deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Restaurante

Trabalho realizado por:

Diogo Alexandre Rocha Domingues, nº114192

João Pedro Moreira Viegas, nº113144

Sistemas Operativos

Prof. José Nuno Panelas Nunes Lau

Ano Letivo 2023/2024

Índice

Introdução

Na sequência do trabalho proposto, este relatório tem como objetivo a explicação dos raciocínios utilizados para formular o código necessário para a correta sincronização dos processos e *threads* fornecidos no programa inicial. O tema do trabalho consiste na simulação de um restaurante onde coexistem quatro entidades que correspondem a processos independentes:

- ***Group* ->** Os grupos são os clientes que irão ser atendidos pelas outras entidades, isto é, dirigir-se-ão ao *Receptionist* para pedir mesa e para pagarem no fim da refeição e dirigir-se-ão ao *Waiter* para fazerem o pedido da comida.

- ***Recepcionist* ->** O rececionista inicialmente espera que os grupos lhe peçam mesa, de seguida, este verifica se existem mesas disponíveis e, se existirem o grupo pode sentar-se, se não existirem o grupo irá para a sala de espera. Quando o rececionista recebe o pagamento de uma das mesas, este irá verificar se existem grupos à espera e, se houver, ele chama-os e indica em que mesa estes se devem sentar.

- ***Waiter* ->** O *Waiter* inicialmente espera que algum dos grupos lhe faça um pedido e de seguida informa o *Chef* que tem um pedido para fazer, quando o *Chef* sinalizar que o pedido está pronto chamará o *Waiter* para o levar para a mesa, sendo que pode haver vários pedidos ao mesmo tempo dependendo do número de mesas sentadas.

- ***Chef* ->** O *Chef* inicialmente espera que o *Waiter* lhe traga um pedido, sendo que quando houver um, esta entidade é responsável por o confecionar e *o entregar ao* Waiter para este levar o pedido para a mesa, ficando a descansar quando não existem pedidos.

O código apresentado no seguimento deste relatório, serão apenas as funções que tinham código sinalizado para nós completarmos.

Comportamento dos Semáforos

Semáforos	*semDown*		*semUp*	
	Entidade	*Função*	*Entidade*	*Função*
mutex	*Todos*	*-*	*Todos*	*-*
receptionistReq	*Receptionist*	*waitForGroup*	*Group*	*checkInAtReception*
				checkOutAtReception
receptionistRequestPossible	*Group*	*checkInAtReception*	*Receptionist*	*waitForGroup*
		checkOutAtReception		
waiterRequest	*Waiter*	*waitForClientOrChef*	*Group*	*orderFood*
			Chef	*processOrder*
waiterRequestPossible	*Group*	*orderFood*	*Waiter*	*waitForClientOrChef*
	Chef	*processOrder*		
waitOrder	*Chef*	*waitForOrder*	*Waiter*	*informChef*
orderReceived	*Waiter*	*informChef*	*Chef*	*waitForOrder*
waitForTable[MAXGROUPS]	*Group*	*checkInAtReception*	*Receptionist*	*receivePayment*
requestReceived[NUMTABLES]	*Group*	*orderFood*	*Waiter*	*informChef*
foodArrived[NUMTABLES]	*Group*	*waitFood*	*Waiter*	*takeFoodToTable*
tableDone[NUMTABLES]	*Group*	*checkOutAtReception*	*Receptionist*	*receivePayment*

O semáforo *mutex* é usado em todas as entidades para entrar numa zona crítica quando faz *semDown* e sair desta quando faz *semUp*. Este só serve para não haver conflitos ao escrever na memoria partilha, pois os processos só podem entrar um por vez nessa zona então os processos que precisam de escrever na memoria partilhada esperam que quem estiver a escrever pare, para que o próximo o possa escrever.

Chef

O código apresentado de seguida é uma implementação possível para o processo de chefe de cozinha, simulando um restaurante. Este é responsável por preparar os pedidos de comida que são feitos pelo processo *Waiter*.

