking out */
#define CHECKOUT 6
/** \brief client is leaving */
#define LEAVING 7
```

Figura 2 - Estados do *group*

```
/* 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*

```
/* Waiter state constants */

/** \brief waiter/receptionist 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
```

Figura 4 - Estados do *waiter*

```
/* Receptionist state constants */

/** \brief receptionist waits for food request */
#define ASSIGNTABLE 1
/** \brief receptionist receives payment */
#define RECVPAY 2
```

Figura 5 - Estados do *receptionist*

Encontram-se também definidos os valores para as condições iniciais do problema - o número máximo de grupos que podem estar presentes, o número de mesas existentes, bem como o tempo máximo permitido para a confecção do pedido. Estes valores estão também contidos no ficheiro `probConst.h`.

```
/* Generic parameters */

/** \brief maximum number of groups */
#define MAXGROUPS 16
/** \brief number of tables */
#define NUMTABLES 2
/** \brief controls time taken to cook */
#define MAXCOOK 100
```

Figura 6 - Condições iniciais do pedido

Já o conteúdo relativo à memória partilhada (`FULL_STAT fst`), bem com uma lista de índices do array de semáforos, podem ser encontrados no ficheiro `sharedDataSync.h`. Nesse array, são fornecidos 11 semáforos:

```
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 receptionist to wait for groups - val = 0 */
    unsigned int receptionistReq;
    /** \brief identification of semaphore used by groups to wait before issuing receptionist request - val = 1 */
    unsigned int receptionistRequestPossible;
    /** \brief identification of semaphore used by waiter to wait for requests - val = 0 */
    unsigned int waiterRequest;
    /** \brief identification of semaphore used by groups and chef to wait before issuing waiter request - val = 1 */
    unsigned int waiterRequestPossible;
    /** \brief identification of semaphore used by chef to wait for order - val = 0 */
    unsigned int waitOrder;
    /** \brief identification of semaphore used by waiter to wait for chef - val = 0 */
    unsigned int orderReceived;
    /** \brief identification of semaphore used by groups to wait for table - val = 0 */
    unsigned int waitForTable[MAXGROUPS];
    /** \brief identification of semaphore used by groups to wait for waiter acknowledge - val = 0 */
    unsigned int requestReceived[NUMTABLES];
    /** \brief identification of semaphore used by groups to wait for food - val = 0 */
    unsigned int foodArrived[NUMTABLES];
    /** \brief identification of semaphore used by groups to wait for payment completed - val = 0 */
    unsigned int tableDone[NUMTABLES];
} SHARED_DATA;

/** \brief number of semaphores in the set */
#define SEM_NU      ( 7 + sh->fSt.nGroups + 3*NUMTABLES )

#define MUTEX      1
#define RECEPTIONISTREQ      2
#define RECEPTIONISTREQUESTPOSSIBLE      3
#define WAITERREQUEST      4
#define WAITERREQUESTPOSSIBLE      5
#define WAITORDER      6
#define ORDERRECEIVED      7
#define WAITFORTABLE      8
#define FOODARRIVED      (WAITFORTABLE+sh->fSt.nGroups)
#define REQUESTRECEIVED      (FOODARRIVED+NUMTABLES)
#define TABLEDONE      (REQUESTRECEIVED+NUMTABLES)
```

Figura 7 - Ficheiro sharedDataSync.h

Implementação da solução

Dado que, como já referido anteriormente, a resolução do problema será feita a partir da utilização de semáforos e de memória partilhada, foi sugerido que, primeiramente, se começasse por analisar cada semáforo, de forma a perceber a sua finalidade e onde este deve ser aplicado. Na tabela seguinte, pode ser encontrada toda essa informação:

	Descrição	Up	Down
<i>mutex</i>	Define se estamos ou não na região crítica.	Usado para sair da região crítica.	Usado para aceder à região crítica.
<i>receptionistReq</i>	Usado pelo <i>receptionist</i> para esperar pelos grupos.	Feito pelo grupo quando o pedido se encontra concluído.	Feito pelo rececionista quando recebe o pedido.
<i>receptionistRequestPossible</i>	Usado pelos grupos para esperar que o <i>receptionist</i> realize o pedido.	Feito pelo <i>receptionist</i> após receber o pedido.	Feito pelo grupo para realizar um pedido.
<i>waiterRequest</i>	Usado pelo <i>waiter</i> para esperar por pedidos, tanto do grupo, como do <i>chef</i> .	Feito pelo <i>chef</i> ou pelo grupo de forma a chamar o <i>waiter</i> .	Feito pelo <i>waiter</i> quando está disponível a receber pedidos.

Segundo Trabalho Prático: Restaurante

<i>waiterRequestPossible</i>	<p>Usado tanto pelos grupos, como pelo <i>chef</i>, de forma a esperar antes de realizar o pedido ao <i>waiter</i>.</p> <p>Sincroniza, também, os acessos ao semáforo <i>waiterRequest</i> (diz se é possível usar)</p>	<p>Feito pelo <i>waiter</i> de forma a dizer se está disponível a receber pedidos.</p>	<p>Feito pelo grupo ou pelo <i>chef</i> quando pretendem chamar o <i>waiter</i>.</p>
<i>waitOrder</i>	<p>Usado pelo <i>chef</i> para esperar por um pedido.</p>	<p>Feito pelo <i>waiter</i> quando faz o pedido (só depois de atribuir o <i>foodOrder - flag</i> que sinaliza um pedido de comida realizado pelo <i>waiter</i> ao <i>chef</i> - e o <i>foodGroup</i> - grupo associado ao pedido de comida). realizado pelo <i>waiter</i> ao <i>chef</i>).</p>	<p>Feito pelo <i>chef</i> quando este se encontra disponível para receber um pedido.</p>

