universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

Trabalho 2 - Restaurante

Sistemas Operativos

Ano Letivo: 2023/2024
Prof. José Nuno Panelas Nunes Lau
Prof. António Guilherme Rocha Campos

Trabalho realizado por:
Rodrigo Abreu, Nº113626
João Neto, Nº113482

Índice

Índice **1**
Introdução **2**
Semáforos **3**
Groups **4**
checkInAtReception(int id) 4
orderFood(int id) 5
waitFood(int id) 6
checkOutAtReception(int id) 7
Receptionist **8**
decideTableOrWait(int n) 8
decideNextGroup() 9
waitForGroup() 9
provideTableOrWaitingRoom(int n) 10
receivePayment(int n) 11
Waiter **12**
waitForClientOrChef() 12
informChef(int n) 13
takeFoodToTable(int n) 14
Chef **15**
waitForOrder() 15
processOrder() 16
Testes **17**
Conclusão **18**

Introdução

Este projeto consiste na compreensão dos mecanismos associados à execução e sincronização de processos e threads, simulando o funcionamento de um restaurante com diversas entidades, em que os grupos (vários) vão a um restaurante e durante a sua estadia passam por diversos estados e comunicam com diversas entidades (um waiter, um recepcionista e um chef), estas que também passam por diferentes estados.

- **Chef**
  - Recebe os pedidos através do *waiter*;
  - Chama o *waiter* quando o pedido estiver pronto.
- **Groups**
  - Chama o *recepcionista* quando chega ao restaurante (para requisitar mesa);
  - Chama o *waiter* quando chega à mesa (para requisitar comida);
  - Volta a chamar o *rececionista* para efetuar o pagamento.
- **Waiter**
  - Recebe os pedidos dos *groups*;
  - Entrega os pedidos ao *chef*;
  - Entrega a comida aos *groups*;
- **Receptionist**
  - Encaminha os *groups* para as mesas (ou caso estejam todas ocupadas, manda-os esperar);
  - Recebe o pagamento dos *groups*.

Ao longo deste relatório, vão ser apresentadas as funções que necessitavam de ser completadas para o correto funcionamento do programa, dos ficheiros **semSharedMemGroup.c**, **semSharedMemChef.c**, **semSharedMemWaiter.c** e **semSharedMemReceptionist.c**, em conjunto com os testes realizados para a comprovação do funcionamento.

Semáforos

Excluindo o *mutex,* este é o esquema dos semáforos utilizados:

Semáforos	**UP**		**DOWN**	
	Entidade	**Função**	**Entidade**	**Função**
receptionistRequestPossible	Receptionist	waitForGroup	Groups	checkInAtReception
				checkOutAtReception
receptionistReq	Groups	checkInAtReception	Receptionist	waitForGroup
		checkOutAtReception		
waitForTable	Receptionist	provideTableOrWaitingRoom	Groups	checkInAtReception
waiterRequestPossible	Waiter	waitForCLientOrChef	Groups	orderFood
			Chef	processOrder
waiterRequest	Groups	orderFood	Waiter	waitForCLientOrChef
	Chef	processOrder		
requestReceived	Waiter	informChef	Groups	orderFood
foodArrived	Waiter	takeFoodToTable	Groups	waitFood
tableDone	Receptionist	receivePayment	Groups	checkOutAtReception
waitOrder	Waiter	informChef	Chef	waitForOrder
orderReceived	Chef	waitForOrder	Waiter	informChef

Groups

Esta entidade é definida no ficheiro semSharedMemGroup.c, e aqui vão ser feitas as atualizações dos estados dos *groups*, consideramos este ficheiro a base de todos os outros. Pois são os *groups* que exigem a comunicação entre as diferentes entidades, comunicando diretamente com o *waiter* e o *receptionist* e indiretamente com o *chef* (pois o *waiter* tem de levar ao *chef* o pedido dos *groups*).

