Projeto SO

Mecanismos associados à execução e sincronização de processos e threads

Pedro Pinto nº115304

Jorge Domingues nº113278

Resumo

O objetivo deste trabalho consistiu em desenvolver uma estratégia eficaz para sincronizar vários processos independentes por meio de utilização de semáforos e memória partilhada. Esta sincronização irá nos permitir simular um restaurante que terá como principais entidades (processos): Grupos, Rececionista, Empregado e Cozinheiro. A implementação deste conceito foi realizada na linguagem de programação C sendo a saída apresentada no terminal de execução ou em um ficheiro introduzido na execução. A estrutura do trabalho foi distribuída em módulos e foram realizados vários testes para garantir um bom funcionamento do sistema.

Keywords: *Semáforos, Memória Partilhada, Sistemas Operativos*

Conteúdo

1. Introdução

No âmbito da disciplina de Sistemas Operativos, o propósito principal deste projeto é a implementação de quatro entidades que constituem um Restaurante. Estas ferramentas possibilitam uma sincronização mútua que, ao ser corretamente resolvida, resultará numa representação da evolução dos estados de cada entidade. Como ponto de partida, seguimos a orientação do enunciado ao compilar o ficheiro "*all_bin*". A análise deste ficheiro foi o ponto inicial, permitindo-nos compreender as tarefas a realizar no projeto, assim como a análise de cada processo independente, contribuindo para uma compreensão mais precisa do problema. Vamos agora apresentar as entidades que fazem parte do funcionamento do programa:

Rececionista (Receptionist): Responsável por indicar os Grupos para as mesas disponíveis, ou no caso de estarem todas ocupadas coloca-os numa "sala de espera". Além disso, gere os pagamentos no final de cada refeição, após a solicitação por parte do Grupo em questão.

Grupos (Groups): Representam as entidades que frequentam o restaurante. Inicialmente, solicitam uma mesa e, posteriormente, realizam o pedido ao Empregado. Após fazerem a sua refeição, procedem ao pagamento com o Rececionista.

Empregado (Waiter): Desempenha a função de receber os pedidos dos Grupos e, em seguida, transmiti-los ao Cozinheiro. Por fim, entrega a comida aos grupos assim que estiver pronta.

Cozinheiro (Chef): Recebe os pedidos do Empregado e inicia o processo de preparação das refeições solicitadas. Ao finalizar, entrega a comida ao Empregado, que, por sua vez, encarrega-se de entregá-la ao Grupo.

2. Mutex e utilização de Semáforos

Para o desenvolvimento deste trabalho é importante explicar o termo "mutex"e para que serve. Mutex (exclusão mútua) é um mecanismo de sincronização frequentemente usado para evitar que vários threads tentem acessar e modificar os mesmos dados simultaneamente. Isto leva a que o programa tenha comportamentos inesperados e bugs. Então foi usado um semáforo mutex para que apenas um thread tenha acesso à memória por vez. Será usado para quando for preciso alterar algum valor de uma variável que se encontra na memória partilhada, não iremos mencionar esta questão durante o relatório, para uma melhor compreensão, mas é útil ter em conta o seu funcionamento, ilustrado na Figura 1.

Para além disso, no âmbito deste projeto, recorremos igualmente à utilização de semáforos para controlar as interações e a sincronização. Ao definir o semáforo como 'down', uma thread procura obter recursos críticos e, caso estes estejam disponíveis, adquire os recursos desejados, decrementando, simultaneamente, o contador. Se os recursos não estiverem disponíveis, a thread aguarda até que estes se tornem acessíveis. Ao definir o semáforo como 'up', a thread liberta recursos, indicando disponibilidade para outras threads. A operação 'up' incrementa o contador, assinalando que um recurso foi libertado.