Função waitForOrder()

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

    // Muda o estado do chef para WAIT_FOR_ORDER
    sh->fSt.st.chefStat = WAIT_FOR_ORDER;
    saveState(nFic, &sh->fSt);

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

    // Espera que o Waiter dê inforções da comida
    if (semDown (semgid, sh->waitOrder) == -1) {
        perror ("error on the down operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

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

    // Indicar que o pedido foi recebido
    sh->fSt.foodOrder = 0;
    // Muda o estado do Chef para COOK
    sh->fSt.st.chefStat = COOK;
    // Guarda o grupo que pediu comida
    lastGroup=sh->fSt.foodGroup ;
    saveState(nFic, &sh->fSt);

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

    // Desbloqueia o Waiter pois já recebeu e guardou a informação do pedido
    if (semUp (semgid, sh->orderReceived) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
}
```

Figura 1 / Função waitForOrder()

Nesta função o *Chef* aguarda que o *Waiter* lhe traga o pedido e depois guarda as informações do pedido.

O *Chef*, dentro da zona critica (*semDown sh->mutex*), passa para o estado *WAIT_FOR_ORDER* e depois de sair (*semUp sh->mutex*) faz *semDown* do semáforo (*sh->waitOrder*) para bloquear o processo até que o *Waiter* de *semUp* quando o pedido estiver pronto.

Depois, quando o *Chef* for desbloqueado, este volta a entrar na zona critica e guarda na variável *lastGroup* o grupo que pediu comida, esse grupo vem do semáforo *sh->fSt.foodGroup*, pois foi guardado pelo *Waiter*.

Após sair da zona critica, faz um *semUp* do semáforo (*sh->orderReceived*) para informar o *Waiter* que o *Chef* já recebeu o pedido.

Função processOrder()

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

    // Espera que o Waiter esteja disponivel para receber um pedido
    if (semDown(semgid, sh->waiterRequestPossible) == -1) {
            perror ("error on the down operation for semaphore access (PT)");
            exit (EXIT_FAILURE);
        }

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

    // Muda o estado do Chef para REST
    sh->fSt.st.chefStat = REST;

    // Guarda no waiterRequest o type FOODREADY para o Waiter saber que tem de receber a comida e o grupo guardado que pediu a comida
    sh->fSt.waiterRequest.reqType = FOODREADY;
    sh->fSt.waiterRequest.reqGroup = lastGroup;
    saveState(nFic, &sh->fSt);

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

    // Liberta o Waiter para processar o pedido
    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
}
```

Figura 2 / Função processOrder()

Nesta função o *Chef* faz a comida durante algum tempo e depois faz um *Request* ao *Waiter* para que este guarde as informações do pedido.

O *Chef* aguarda (*usleep*) por um período de tempo aleatório, simulando o processo de preparação da refeição.

Depois faz *semDown* do semáforo (*sh->waiterRequestPossible*) para espera que o *Waiter* esteja disponível para o *Chef* entregar a comida e informar o grupo que a pediu.

Quando o *Waiter* estiver disponível, o *Chef* entra na zona critica (*semDown mutex*), depois indica no *reqType* que a comida está pronta (*FOODREADY*) e no *reqGroup* indica o grupo que a pediu pois este ficou guardado na variável *lastGroup* e o *Chef* passa para o estado (*REST*).

Ao sair da zona critica, faz um *semUp* do semáforo (*sh-> waiterRequest*) libertando o *Waiter* para que este possa receber a comida.

O *Chef* encerra o seu ciclo de vida quando tiver cozinhado para todos os grupos.

Group

O código apresentado de seguida é uma implementação possível para o processo *Group*, simulando um restaurante. Este fica responsável por se dirigir ao restaurante e requisitar uma mesa ao rececionista. De seguida, efetua o pedido da comida ao *Waiter* e espera que este chegue, quando chegar começa a refeição e quando esta acaba o grupo dirigir-se-á ao rececionista para efetuar o pagamento do jantar.

Função checkInAtReception()

```
static void checkInAtReception(int id)
{
    // Espera que o rececionsita esteja disponível para receber um pedido
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // Muda o estado do grupo Nº(id) para ATRECEPTION
    sh->fSt.st.groupStat[id] = ATRECEPTION;
    // Guarda no receptionistRequest o id do grupo e o type TABLEREQ para pedir uma mesa
    sh->fSt.receptionistRequest.reqGroup = id;
    sh->fSt.receptionistRequest.reqType = TABLEREQ;
    saveState(nFic, &sh->fSt);

    // Liberta o rececionista para processar o request (pedido de mesa)
    if (semUp (semgid, sh->receptionistReq) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // O grupo espera que lhe seja atribuída uma mesa
    if (semDown (semgid, sh->waitForTable[id]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}
```

Figura 3 / Função checkInAtReception()

O *Group(id)* faz *semDown* do semáforo (*sh->receptionistRequestPossible*) para esperar que o rececionista esteja disponível para receber um pedido.

Quando estiver, o *Group(id)* entra na zona crítica e passa o seu estado para *ATRECEPTION* e pede uma mesa ao rececionista (*reqType = TABLEREQ*) e indica no *reqGroup* o seu *id* para identificar que grupo é que pediu a mesa e depois liberta (*semUp*) o semáforo (*sh->receptionistReq*) para que o rececionista receba o pedido de mesa.

