

Sistemas Operativos

RESTAURANTE

TRABALHO REALIZADO POR:

DIOGO FERNANDES 114137

RAQUEL VINAGRE 113736

PROF. JOSÉ NUNO PANELAS NUNES LAU

ANO LETIVO 2023/2024

Índice

Sistemas Operativos.....	1
Introdução.....	3
Comportamento dos semáforos	4
checkInAtReception().....	5
orderFood()	6
waitFood()	8
checkOutAtReception ()	9
Waiter	12
waitForClientOrChef()	12
informChef().....	14
takeFoodToTable().....	16
Chef	17
waitForOrder()	17
processOrder()	19
Receptionist	20
decideTableOrWait().....	20
decideNextGroup()	22
waitForGroup()	23
provideTableOrWaitingRoom().....	25
receivePayment()	26
Testes e validação de resultados	27
Conclusão	29

Introdução

Este trabalho tem como objetivo a compreensão dos mecanismos associados à execução e sincronização de processos e threads. O tema deste projeto consiste em vários grupos de pessoas que jantam num restaurante com duas mesas. No total, há quatro processos independentes: grupos, receptionist, empregado e chef. A sua sincronização será realizada através de semáforos e memória partilhada.

Neste relatório serão apresentados os segmentos de código completado, sendo os outros ficheiros também essenciais ao funcionamento do programa.

Comportamento dos semáforos

Semáforo	Up			Down		
	Processo	Funções	Número	Processo	Funções	Número
mutex	Group	checkInAtReception	1	Group	checkInAtReception	1
		orderFood	1		orderFood	1
		waitFood	2		waitFood	1
		checkOutAtReception	2		checkOutAtReception	2
	Waiter	waitForClientOrChef	2	Waiter	waitForClientOrChef	2
		informChef	1		informChef	1
		takeFoodToTable	1		takeFoodToTable	1
	Chef	waitForOrder	1	Chef	waitForOrder	1
		processOrder	1		processOrder	2
	Receptionist	waitForGroup	2	Receptionist	waitForGroup	2
		provideTableOrWaitingRoom	1		provideTableOrWaitingRoom	1
		receivePayment	1		receivePayment	1
receptionistRequestPossible	Receptionist	waitForGroup	1	Group	checkInAtReception	1
					checkOutAtReception	1
receptionistReq	Group	checkInAtReception	1	Receptionist	waitForGroup	1
		checkOutAtReception	1			
waiterRequest	Chef	processOrder	1	Waiter	waitForClientOrCHEF	1
	Group	orderFood	1			
waiterRequestPossible	Waiter	waitForClientOrChef	1	Group	orderFood	1
				Chef	processOrder	1
waitOrder	Waiter	informChef	1	Chef	waitForOrder	1
orderReceived	Chef	waitForOrder	1	Waiter	informChef	1
waitForTable	Receptionist	provideTableOrWaitingRoom	1	Group	checkInAtReception	1
		receivePayment	1			
foodArrived	Waiter	takeFoodToTable	1	Group	waitFood	1
tableDone	Receptionist	receivePayment	1	Group	checkOutAtReception	1
requestReceived	Waiter	informChef	1	-	-	-

Group

Cada grupo deve dirigir-se em primeiro lugar ao receptionist, que lhe indicará a mesa a ocupar ou se deve esperar, no caso de as duas mesas estarem ocupadas. Após obter mesa, o grupo pede a comida ao waiter, espera que esta chegue e começa a comer, assim que o waiter a leva à mesa. No final, o grupo contacta novamente o receptionist para pagar a conta e sai.

checkInAtReception()

Nesta função o grupo vai para a receção e espera pelo receptionist.

Inicialmente, decrementamos o semáforo receptionistRequestPossible pois o receptionist precisa de estar disponível. Em seguida, entra-se na região crítica de memória, atualizamos o estado do grupo para ATRECEPTION e simulamos o pedido de mesa, alterando o receptionistRequest type para TABLEREQ. Seleciona-se o ID do último grupo e, por fim guardamos o estado.

```
1 // TODO insert your code here
2
3 if (semDown (semgid, sh->receptionistRequestPossible) == -1)
4 {
5     perror ("error on the up operation for semaphore access (PT)");
6     exit (EXIT_FAILURE);
7 }
8
9 //end of TODO
10
11 if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
12     perror ("error on the down operation for semaphore access (CT)");
13     exit (EXIT_FAILURE);
14 }
15
16 // TODO insert your code here
17
18 // Change the group state because it's now at the reception
19 sh->fSt.groupStat[id] = ATRECEPTION;
20
21 // Request a table
22 sh->fSt.receptionistRequest.reqType = TABLEREQ;
23 sh->fSt.receptionistRequest.reqGroup = id;
24 saveState(nFic, &sh->fSt);
25
26 //end of TODO
```