checkInAtReception(int id)

```
static void checkInAtReception(int id)
{
    //=============================================================================================
    // TODO insert your code here
    request info;
    info.reqType = TABLEREQ;
    info.reqGroup = id;
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================================

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

    //=============================================================================================
    // TODO insert your code here
    sh->fSt.st.groupStat[id] = ATRECEPTION;
    saveState(nFic, &sh->fSt);
    sh->fSt.receptionistRequest = info;
    if (semUp (semgid, sh->receptionistReq) == -1) {
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================================

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

    //=============================================================================================
    // TODO insert your code here
    if (semDown (semgid, sh->waitForTable[id]) == -1) {
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================================

}
```

Esta função é usada pelos grupos para darem check in na receção, isto é, serem atendidos pelo *receptionist* (comunicarem com ele).

Inicialmente começamos por editar o tipo de pedido feito ao *receptionist*, que neste caso é do tipo TABLEREQ (pedido de mesa), e quem foi o grupo (id) que fez o pedido. Após isso esperamos que o *receptionist* esteja disponível.

Após o *receptionist* mostrar a sua disponibilidade, procedemos à atualização do estado do grupo para ATRECEPTION e informamos o *receptionist* do tipo de pedido e aguardamos que o mesmo guarde essa informação. Esta parte foi efetuada na região crítica do programa e as mudanças de estado do grupo e das variáveis usadas foram salvas simultaneamente.

Finalmente o *group* aguarda que o *receptionist* lhe atribua uma mesa.

orderFood(int id)

```
static void orderFood (int id)
{
    //=============================================================================================
    // TODO insert your code here
    request info;
    info.reqType = FOODREQ;
    info.reqGroup = id;

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


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

    //=============================================================================================
    // TODO insert your code here
    sh->fSt.st.groupStat[id]= FOOD_REQUEST;
    saveState(nFic, &sh->fSt);
    sh->fSt.waiterRequest = info;
    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================================

    if (semUp (semgid, sh->mutex) == -1) {                                                  /* exi
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    //=============================================================================================
    // TODO insert your code here
    if (semDown(semgid, sh->requestReceived[sh->fSt.assignedTable[id]]) == -1) {
        perror("error on the down operation for tableDone");
        exit(EXIT_FAILURE);
    }
    //completed
    //=============================================================================================
}
```

Esta função é usada pelos Grupos, já nas mesas, para efetuarem o pedido, de comida, ao *waiter* (comunicando com ele).

Inicialmente começamos por editar o tipo de pedido feito ao *waiter*, que neste caso é do tipo FOODREQ (pedido de comida), e quem foi o grupo (id) que fez o pedido. Após isso esperamos que o *waiter* esteja disponível.

Após o *waiter* mostrar a sua disponibilidade, procedemos à atualização do estado do grupo para FOOD_REQUEST e informamos o *waiter* do tipo de pedido e aguardamos que o mesmo guarde essa informação. Esta parte foi efetuada na região crítica do programa e as mudanças de estado do grupo e das variáveis usadas foram guardadas simultaneamente.

Finalmente, o *group* aguarda que o *waiter* confirme que recebeu o pedido da mesa onde o grupo está situado.

waitFood(int id)

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

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

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

    //=============================================================================
    // TODO insert your code here
    if (semDown(semgid, sh->foodArrived[sh->fSt.assignedTable[id]]) == -1) {
        perror("error on the down operation for tableDone");
        exit(EXIT_FAILURE);
    }
    //completed
    //=============================================================================

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

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

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

Esta função é usada pelos grupos, já nas mesas, para atualizarem os seus estados enquanto esperam pela comida e quando começam a comer.

Inicialmente procedemos à atualização do estado do grupo para WAIT_FOR_FOOD e aguardamos até a comida chegar à mesa onde está o grupo correspondente. Quando a comida chegar, procederemos à atualização do estado do grupo para EAT.