Segundo Trabalho Prático: Restaurante

<i>orderReceived</i>	Usado pelo <i>waiter</i> de forma a esperar pelo <i>chef</i> .	Feito pelo <i>chef</i> para confirmar que recebeu o pedido por parte do <i>waiter</i> .	Feito pelo <i>waiter</i> para esperar que o pedido que realizou seja aceite.
<i>waitForTable[MAXGROUPS]</i>	Usado pelos grupos para esperar por mesa (<i>MAXGROUPS</i> corresponde ao número máximo de grupos que é possível configurar - cada grupo tem um número associado).	Feito pelo <i>receptionist</i> de forma a atribuir mesa.	Feito pelo grupo enquanto espera.
<i>requestReceived[NUMTABLES]</i>	Usado pelos grupos para esperar pelo <i>waiter</i> (<i>NUMTABLES</i> corresponde ao número da mesa respetiva, que pode ser 0 ou 1).	Feito pelo <i>waiter</i> após concluir o pedido de <i>NUMTABLES</i> .	Feito pelo grupo enquanto espera.

<i>foodArrived[NUMTABLES]</i>	Usado pelos grupos para esperar pela comida.	Feito pelo <i>waiter</i> quando entrega o pedido (leva a comida à mesa).	Feito pelo grupo após fazer o seu pedido ao <i>waiter</i> .
<i>tableDone[NUMTABLES]</i>	Usado pelos grupos para esperar que o pagamento seja concluído.	Feito pelo <i>receptionist</i> quando o pagamento fica concluído.	Feito pelo grupo após realizar pedido para pagamento.

Tabela 1 - Semáforos existentes e respetiva explicação

A partir da utilização de semáforos, permitir-se-à que todas as entidades troquem informação entre elas, sem nunca colidirem. Desta forma irá estar-se, também, em contínua atualização do estado das mesmas.

semSharedMemChef.c

A primeira entidade tratada foi o *chef*. Verificou-se que esta seria uma boa forma de começar a resolução do problema, dado que o cozinheiro apenas interage com o empregado de mesa. Isto permitiria um bom começo pois, a partir da entidade mais simples, iria perceber-se o modo de funcionamento de todo o programa, tornando mais fáceis e rápidas as próximas implementações.

Função *waitForOrder()*

Esta é, então, a primeira função do ficheiro a necessitar de código. Aqui, o *chef* vai esperar por receber um pedido de confeção por parte do *waiter*, atualizar o seu estado e guardar o seu estado interno, e deve também sinalizar que recebeu um pedido.

Neste relatório irão ser analisadas, apenas, as zonas em que foi pedido para inserir código. Assim, o primeiro semáforo usado (na primeira zona destinada à inserção de código, *TODO*) é o *waitOrder*, fazendo-lhe *down*, uma vez que o *chef* se encontra disponível para receber um pedido.

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

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

    /*-----END TODO-----*/
}
```

Figura 8 - Primeiro *TODO* da função *waitForOrder()*

Após entrar na região crítica, atribui-se à variável inteira *lastGroup* o valor de *foodGroup* (ou seja, o grupo mais recente é o grupo associado àquele pedido). O estado do *chef* é atualizado para *COOK*, dado que se encontra a confeccionar o pedido, e a *flag foodOrder* é colocada a 0. O estado do *chef* é, então, guardado.

```
//TODO insert your code here

lastGroup=sh->fSt.foodGroup;

sh->fSt.chefStat = COOK;    // altera o estado para COOK (cozinha e depois informa que está livre)
sh->fSt.foodOrder = 0;      // atualiza flag
saveState (nFic, &sh->fSt); // guarda estado

/*-----END TODO-----*/
```

Figura 9 - Segundo *TODO* da função *waitForOrder()*

Por último, é usado o semáforo *orderReceived*, realizando-se um *up* (agora já fora da região crítica) - o *chef* informa que recebeu um pedido por parte do *waiter*.

```
//TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 10 - Terceiro e último *TODO* da função *waitForOrder()*

Função *processOrder()*

Na segunda função do ficheiro *semSharedMemChef.c*, o *chef* cozinha e entrega o pedido ao *waiter* - o *chef* leva algum tempo a cozinhar, dizendo ao *waiter*, seguidamente, que a confeção terminou e o pedido está pronto (isto só pode acontecer se o *waiter* estiver disponível); posteriormente, atualiza o seu estado e o seu estado interno é guardado.

Nesta função, começa-se por fazer um *down* no semáforo *waiterRequestPossible*, de forma a chamar o *waiter* e aguardar pela vinda deste.

```
//TODO insert your code here

if (semDown (semgid, sh->waiterRequestPossible) == -1) { //verificar disponibilidade do waiter
    perror ("error on the down operation for semaphore access (PT)");
    exit (EXIT_FAILURE);
}

/*-----END TODO-----*/
```

Figura 11 - Primeiro *TODO* da função *processOrder()*

Em seguida, acede-se a *fSt* (estrutura *FULL_STAT*), onde se vai aceder a *waiterRequest* (variável da estrutura *request*) e alterar o *reqType* (id do pedido, ou seja, especifica o tipo de pedido) e o *reqGroup* (grupo que fez o respetivo pedido). Desta forma, o *waiter* é informado que o *reqGroup* de que tem que tratar é o *lastGroup*, e que o id do pedido corresponde ao id de um pedido quando este se encontra pronto (*FOODREADY*).