No contexto da comunicação entre entidades, os semáforos presentes em *sharedDataSync* gerem as ações. Ao definir um semáforo como 'down', uma entidade solicita serviço ou indica que se encontra ocupada. Ao definir como 'up', sinaliza uma nova tarefa ou a conclusão de uma tarefa. É pertinente salientar que a implementação de cada semáforo pode variar consoante o caso de uso.

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

// Acess shared memory here!
sh->fSt.waiterRequest.reqType = FOODREADY;
sh->fSt.waiterRequest.reqGroup = lastGroup;

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

Figura 1. Exemplo de utilização do mutex (Chef)

3. Comunicação: Groups e Receptionist

A comunicação entre Grupos e o Rececionista depende da utilização da estrutura *request*, que é usada para filtrar os pedidos e identificar o remetente do pedido. Esta estrutura possui dois campos: *reqType* e *reqGroup*. É armazenada na memória partilhada como *sh->fSt.receptionistRequest* e, localmente, pelo Rececionista na variável *req*. Ambas as comunicações apresentadas a seguir seguem esta metodologia, apresentada na Figura 2 para solicitar e processar um pedido com o Rececionista.

Esta comunicação depende diretamente do uso de cinco funções: *waitForGroup*, *checkInAtReception*, *provideTableOrWaitingRoom*, *checkOutAtReception* e *receivePayment*. É importante salientar que todas são utilizadas pelos Grupos e pelo Rececionista.

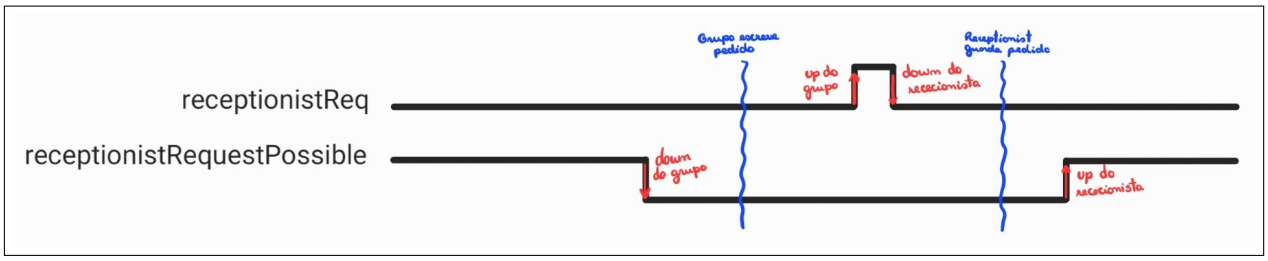

Figura 2. Metodologia de comunicação Groups - Receptionist

3.1. Entrada no restaurante

A função *waitForGroup* é utilizada pelo Rececionista para aguardar pelos grupos. Esta função utiliza dois semáforos que são inversamente utilizados pelos Grupos. Inicialmente, o Rececionista faz um 'down' do semáforo **receptionistReq** que simula a espera pelos grupos e, quando sinalizado que um grupo chegou, começa o processamento do pedido para a variável local, Figura 2. Após isso, faz um 'up' do semáforo **receptionistRequestPossible** para sinalizar que está pronto para receber novos pedidos. Por sua vez, na função *checkInAtReception*, os Grupos fazem o inverso. Um 'down' do semáforo **receptionistRequestPossible** para verificar se podem ou não fazer o pedido e, quando puderem, um 'up' do semáforo **receptionistReq**, para sinalizar ao Rececionista que pode começar a processar o pedido. É importante notar que, durante a fase entre os dois semáforos, os Grupos colocam o seu respetivo ID no campo *reqGroup* e o tipo de pedido (TABLEREQ) no campo *reqType* da estrutura explicada anteriormente. Após a solicitação, o grupo fica à espera com o 'down' do semáforo **sh->waitForTable[id]**, aguardando a atribuição de mesa. Para que essa atribuição aconteça, o Rececionista executa a função *provideTableOrWaitingRoom*, que irá, como o nome indica, redirecionar o Grupo para uma mesa livre ou simplesmente colocá-lo na sua própria lista de espera (fazendo $groupRecord[n] = WAIT$).