Depois de sair da zona critica o *Group(id)* faz *semDown* do (*sh->waitForTable[id]*) para esperar que lhe seja atribuída uma mesa.

Função orderFood()

```c
static void orderFood (int id)
{
    // Espera que o Waiter esteja disponivel para receber um request
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // Muda o estado do grupo Nº(id) para FOOD_REQUEST
    sh->fSt.st.groupStat[id] = FOOD_REQUEST;
    // Guarda no waiterRequest o id do grupo e o type FOODREQ para pedir comida
    sh->fSt.waiterRequest.reqGroup = id;
    sh->fSt.waiterRequest.reqType = FOODREQ;
    saveState(nFic, &sh->fSt);

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

    // Liberta o Waiter para prcessar o request(pedido da comida)
    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

Figura 4 / Função orderFood()

O *Group(id)* faz *semDown* do semáforo (*sh-> waiterRequestPossible*) para esperar que o *Waiter* esteja disponível para receber um pedido.

Depois do *Waiter* estar disponível, o *Group(id)* entra na zona critica e passa o seu estado para *FOOD_REQUEST* e diz que o pedido que vai fazer ao *Waiter* é para pedir comida (*reqType = FOODREQ*) e indica no *reqGroup* o seu *id* para identificar que grupo é que pediu a comida e depois liberta (*semUp*) o semáforo (*sh-> waiterRequest*) para que o *Waiter* receba o pedido de mesa.

Depois de sair da zona critica o *Group(id)* faz *semDown* do (*sh ->requestReceived [sh-> fSt.assignedTable[id]]*)) para esperar que o *Waiter* receba o pedido.

Função waitFood()

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

    // Muda o estado do grupo Nº(id) para WAIT_FOR_FOOD
    sh->fSt.st.groupStat[id] = WAIT_FOR_FOOD;
    saveState(nFic, &sh->fSt);

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

    // Espera que pela comida que será trazida pelo Waiter
    if (semDown (semgid, sh->foodArrived[sh->fSt.assignedTable[id]]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // Muda o estado do grupo(id) para EAT
    sh->fSt.st.groupStat[id] = EAT;
    saveState(nFic, &sh->fSt);

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

Figura 5 / Função waitFood()

O *Group(id)* entra na zona critica (*semDown sh->mutex*) para mudar o seu estado para *WAIT_FOR_FOOD* e depois sai (*semUp sh->mutex*).

Em seguida, faz *semDown* do semáforo (*sh->foodArrived[sh->fSt.assignedTable[id]]*)) para esperar que lhe seja trazida a comida pelo *Waiter* à mesa em que está (*sh->fSt.assignedTable[id]*).

O *Group(id)* volta á zona critica (*semDown sh->mutex*) para mudar o seu estado para *EAT* e o *Group(id)* começa a comer e depois sai da zona critica (*semUp sh->mutex*).

Função checkOutAtReception()

```c
static void checkOutAtReception (int id)
{
    // Espera que o receptionist esteja disponivel para receber um request
   if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // Muda o estado do grupo Nº(id) para CHECKOUT
    sh->fSt.st.groupStat[id] = CHECKOUT;
    //Guarda no receptionistRequest o id do grupo e o type BILLREQ para pedir para pagar
    sh->fSt.receptionistRequest.reqGroup = id;
    sh->fSt.receptionistRequest.reqType = BILLREQ;
    saveState(nFic, &sh->fSt);

    // Liberta o receptionist para ir buscar o pagamento
    if (semUp (semgid, sh->receptionistReq) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // Espera que receptionist libere a mesa em que está.
    if (semDown (semgid, sh->tableDone[sh->fSt.assignedTable[id]]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // Muda o estado do grupo Nº(id) para LEAVING
    sh->fSt.st.groupStat[id] = LEAVING;
    saveState(nFic, &sh->fSt);

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

Figura 6 / Função checkOutAtReception()

O *Group(id)* faz *semDown* do semáforo (*sh->receptionistRequestPossible*) para esperar que o rececionista esteja disponível para receber um pedido para pagar a conta.

Quando estiver disponível, o *Group(id)* entra na zona critica e passa o seu estado para *CHECKOUT* e diz que o pedido que está a fazer ao rececionista é pagar a conta (*reqType = BILLREQ*) e indica no *reqGroup* o seu *id* para identificar quem pediu a conta e depois liberta (*semUp*) o semáforo (*sh->receptionistReq*) para que o rececionista receba o pedido da conta.