Seguidamente, realizam-se duas comparações:

1. Se ainda existirem grupos por atender, o estado do *chef* será atualizado para *WAIT_FOR_ORDER*,
2. Se já não existem mais grupos à espera de serem atendidos, o estado do *chef* é atualizado para *REST*.

```
//TODO insert your code here

//dizer que o pedido está pronto p este grupo
sh->fSt.waiterRequest.reqGroup = lastGroup ;
sh->fSt.waiterRequest.reqType = FOODREADY; //o id do redType vai corresponder a FOODREADY que é = 4 (ou seja, id a significar que a comida e

if (count < sh->fSt.nGroups)
{
    //se o contador for menor que o numero de grupos, os pedidos ainda nao acabaram
    sh->fSt.st.chefStat = WAIT_FOR_ORDER; //por o estado em espera
    saveState (nFic, &sh->fSt); //guardar estado
    count++;
}
else if (count == sh->fSt.nGroups)
{
    //caso o contador seja igual ao numero de grupos, significa que todos os pedidos estão feitos e que o chef pode descansar
    sh->fSt.st.chefStat = REST; //por o estado em descanso
    saveState (nFic, &sh->fSt); //guardar estado
}

/*-----END TODO-----*/
```

Figura 12 - Segundo *TODO* da função *processOrder()*

Por último, realiza-se um *up* do semáforo *waiterRequest*, de forma a chamar o *waiter*.

```
//TODO insert your code here

if (semUp (semgid, sh->waiterRequest) == -1) { //desbloqueia o semaforo do waiterRequest
    perror ("error on the up operation for semaphore access (PT)");
    exit (EXIT_FAILURE);
}

/*-----END TODO-----*/
```

Figura 13 - Terceiro e último *TODO* da função *processOrder()*

semSharedMemGroup.c

Optou-se por, em segundo lugar, tratar do grupo, dado que este interage com o *receptionist* e com o *waiter*, o que pareceu que seria o mais simples, de entre as restantes entidades.

Função *checkInAtReception(int id)*

Esta função recebe como parâmetro um *id*, que identificará o grupo que está a ser tratado, no momento. Em *checkInAtReception(int id)*, o grupo deve, assim que o *receptionist* se encontre disponível, pedir uma mesa. O grupo pode, ou não, ter que esperar que o seu pedido se realize. O estado interno do grupo deverá ser guardado.

O primeiro *TODO* passa por se fazer um *down* ao semáforo *receptionistRequestPossible*, indicando que o grupo realiza um pedido.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 14 - Primeiro *TODO* da função *checkInAtReception(int id)*

Posteriormente, na região crítica, é atribuído a *groupStat[id]* o estado *ATRECEPTION*, ou seja, o estado do grupo (identificado pelo *id* respetivo) será, nesse momento, *ATRECEPTION*. Quanto a *reqGroup* e *reqType* (explicados anteriormente) da variável do tipo *request*, relativos a *receptionistRequest*, terão os valores *id* e *TABLEREQUEST* (dado que foi pedida uma mesa), respetivamente, indicando que o *receptionist* se encontra a receber um pedido de mesa para aquele grupo.

O estado do grupo é, assim, guardado.

```
// TODO insert your code here

sh->fSt.st.groupStat[id]=ATRECEPTION;
sh->fSt.receptionistRequest.reqGroup = id ;
sh->fSt.receptionistRequest.reqType = TABLEREQ;
saveState (nFic, &sh->fSt);

/*-----END TODO-----*/
```

Figura 15 - Segundo *TODO* da função *checkInAtReception(int id)*

Vão agora ser tratados dois semáforos: *receptionistReq* e *waitForTable[id]*. No primeiro, é feito um *up*, onde o grupo indica que o pedido de mesa está concluído. Em *waitForTable[id]*, é feito um *down*, e indica que o grupo se encontra a aguardar por uma mesa.

```
// TODO insert your code here

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

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

/*-----END TODO-----*/
```

Figura 16 - Terceiro e último *TODO* da função *checkInAtReception(int id)*

Função *orderFood(int id)*

Aqui, o grupo deve atualizar o seu estado, realizando um pedido de comida ao *waiter* e esperando que o *waiter* receba o seu pedido. O estado interno deverá ser guardado.

No primeiro *TODO*, é apenas feito um *down* do semáforo *waiterRequestPossible*, indicando que o grupo pretende chamar o *waiter*.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 17 - Primeiro *TODO* da função *orderFood(int id)*

Seguidamente, ao entrar na região crítica, criamos a variável inteira *mesa*, que vai ter como valor um elemento do array *assignedTable*. Este array pode ter como elementos 0 (indica que a mesa 0 está vaga), 1 (indica que a

mesa 1 está vaga) e -1 (indica que todas as mesas existentes estão ocupadas). Ou seja, a variável *mesa* irá conter a mesa ocupada pelo grupo passado em parâmetro. Também aqui é atualizado o estado do grupo para *FOOD_REQUEST* (indicando que fez um pedido de comida), e o *reqGroup* e *reqType* de *waiterRequest* terão como valores *id* e *FOODREQ* (visto que o grupo solicitou comida), respetivamente. O estado atual é guardado.

```
// TODO insert your code here
int mesa = sh->fSt.assignedTable[id];

sh->fSt.st.groupStat[id]=FOOD_REQUEST;
sh->fSt.waiterRequest.reqGroup = id ;
sh->fSt.waiterRequest.reqType = FOODREQ;
saveState (nFic, &sh->fSt);

/*-----END TODO-----*/
```

Figura 18 - Segundo *TODO* da função *orderFood(int id)*

Agora, vai ser realizado um *up*, no semáforo *waiterRequest*, que indica que o grupo está a chamar o *waiter*, e um *down*, no semáforo *requestReceived*, que indica que o grupo está à espera que o seu pedido seja realizado.

```
// TODO insert your code here

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

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

/*-----END TODO-----*/
```

Figura 19 - Terceiro e último *TODO* da função *orderFood(int id)*

Função *waitFood(int id)*

Nesta função, o grupo atualiza o seu estado e espera até que a sua comida chegue. Deve, também, atualizar o seu estado depois da comida chegar. O estado interno tem que, desta forma, ser guardado duas vezes.