Por fim, sai-se da região crítica, incrementamos o semáforo receptionistReq, simulando que o receptionist recebeu o pedido, e decrementamos o semáforo waitForTable para garantir que o grupo tem de esperar até uma mesa estar livre.

```
1  if (semUp (semgid, sh->mutex) == -1) {                                /* exit critical region */
2      perror ("error on the up operation for semaphore access (CT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7
8  // Receptionist receives the request
9
10 if (semUp (semgid, sh->receptionistReq) == -1)
11 {
12     perror ("error on the up operation for semaphore access (PT)");
13     exit (EXIT_FAILURE);
14 }
15
16 // Make sure the groups wait till there is a table available
17
18 if (semDown (semgid, sh->waitForTable[id]) == -1)
19 {
20     perror ("error on the up operation for semaphore access (PT)");
21     exit (EXIT_FAILURE);
22 }
23
24 //end of TODO
```

orderFood()

Nesta função, após conseguir uma mesa, o grupo espera pelo waiter.

Inicialmente, decrementamos o semáforo waiterRequestPossible para garantir que o waiter está disponível. E entra-se na região crítica de memória.

```
1  // TODO insert your code here
2
3  if (semDown (semgid, sh->waiterRequestPossible) == -1)
4  {
5      perror ("error on the up operation for semaphore access (PT)");
6      exit (EXIT_FAILURE);
7  }
8
9  //end of TODO
10
11 if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
12     perror ("error on the down operation for semaphore access (CT)");
13     exit (EXIT_FAILURE);
14 }
```

Em seguida, atualizamos o estado do grupo para FOOD_REQUEST, guardamos o estado e atualizamos o waiterRequest reqType e reqGroup para FOODREQ e para o ID dado na função, respetivamente.

```
1 // TODO insert your code here
2
3 // Change the group state because it's now ordering food
4 sh->fSt.groupStat[id] = FOOD_REQUEST;
5 saveState(nFic, &sh->fSt);
6
7 // Waiter receives the request
8 sh->fSt.waiterRequest.reqType = FOODREQ;
9 sh->fSt.waiterRequest.reqGroup = id;
10
11 //end of TODO
```

Por fim sai-se da região crítica da memória e incrementamos o semáforo waiterRequest para garantir que o waiter pode voltar a receber pedidos.

```
1 if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
2     perror ("error on the up operation for semaphore access (CT)");
3     exit (EXIT_FAILURE);
4 }
5
6 // TODO insert your code here
7
8 // Waiter can now receive requests again
9
10 if (semUp (semgid, sh->waiterRequest) == -1)
11 {
12     perror ("error on the up operation for semaphore access (PT)");
13     exit (EXIT_FAILURE);
14 }
15
16 //end of TODO
```

waitFood()

Após o pedido ser entregue ao waiter, o grupo espera pela comida e por fim come.

Entramos na região de memória crítica e alteramos o estado do grupo para WAIT_FOR_FOOD e guardamos o ID da mesa que foi atribuída.

```
1  if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
2      perror ("error on the down operation for semaphore access (CT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7
8  // Group is now waiting for food
9  sh->fSt.groupStat[id] = WAIT_FOR_FOOD;
10 saveState(nFic, &sh->fSt);
11
12 int table = sh->fSt.assignedTable[id];
13
14 //end of TODO
```

De seguida, sai-se da região crítica e incrementamos o semáforo foodArrived com o ID da mesa que guardamos.

```
1  if (semUp (semgid, sh->mutex) == -1) {                                /* enter critical region */
2      perror ("error on the down operation for semaphore access (CT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7
8  // Wait for the food to arrive
9  if (semDown (semgid, sh->foodArrived[table]) == -1)
10 {
11     perror ("error on the up operation for semaphore access (PT)");
12     exit (EXIT_FAILURE);
13 }
14
15 //end of TODO
```


Por fim, voltamos a entrar na região crítica, o grupo come, atualizamos o estado para EAT e guardamos o mesmo.

```
1  if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
2      perror ("error on the down operation for semaphore access (CT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7
8  // Group is now eating
9  sh->fst.st.groupStat[id] = EAT;
10 saveState(nFic, &sh->fst);
11
12 //end of TODO
13
14
15 if (semUp (semgid, sh->mutex) == -1) { /* enter critical region */
16     perror ("error on the down operation for semaphore access (CT)");
17     exit (EXIT_FAILURE);
18 }
```

checkOutAtReception ()

O grupo espera pelo waiter até este estar disponível para receber o pagamento.