Para a seleção da mesa, o Rececionista utiliza uma função auxiliar designada por *decideTableOrWait*, Figura 3, que faz uso do array em memória partilhada, designado por *assignedTable*, que guarda as mesas atribuídas a cada Grupo e verifica se existe alguma mesa não ocupada. No caso de ser selecionada uma mesa, o Rececionista avisa o Grupo fazendo o 'up' do semáforo **waitForTable[n]**, onde n é o ID do respetivo grupo. Assim, com estas funções, conseguimos gerir a receção do restaurante.

```
158 static int decideTableOrWait(int n)
159 {
160     // TODO insert your code here
161     int g;
162     for (int table = 0; table < NUMTABLES; table++) {
163         for (g = 0; g < sh->fSt.nGroups && sh->fSt.assignedTable[g] != table; g++);
164         if (g == sh->fSt.nGroups) return table;
165     }
166     return -1;
167 }
```

Figura 3. Atribuição das mesas - *decideTableOrWait*

3.2. Receber pagamentos

No final da refeição, o Rececionista é solicitado mais uma vez pelo Grupo para realizar o pagamento da refeição. Esta comunicação depende diretamente de duas funções: *checkOutAtReception* e *receivePayment*. O Grupo que pretende solicitar o pagamento executa a função *checkOutAtReception*, que verifica se o Rececionista está disponível fazendo um 'down' do semáforo **receptionistRequestPossible**. Após isso, como explicado anteriormente, o Grupo coloca as informações do pedido na variável indicada da memória partilhada e faz um 'up' do semáforo **receptionistReq** para sinalizar ao Rececionista que pode processar o pedido. Assim, como no subcapítulo anterior, o Grupo aguarda, neste caso pela confirmação, fazendo um 'down' do semáforo **tableDone[table]**. Por outro lado, o Rececionista faz o processo inverso. Utilizando a função já explicada *waitForGroup*, ele aguarda pelo pedido do Grupo e, após isso, filtra o pedido e executa a função *receivePayment*, que irá confirmar o pagamento e, se necessário, redirecionar um Grupo que está à espera para a mesa que acabou de ser desocupada. Para isso, esta função utiliza uma função auxiliar, definida como *decideNextGroup*, Figura 4, que utilizando os estados WAIT do array *groupRecord* verifica se a mesa deve ser reocupada. Esta seleção toma como prioridade os Grupos com menor ID, numa futura implementação, seria útil considerar o uso de uma fila para manter a ordem de espera dos Grupos. Depois disso, à semelhança da função *provideTableOrWaitingRoom*, faz uso do semáforo **waitForTable[n]** para informar que o Grupo pode utilizar essa mesa. E logo a seguir, confirma o Grupo que terminou a refeição do pagamento, fazendo assim uma troca na mesma mesa entre dois Grupos."

```
177 static int decideNextGroup()
178 {
179     // TODO insert your code here
180     if(sh->fSt.groupsWaiting == 0) return -1;
181 
182     int r = -1;
183     for (int n = 0; n < sh->fSt.nGroups; n++) {
184         if (groupRecord[n] == WAIT) {
185             r = n; // next group = smallest id that is waiting
186             break;
187         }
188     }
189 
190     return r;
191 }
```

Figura 4. Decidir o próximo grupo - *decideNextGroup*

4. Comunicação: Groups, Waiter e Chef

A comunicação entre Grupos, Empregado e Cozinheiro depende da utilização de uma estrutura *request*, que é usada para filtrar os pedidos e identificar o remetente do pedido. Esta estrutura possui dois campos: *reqType* e *reqGroup*. É armazenada na memória partilhada como *sh->fSt.waiterRequest* e, localmente, pelo Empregado na variável local *req*. Ambas as comunicações apresentadas a seguir seguem a metodologia, apresentada na Figura 5 para solicitar e processar um pedido com o Empregado. Neste exemplo é apresentada a comunicação de um Grupo com o Empregado.

