universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Sistemas Operativos

Trabalho Prático 2

Jantar de Amigos (Restaurant)

Professor:

Nuno Lau (*nunolau@ua.pt*)

Realizado por:

Diogo Falcão, 108712, P3

José Gameiro, 108840, P3

02/01/2023

Índice

1. Introdução

No âmbito da Unidade Curricular de Sistemas Operativos, foi-nos proposto realizar um trabalho prático, que consiste em simular um jantar de amigos num restaurante envolvendo três entidades: clients (clientes), waiter (empregado/a) e chef (chefe). Todas estas entidades são processos independentes, sendo que a sua sincronização e comunicação é efetuada através de vários semáforos e de memória partilhada.

A simulação começa com a chegada de todos os amigos ao restaurante. Nota-se que o primeiro amigo a chegar será o que irá fazer o pedido da comida, no entanto, só o poderá fazer quando todos os amigos tiverem chegado. O último amigo a chegar será o que irá pagar a conta e só poderá pedi-la quando todos os amigos tiverem terminado a sua refeição. De modo a evitar situações em que dois ou mais processos sejam bloqueados (visto que estão à espera de um evento que apenas pode ser despoletado por um dos processos em bloqueio, ou seja, uma situação de deadlock), usamos semáforos.

A utilização de semáforos serve essencialmente o controlo de acesso à memória partilhada, de modo a evitar choques entre as três entidades que participam na simulação. As notificações entre entidades são feitas através desses mesmos semáforos para que o programa execute sem problemas.

Com a realização deste trabalho prático, esperamos conseguir cumprir todos os pontos essenciais que são propostos no guião e alargar os nossos conhecimentos relativamente a programar com semáforos em C, visto ser um aspeto importante no que toca a controlar o acesso a determinadas regiões por parte de vários processos.

2. Material Fornecido

É disponibilizado um conjunto de ficheiros para a resolução do problema, em que está presente uma pasta src com ficheiros incompletos com código em C para a simulação do jantar (semSharedMemChef.c, semSharedMemClient.c e semSharedMemWaiter.c). Ao longo deste relatório iremos explicar como é que completámos as funções que se encontravam incompletas em cada um destes ficheiros.

Existem também outros ficheiros que contém dados fulcrais à simulação do jantar:

- probConst.h: este ficheiro contém variáveis que irão ser utilizadas nos 3 ficheiros incompletos. É caso de: estados dos client, estados do waiter, estados do chef e outras como o tamanho máximo da mesa e os tempos máximos para comer ou cozinhar.

```
/* Client state constants */

/** \brief client initial state */
#define  INIT              1
/** \brief client is waiting for friends to arrive at table */
#define  WAIT_FOR_FRIENDS  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 waiting for others to finish */
#define  WAIT_FOR_OTHERS   6
/** \brief client is waiting to complete payment */
#define  WAIT_FOR_BILL     7
/** \brief client finished meal */
#define  FINISHED          8


/* 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
/** \brief waiter reiceives payment */
#define  RECEIVE_PAYMENT    3
```

```
/* 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



/* Generic parameters */

/** \brief table capacity, equal to number of clients */
#define  TABLESIZE        20
/** \brief controls time taken to eat */
#define  MAXEAT       500000
/** \brief controls time taken to cook */
#define  MAXCOOK     3000000
```

Fig.1 – Constantes definidas no ficheiro probConst.h.

- probDataStruct.h: neste ficheiro estão definidas duas estruturas que são a estrutura STAT e a FULL_STAT. A estrutura STAT tem como parâmetros 3 variáveis inteiras, que são os estados de cada entidade envolvente no jantar - e como existem 20 clientes, a variável para o estado de cada cliente será um array com o tamanho de 20 (TABLESIZE). A estrutura FULL_STAT apresenta três tipos de parâmetros (que podemos observar na figura dois): nas linhas 43 e 45 funcionam como contadores para que sempre que um cliente chegue ao restaurante ou sempre que um cliente acaba de comer, estas variáveis incrementam; as variáveis nas linhas 48, 50, 52 e 54 irão funcionar com sinais, ou seja, o seu valor inicial é zero e quando for necessário estas