Primeiramente, incrementamos o semáforo receptionistRequestPossible e entramos na região crítica da memória. Depois, atualizamos o estado do grupo para CHECKOUT e guardamos o estado. Atualizamos o receptionistRequest type e group para BILLREQ e para o ID, respetivamente, e também guardamos o ID da mesa que foi atribuída.

```

1 // TODO insert your code here
2
3 if (semDown (semgid, sh->receptionistRequestPossible) == -1)
4 {
5     perror ("error on the up operation for semaphore access (PT)");
6     exit (EXIT_FAILURE);
7 }
8
9 //end of TODO
10
11 if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
12     perror ("error on the down operation for semaphore access (CT)");
13     exit (EXIT_FAILURE);
14 }
15
16 // TODO insert your code here
17
18 // Change the group state because it's now at the reception
19 sh->fSt.st.groupStat[id] = CHECKOUT;
20 saveState(nFic, &sh->fSt);
21
22 // Receptionist receives the bill request
23 sh->fSt.receptionistRequest.reqType = BILLREQ;
24 sh->fSt.receptionistRequest.reqGroup = id;
25
26 int table = sh->fSt.assignedTable[id];
27
28 //end of TODO

```

Sai-se da região crítica da memória, incrementamos o receptionistReq (o receptionist recebe o pedido) e decrementamos o semáforo tableDone com o ID da mesa guardado, para que depois do pagamento a mesa esteja livre.

```

1  if (semUp (semgid, sh->mutex) == -1) {                               /* enter critical region */
2      perror ("error on the down operation for semaphore access (CT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7
8  // Receptionist receives the request
9  if (semUp (semgid, sh->receptionistReq) == -1)
10 {
11     perror ("error on the up operation for semaphore access (PT)");
12     exit (EXIT_FAILURE);
13 }
14
15 // Table will be ready after the payment is done
16 if (semDown (semgid, sh->tableDone[table]) == -1)
17 {
18     perror ("error on the up operation for semaphore access (PT)");
19     exit (EXIT_FAILURE);
20 }
21
22 //end of TODO

```

Por fim, atualizamos o estado do group para LEAVING, guardamos o mesmo e sai-se da região crítica da memória.

```

1  if (semDown (semgid, sh->mutex) == -1) {                               /* enter critical region */
2      perror ("error on the down operation for semaphore access (CT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7
8  // Group is now leaving
9  sh->fSt.st.groupStat[id] = LEAVING;
10 saveState(nFic, &sh->fSt);
11
12 //end of TODO
13
14
15 if (semUp (semgid, sh->mutex) == -1) {                               /* enter critical region */
16     perror ("error on the down operation for semaphore access (CT)");
17     exit (EXIT_FAILURE);
18 }

```

Waiter

O waiter de mesa deve levar o pedido dos grupos ao chef e trazer a comida para a mesa quando esta estiver pronta.

Serão analisadas as três funções completadas que permitem o funcionamento deste processo: `waitForClientOrChef()`, `informChef()` e `takeFoodToTable()`.

`waitForClientOrChef()`

Numa primeira fase, o waiter espera pelo próximo pedido. É necessário entrar na região crítica, de modo a atualizar e guardar o seu estado como `WAIT_FOR_REQUEST`. De seguida, sai-se da região crítica.

```
1 static request waitForClientOrChef() //waiter waits for next request
2 {
3     request req;
4     if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
5         perror ("error on the up operation for semaphore access (WT)");
6         exit (EXIT_FAILURE);
7     }
8
9     // TODO insert your code here
10    // waiter updates state
11    sh->fSt.waiterStat = WAIT_FOR_REQUEST;
12    saveState(nFic, &sh->fSt);
13    //end of TODO
14
15    if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
16        perror ("error on the down operation for semaphore access (WT)");
17        exit (EXIT_FAILURE);
18    }
```

Entra-se, de novo, na região crítica, e o waiter guarda o pedido efetuado.

```
1 // TODO insert your code here
2 // waiter waits for request
3 if (semDown (semgid, sh->waiterRequest) == -1) { /* enter critical region */
4     perror ("error on the down operation for semaphore access (WT)");
5     exit (EXIT_FAILURE);
6 }
7
8 if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
9     perror ("error on the up operation for semaphore access (WT)");
10    exit (EXIT_FAILURE);
11 }
12
13 // TODO insert your code here
14 // waiter reads request
15 req = sh->fst.waiterRequest;
16 //end of TODO
```