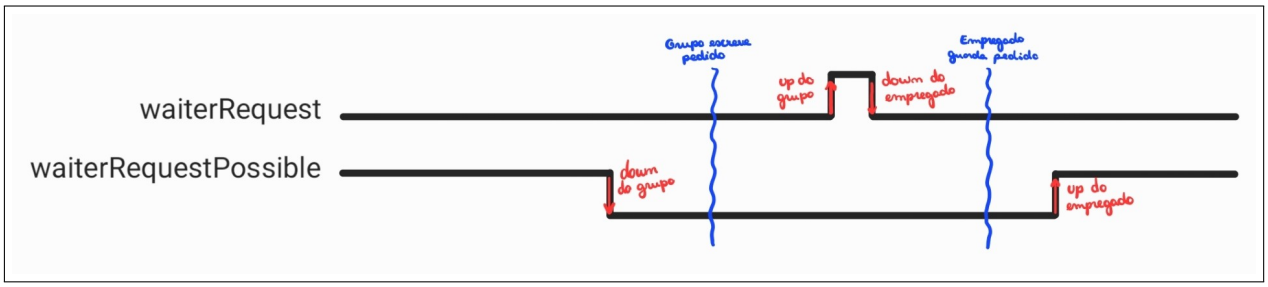

Figura 5. Metodologia de comunicação Groups - Waiter

4.1. Pedido de comida ao Empregado

O processo de realização de pedidos de comida ao Empregado envolve o uso de duas funções: *orderFood* e *waitForClientOrChef*.

A função *orderFood* é utilizada pelos Grupos para simular a solicitação da comida ao Empregado. Esta função faz uso de três semáforos, os quais são inversamente utilizados na função *waitForClientOrChef* utilizada pelo Empregado. Inicialmente, o Grupo faz um 'down' do semáforo **waiterRequestPossible** para verificar se o pedido é possivél e depois atualiza os dados do *waiterRequest*, conforme explicado anteriormente, indicando o tipo, *reqType*, e o seu ID, *reqGroup*, assim como atualiza o seu estado internamente. Após isso faz um 'up' do semáforo **waiterRequest** para avisar que já fez o pedido e depois coloca-se em espera com o 'down' do semáforo **requestReceived[table]**. Por outro lado o Empregado que estará preso em um 'down' do **waiterRequest**, irá começar a processar esse pedido. É relevante ressaltar que o Empregado voltar a ficar disponível após este processo, colocando o semáforo **waiterRequestPossible** para 'up', antes de executar o pedido solicitado.

```
sh->fSt.waiterRequest.reqType = FOODREQ;
sh->fSt.waiterRequest.reqGroup = id;
```

Figura 6. Envio do pedido - Grupo

```
req.reqGroup = sh->fSt.waiterRequest.reqGroup;
req.reqType = sh->fSt.waiterRequest.reqType;
```

Figura 7. Leitura do pedido - Empregado

4.2. Envio do pedido ao Cozinheiro

Após o pedido ser concluído, inicia-se o processo em que o Empregado informa o Cozinheiro sobre a solicitação. Neste processo são usadas duas funções. A função *informChef* utilizada pelo Empregado e a função *waitForOrder* utilizada pelo Cozinheiro. É importante relembrar que as duas função interagem entre si para que cada uma das entidades se possam comunicar. No princípio do processo, o Cozinheiro tem o seu semáforo **waitOrder** em 'down' indicando que está à espera que o Empregado lhe entregue algum pedido. Antes do Empregado entregar o pedido ao Cozinheiro, este avisa o Grupo que entregou o pedido com o 'up' do semáforo **requestReceived[table]**, e de seguida entrega o pedido ao Cozinheiro, Figura 8. Para indicar ao Cozinheiro que fez o pedido, o Empregado faz um 'up' do semáforo **waitOrder**. É importante ressaltar que o Empregado envia a informação sobre o Grupo ao Cozinheiro para que ele saiba de quem é cada pedido,