Depois de sair da zona critica o *Group(id)* faz *semDown* do (*sh->tableDone[sh->fSt.assignedTable [id]]*)) para esperar que a mesa onde se encontra seja libertada para que outro grupo se possa sentar.

O *Group(id)* volta á zona critica (*semDown sh->mutex*) para mudar o seu estado para *LEAVING* e o *Group(id)* vai se embora e depois sai da zona critica (*semUp sh->mutex*).

Após o fim da função *checkOutAtReception()* o *Group(id)* encerra o seu ciclo de vida.

Waiter

O código apresentado de seguida é uma implementação possível para o processo *Waiter*, simulando um restaurante. Este é responsável por receber o pedido de comida por parte do grupo, de informar o Chef de tal pedido e aquando da confeção deste de o levar para a mesa.

Função waitForClientOrChef()

```c
static request waitForClientOrChef()
{
    request req;

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

    // Atualizar e guardar o estado do Waiter para WAIT_FOR_REQUEST
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
    saveState(nFic, &sh->fSt);

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

    // Bloquear o Waiter até que haja um pedido
    if (semDown(semgid, sh->waiterRequest) == -1){
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

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

    // Atualizar a variável req com o pedido com o pedido do grupo e reiniciar o pedido do Waiter, guardando o estado
    req = sh->fSt.waiterRequest;
    sh->fSt.waiterRequest.reqType = -1;
    sh->fSt.waiterRequest.reqGroup = -1;
    saveState(nFic, &sh->fSt);

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

    // Sinalizar que o Waiter pode receber pedidos
    if (semUp(semgid, sh->waiterRequestPossible) == -1){
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }
    return req;
}
```

Figura 7 / Função waitForClientOrChef()

Nesta função o *Waiter* espera que um grupo ou que o chefe lhe faça um pedido.

Começamos por definir a variável de retorno *request req*, e de seguida entramos na região crítica da função apenas para atualizar e guardar o estado do *Waiter*, saindo logo de seguida desta.

Após sair da região crítica bloqueamos o *Waiter* até que um grupo ou o chefe lhe precise de fazer um pedido.

Depois voltamos a entrar na região crítica da função para atualizar a variável *req* para o pedido efetuado pelo grupo e atribuímos os valores –1 ao semáforo *waiterRequest* para o reiniciar, guardando o seu estado atual e voltamos a sair da região crítica.

Finalmente, sinalizamos que o *Waiter* está disponível para receber mais pedidos e retornamos a variável *req*.

Função informChef()

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

    // Atualizar o estado do Waiter para INFORM_CHEF, indicar que há um pedido de comida e qual o grupo que o pediu e guardar os respetivos estados
    sh->fSt.st.waiterStat = INFORM_CHEF;
    sh->fSt.foodOrder = 1;
    sh->fSt.foodGroup = n;
    saveState(nFic, &sh->fSt);

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

    // Sinalizar que o pedido do grupo foi recebido pelo Waiter
    if (semUp(semgid, sh->requestReceived[sh->fSt.assignedTable[n]]) == -1){
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    // Informar o Chef que há um pedido de comida
    if (semUp(semgid, sh->waitOrder) == -1){
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    // Bloquear o Waiter até que o Chef receba o pedido
    if (semDown(semgid, sh->orderReceived) == -1){
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }
}
```

Figura 8 / Função informChef()

Nesta função o *Waiter* informará o chefe de cozinha que tem um pedido para ele.

Primeiramente, começamos por entrar na região crítica da função para atualizar o estado do *Waiter* para *INFORM_CHEF*, para atualizar o semáforo *foodOrder* que indica que foi feito um pedido de comida ao chefe e atualizamos o semáforo *foodGroup* para saber qual dos grupos pediu a comida. Após guardar o estado dos semáforos, saímos da região crítica da função.

Já fora da região crítica da função iremos informar o grupo que o *Waiter* recebeu o pedido que eles fizeram através do *semUp* do semáforo *requesteReceived*. Iremos, também, informar o chefe que há um pedido de comida pendente. Finalmente, bloqueamos o *Waiter* através do *semDown* do semáforo *orderReceived* até que o chefe receba o pedido e o desbloqueie através do *semUp* do mesmo semáforo.

Função takeFoodToTable()

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

    // Atualizar o estado do Waiter para TAKE_TO_TABLE e guarda-o
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &sh->fSt);

    // Informar o grupo que a comida chegou
    if (semUp(semgid, sh->foodArrived[sh->fSt.assignedTable[n]]) == -1){
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

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

Figura 9 / Função takeFoodToTable()

Esta função leva a comida à mesa que a pediu assim que esta estiver feita.

Começamos por entrar na região crítica da função para atualizar e guardar o estado do *Waiter* para *TAKE_TO_TABLE*. De seguida, informamos o grupo que a comida chegou através do *semUp* do semáforo *foodArrived* e saímos da região crítica.

O ciclo de vida do *Waiter* terminará quando o número de *Requests* for igual ao dobro do número de grupos.

Receptionist

O código apresentado de seguida é uma implementação possível para o processo de rececionista, simulando um restaurante. Este fica responsável por indicar aos grupos qual mesa usar, assim como receber o pagamento no fim do jantar.

Função decideTableOrWait()