No primeiro *TODO* é, novamente, criada a variável *mesa*, com a mesma finalidade, e sempre criada dentro da região crítica. O estado do grupo é atualizado para *WAIT_FOR_FOOD* e, seguidamente, guardado.

```
// TODO insert your code here

int mesa = sh->fSt.assignedTable[id];
sh->fSt.st.groupStat[id]=WAIT_FOR_FOOD;
saveState (nFic, &sh->fSt);

/*-----END TODO-----*/
```

Figura 20 - Primeiro *TODO* da função *waitFood(int id)*

Posteriormente, é realizado um *down* no semáforo *foodArrived[mesa]*, que o grupo realiza após fazer o seu pedido ao *waiter*.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 21 - Segundo *TODO* da função *waitFood(int id)*

Por último, o estado do grupo é atualizado para *EAT*, dado que já se encontra a comer. O seu estado é novamente guardado.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 22 - Terceiro e último *TODO* da função *waitFood(int id)*

Função *checkOutAtReception(int id)*

Agora, o grupo deve, assim que o *receptionist* esteja disponível, atualizar o seu estado e realizar um pedido de pagamento ao *receptionist*. O grupo deve esperar que o *receptionist* permita o pagamento e deve, depois, mudar o seu estado para *LEAVING*. Novamente, o estado interno deve ser guardado sempre que alterado.

Primeiramente, é feito um *down* ao semáforo *receptionistRequestPossible*, indicando que o grupo deseja realizar um pedido.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 23 - Primeiro *TODO* da função *checkOutAtReception(int id)*

No segundo *TODO*, já dentro da região crítica, é novamente criada a variável inteira *mesa*. O estado do grupo é atualizado para *CHECKOUT*, o *reqGroup* e *reqType* de *receptionistRequest* terão como valores *id* e *BILLREQ* (visto que o grupo fez um pedido de pagamento), respetivamente, e o estado é guardado.

```
// TODO insert your code here

int mesa = sh->fSt.assignedTable[id];
sh->fSt.st.groupStat[id]=CHECKOUT;
sh->fSt.receptionistRequest.reqGroup = id ;
sh->fSt.receptionistRequest.reqType = BILLREQ;
saveState (nFic, &sh->fSt);

/*-----END TODO-----*/
```

Figura 24 - Segundo *TODO* da função *checkOutAtReception(int id)*

No próximo *TODO*, fora da região crítica, faz-se um *up* ao semáforo *receptionistReq*, indicando que o pedido se encontra concluído, e um *down* a *tableDone[mesa]*, visto que o grupo já realizou o pagamento.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 25 - Terceiro *TODO* da função *checkOutAtReception(int id)*

Por último, novamente dentro da região crítica, o estado do grupo é atualizado para *LEAVING* e, logo a seguir, este é guardado.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 26 - Quarto e último *TODO* da função *checkOutAtReception(int id)*

semSharedMemReceptionist.c

A próxima entidade a ser tratada foi o receptionist, que interage com o grupo e com o waiter.

Função *decideTableOrWait(int n)*

Nesta função, é decidido se o grupo *n* ocupa uma mesa ou se deve esperar. O estado atual das mesas é verificado, bem como o dos grupos, de forma a tomar a decisão. Será retornado o *id* da mesa ou -1, se ambas as mesas estiverem ocupadas e o grupo for obrigado a esperar.

Começa-se por inicializar duas variáveis inteiras: *contador_zero* e *contador_um*.

Seguidamente, é realizado um *for*, a iterar sobre *array assignedTable* (conterá o valor 0 se a mesa 0 estiver ocupada, 1 se a mesa 1 estiver ocupada, e -1 se ambas as mesas estiverem ocupadas). Se o valor encontrado for zero, incrementamos a variável *contador_zero*, pois significa que a mesa 0 já foi atribuída. Se o valor encontrado for um, significa que a mesa 1 está ocupada, e que podemos incrementar a variável *contador_um*.

Realizam-se, agora, duas verificações:

1. Se o *contador_zero* for diferente de um, significa que a mesa 0 estará livre. Assim, atribui-se essa mesa ao grupo atual (*assignedTable[n]=0*) e atualiza-se o estado do grupo para *ATABLE*, visto que lhe foi atribuída mesa. Esta atualização também poderia ser feita mais tarde, após chamada a função que estamos a analisar, no entanto, por organização, optou-se por fazê-la aqui. É retornado o valor 0 (*id* da mesa ocupada).
2. Se o *contador_um* for diferente de um, significa que a mesa 1 estará livre. Assim, vai-se atribuir essa mesa ao grupo atual (*assignedTable[n]=1*) e atualiza-se o estado do grupo para *ATABLE*, visto que lhe foi atribuída mesa. É retornado 1 (*id* da mesa ocupada).