Por último, o waiter avisa que é possível realizar novos pedidos, sai-se da região crítica e a função retorna o pedido atual (req).

```
1 if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
2     perror ("error on the down operation for semaphore access (WT)");
3     exit (EXIT_FAILURE);
4 }
5
6 // TODO insert your code here
7 // waiter signals that new requests are possible
8 if (semUp (semgid, sh->waiterRequestPossible) == -1) { /* exit critical region */
9     perror ("error on the down operation for semaphore access (WT)");
10    exit (EXIT_FAILURE);
11 }
12
13 return req;
14
15 }
```

informChef()

Neste momento, o waiter leva o pedido ao chef, e atualiza o seu estado para INFORM_CHEF, guardando-o. É alterada a flag foodOrder, indicando que o pedido tem de ser preparado e é ainda guardado o grupo (n) associado ao mesmo. É também atribuída uma mesa ao grupo, através do seu ID.

```
1 static void informChef (int n) //waiter takes food order to chef
2 {
3     if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
4         perror ("error on the up operation for semaphore access (WT)");
5         exit (EXIT_FAILURE);
6     }
7
8     // TODO insert your code here
9     // waiter updates state
10    sh->fSt.st.waiterStat = INFORM_CHEF;
11    saveState(nFic, &sh->fSt);
12
13    // saves group order
14    sh->fSt.foodGroup = n;
15    sh->fSt.foodOrder = 1;
16
17    // table assigned to group
18    int table_id = sh->fSt.assignedTable[n];
19
20    if (semUp (semgid, sh->mutex) == -1) /* exit critical region */
21    { perror ("error on the down operation for semaphore access (WT)");
22      exit (EXIT_FAILURE);
23    }
```

De seguida, o grupo é notificado de que o pedido foi recebido, através do semáforo requestReceived, que permite entrar na região crítica. O chef espera até receber o pedido (semáforo waitOrder) e o waiter espera que o chef o receba (semáforo orderReceived).

```
1 // TODO insert your code here
2 // notify group that request is received
3 if (semUp (semgid, sh->requestReceived[table_id]) == -1)
4 {
5     perror ("error on the up operation for semaphore access (PT)");
6     exit (EXIT_FAILURE);
7 }
8
9 // chef waits for order
10 if (semUp (semgid, sh->waitOrder) == -1)
11 {
12     perror ("error on the up operation for semaphore access (PT)");
13     exit (EXIT_FAILURE);
14 }
15
16 // waiter waits for chef receiving request
17 if (semDown (semgid, sh->orderReceived) == -1)
18 {
19     perror ("error on the up operation for semaphore access (PT)");
20     exit (EXIT_FAILURE);
21 }
22 //end of TODO
23
24 }
```

takeFoodToTable()

Na última parte deste processo, entra-se, mais uma vez, na região crítica. O waiter leva a comida à mesa e atualiza o seu estado (TAKE_TO_TABLE). Informa ao grupo que a comida está disponível (foodArrived) e, por fim, sai-se da região crítica da memória.

```
1 static void takeFoodToTable (int n)
2 {
3     if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
4         perror ("error on the up operation for semaphore access (WT)");
5         exit (EXIT_FAILURE);
6     }
7
8     // TODO insert your code here
9     // waiter takes food to table
10    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
11    saveState(nFic, &sh->fSt);
12
13    // inform group that food is available
14    if (semUp (semgid, sh->foodArrived [sh->fSt.assignedTable[n]]) == -1)
15    {
16        perror ("error on the up operation for semaphore access (PT)");
17        exit (EXIT_FAILURE);
18    }
19    //end of TODO
20
21    if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
22        perror ("error on the down operation for semaphore access (WT)");
23        exit (EXIT_FAILURE);
24    }
25 }
```


Chef

O chef recebe pedidos do waiter de cada mesa e trata de preparar a comida para o grupo. Após esta ficar pronta, pede ao waiter para a levar à mesa.

Serão analisadas as duas funções completadas que permitem o funcionamento deste processo: `waitForOrder()` e `processOrder()`,

`waitForOrder()`

Esta função trata de gerir os pedidos de comida feitos pelo waiter para cada mesa. O chef espera até o waiter fazer esse pedido.