foodGroup. Posteriormente o Empregado coloca o semáforo **orderReceived** para 'down', para aguardar resposta vinda do Cozinheiro a dizer que irá processar o pedido. Isso acontece quando o Cozinheiro começa a cozinhar o pedido, fazendo um 'up' do semáforo **orderReceived**.

```
sh->fSt.foodGroup = n;
sh->fSt.foodOrder = 1;
```

Figura 8. Envio do pedido - Empregado

```
lastGroup = sh->fSt.foodGroup;
sh->fSt.foodOrder = 0;
```

Figura 9. Leitura do pedido - Chef

4.3. Cozinheiro entrega a comida pronta ao Empregado

Quando o Cozinheiro conclui o pedido, comunica ao Empregado que este já pode proceder à recolha da comida para entregá-la à mesa. Este processo envolve duas funções que se sincronizam por meio de semáforos inversos: *processOrder*, utilizada pelo Chef, e *waitForClientOrChef*, utilizada pelo Empregado. O Cozinheiro começa por colocar o semáforo **waiterRequestPossible** para 'down', para verificar a disponibilidade do Empregado. Fica à espera até o Empregado ficar livre e só depois entrega o pedido ao Empregado e indica-lhe o grupo a que se destina, Figura 10, anunciando esse pedido com o 'up' do semáforo **waiterRequest**. Por outro lado, o Empregado depois de ficar disponível vai recolher o pedido do Cozinheiro. Com isto ele coloca o semáforo **waiterRequest** para 'down' para iniciar a tarefa. Ele recebe o pedido e prepara-se para o entregar à mesa. Podemos verificar que no fim da função *waitForClientOrChef*, o Empregado coloca-se outra vez disponível para outros pedidos que forem surgindo, **waiterRequestPossible** para 'up'. Agora só falta entregar a comida ao Grupo correto.

```
sh->fSt.waiterRequest.reqType = FOODREADY;
sh->fSt.waiterRequest.reqGroup = lastGroup;
```

Figura 10. Envio do pedido - Chef

```
req.reqGroup = sh->fSt.waiterRequest.reqGroup;
req.reqType = sh->fSt.waiterRequest.reqType;
```

Figura 11. Leitura do pedido - Empregado

4.4. Entrega da comida à mesa

No momento da entrega da comida ao Grupo, são envolvidas também duas funções: *takeFoodToTable* utilizada pelo Empregado e *waitFood* utilizada pelo Grupo. Voltando um pouco atrás, após o Grupo efetuar o pedido entra em estado de espera e faz um 'down' do semáforo **foodArrived[table]**, indicando que o grupo a partir desse momento está à espera que a refeição chegue. Quando o Empregado chega com o pedido, este faz o 'up' do semáforo **foodArrived[table]**, que sinaliza o Grupo que pode começar a comer. Assim, o Empregado consegue entregar à mesa correta utilizando o *reqGroup* (lastGroup) fornecido pelo Cozinheiro na entrega da comida. Após isto, o Grupo entra em estado de 'EAT' até terminar o jantar e depois, começa o processo de pagamento conforme explicado anteriormente.

5. Testes

A secção de testes é uma parte muito importante deste projeto pois assegura que nenhuma das entidades fica presa em alguma fase. Esta secção, ao fazer muitos testes, verifica por exemplo a existência de *DeadLocks*, que trabalhámos ao máximo para que não acontecessem.