```c
int decideTableOrWait(int n)
{
    // TODO insert your code here

    // ID da mesa em que o grupo se irá sentar
    int tableID = -1;
    // Numero de mesa a atribuir (0 ou 1) e váriavel para saber se a mesa esta em uso
    int numTable, notInUse;

    // Ciclos para verificar se existem mesas disponiveis ou nao
    for (numTable = 0; numTable < NUMTABLES; numTable++){
        notInUse = 1;
        for (int i = 0; i < sh->fSt.nGroups; i++){
            if (sh->fSt.assignedTable[i] == numTable){
                notInUse = 0;
                break;
            }
        }
        if (notInUse){
            // Se existirem mesas disponiveis, atribuir ao ID o número disponivel (0 ou 1)
            tableID = numTable;
            break;
        }
    }

    return tableID;

    /* fim */
}
```

Figura 10 / Função decideTableOrWait()

Esta função é usada para verificar se existem mesas disponíveis para sentar um grupo ou se estão todas ocupadas.

Começamos por definir a variável de retorno que é a *tabelID* e depois definir as variáveis auxiliares numTable e notInUse. A variável de retorno variará entre –1 (se não houver mesas disponíveis) e *NUMTABLES – 1*, *NUMTABLES* está definido em *"probConst.h" e representa o número máximo de mesas que se pode atender ao mesmo tempo. A segunda varia entre 0 e NUMTABLES – 1*. Já a terceira variável está a funcionar como um *boolean* por isso toma os valores 0 ou 1.

De seguida, iniciamos um ciclo *for* para iterar sobre o número máximo de mesas que podem ser atendidas ao mesmo tempo para verificar o estado das mesas, assumimos aqui que uma mesa não está ocupada. No segundo ciclo *for* iteramos sobre o número de grupos e verificamos com um *if statement* se a mesa que estamos a verificar está a ser usada e se estiver mudamos a variável notInUse para zero e a variável *tableID* continua com o valor –1. No entanto, se a mesa verificada não estiver a ser usada atribuiremos o número desta ao tableID e retornamos o valor obtido. Este valor será usado para determinar se um grupo se pode sentar ou se deve esperar na função *"provideTableOrWaitingRoom"*.

Função decideNextGroup()

```c
static int decideNextGroup()
{
    // TODO insert your code here

    // Próximo grupo a ser atendido
    int nextGroup = -1;

    // Ciclo para verificar se existem grupos à espera
    for (int i = 0; i < sh->fSt.nGroups; i++){
        if (groupRecord[i] == WAIT){
            // Se existirem grupos a espera, atribuir ao nextGroup o número do grupo
            nextGroup = i;
            break;
        }
    }

    return nextGroup;

    /* fim */
}
```

Figura 11 / Função decideNextGroup()

Nesta função verificamos se todos os grupos já foram atendidos ou se ainda existem grupos à espera de uma mesa.

Começamos por definir a variável de retorno *nextGroup* e defini-la com o valor –1 que é o valor que irá ser retornado se não se encontrar nenhum grupo à espera.

De seguida, iniciamos um *for loop* que iterará sobre o número de grupos e irá verificar através de um *if statement* se existem grupos à espera, isto é possível pois temos o *array groupRecord* que nos permite guardar o estado em que um determinado grupo se encontra que podem ser *TOARRIVE, WAIT, ATTABLE, DONE*, cada um destes estados tem um número que o representa de 0 a 3, respetivamente. Se se verificar que existem grupos à espera a variável *nextGroup* irá ficar com o número do grupo que se encontra à espera e retornará esse valor que irá ser usado na função *“receivePayment”*, para sentar um grupo que esteja à espera quando uma mesa ficar livre.

Função waitForGroup()

```c
static request waitForGroup()
{
    request ret;

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

    // TODO insert your code here

    // Aualizar e guardar estado do rececionista para esperar por um pedido
    sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;
    saveState(nFic, &sh->fSt);