Caso nenhuma destas situações se verifique, será retornado -1, indicando que o grupo necessita de esperar até que uma mesa vague.

```

| //TODO insert your code here
| //se houver mesa livre, ele vai decidir qual mesa atribuir
| //se nao houver, poe os grupos na sala de espera

int contador_zero=0;
int contador_um=0;

for (int g=0; g < sh->fSt.nGroups; g++) {
    if (sh->fSt.assignedTable[g] == 0){
        contador_zero++;
    }
    else if (sh->fSt.assignedTable[g] == 1){
        contador_um++;
    }
}

if (contador_zero != 1){
    sh->fSt.assignedTable[n]=0;
    groupRecord[n] = ATTABLE;
    return 0; //se a mesa 0 estiver livre
}

else if (contador_um != 1){
    sh->fSt.assignedTable[n]=1;
    groupRecord[n] = ATTABLE;
    return 1; //se a mesa 1 estiver livre
}

/*-----END TODO-----*/
```

Figura 27 - Único *TODO* da função *decideTableOrWait(int n)*

Função *decideNextGroup()*

Esta função será chamada quando uma mesa fica vaga e há grupos que se encontram à espera, ou seja, vai decidir qual o próximo grupo (se é que este existe) a ocupar a mesa. O estado atual das mesas e dos grupos deve ser verificado de forma a ser tomada uma decisão.

O único *TODO* existente consiste em iterar, até ao tamanho de *nGroups* (grupos existentes), o array *groupRecord*, que indica o estado de cada grupo. Se, neste array, se encontrar um grupo que se encontra à espera, ou seja, no estado *WAIT*, esse grupo é retornado; caso contrário, é retornado -1, indicando que a decisão tomada foi a de espera.

```
static int decideNextGroup()
{
    for (int g=0; g < sh->fSt.nGroups; g++) {
        if (groupRecord[g] == WAIT){
            return g;
        }
    }
    return -1;
}
```

Figura 28 - Único *TODO* da função *decideNextGroup(int id)*

Função *waitForGroup()*

Nesta função, o *receptionist* espera pelo próximo pedido - atualiza o seu estado, espera por um pedido por parte do grupo e, seguidamente, lê o pedido. Vai, agora, sinalizar que está disponível para um novo pedido. O estado interno deve ser salvo. A função irá retornar o pedido realizado pelo grupo.

Começa-se por atualizar o estado do *receptionist*, dentro da região crítica, para *WAIT_FOR_REQUEST*, visto que este se encontra a esperar por um pedido.

```
// TODO insert your code here

sh->fSt.st.receptionistStat=WAIT_FOR_REQUEST;
saveState (nFic, &sh->fSt);

/*-----END TODO-----*/
```

Figura 29 - Primeiro *TODO* da função *waitForGroup()*

No *TODO* seguinte, já fora da região crítica, realiza-se um *down* no semáforo *receptionistReq*, realizado pelo *receptionist* quando aguarda um pedido.


```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 30 - Segundo *TODO* da função *waitForGroup()*

Seguidamente, e mais uma vez dentro da região crítica, são feitas duas comparações (de forma a especificar o tipo de pedido respetivo de cada grupo):

1. Se o pedido realizado ao *receptionist* for um pedido de mesa, o *reqType* de *ret* (variável do tipo *request* que já havia sido inicializada pelo professor, no início da função, e que será retornada por esta) é atualizado para *TABLEREQ* e o *reqGroup* será o mesmo de *receptionistRequest*.
2. Se o pedido realizado ao *receptionist* for um pedido de pagamento, o *reqType* de *ret* será atualizado para *BILLREQ*, e o *reqGroup* será, novamente, o mesmo de *receptionistRequest*.

```
// TODO insert your code here

if (sh->fSt.receptionistRequest.reqType == TABLEREQ){
    ret.reqType = TABLEREQ;
    ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
}

if (sh->fSt.receptionistRequest.reqType == BILLREQ){
    ret.reqType = BILLREQ;
    ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
}

/*-----END TODO-----*/
```

Figura 31 - Terceiro *TODO* da função *waitForGroup()*

No último *TODO*, fora da região crítica, é feito um *up* no semáforo *receptionistRequestPossible*, dado que o *receptionist* se encontra disponível para receber um novo pedido.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 32 - Quarto e último *TODO* da função *waitForGroup()*

Função *provideTableOrWaitingRoom(int n)*

Aqui, o *receptionist* vai decidir se o grupo deve ocupar uma mesa ou esperar. Deve começar por atualizar o seu estado e tomar a decisão. Tanto a memória partilhada como a interna devem ser atualizadas. Se o grupo decidir ocupar uma mesa, deve ser informado se pode ou não proceder. Mais uma vez, o estado interno deve ser guardado. A função recebe como parâmetro *n*, que indica o grupo a ser tratado no momento.

No único *TODO* existente nesta função, que se realiza dentro da região crítica, começa-se por atualizar o estado do *receptionist* para *ASSIGNTABLE*, dado que ele se encontra a atribuir uma mesa, e guardamos o estado.

Seguidamente, inicializa-se uma variável inteira *mesa*, desta vez igualada ao valor de retorno de *decideTableOrWait(n)*. Ou seja, vai conter 0, 1 ou -1.

Agora, voltam-se a realizar duas comparações:

1. Caso o valor de *mesa* seja diferente de -1 (significa que uma das outras mesas está livre), atribuímos esse valor ao grupo atual, e guardamos o seu estado. Fazemos também *up* do semáforo *waitForTable[n]*, visto que o grupo atual se encontra à espera que uma mesa seja atribuída.
2. Caso não se verifique a condição "*mesa != -1*", colocamos o estado do grupo em *WAIT* e incrementamos a variável *groupsWaiting*, que está

contida na estrutura *fSt* e que indica quantos grupos se encontram em espera.