Embora não seja o nosso foco neste projeto, durante a implementação tivemos o cuidado de atribuir estados a cada entidade para poder distinguir um resultado correto. Estes estados são guardados em memória partilhada e permitem visualizar nos testes o correto funcionamento. Na Figura 12 são apresentados os estados para cada entidade.

```
/* Client state constants */

/** \brief group initial state */
#define GOTOREST 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

/* 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

/* Waiter state constants */

/** \brief waiter 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

/* Receptionist state constants */

/** \brief receptionist waits for request */
#define WAIT_FOR_REQUEST        0  // (new!)
/** \brief receptionist assign a table */
#define  ASSIGNTABLE        1
/** \brief receptionist receives payment */
#define  RECVPAY            2
```

Figura 12. Estados das entidades, localizados em *probConst.h*

Para a realização dos testes utilizámos os scripts já fornecidos da diretoria *run* e modificamos alguns para se ajustarem aos testes que pretendiamos fazer. Como por exemplo o **clean.sh**, para terminar a memória partilhada e os semáforos que foram utilizados e não foram terminados, o **execute.sh**, criado por nós para fazer testes unitários a cada uma das entidades, aceitando por argumento as siglas especificadas no *Makefile*, o **filter.sh**, que executa o restaurante mas mostra apenas a informação mais importante e o **run.sh** que executa enumeras vezes o restaurante para verificar possíveis erros, util para verificar *Deadlocks*. Foi alterado também o ficheiro *config.txt* para adicionar ou remover grupos e alterar os seus tempos de execução, um exemplo é ilustrado na Figura 13.

```
#ngroups
4
#startTime timeToEat
50000 120000
55000 80000
70000 100000
45000 120000
```