Inicialmente, atualizamos o estado do chef para `WAIT_FOR_ORDER`, e guardamos o mesmo. Em seguida, decrementamos o semáforo `waitOrder` para que o chef, que estava previamente à espera de um pedido, o possa receber e continuar a sua execução.

```
1 //TODO insert your code here
2
3 //update chef state
4 sh->fSt.st.chefStat = WAIT_FOR_ORDER; //chef waits for order
5 saveState(nFic, &sh->fSt);
6
7 //block until waiter signals chef that there is a new order
8 if (semDown (semgid, sh->waitOrder) == -1)
9 {
10     perror ("error on the up operation for semaphore access (PT)");
11     exit (EXIT_FAILURE);
12 }
13
14
15 //end of TODO
```

Posteriormente, entra-se na região crítica da memória do programa. O `lastGroup` é atualizado para o devido grupo, a `foodOrder` também é alterada para 0, pois acaba de receber o pedido, e o estado do chef é atualizado para `COOK` e este é guardado.

```

1  if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
2      perror ("error on the up operation for semaphore access (PT)");
3      exit (EXIT_FAILURE);
4  }
5
6      //TODO insert your code here
7      //chef has received an order
8
9      //update chef state
10     lastGroup = sh->fSt.foodGroup;
11     sh->fSt.foodOrder = 0;
12
13     sh->fSt.st.chefStat = COOK; //chef is cooking
14     saveState(nFic, &sh->fSt);
15     //end of TODO

```

Por fim, sai-se da região crítica da memória e também incrementamos o semáforo orderReceived para que o waiter tenha a informação que o pedido foi recebido pelo chef.

```

1  if (semUp (semgid, sh->mutex) == -1) {                                /* exit critical region */
2      perror ("error on the up operation for semaphore access (PT)");
3      exit (EXIT_FAILURE);
4  }
5
6      //TODO insert your code here
7      //acknowledge waiter that order was received
8      if (semUp (semgid, sh->orderReceived) == -1)
9      {
10         perror ("error on the up operation for semaphore access (PT)");
11         exit (EXIT_FAILURE);
12     }
13     //end of TODO

```

processOrder()

Esta função trata do processamento do pedido que foi aceite pelo chef.

Inicialmente, é simulado algum tempo para a chef cozinhar. Em seguida, decrementamos (semDown) o semáforo waiterRequestPossible, para que o waiter saiba que a comida está pronta.

```
1  usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));
2
3  //TODO insert your code here
4  //signal waiter that food is ready (this may only happen when waiter is available)
5  if (semDown (semgid, sh->waiterRequestPossible) == -1)
6  {
7      perror ("error on the up operation for semaphore access (PT)");
8      exit (EXIT_FAILURE);
9  }
10 //end of TODO
```

Após decrementar (semDown) o semáforo mutex, ou seja, entrando na região crítica da memória, atualizamos o request type do waiter para FOODREADY e atualizamos o grupo para lastGroup, ou seja, o grupo que irá receber a comida.

```
1  if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
2      perror ("error on the up operation for semaphore access (PT)");
3      exit (EXIT_FAILURE);
4  }
5
6  //TODO insert your code here
7  //update waiter request
8  sh->fSt.waiterRequest.reqType = FOODREADY; //food is ready
9  sh->fSt.waiterRequest.reqGroup = lastGroup; //group that requested food
```

Em seguida, atualizamos o estado do chef para WAIT_FOR_ORDER para que possa receber outro pedido e guardamos o devido estado. Finalmente, incrementamos o semáforo waiterRequest para sinalizar ao waiter.

```
1 //internal update of chef state
2 sh->fSt.st.chefStat = WAIT_FOR_ORDER; //chef is waiting again
3 saveState(nFic, &sh->fSt);
4 //end of TODO
5
6 if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
7     perror ("error on the up operation for semaphore access (PT)");
8     exit (EXIT_FAILURE);
9 }
10
11 //TODO insert your code here
12 //signal waiter that food is ready
13 if (semUp (semgid, sh->waiterRequest) == -1)
14 {
15     perror ("error on the up operation for semaphore access (PT)");
16     exit (EXIT_FAILURE);
17 }
18 // end of TODO
```

Receptionist

O receptionist trata da atribuição das mesas e de receber os pagamentos.

Serão analisadas as cinco funções completadas que permitem o funcionamento deste processo: decideTableOrWait(), decideNextGroup(), waitForGroup(), provideTableOrWaitingRoom() e receivePayment().

decideTableOrWait()

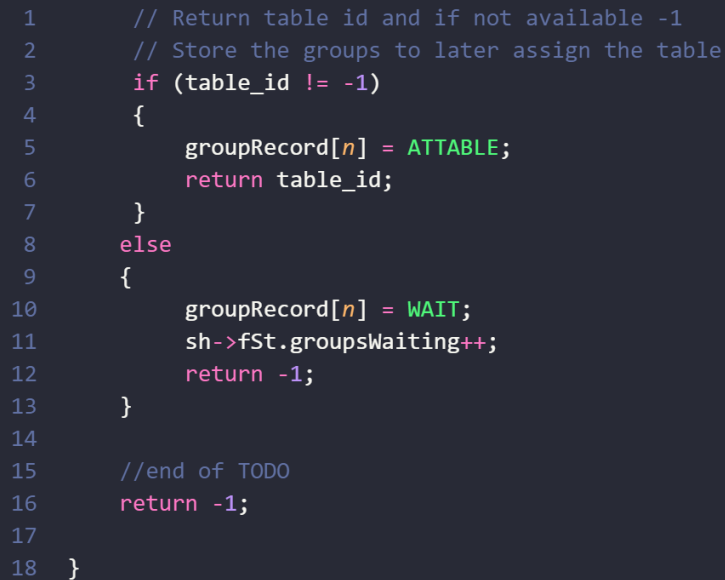
Nesta função, associamos uma mesa, caso possível, a um grupo n. Caso não seja possível, fica na lista de espera.