```
// TODO insert your code here

(sh->fSt.st.receptionistStat)=ASSIGNTABLE;
saveState (nFic, &sh->fSt);

int mesa = decideTableOrWait(n);

if (mesa != -1){
    sh->fSt.assignedTable[n]=mesa;
    saveState (nFic, &sh->fSt);

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

/*-----END TODO-----*/
```

Figura 33 - Único *TODO* da função *provideTableOrWaitingRoom(int n)*

Função *receivePayment(int n)*

Nesta função, o *receptionist* irá receber o pagamento. Atualizará o seu estado e receberá o pagamento. Se existirem grupos à espera, deve verificar se existe alguma mesa vaga que possa ser ocupada. Tanto a memória interna como a partilhada devem ser atualizadas e o estado guardado.

Apesar de esta função conter apenas um *TODO* (todo situado dentro da região crítica), neste relatório irá ser apresentado em subpartes, de forma a ser mais fácil de interpretar.

Começa-se por inicializar uma variável inteira, *grupo_ret*, que será mais tarde usada. O estado do *receptionist* é atualizado para *RECVPAY*, visto que se encontra a receber pagamento. O estado é, assim, salvo.

É realizado um *up* no semáforo *tableDone[sh->fst.assignedTable[n]]*, indicando que o pagamento foi realizado para o grupo em questão.

Em seguida, é criada uma variável inteira, *mesa_disp*, que guarda o valor da mesa que vagou, ou seja, a mesa onde se encontrava o grupo que

acabou de pagar a sua conta. Igualar-se, então, *sh->fst.assignedTable[n]* a -1, visto que é necessário libertar a mesa do grupo que acabou de realizar o pagamento. O estado é guardado.

```
// TODO insert your code here
int grupo_ret;
sh->fst.st.receptionistStat=RECVPAY; //recebe pagamento
saveState(nFic, &sh->fst);

if (semUp (semgid, sh->tableDone[sh->fst.assignedTable[n]]) == -1) {
    perror ("error on the up operation for semaphore access (WT)"); //rececionista da pagamento como concluido
    exit (EXIT_FAILURE);
}

int mesa_disp = sh->fst.assignedTable[n]; //guarda valor da mesa em que o grupo estava
sh->fst.assignedTable[n]=-1; //liberta a mesa pois o grupo foi embora
saveState(nFic, &sh->fst);
```

Figura 34 - Primeira parte do *TODO* da função *receivePayment(int n)*

Vai, agora, verificar-se se existem grupos em espera. Se existirem, a variável *grupo_ret* vai tomar o valor de retorno da função *decideNextGroup()*, de forma a guardar o próximo grupo que irá ocupar uma mesa, de entre aqueles que se encontravam à espera. Assim, a mesa que será atribuída a esse grupo, será a mesa guardada em *mesa_disp*. Posteriormente, atualizamos o estado do grupo que realizou o pagamento (grupo *n*) para *DONE*, e o estado do grupo a quem foi atribuída mesa (*grupo_ret*) para *ATTABLE*, visto que já se encontra na sua mesa. A variável *groupsWaiting* é, então, decrementada. O estado do *receptionist* é atualizado para *ASSIGNTABLE*, dado que acabou de atribuir uma mesa.

```
if (sh->fst.groupsWaiting != 0){ // se houver grupos à espera
    grupo_ret = decideNextGroup(); // vê o 1º grupo a espera, grupo_ret
    sh->fst.assignedTable[grupo_ret] = mesa_disp; //grupo que esta a espera ocupa a mesa que ficou livre
    groupRecord[n] = DONE; //atualização do groupRecord
    groupRecord[grupo_ret] = ATTABLE;
    (sh->fst.groupsWaiting) = (sh->fst.groupsWaiting) - 1; //menos um grupo à espera
    sh->fst.st.receptionistStat=ASSIGNTABLE;
```

Figura 35 - Segunda parte do *TODO* da função *receivePayment(int n)*

Por último, faz-se um up do semáforo *waitForTable[grupo_ret]*, ainda dentro da condição anterior, pois o *receptionist* atribui uma mesa. Guarda-se o estado do mesmo.

```
    if (semUp (semgid, sh->waitForTable[grupo_ret]) == -1) {  
        perror ("error on the up operation for semaphore access (WT)");  
        exit (EXIT_FAILURE);  
    }  
  
    saveState(nFic, &sh->fSt);  
}
```

Figura 34 - Terceira parte do *TODO* da função *receivePayment(int n)*

semSharedMemWaiter.c

A última entidade a ser tratada foi o waiter, que interage com o grupo e com o chef.

Função waitForClientOrChef()

Aqui, o *waiter* vai atualizar o seu estado e esperar por um pedido do grupo ou do *chef*. Lê o pedido, e assinala que está disponível para novos pedidos. O estado interno deve ser salvo. O valor de retorno desta função será a variável *req*, que indica o pedido feito pelo grupo ou pelo *chef*.

No primeiro *TODO*, dentro da região crítica, coloca-se o estado do *waiter* a *WAIT_FOR_REQUEST* e guarda-se o mesmo.

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

Figura 35 - Primeiro *TODO* da função *waitForClientOrChef()*

Seguidamente, é feito um *down* no semáforo *waiterRequest*, visto que o *waiter* se encontra disponível para receber pedidos.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 36 - Segundo *TODO* da função *waitForClientOrChef()*

Voltando a entrar na região crítica, vai-se tratar de *reqType* e *reqGroup*.

1. Se o pedido realizado ao *waiter* for um pedido de comida por parte do grupo, o *reqType* do pedido atual será *FOODREQ* e o *reqGroup* será o mesmo de *waiterRequest*.
2. Se o pedido realizado ao *waiter* for um pedido por parte do *chefe*, o *reqtype* do pedido atual será *FOODREADY* e o *reqGroup* será o mesmo de *waiterRequest*.