As mudanças de estado do grupo foram feitas dentro das regiões críticas do programa.

checkOutAtReception(int id)

```
 */
static void checkOutAtReception (int id)
{
    //=============================================================================
    // TODO insert your code here
    request info;
    info.reqType = BILLREQ;
    info.reqGroup = id;
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================
    if (semDown (semgid, sh->mutex) == -1) {                                        /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    //=============================================================================
    // TODO insert your code here
    sh->fSt.st.groupStat[id] = CHECKOUT;
    saveState(nFic, &sh->fSt);
    //completed
    //=============================================================================
    if (semUp (semgid, sh->mutex) == -1) {                                        /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    //=============================================================================
    // TODO insert your code here
    sh->fSt.receptionistRequest = info;
    if (semUp (semgid, sh->receptionistReq) == -1) {
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }
    if (semDown(semgid, sh->tableDone[sh->fSt.assignedTable[id]]) == -1) {
        perror("error on the down operation for tableDone");
        exit(EXIT_FAILURE);
    }
    //completed
    //=============================================================================
    if (semDown (semgid, sh->mutex) == -1) {                                        /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    //=============================================================================
    // TODO insert your code here
    sh->fSt.st.groupStat[id] = LEAVING;
    saveState(nFic, &sh->fSt);
    //completed
    //=============================================================================
    if (semUp (semgid, sh->mutex) == -1) {                                        /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

}
```

Esta função é usada pelos *groups* para darem Check Out na receção, isto é, pagarem ao *receptionist* (comunicarem com ele).

Inicialmente começamos por editar o tipo de pedido feito ao *receptionist*, que neste caso é do tipo BILLREQ (pedido da conta), e quem foi o grupo (id) que fez o pedido. Após isso esperamos que o *receptionist* esteja disponível.

Após o *receptionist* mostrar a sua disponibilidade, procedemos à atualização do estado do grupo para CHECKOUT e informamos o *receptionist* do tipo de pedido e aguardamos que o mesmo guarde essa informação.

Finalmente o *group* aguarda que o *receptionist* lhes confirme a receção do pagamento, para (mais uma vez na região crítica do programa) o grupo atualizar o seu estado para LEAVING.

Receptionist

Esta entidade é desenvolvida no ficheiro semSharedMemReceptionist.c, e assegura que cada grupo tenha a sua mesa e que, caso não haja mesas disponíveis, os grupos seguintes têm de esperar que as mesas fiquem livres.

decideTableOrWait(int n)

```
static int decideTableOrWait(int n)
{
    //==================================================================
    //TODO insert your code here
    int mesa = -1;

    for (int table = 0; table < NUMTABLES; table++) {
        int isTableAssigned = 0;
        for (int id = 0; id < MAXGROUPS; id++) {
            if (sh->fSt.assignedTable[id] == table) {
                isTableAssigned = 1;
                break;
            }
        }
        if (!isTableAssigned) {
            mesa = table;
            sh->fSt.assignedTable[n] = table;
            break;
        }
    }
    return mesa;
    //completed
    //==================================================================
}
```

Esta função serve para decidir se um grupo vai para uma mesa ou tem de esperar, retorna -1 se tiver de esperar, e retorna o id de uma mesa caso esta esteja livre, e define essa mesa como a mesa correspondente ao grupo n.

decideNextGroup()

```
*/
static int decideNextGroup()
{
    //TODO insert your code here
    for (int id = 0; id < MAXGROUPS; id++) {
       if (groupRecord[id] == WAIT) {
           return id;
       }
    }
    return -1;
}
```

Esta função é para decidir qual o grupo (na fila de espera) que vai ocupar a próxima mesa livre. Caso não haja grupos à espera retorna -1.

waitForGroup()

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

    if (semDown (semgid, sh->mutex) == -1)  {                                        /* enter
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    //====================================================================================
    // TODO insert your code here:
    sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;
    saveState(nFic,&sh->fSt);
    // Code completed
    //====================================================================================

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

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

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

    //====================================================================================
    // TODO insert your code here
    ret = sh->fSt.receptionistRequest;
    //completed
    //====================================================================================

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

    //====================================================================================
    // TODO insert your code here
    if (semUp (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access");
        exit (EXIT_FAILURE);
    }
    //completed
    //====================================================================
    return ret;