    /* fim */

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

    // TODO insert your code here

    // Bloquear o rececionista até que um grupo faça um pedido
    if (semDown(semgid, sh->receptionistReq) == -1){
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    /* fim */

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

    // TODO insert your code here

    // Atualizar a variavel ret com o pedido do grupo e reiniciar o pedido do rececionista, guardando-o
    ret = sh->fSt.receptionistRequest;
    sh->fSt.receptionistRequest.reqType = -1;
    sh->fSt.receptionistRequest.reqGroup = -1;
    saveState(nFic, &sh->fSt);

    /* fim */

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

    // TODO insert your code here

    // Aualizar o semáforo para sinalizar que o rececionista pode receber um pedido
    if (semUp(semgid, sh->receptionistRequestPossible) == -1){
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    /* fim */

    return ret;
}
```

Figura 12 / Função waitForGroup()

Nesta função o rececionista espera que um grupo lhe faça o pedido de uma mesa.

Para isto, começamos por definir a variável *request ret que é onde será guardado o pedido da mesa efetuado pelo grupo e de seguida* devemos entrar na região crítica da função para alterar o estado do rececionista para *WAIT_FOR_REQUEST*, guardando-o e saindo da região crítica da função.

Após sairmos da região crítica faremos s*emDown* do semáforo *receptionistReq* para bloquear o rececionista até que um grupo lhe faça um pedido.

Depois do semáforo mencionado no paragrafo anterior estar bloqueado, voltamos a entrar na região crítica da função para atualizar a variável *ret* para o pedido efetuado pelo grupo e atribuímos os valores –1 ao semáforo *receptionistRequest* para o reiniciar, guardando o seu estado atual.

Finalmente, saímos da região crítica e fazemos s*emUp* do semáforo *receptionistRequestPossible* para sinalizarmos aos grupos que o rececionista pode receber pedidos.

Função provideTableOrWaitingRoom()

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

    // TODO insert your code here

    // Atualizar e guardar o estado do rececionista para atribuição de mesas
    sh->fSt.st.receptionistStat = ASSIGNTABLE;
    saveState(nFic, &sh->fSt);

    // Chamar a função decideTableOrWait para saber se existem mesas disponiveis
    int table = decideTableOrWait(n);

    // Verificar se existem mesas disponiveis
    if (table != -1){
        // Se existirem mesas disponiveis, atribuir ao grupo a mesa disponivel e atualizar o estado deste
        sh->fSt.assignedTable[n] = table;
        groupRecord[n] = ATTABLE;
        // Sinalizar que o grupo pode prosseguir
        if (semUp(semgid, sh->waitForTable[n]) == -1){
            perror("error on the up operation for semaphore access (WT)");
            exit(EXIT_FAILURE);
        }
    }
    else{
        // Se não existirem mesas disponiveis, atualizar o estado do grupo para WAIT e incrementar o numero de grupos à espera
        sh->fSt.groupsWaiting++;
        groupRecord[n] = WAIT;
    }

    /* fim */

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

Figura 13 / Função provideTableOrWaitingRoom

Como referido anteriormente, esta função usará os valores de retorno de *"decideTableOrWait"* e atualizará os estados dos grupos em concordância com esses.

Primeiramente, entramos na região crítica da função e atualizamos o estado do rececionista para *ASSIGNTABLE*, guardando o estado do semáforo. Chamamos a função *"decideTableOrWait"* e guardamos o seu valor na variável *table*. De seguida, verificamos se existem mesas disponíveis (isto acontece se o valor de retorno for diferente de -1) e se existirem atribuímos ao grupo a mesa disponível através do semáforo *assignedTable*, atualizamos o estado desse grupo para *ATTABLE* e sinalizamos ao grupo que este pode prosseguir através do semáforo *waitForTable*. No entanto, se não existirem mesas disponíveis aumentamos a variável que guarda o número de grupos à espera e guardamos o estado do grupo que será *WAIT* no *array groupRecord* referido anteriormente.

Sendo assim, para terminar a função basta apenas sair da região crítica da função e o ciclo de vida do *Receptionist* terminará quando o número de *Requests* for igual ao dobro do número de grupos.

Função receivePayment()

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

    // TODO insert your code here

    // Atualizar e guardar o estado do rececionista para receber pagamento
    sh->fSt.st.receptionistStat = RECVPAY;
    saveState(nFic, &sh->fSt);

    // Libertar a mesa que o grupo estava a ocupar
    if (semUp(semgid, sh->tableDone[sh->fSt.assignedTable[n]]) == -1){
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    // Sinalizar que a mesa está disponivel e atualizar o estado do grupo
    sh->fSt.assignedTable[n] = -1;
    groupRecord[n] = DONE;
    saveState(nFic, &sh->fSt);

    /* fim */

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

    // TODO insert your code here

    // Chamar a função decideNextGroup para saber se existem grupos à espera
    int group = decideNextGroup();

    // Verificar se existem grupos à espera
    if (group != -1){
        // Se existirem grupos à espera, atribuir uma mesa ao grupo e atualizar o estado deste
        provideTableOrWaitingRoom(group);
        sh->fSt.groupsWaiting--;
    }

    // Se não existirem grupos à espera o programa termina

    /* fim */
}
```

Figura 14 / Função receivePayment()

Com esta função, o rececionista fica responsável de recolher o pagamento de todos os grupos.