```
// TODO insert your code here

if (sh->fSt.waiterRequest.reqType == FOODREQ){
    req.reqType = FOODREQ;
    req.reqGroup = sh->fSt.waiterRequest.reqGroup;
}

if (sh->fSt.waiterRequest.reqType == FOODREADY){
    req.reqType = FOODREADY;
    req.reqGroup = sh->fSt.waiterRequest.reqGroup;
}

/*-----END TODO-----*/
```

Figura 37 - Terceiro *TODO* da função *waitForClientOrChef()*

Por último, é feito um *up* no semáforo *waiterRequestPossible*, de forma a indicar que o *waiter* está disponível para receber pedidos.

```
// TODO insert your code here

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

/*-----END TODO-----*/
```

Figura 38 - Quarto e último *TODO* da função *waitForClientOrChef()*

Função *informChef(int n)*

Nesta função, o *waiter* deve atualizar o seu estado e levar o pedido de comida ao *chef*. Posteriormente, deve informar o grupo que o pedido foi recebido. Deve, também, esperar que o *chef* receba o pedido. O estado interno deve ser salvo.

Começa-se, então, por alterar o estado do *waiter* para *INFORM_CHEF*, pois este leva um pedido ao *chef*. Desta forma, atualiza-se a flag *foodOrder* para 1 (há um pedido de comida do *waiter* para o *chef*), e o *foodGroup* para *n* (parâmetro passado na função e que representa o grupo a ser tratado). O estado do *waiter* é, então, salvo.

É criada uma variável inteira, *mesa_grupo*, que guardará a mesa em que o grupo (*n*) se encontra (esta variável é usada mais à frente, mas é criada dentro da região crítica pois apenas aqui se consegue aceder a *sh->fSt.assignedTable[n]*).

Por último, o estado do *waiter* é atualizado para *WAIT_FOR_ORDER*, visto que se encontra à espera de um pedido. Este estado é, seguidamente, guardado.


```
// TODO insert your code here

sh->fSt.st.waiterStat = INFORM_CHEF;
sh->fSt.foodOrder = 1;           // atualiza flag
sh->fSt.foodGroup = n;
saveState (nFic, &sh->fSt);     // guarda estado

int mesa_grupo= sh->fSt.assignedTable[n];
saveState (nFic, &sh->fSt);

sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
saveState (nFic, &sh->fSt);

/*-----END TODO-----*/
```

Figura 39 - Primeiro *TODO* da função *informChef(int n)*

Seguidamente, faz-se *up* no semáforo *requestReceived[mesa_grupo]*, pois o *waiter* recebeu o pedido associado àquela mesa.

Por fim, faz-se um *up* no semáforo *waitOrder*, o *waiter* realiza um pedido ao chef, e um *down* no semáforo *orderReceived*, depois de realizar o pedido, o *waiter* espera que este seja aceite.

```
// TODO insert your code here

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

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

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

/*-----END TODO-----*/
```

Figura 40 - Segundo e último *TODO* da função *informChef(int n)*

Função takeFoodToTable(int n)

Aqui, o *waiter* atualiza o seu estado e leva o pedido de comida à respetiva mesa, dando permissão para a refeição começar. O grupo deverá ser informado que a comida se encontra pronta. O estado interno deve ser salvo.

O estado do *waiter* é, então, atualizado para *TAKE_TO_TABLE*, e salvo de imediato. Seguidamente, é feito um *up* no semáforo *foodArrived[sh->fSt.assignedTable[n]]* sinalizando que o *waiter* entregou a comida na respetiva mesa.

```
// TODO insert your code here

sh->fSt.st.waiterStat = TAKE_TO_TABLE;
saveState (nFic, &sh->fSt);