Figura 13. Exemplo do ficheiro *config.txt*

Um exemplo de teste unitário ao Rececionista, seria por exemplo, executar o comando **./execute rt** e verificar o 'ouput' e concluir se correu ou não como esperado. A Figura 14 apresenta o 'output' desse comando para apenas quatro grupos. Foram feitos os mesmos testes para todas as entidades com os argumentos: ch (Chef), gr (Groups) e wt (Waiter).

```
pedro@pedroh:~/Documents/Universidade/2ano/1semestre/SO/SO-project-2/run$ ./execute.sh rt
(...) - Compilacao
                                Restaurant - Description of the internal state

 CH WT RC  G00 G01 G02 G03  gWT T00 T01 T02 T03
  0  0  0    1   1   1   1    0   .   .   .   .
  0  0  0    1   1   1   1    0   .   .   .   .
  0  0  0    1   1   1   1    0   .   .   .   .
  0  0  0    1   1   1   2    0   .   .   .   .
  0  0  1    1   1   1   2    0   .   .   .   .
  0  0  0    1   1   1   2    0   .   .   .   0
  0  0  0    1   1   1   3    0   .   .   .   0
  0  1  0    1   1   1   3    0   .   .   .   0
  1  1  0    1   1   1   3    0   .   .   .   0
  1  1  0    1   1   1   4    0   .   .   .   0
  1  0  0    1   1   1   4    0   .   .   .   0
  0  0  0    1   1   1   4    0   .   .   .   0
  0  2  0    1   1   1   4    0   .   .   .   0
  0  0  0    1   1   1   4    0   .   .   .   0
  0  0  0    1   1   1   5    0   .   .   .   0
  0  0  0    2   1   1   5    0   .   .   .   0
  0  0  1    2   1   1   5    0   .   .   .   0
  0  0  0    2   1   1   5    0   1   .   .   0
  0  0  0    3   1   1   5    0   1   .   .   0
  0  1  0    3   1   1   5    0   1   .   .   0
  0  1  0    4   1   1   5    0   1   .   .   0
  1  1  0    4   1   1   5    0   1   .   .   0
  1  0  0    4   1   1   5    0   1   .   .   0
  0  0  0    4   1   1   5    0   1   .   .   0
  0  2  0    4   1   1   5    0   1   .   .   0
  0  0  0    4   1   1   5    0   1   .   .   0
  0  0  0    5   1   1   5    0   1   .   .   0
  0  0  0    5   2   1   5    0   1   .   .   0
  0  0  1    5   2   1   5    0   1   .   .   0
  0  0  0    5   2   1   5    1   1   .   .   0
  0  0  0    5   2   2   5    1   1   .   .   0
  0  0  1    5   2   2   5    1   1   .   .   0
  0  0  0    5   2   2   5    2   1   .   .   0
  0  0  0    5   2   2   6    2   1   .   .   0
  0  0  2    5   2   2   6    2   1   .   .   0
  0  0  0    5   2   2   6    1   1   0   .   .
  0  0  0    5   2   2   7    1   1   0   .   .
  0  0  0    5   3   2   7    1   1   0   .   .
  0  1  0    5   3   2   7    1   1   0   .   .
  1  1  0    5   3   2   7    1   1   0   .   .
  1  1  0    5   4   2   7    1   1   0   .   .
  1  0  0    5   4   2   7    1   1   0   .   .
  0  0  0    5   4   2   7    1   1   0   .   .
  0  2  0    5   4   2   7    1   1   0   .   .
  0  0  0    5   4   2   7    1   1   0   .   .
  0  0  0    5   5   2   7    1   1   0   .   .
  0  0  0    6   5   2   7    1   1   0   .   .
  0  0  2    6   5   2   7    1   1   0   .   .
  0  0  0    6   5   2   7    0   .   0   1   .
  0  0  0    7   5   2   7    0   .   0   1   .
  0  0  0    7   5   3   7    0   .   0   1   .
  0  1  0    7   5   3   7    0   .   0   1   .
  1  1  0    7   5   3   7    0   .   0   1   .
  1  1  0    7   5   4   7    0   .   0   1   .
  1  0  0    7   5   4   7    0   .   0   1   .
  0  0  0    7   5   4   7    0   .   0   1   .
  0  2  0    7   5   4   7    0   .   0   1   .
  0  2  0    7   5   5   7    0   .   0   1   .
  0  2  0    7   6   5   7    0   .   0   1   .
  0  2  2    7   6   5   7    0   .   0   1   .
  0  2  0    7   6   5   7    0   .   .   1   .
  0  2  0    7   7   5   7    0   .   .   1   .
  0  2  0    7   7   6   7    0   .   .   1   .
  0  2  2    7   7   6   7    0   .   .   1   .
  0  2  2    7   7   7   7    0   .   .   .   .
```

Figura 14. Teste unitário ao Rececionista (simplificado)