Começamos por assumir uma mesa disponível, no caso a mesa 0, e iniciamos a variável `table_occupied` a zero. No ciclo `for`, percorremos todos os grupos para conseguirmos saber se existe alguma mesa disponível. Dentro do ciclo é verificado se o grupo está na mesa 0 ou na mesa 1. Caso esteja na mesa 0, na primeira iteração, o `table_id` será agora 1 (pois é a mesa atualmente disponível, neste ciclo) e incrementamos `table_occupied`.

Apartir de agora, nos próximos ciclos, será verificado se o grupo está na mesa 1 (na mesa 0 também, mas nunca estará nenhum grupo), caso esteja, como uma mesa já foi ocupada, o `table_id` será -1, caso contrário, o `for` loop irá continuar até encontrar um grupo que esteja na mesa 1 e atualizar o `table_id` para -1. Se nenhum grupo estiver na mesa 1, o `table_id` continuará 1.

```
1 static int decideTableOrWait(int n) // Associates a table to a group, if it's able to
2 {
3     //TODO insert your code here
4
5     int table_id = 0;
6     int table_occupied = 0;
7
8     // This is for the assignments of the tables
9     for (int i = 0; i < MAXGROUPS; i++) {
10        // Checking for free tables
11
12        if(sh->fSt.assignedTable[i] == 0) { // If table 0 is occupied
13
14            if (table_occupied == 0) { // If we have not found an occupied table yet
15
16                table_id = 1; // Assign table 1
17                table_occupied++;
18            }
19            else
20                // Wait decision
21                table_id = -1;
22        }
23        else if (sh->fSt.assignedTable[i] == 1) { // If table 1 is occupied
24
25            if (table_occupied == 0) { // If we have not found an occupied table yet
26                table_id = 0; // Assign table 0
27                table_occupied++;
28            } else
29                // Wait decision
30                table_id = -1;
31        }
32    }
```

Por fim, se o ID da mesa for diferente de -1, ou seja, se estiver atribuída, o grupo passa a estar no estado `ATTABLE` e o ID da mesa é retornado. Caso contrário, o grupo fica no estado `WAIT` e o número de grupos em espera aumenta, retorna-se o valor -1.



```
1 // Return table id and if not available -1
2 // Store the groups to later assign the table
3 if (table_id != -1)
4 {
5     groupRecord[n] = ATTABLE;
6     return table_id;
7 }
8 else
9 {
10     groupRecord[n] = WAIT;
11     sh->fSt.groupsWaiting++;
12     return -1;
13 }
14
15 //end of TODO
16 return -1;
17
18 }
```

decideNextGroup()

Esta função serve para decidir que grupo deverá ocupar a mesa. Primeiro verifica-se se o número de grupos à espera é diferente de 0. Depois, através de um ciclo for, percorre-se o número máximo de grupos. Caso o grupo atual esteja em espera (WAIT), altera-se o seu estado para ATTABLE e este é removido da lista de grupos em fila. Retorna-se o índice do grupo. Caso nenhuma condição anterior se verifique, retorna-se -1.

```

1  static int decideNextGroup() // Decides which group is next to occupy a table
2  {
3      //TODO insert your code here
4
5      if (sh->fSt.groupsWaiting != 0)
6      {
7          for (int i = 0; i < MAXGROUPS; i++)
8          {
9              if (groupRecord[i] == WAIT)
10             {
11                 // Change group to the table
12                 groupRecord[i] = ATTABLE;
13                 // Remove from waiting list
14                 sh->fSt.groupsWaiting--;
15
16                 return i;
17             }
18         }
19         return -1;
20     }
21     //end of TODO
22     return -1;
23 }

```

waitForGroup()