}
```

Esta função é usada pelo *receptionist* para esperar pelos grupos.

Inicialmente procedemos à atualização do estado do *receptionist* para WAIT_FOR_REQUEST.

Depois aguardamos pelo pedido do grupo, e quando esse pedido é feito guardamos o tipo de pedido numa variável **ret** que pode ser (BILLREQ ou TABLEREQ).

Finalmente o *receptionist*, torna-se disponível para receber novos pedidos.

provideTableOrWaitingRoom(int n)

```
static void provideTableOrWaitingRoom (int n)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    //====================================================================
    // TODO insert your code here
    sh->fSt.st.receptionistStat = ASSIGNTABLE;
    int table = decideTableOrWait(n);
    if(table != -1){
        groupRecord[n] = ATTABLE;
        if (semUp (semgid, sh->waitForTable[n]) == -1) {
            perror ("error on the down operation for semaphore access");
            exit (EXIT_FAILURE);
        }
    }
    else{
        groupRecord[n] = WAIT;
        sh->fSt.groupsWaiting ++;
    }
    saveState(nFic,&sh->fSt);
    //completed
    //====================================================================

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

}
```

Esta função é usada pelo *receptionist* para informar aos grupos se devem esperar, caso contrário, atribuem-lhes uma mesa.

Inicialmente procedemos à atualização do estado do *receptionist* para ASSIGNTABLE.

Depois procura uma mesa disponível para o grupo n, se não houver, o grupo é adicionado à fila de espera, e o *receptionist* guarda a sua evolução como WAIT, caso contrário, guarda a sua evolução como ATTABLE e avisa o grupo correspondente que podem ir para a mesa.

receivePayment(int n)

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

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

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

    // TODO insert your code here
    if (semUp(semgid, sh->tableDone[sh->fSt.assignedTable[n]]) == -1) {
        perror("error on the down operation for tableDone");
        exit(EXIT_FAILURE);
    }
    if(sh->fSt.groupsWaiting > 0){
        int grupo = decideNextGroup();
        groupRecord[grupo] = ATTABLE;
        sh->fSt.assignedTable[grupo] = sh->fSt.assignedTable[n];
        sh->fSt.groupsWaiting --;
        if (semUp (semgid, sh->waitForTable[grupo]) == -1)  {
            perror ("error on the up operation for semaphore access");
            exit (EXIT_FAILURE);
        }
    }
    sh->fSt.assignedTable[n] = -1;
}
```

Esta função é usada pelo *receptionist* para receber os pagamentos de cada grupo.

Inicialmente procedemos à atualização do estado do *receptionist* para RECVPAY e guardamos a evolução do grupo para DONE.

Posteriormente avisamos o grupo que o pagamento foi concluído e se houverem grupos à espera, com ajuda da função decideNextGroup(), decide qual o próximo grupo a ir para uma mesa e guarda a sua evolução para ATTABLE.

Para evitar conflitos, atribuí ao novo grupo a mesa do grupo que saiu e avisa-os que podem ir para a mesa, e diminui o número de grupos à espera.

Finalmente retiramos a mesa ao grupo que saiu.

Waiter

Este código desenvolvido no ficheiro semSharedMemWaiter.c, tem como objetivo simular a atividade de um empregado de mesa (waiter) num restaurante. Este é responsável por registar os pedidos dos grupos, entregar os pedidos ao cozinheiro e levar os pedidos preparados pelo cozinheiro às respetivas mesas.