De forma a simplificar a visualização, embora tenhamos feito testes que abrangiam mais grupos, o teste apresentado na Figura 15 apresenta a última iteração do *output* ao executar **./run.sh 5000** para duas mesas e quatro grupos com as constantes que influenciam as distribuições aleatórias: $MAXCOOK = 10^5, STARTDEV = 4 \times 10^4, EATDEV = 4 \times 10^4$, e o ficheiro *config.txt* como apresentado na Figura 13.

```
Run n. 5000
                                  Restaurant - Description of the internal state

 CH  WT  RC   G00  G01  G02  G03   gWT  T00  T01  T02  T03
  0   0   0    1    1    1    1     0    .    .    .    .
  0   0   0    1    1    1    1     0    .    .    .    .
  0   0   0    1    1    1    1     0    .    .    .    .
  0   0   0    1    2    1    1     0    .    .    .    .
  0   0   1    1    2    1    1     0    .    .    .    .
  0   0   0    1    2    1    1     0    .    0    .    .
  0   0   0    1    3    1    1     0    .    0    .    .
  0   1   0    1    3    1    1     0    .    0    .    .
  0   1   0    1    4    1    1     0    .    0    .    .
  1   1   0    1    4    1    1     0    .    0    .    .
  1   0   0    1    4    1    1     0    .    0    .    .
  1   0   0    1    4    2    1     0    .    0    .    .
  1   0   1    1    4    2    1     0    .    0    .    .
  1   0   0    1    4    2    1     0    .    0    1    .
  1   0   0    1    4    3    1     0    .    0    1    .
  1   1   0    1    4    3    1     0    .    0    1    .
  1   1   0    1    4    4    1     0    .    0    1    .
  0   1   0    1    4    4    1     0    .    0    1    .
  1   1   0    1    4    4    1     0    .    0    1    .
  1   0   0    1    4    4    1     0    .    0    1    .
  1   2   0    1    4    4    1     0    .    0    1    .
  1   0   0    1    4    4    1     0    .    0    1    .
  1   0   0    1    5    4    1     0    .    0    1    .
  0   0   0    1    5    4    1     0    .    0    1    .
  0   2   0    1    5    4    1     0    .    0    1    .
  0   0   0    1    5    4    1     0    .    0    1    .
  0   0   0    1    5    5    1     0    .    0    1    .
  0   0   0    2    5    5    1     0    .    0    1    .
  0   0   1    2    5    5    1     0    .    0    1    .
  0   0   0    2    5    5    1     1    .    0    1    .
  0   0   0    2    5    5    2     1    .    0    1    .
  0   0   1    2    5    5    2     1    .    0    1    .
  0   0   0    2    5    5    2     2    .    0    1    .
  0   0   0    2    5    6    2     2    .    0    1    .
  0   0   2    2    5    6    2     2    .    0    1    .
  0   0   0    2    5    6    2     1    1    0    .    .
  0   0   0    2    5    7    2     1    1    0    .    .
  0   0   0    3    5    7    2     1    1    0    .    .
  0   1   0    3    5    7    2     1    1    0    .    .
  0   1   0    4    5    7    2     1    1    0    .    .
  1   1   0    4    5    7    2     1    1    0    .    .
  1   0   0    4    5    7    2     1    1    0    .    .
  0   0   0    4    5    7    2     1    1    0    .    .
  0   2   0    4    5    7    2     1    1    0    .    .
  0   0   0    4    5    7    2     1    1    0    .    .
  0   0   0    5    5    7    2     1    1    0    .    .
  0   0   0    5    6    7    2     1    1    0    .    .
  0   0   2    5    6    7    2     1    1    0    .    .
  0   0   0    5    6    7    2     0    1    .    .    0
  0   0   0    5    7    7    2     0    1    .    .    0
  0   0   0    5    7    7    3     0    1    .    .    0
  0   1   0    5    7    7    3     0    1    .    .    0
  0   1   0    5    7    7    4     0    1    .    .    0
  1   1   0    5    7    7    4     0    1    .    .    0
  1   0   0    5    7    7    4     0    1    .    .    0
  0   0   0    5    7    7    4     0    1    .    .    0
  0   2   0    5    7    7    4     0    1    .    .    0
  0   2   0    5    7    7    5     0    1    .    .    0
  0   2   0    6    7    7    5     0    1    .    .    0
  0   2   2    6    7    7    5     0    1    .    .    0
  0   2   0    6    7    7    5     0    .    .    .    0
  0   2   0    7    7    7    5     0    .    .    .    0
  0   2   0    7    7    7    6     0    .    .    .    0
  0   2   2    7    7    7    6     0    .    .    .    0
  0   2   2    7    7    7    7     0    .    .    .    .
```

Figura 15. Teste n.º 5000

6. Conclusão

Em conclusão, acreditamos que conseguimos cumprir com os objetivos propostos e adquirimos bastantes conhecimentos no desenvolvimento deste projeto pois colocámos em prática conceitos aprendidos nas aulas teóricas e práticas. Apesar disso, consideramos que uma versão melhorada deste restaurante devia implementar uma fila com prioridade na entrada do restaurante, ao invés de dar prioridade aos que têm menor ID, seria uma futura implementação utilizando por exemplo *Message Queues*.