variáveis serão postas a um; por fim, as variáveis tableFirst e tableLast irão guardar os ID's do primeiro e último cliente a chegar, respetivamente.

```
20
21  /**
22   *  \brief Definition of <em>state of the intervening entities</em> data type.
23   */
24  typedef struct {
25      /** \brief waiter state */
26      unsigned int waiterStat;
27      /** \brief chef state */
28      unsigned int chefStat;
29      /** \brief client state array */
30      unsigned int clientStat[TABLESIZE];
31
32  } STAT;
34
35  /**
36   *  \brief Definition of <em>full state of the problem</em> data type.
37   */
38  typedef struct
39  {   /** \brief state of all intervening entities */
40      STAT st;
41
42      /** \brief number of clients at table */
43      int tableClients;
44      /** \brief number of clients that finished eating */
45      int tableFinishEat;
46
47      /** \brief flag of food request from client to waiter */
48      int foodRequest;
49      /** \brief flag of food order from waiter to chef */
50      int foodOrder;
51      /** \brief flag of food ready from chef to waiter */
52      int foodReady;
53      /** \brief flag of payment request from client to waiter */
54      int paymentRequest;
55
56      /** \brief id of first client to arrive */
57      int tableFirst;
58      /** \brief id of last client to arrive */
59      int tableLast;
60
61  } FULL_STAT;
```

Fig.2 –Estruturas definidas no ficheiro probDataStruct.h.

- sharedDataSync.h: este ficheiro compreende a estrutura SHARED_DATA, que inclui como parâmetros uma estrutura do tipo FULL_STAT (fSt) e os semáforos que irão ser utilizados para evitar situações de deadlock. Estes são:

  - mutex (valor inicial um) – este semáforo é usado para identificar a entrada na região crítica e a saída da mesma. Uma região crítica é uma zona de código que manipula dados partilhados e que não pode ser executada concorrentemente por mais do que um processo;

- friendsArrived (valor inicial zero) – identifica o semáforo usado pelos clientes para esperarem que os amigos cheguem;
- requestReceived (valor inicial zero) – este semáforo é usado pelos clientes para esperar pelo empregado depois de este ter feito um pedido;
- foodArrived (valor inicial zero) – identifica o semáforo usado pelos clientes para esperarem que a comida chegue;
- allFinished (valor inicial zero) – este semáforo é usado pelos clientes para esperarem que cada um termine a sua refeição;
- waiterRequest (valor inicial zero) – identifica o semáforo usado pelo empregado para esperar por um pedido vindo ou de um cliente ou do chefe;
- waitOrder (valor inicial zero) – este semáforo é usado pelo chefe para esperar por um pedido.

```
26  typedef struct
27          { /** \brief full state of the problem */
28            FULL_STAT fSt;
29
30            /* semaphores ids */
31            /** \brief identification of critical region protection semaphore - val = 1 */
32            unsigned int mutex;
33            /** \brief identification of semaphore used by clients to wait for friends to arrive - val = 0 */
34            unsigned int friendsArrived;
35            /** \brief identification of semaphore used by client to wait for waiter after a request - val = 0 */
36            unsigned int requestReceived;
37            /** \brief identification of semaphore used by clients to wait for food - val = 0 */
38            unsigned int foodArrived;
39            /** \brief identification of semaphore used by clients to wait for friends to finish eating - val = 0  */
40            unsigned int allFinished;
41            /** \brief identification of semaphore used by waiter to wait for requests - val = 0  */
42            unsigned int waiterRequest;
43            /** \brief identification of semaphore used by chef to wait for order - val = 0  */
44            unsigned int waitOrder;
45
46          } SHARED_DATA;
```

Fig.3 –Estrutura definida no ficheiro sharedDataSync.h.