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);
    }
    //=============================================================================
    // TODO insert your code here
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
    saveState(nFic, &sh->fSt);
    //completed
    //=============================================================================
    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
    if (semDown (semgid, sh->waiterRequest) == -1)  {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================
    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
    req = sh->fSt.waiterRequest;
    //completed
    //=============================================================================
    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
    if (semUp (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    //completed
    //=============================================================================
    return req;
}
```

Nesta função, começamos por decrementar o mutex para entrar na região crítica e definir o estado do waiter como *WAIT_FOR_REQUEST*, guarda-se o estado e incrementa-se o mutex para sair da região crítica.

De seguida, decrementa-se o *waiterRequest* para o waiter poder fornecer os seus serviços ao cozinheiro ou aos grupos, dependendo de quem incrementou o semáforo antes. Assim, entramos outra vez na região crítica e registamos o *waiterRequest* na variável req, aqui estará a informação do tipo de serviço que o empregado vai realizar, e sai-se da região crítica.

Por último, incrementa-se o semáforo *waiterRequestPossible* para obter os serviços do empregado de mesa e retorna req. Dependendo do valor de req será invocada a função *informChef* se for "*FOODREQ*" ou a função *takeFoodToTable* se for "*FOODREADY*".

informChef(int n)

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

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

    //=============================================================================
    // TODO insert your code here
    if (semUp (semgid, sh->requestReceived[sh->fSt.assignedTable[n]]) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    sh->fSt.foodGroup = n;
    if (semUp (semgid, sh->waitOrder) == -1)  {
        perror ("error on the up 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);
    }
    //completed
    //=============================================================================


}
```

Esta função recebe como argumento "*n*" que corresponde ao grupo que efetua o pedido, e consiste em informar o cozinheiro sobre o pedido a preparar e o grupo correspondente a esse pedido.

Para começar, entra-se na região crítica e atualiza-se o estado do empregado para *INFORM_CHEF*, atribui-se o valor 1 à variável *foodOrder*, para indicar que existe um pedido, guarda-se o estado e sai-se da região crítica.

De seguida, incrementa-se o semáforo *requestReceived[table]*, que indica que o empregado recebeu o pedido, sendo *table* o número da mesa onde o grupo que efetuou o pedido está sentado. Depois atribui-se o valor da variável n à variável *foodgroup,* para registar o grupo correspondente ao pedido, e incrementa-se o semáforo *waitOrder,* para que o cozinheiro aguarde o pedido do empregado. Por fim, decrementa-se o *orderReceived*, para que o empregado espere que o cozinheiro prepare o pedido.

takeFoodToTable(int n)

```
static void takeFoodToTable (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
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &sh->fSt);
    //completed
    //=================================================================

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

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

Esta função recebe como argumento "*n*" que corresponde ao grupo ao qual terá de ser entregue o pedido, e consiste em o empregado trazer o pedido preparado pelo cozinheiro à mesa correspondente ao grupo passado como argumento.

No início, acessa-se a região crítica e atualiza-se o estado do empregado para *TAKE_TO_TABLE,* guarda-se o estado e sai-se da região crítica, de se seguida incrementa-se o semáforo *foodArrived[table]*, para que o empregado traga o pedido à mesa, sendo table a mesa correspondente ao grupo do respetivo pedido.

Chef

Este código desenvolvido no ficheiro semSharedMemChef.c, tem como objetivo simular a atividade de um cozinheiro (Chef) num restaurante. Este é responsável por preparar os pedidos indicados pelo empregado de mesa (Waiter) e avisá-lo quando o pedido estiver pronto para ser levado à respetiva mesa.

waitForOrder()

```
static void waitForOrder ()
{

    //=============================================================
    //TODO insert your code here
    //Waiting for order
    if (semDown (semgid, sh->waitOrder) == -1) {                                    /*
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
    lastGroup = sh->fSt.foodGroup;
    //completed
    //=============================================================

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

    //=============================================================
    //TODO insert your code here

    sh->fSt.foodOrder = 0;
    sh->fSt.st.chefStat=COOK;
    saveState(nFic, &sh->fSt);
    //completed
    //=============================================================

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

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

Esta função representa a espera do cozinheiro até receber o pedido do empregado de mesa.

O primeiro passo é decrementar-se os semáforos *waitOrder,* atribuindo-se o valor do grupo do pedido à variável *lastGroup,* e *mutex* e, dentro da região crítica e atualiza-se o estado do cozinheiro para *COOK* (guardando o estado), atribuindo o valor 0 a *foodOrder* de forma a indicar que o pedido foi recebido.