Nesta fase, o receptionist espera que o grupo faça um pedido. Primeiramente, inicializa-se o pedido do tipo *request* e entra-se na região crítica através do semáforo *mutex*. É, então, atualizado o estado do receptionist para *WAIT_FOR_REQUEST* e é guardado. Sai-se da região crítica.

```

1 static request waitForGroup() // Receptionist waits for request from group
2 {
3     request req;
4
5     if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
6         perror ("error on the up operation for semaphore access (WT)");
7         exit (EXIT_FAILURE);
8     }
9
10    // TODO insert your code here
11
12    // Receptionist waits for request from group
13    sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;
14    saveState(nFic, &sh->fSt);
15
16    // END OF TODO
17
18    if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
19        perror ("error on the down operation for semaphore access (WT)");
20        exit (EXIT_FAILURE);
21    }

```

Entra-se, de novo, na região crítica através do semáforo de espera do receptionist pelo pedido. Faz-se semDown do semáforo mutex, mais uma vez, para o receptionist ler e guardar o pedido. Faz-se semUp do mesmo semáforo. Por fim, o receptionist assinala que é possível receber novos pedidos, saindo da região crítica. Retorna-se o pedido, req.

```

1 // TODO insert your code here
2
3 // Receptionist waits for request from group
4 if (semDown (semgid, sh->receptionistReq) == -1) /* enter critical region*/
5 {
6     perror ("error on the up operation for semaphore access (PT)");
7     exit (EXIT_FAILURE);
8 }
9
10 // END OF TODO
11
12 if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
13     perror ("error on the up operation for semaphore access (WT)");
14     exit (EXIT_FAILURE);
15 }
16
17 // TODO insert your code here
18 // Receptionist reads request
19 req = sh->fSt.receptionistRequest;
20
21 if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
22     perror ("error on the down operation for semaphore access (WT)");
23     exit (EXIT_FAILURE);
24 }
25
26 // TODO insert your code here
27 // Receptionist signals that new requests are possible
28 if (semUp (semgid, sh->receptionistRequestPossible) == -1) /* exit critical region */
29 {
30     perror ("error on the up operation for semaphore access (PT)");
31     exit (EXIT_FAILURE);
32 }
33
34 return req;
35
36 }

```


provideTableOrWaitingRoom()

Agora, o receptionist irá decidir se o grupo pode ocupar a mesa ou deve esperar. Primeiramente, entra-se na região crítica da memória do programa, decrementando o semáforo *mutex*. De seguida, atualiza-se o estado do receptionist para *ASSIGNTABLE*, para que ele possa atribuir mesas aos grupos.

```
1 static void provideTableOrWaitingRoom (int n) // Receptionist decides if group should occupy table or wait
2 {
3     if (semDown (semgid, sh->mutex) == -1) {                               /* enter critical region */
4         perror ("error on the up operation for semaphore access (WT)");
5         exit (EXIT_FAILURE);
6     }
7
8     // TODO insert your code here
9
10    // Change receptionist state so it can assign tables
11    sh->fSt.receptionistStat = ASSIGNTABLE;
12    saveState(nFic, &sh->fSt);
13 }
```

O ID da mesa passa a ser o retornado na função *decideTableOrWait()*. Se este for diferente de -1, a mesa é atribuída ao grupo, caso contrário, este continua em espera. No fim, sai-se da região crítica, incrementando os semáforos *waitForTable* e *mutex*.

```
1 // Check all tables and checks if it's either assigned or put on the waiting list
2 // If there's an available table, it lets the group know
3 int table_id = decideTableOrWait(n);
4 if (table_id != -1)
5 {
6     sh->fSt.assignedTable[n] = table_id;
7
8     if (semUp (semgid, sh->waitForTable[n]) == -1)                       /* exit critical region */
9     {
10         perror ("error on the up operation for semaphore access (PT)");
11         exit (EXIT_FAILURE);
12     }
13 }
14
15 //end of TODO
16
17 if (semUp (semgid, sh->mutex) == -1) {                                   /* exit critical region */
18     perror ("error on the down operation for semaphore access (WT)");
19     exit (EXIT_FAILURE);
20 }
21
22 }
```

receivePayment()

O receptionist irá finalmente receber o pagamento dos clientes. Entramos na região crítica da memória e atualizamos o estado do receptionist para RECVPAY.

```
1 static void receivePayment (int n) // Receptionist receives the payment, checks if there are any groups waiting
2 {                                // and if so places them in a now free table
3
4     if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
5         perror ("error on the up operation for semaphore access (WT)");
6         exit (EXIT_FAILURE);
7     }
8
9     // TODO insert your code here
10    // Receptionist changes state to receive the payment
11    sh->fSt.receptionistStat = RECVPAY;
12    saveState(nFic, &sh->fSt);
13
```

Seguidamente, o receptionist irá verificar se a mesa está livre. O ID da mesa passa a ser o da mesa atribuída e o ID do grupo é o retornado na função decideNextGroup().