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

/*-----END TODO-----*/
```

Figura 41 - Primeiro e único *TODO* da função *takeFoodToTable(int n)*

Resultados obtidos

De forma a verificar que a solução implementada é a correta, deve-se ir testando o programa usando a *makefile* fornecida e realizando os testes dentro do diretório */run*. Como os resultados são sempre diferentes, é uma boa prática realizar os testes diversas vezes e confirmar se o *output* é, realmente, o correto. Começou-se, então, por testar apenas com um grupo (o número de grupos pode ser mudado no ficheiro *config.txt* do diretório */run*) e, quando os resultados deste foram confirmados, começou por se analisar a solução para três grupos. Para verificar a existência de *deadlocks*, o programa foi também executado usando o comando *./run*, executando assim mil vezes.

CH	WT	RC	G00	gWT	T00
0	0	0	1	0	.
0	0	0	1	0	.
0	0	0	1	0	.
0	0	0	2	0	.
0	0	1	2	0	.
0	0	1	2	0	0
0	0	0	2	0	0
0	0	0	3	0	0
0	1	0	3	0	0
0	1	0	3	0	0
0	0	0	3	0	0
1	0	0	3	0	0
1	0	0	4	0	0
1	0	0	4	0	0
2	0	0	4	0	0
2	2	0	4	0	0
2	2	0	5	0	0
2	2	0	6	0	0
2	2	2	6	0	0
2	2	2	6	0	.
2	2	2	7	0	.

Figura 42 - *Output* da solução para um grupo

CH	WT	RC	G00	G01	G02	gWT	T00	T01	T02
0	0	0	1	1	1	0	.	.	.
0	0	0	1	1	1	0	.	.	.
0	0	0	1	1	1	0	.	.	.
0	0	0	1	1	2	0	.	.	.
0	0	1	1	1	2	0	.	.	.
0	0	1	1	1	2	0	.	.	0
0	0	1	2	1	2	0	.	.	0
0	0	0	2	1	2	0	.	.	0
0	0	0	2	1	3	0	.	.	0
0	1	0	2	1	3	0	.	.	0
0	1	0	2	1	3	0	.	.	0
0	0	0	2	1	3	0	.	.	0
0	0	1	2	1	3	0	.	.	0
0	0	1	2	1	3	0	1	.	0
0	0	1	2	2	3	0	1	.	0
1	0	1	2	2	3	0	1	.	0
1	0	1	3	2	3	0	1	.	0
1	0	1	3	2	4	0	1	.	0
1	0	0	3	2	4	0	1	.	0
1	0	0	3	2	4	0	1	.	0
1	0	1	3	2	4	0	1	.	0
1	1	1	3	2	4	1	1	.	0
1	1	1	3	2	4	1	1	.	0
1	0	1	3	2	4	1	1	.	0
1	0	0	3	2	4	1	1	.	0
0	0	0	3	2	4	1	1	.	0
0	0	0	4	2	4	1	1	.	0
1	0	0	4	2	4	1	1	.	0
1	0	0	4	2	4	1	1	.	0
1	2	0	4	2	4	1	1	.	0
1	0	0	4	2	4	1	1	.	0
1	0	0	4	2	5	1	1	.	0
0	0	0	4	2	5	1	1	.	0
0	2	0	4	2	5	1	1	.	0
0	0	0	4	2	5	1	1	.	0
0	0	0	5	2	5	1	1	.	0
0	0	0	6	2	5	1	1	.	0
0	0	2	6	2	5	1	1	.	0
0	0	2	6	2	5	1	.	.	0
0	0	1	6	2	5	0	.	1	0
0	0	1	6	3	5	0	.	1	0
0	0	1	7	3	5	0	.	1	0
0	0	0	7	3	5	0	.	1	0
0	1	0	7	3	5	0	.	1	0
0	1	0	7	3	5	0	.	1	0
0	0	0	7	3	5	0	.	1	0
0	0	0	7	4	5	0	.	1	0
1	0	0	7	4	5	0	.	1	0
1	0	0	7	4	5	0	.	1	0
2	0	0	7	4	5	0	.	1	0
2	2	0	7	4	5	0	.	1	0
2	2	0	7	5	5	0	.	1	0
2	2	0	7	6	5	0	.	1	0
2	2	2	7	6	5	0	.	1	0
2	2	2	7	6	5	0	.	.	0
2	2	2	7	7	5	0	.	.	0
2	2	0	7	7	5	0	.	.	0
2	2	0	7	7	6	0	.	.	0
2	2	2	7	7	6	0	.	.	.
2	2	2	7	7	7	0	.	.	.

Figura 43 - *Output* da solução para três grupos

É importante salientar que:

1. A primeira coluna corresponde aos estados do *chef*,
2. A segunda coluna corresponde aos estados do *waiter*,
3. A terceira coluna corresponde aos estados do *receptionist*,
4. As colunas seguintes variam consoante o número de grupos (por exemplo, *G00* corresponde ao grupo 0, *G01* ao grupo 1, etc.);
5. A coluna *gWT* corresponde ao número de grupos que se encontram à espera de mesa, naquele momento;
6. As colunas finais correspondem aos estados das mesas dos respetivos grupos (por exemplo, a coluna *T00* corresponde ao estado da mesa do grupo 0, a coluna *T01* ao estado da mesa do grupo 1, etc.).

Desta forma, os resultados obtidos parecem ser os corretos. Analisando o *output* da solução para um único grupo, verifica-se que:

Inicialmente, aguarda-se que um grupo chegue à receção. Quando chega, altera o seu estado de 1, *GOTOREST*, para 2, *ATRECEPTION*. Já na receção, o grupo dirige-se ao *receptionist* para pedir uma mesa. Aqui, o *receptionist* encontra-se no estado 1, *ASSIGNTABLE*. Vai, então, verificar se existem mesas disponíveis e, em caso positivo, atribui a mesa ao grupo em questão, passando este para o estado 3, *FOOD_REQUEST*. Após lhe ter sido atribuída mesa, o grupo chama o *waiter*.

Ao ser chamado, o *waiter*, que até agora se encontrava no estado 0, *WAIT_FOR_REQUEST*, recebe um pedido, alterando o seu estado para 1, *INFORM_CHEF*, pois irá transmitir ao *chef* o pedido que necessita de ser confeccionado.

Quando o *chef* recebe o pedido, dois estados são alterados - o estado do grupo, que passa para 4, *WAIT_FOR_FOOD* (espera que o seu pedido seja concluído), e o estado do *chef*, que transita de 0, *WAIT_FOR_ORDER*, para 1, *COOK* (cozinha o pedido que recebeu).

Terminada a confecção, o *chef* passa para o estado 2, *REST*, pois, neste caso, só existia um grupo para servir (no entanto, para um número mais elevado de grupos, o *chef* só transita para o estado *REST* depois de tratar dos pedidos de todos - como pode ser verificado na figura 43). Posteriormente, chama o *waiter*, e este entrega a comida ao grupo respetivo - o estado do empregado de mesa passa, então, para 2 - *TAKE_TO_TABLE*.

Agora, o grupo, após receber o seu pedido já confeccionado, transita para o estado 5 - *EAT*. Quando termina a sua refeição, chama o *receptionist*, com o intuito de pagar a conta. Novamente, dois estados são alterados- o grupo passa para o estado 6, *CHECKOUT*, e o *receptionist* passa para 2, *RECVPAY*, visto que foi notificado de um pedido de pagamento.

Finalmente, o *receptionist* dá o pagamento como concluído e, consequentemente, o estado do grupo transita para 7 - *LEAVING*.

Conclusão

Concluído o trabalho, foi verificado que, de facto, as metas propostas no guião foram todas alcançadas. Foi notória a importância que este trabalho teve, para os alunos, na aprendizagem de mecanismos associados à execução e sincronização de processos e *threads*.

É de notar que o código mostrado não foi desenvolvido com o intuito de obter uma otimização do mesmo, mas sim o de ter todas as funcionalidades bem implementadas.