Desta forma, o cozinheiro inicia a preparação do pedido, e terminando este processo, incrementa-se o valor do *mutex* para sair da região crítica e o semáforo *orderReceived*, para que o empregado espere que o pedido seja preparado.

processOrder()

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

    //==============================================================
    //TODO insert your code here
    request info;
    info.reqGroup = lastGroup;
    info.reqType =FOODREADY;

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

    if (semDown (semgid, sh->mutex) == -1) {                                    /* en
        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; //atualizar estado do chefe
    sh->fSt.waiterRequest = info;
    saveState(nFic, &sh->fSt); //guardar estado interno
    //completed
    //==============================================================


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

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

Esta função representa a preparação e a entrega do pedido ao empregado de mesa depois de ele estar preparado.

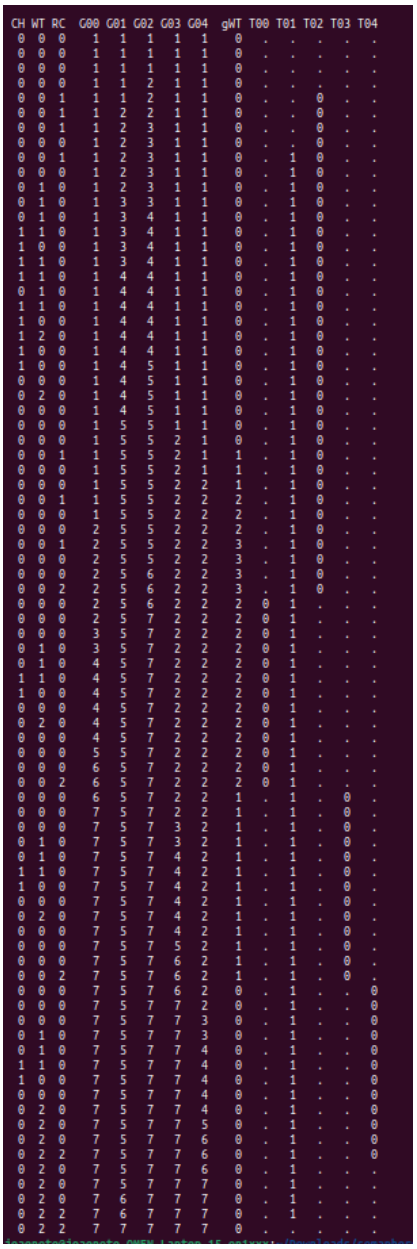
A função começa por simular o tempo que a preparação de cada pedido pelo cozinheiro demora, e depois deste tempo terminar, guarda-se o valor do grupo ao qual será entregue o pedido e atualiza-se o tipo de pedido para *FOODREADY.*

De seguida, os semáforos *waiterRequestPossible* e *mutex* são decrementados, respetivamente para, chamar o empregado de mesa e aceder à região crítica, onde se atualiza o estado do cozinheiro para *WAIT_FOR_ORDER* (guardando o estado), pois este terá outros pedidos para preparar.

Depois disto, incrementa-se o mutex para sair da região crítica e incrementa-se o semáforo *waiterRequest*, para que o empregado entregue o pedido de acordo com as indicações.

Testes

Teste *default*: make com tudo normal



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

Os resultados, mostraram-se semelhantes aos resultados da versão pré-compilada.

Através do comando ./run.sh fomos realizando sucessivos testes. Inicialmente fomos nos deparando com alguns erros, esses que foram sendo corrigidos. Testámos também o programa para diferentes ocasiões e situações, alterando o ficheiro config.txt, como por exemplo, para um maior número de grupos.

Conclusão

Este projeto revelou ser uma boa abordagem para compreender os mecanismos associados à execução e sincronização de processos e threads. Ao criar uma representação virtual de um ambiente dinâmico como um restaurante, onde diversas entidades interagem simultaneamente, tivemos oportunidade de explorar melhor os conceitos fundamentais sobre esta matéria.

A gestão eficiente de recursos, a prevenção de condições de corrida e a implementação de estratégias de sincronização foram aspetos cruciais explorados durante o desenvolvimento.