Primeiramente, entramos na região crítica da função para alterar o estado do rececionista para *RECVPAY*, guardando-o. De seguida, faremos *semUp* ao semáforo tableDone para libertar a mesa que o grupo estava a ocupar, para outro grupo se poder sentar nela, depois de libertar a mesa, sinalizamos que a mesa está disponível atualizando o valor do semáforo *assignedTable* para –1 e atualizamos o estado do grupo no *array groupRecord* para *DONE*. Basta apenas guardar o estado dos semáforos e sair da região crítica da função.

Finalmente, chamamos a função "*decideNextGroup"* para verificar se existem grupos à espera, e se existirem, chamamos a função *"provideTableOrWaitingRoom"* para sentar o grupo que estava à espera e decrescemos a variável que guarda o número de grupos à espera e se não existirem grupos à espera programa termina.

Validação de Resultados

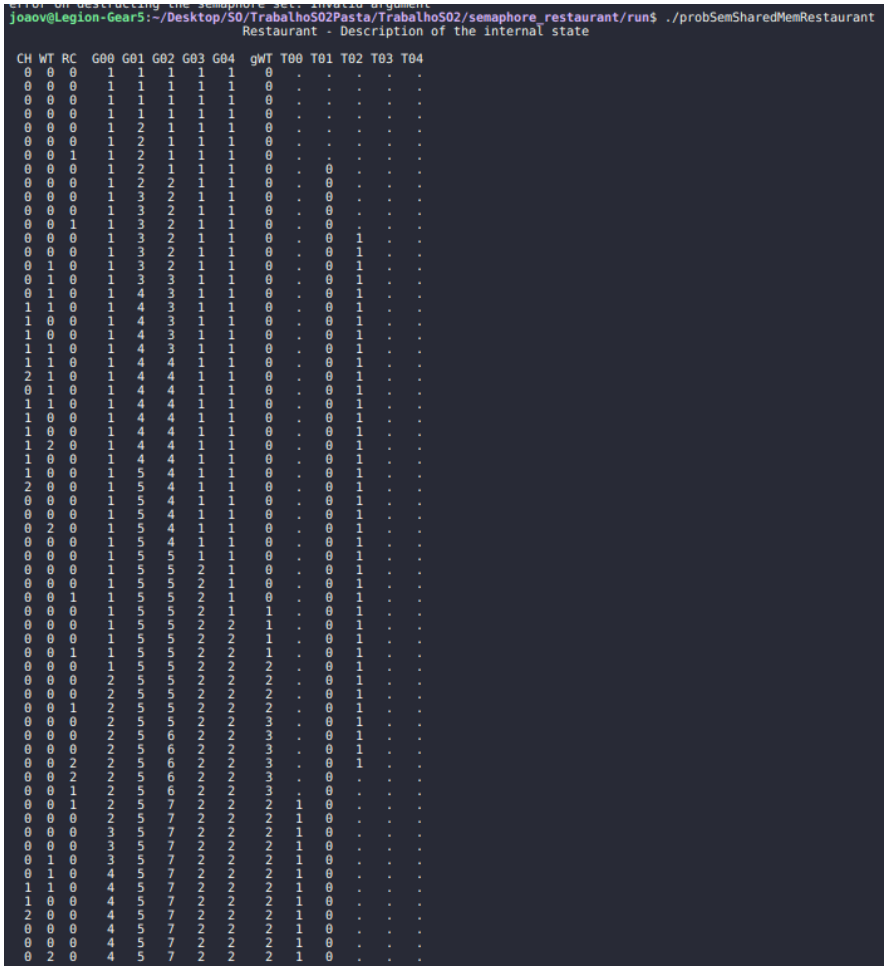

Figura 15 / Exemplo de execução do programa – Parte I

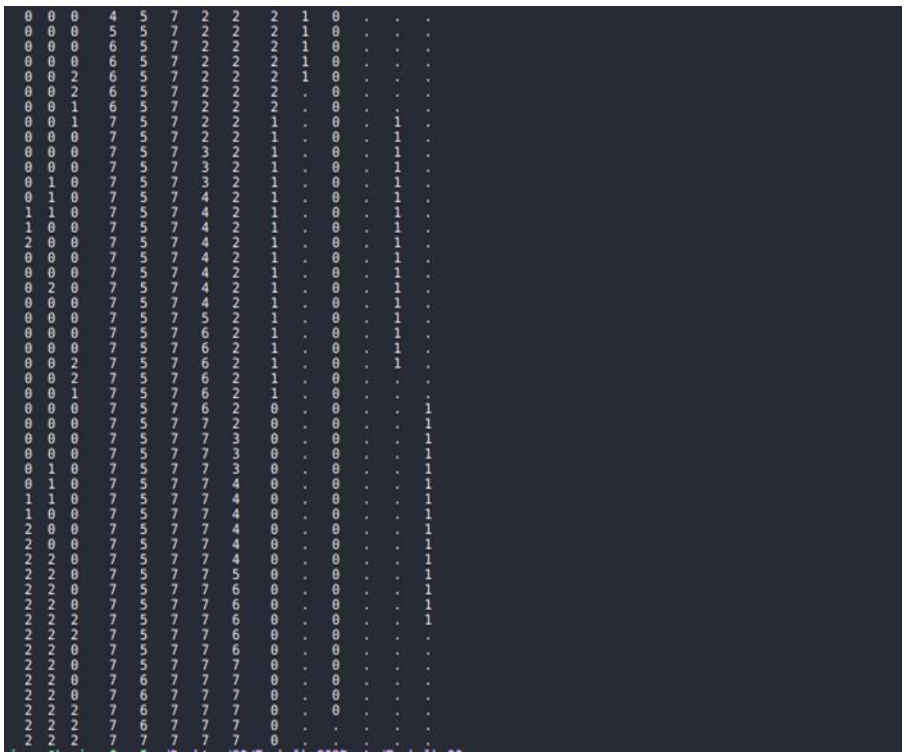

Figura 16 / Exemplo de execução do programa – Parte II

PS: São 2 figuras pois não conseguimos tirar foto aos resultados numa única imagem.

A figura 15 e a figura 16 representam o teste efetuado ao "*probSemSharedMemRestaurant*" para 5 grupos ao fazer-se o "*make all*" para testar os 4 códigos *C* que modificamos neste trabalho.

CH, *WT* e *RT* representam respetivamente os estados do *Chef*, *Waiter* e *Receptionist*.

Os *G(id)* representam o estado do *Group(id)*.

O *gWT* representa o número de grupos á espera de uma mesa (*fSt->groupsWaiting*).

E os *T(id)* representam se o *Group(id)* tem uma mesa. Se for um ponto não têm mesa atribuída e se for um número indica o *id* da mesa.