Se este for diferente de -1, atribui-se a mesa ao próximo grupo e o grupo atual encontra-se como DONE, ou seja, “terminado” e a mesa deixa de estar atribuída a este (linha 18).

Se a mesa ainda estiver ocupada, o próximo grupo terá de esperar.

```
1 // Check if the table is free
2 int table_id = sh->fSt.assignedTable[n];
3 int group_id = decideNextGroup();
4
5 if (group_id != -1)
6 {
7     // Next group gets the table that's available
8     sh->fSt.assignedTable[group_id] = table_id;
9     groupRecord[n] = DONE;
10
11     if (semUp (semgid, sh->waitForTable[group_id]) == -1)
12     {
13         perror ("error on the up operation for semaphore access (PT)");
14         exit (EXIT_FAILURE);
15     }
16 }
17
18 sh->fSt.assignedTable[n] = -1;
19 //end of TODO
```

Para terminar, incrementamos os semáforos *mutex* e *tableDone*, de modo a esperar que o pagamento seja concluído e sair da região crítica.

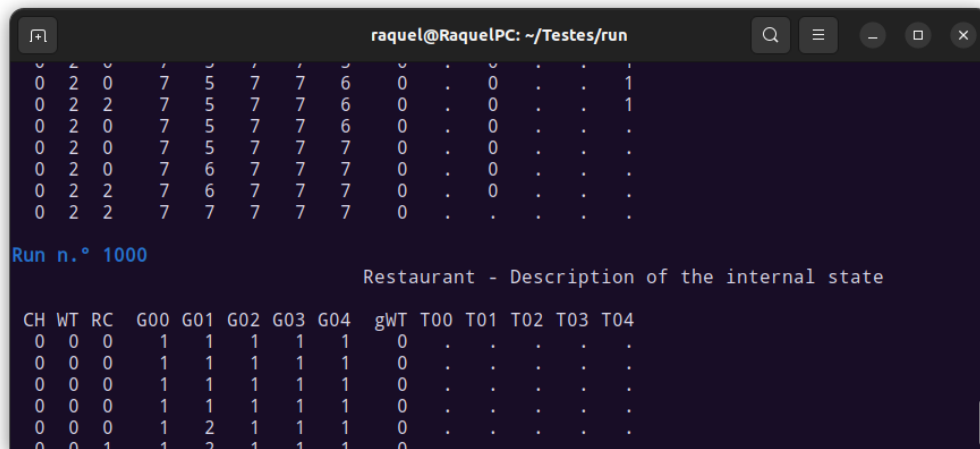
```
1  if (semUp (semgid, sh->mutex) == -1) {                                /* exit critical region */
2      perror ("error on the down operation for semaphore access (WT)");
3      exit (EXIT_FAILURE);
4  }
5
6  // TODO insert your code here
7  // Let the groups know that they are waiting for payment
8  if (semUp (semgid, sh->tableDone[table_id]) == -1)                    /* exit critical region */
9  {
10     perror ("error on the up operation for semaphore access (PT)");
11     exit (EXIT_FAILURE);
12 }
13
14 //end of TODO
15 }
```

Testes e validação de resultados

Como as soluções obtidas são aleatórias, a validação das mesmas baseou-se na verificação da cronologia dos processos e nas transições dos estados, mediante o que foi estabelecido nas funções.

Foi executado o ficheiro *probSemSharedMemRestaurant*, e analisadas com detalhe as transições de estados. Como todas correspondiam ao pretendido, concluiu-se que o programa passava ao teste.

Foi executado, ainda, o ficheiro *run.sh*, que corria o teste anterior 1000 vezes. Como este executou todas até ao fim, concluímos que também foi um sucesso.



```
raquel@RaquelPC: ~/Testes/run
0 2 0 7 5 7 7 6 0 . 0 . . 1
0 2 2 7 5 7 7 6 0 . 0 . . 1
0 2 0 7 5 7 7 6 0 . 0 . . .
0 2 0 7 5 7 7 7 0 . 0 . . .
0 2 0 7 6 7 7 7 0 . 0 . . .
0 2 2 7 6 7 7 7 0 . 0 . . .
0 2 2 7 7 7 7 7 0 . . . . .

Run n.º 1000

Restaurant - Description of the internal state

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

Conclusão

A realização deste trabalho permitiu-nos desenvolver e aperfeiçoar conhecimentos sobre sincronização de processos e threads. Foram realizados múltiplos testes, de modo a garantir o bom funcionamento do código, verificando a cronologia dos ciclos de vida de todas as entidades envolvidas.