Fizemos uma validação para 11 grupos e os códigos funcionaram corretamente, contudo daria várias figuras visto que não conseguiríamos tirar tudo numa única imagem então decidimos não a colocar no relatório.

Nós testamos os códigos com apenas duas mesas pois é o que pedia no enunciado. Caso tentássemos usar mais mesas daria erro pois faltaria um semáforo para impedir que um terceiro grupo pedisse comida ao *Waiter* ao mesmo tempo que o *Waiter* está á tentar informar o *Chef* do pedido de um segundo grupo e o *Chef* está a processar/fazer a comida do primeiro grupo e assim o Chef ficaria bloqueado no *waiterRequestPossible*, o *Waiter* ficaria bloqueado no *orderReceived* e o terceiro grupo ficaria bloqueado no *requestReceived[sh->fSt.assignedTable[id]]* e assim dependeriam de processos bloqueados para serem desbloqueados.

Isto não acontece com duas mesas pois com duas mesas só 2 grupos poderiam pedir comida a mesmo tempo e o nosso código está preparado para isso pois enquanto o *Chef* cozinha, o *Waiter* processa o 2º pedido de comida e depois o *Chef* pode fazer um pedido para entregar a comida ao *Waiter* sem o bloquear permanentemente.

Conclusão

Concluindo, com este trabalho podemos desenvolver os nossos conhecimentos sobre os mecanismos associados à execução e sincronização de processos e *threads*, tais como a utilização de semáforos e de memória partilhada. Dito isto, conseguimos fazer um código funcional e de acordo com os requisitos propostos, apesar de a execução do código com mais de duas mesas falhar.

De um modo geral, a maior dificuldade foi em entender como as diferentes entidades se comunicavam a partir dos semáforos e da memoria partilhada.