- probSemSharedMemRestaurant.c: Neste ficheiro é criada a memória partilhada, são inicializados os estados de cada uma das entidades que participam na simulação, as variáveis da estrutura FULL_STAT e os semáforos, são gerados também os processos clientes, empregado e chefe. Existem também outras operações que são realizadas neste ficheiro que irão ser importantes para a simulação. No fim de ser executado tudo, todos os semáforos e a memória partilhada são destruídos.

```
83      /* creating and initializing the shared memory region and the log file */
84      if ((shmid = shmemCreate (key, sizeof (SHARED_DATA))) == -1) {
85          perror ("error on creating the shared memory region");
86          exit (EXIT_FAILURE);
87      }
88      if (shmemAttach (shmid, (void **) &sh) == -1) {
89          perror ("error on mapping the shared region on the process address space");
90          exit (EXIT_FAILURE);
91      }

96      /* initialize problem internal status */
97      sh->fSt.st.chefStat   = WAIT_FOR_ORDER;                    /* the chef waits for an order */
98      sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;                /* the waiter waits for a request */
99      for (c = 0; c < TABLESIZE; c++) {
100         sh->fSt.st.clientStat[c] = INIT;                            /* clients are initialized */
101     }
102     sh->fSt.tableClients = 0;
103     sh->fSt.tableFinishEat = 0;
104
105     sh->fSt.foodRequest = 0;
106     sh->fSt.foodOrder = 0;
107     sh->fSt.foodReady = 0;
108     sh->fSt.paymentRequest = 0;
109
110     sh->fSt.tableLast = -1;

150     /* waiter process */
151     strcpy (nFicErr + 6, "WT");
152     if ((pidWT = fork ()) < 0)  {
153         perror ("error on the fork operation for the waiter");
154         exit (EXIT_FAILURE);
155     }
156     if (pidWT == 0) {
157         if (execl (WAITER, WAITER, nFic, num[1], nFicErr, NULL) < 0) {
158             perror ("error on the generation of the waiter process");
159             exit (EXIT_FAILURE);
160         }
161     }
162     /* chef process */
163     strcpy (nFicErr + 6, "CH");
164     if ((pidCH = fork ()) < 0) {
165         perror ("error on the fork operation for the chef");
166         exit (EXIT_FAILURE);
167     }
168     if (pidCH == 0)
169         if (execl (CHEF, CHEF, nFic, num[1], nFicErr, NULL) < 0) {
170             perror ("error on the generation of the chef process");
171             exit (EXIT_FAILURE);
172         }
173

191     /* destruction of semaphore set and shared region */
192     if (semDestroy (semgid) == -1) {
193         perror ("error on destructing the semaphore set");
194         exit (EXIT_FAILURE);
195     }
196     if (shmemDettach (sh) == -1) {
197         perror ("error on unmapping the shared region off the process address space");
198         exit (EXIT_FAILURE);
199     }
200     if (shmemDestroy (shmid) == -1) {
201         perror ("error on destructing the shared region");
202         exit (EXIT_FAILURE);
```

Fig.4 – Algum código presente no ficheiro probSemSharedMemRestaurant.c

3. Desenvolvimento

Nesta parte do relatório iremos explicar o código que implementámos nos ficheiros semSharedMemClient.c, semSharedMemWaiter.c e semSharedMemChef.c, as ideias que tivemos para evitar situações de deadlock e mostrar também os ciclos de vida de cada uma das entidades.

3.1. Client

Começamos pelo client visto que a primeira ação que ocorre na simulação é a chegada de todos os clientes ao restaurante e o ciclo de vida do cliente é o maior de entre as 3 entidades envolventes na simulação, como podemos observar na figura 5.

```
107     /* simulation of the life cycle of the client */
108     travel(n);
109     bool first = waitFriends(n);
110     if (first) orderFood(n);
111     waitFood(n);
112     eat(n);
113     waitAndPay(n);
```

Fig.5 – Ciclo de vida da entidade client.

Ao analisar a figura 5 podemos concluir que a primeira função que o cliente executa é travel (), que consiste em fazer o client viajar de um determinado lugar até ao restaurante, esta função é a única, de entre todas, que se encontra completa.

A segunda função, waitFriends (), é usada para fazer com que o cliente avance para a próxima função apenas quando todos os clientes chegarem, através de um booleano. Se todos tiverem chegado o client avança para a próxima função, orderFood (), em que irá se efetuado o pedido da comida.

Depois de se ter feito o pedido, os clientes esperam que a comida chegue, através da função waitFood (). Ao chegar a comida, todos começam a comer, com a função eat (). Por último, todos os clientes terão de esperar que todos à mesa terminem para que se possa pedir a conta e acabar o jantar - é o que acontece na função waitAndPay (). Desta forma, termina o ciclo de vida do client.

3.1.1. waitFriends ()

Tal como foi referido em cima, esta função tem como objetivo fazer com que os clientes esperem que a mesa fique completa, ou seja, que todos os clientes cheguem para depois se puder fazer o pedido da comida. O pedido terá de ser feito pelo primeiro client e, por isso, é necessário guardar o estado de cada um dos clientes pois este irá alterar para todos.

```
148  /**
149   *  \brief client waits until table is complete
150   *
151   *  Client should udpate state, first and last clients should register their values in shared data,
152   *  last client should, in addition, inform the others that the table is complete.
153   *  Client must wait in this function until the table is complete.
154   *  The internal state should be saved.
155   *
156   *  \param id client id
157   *
158   *  \return true if first client, false otherwise
159   */
160  static bool waitFriends(int id)
161  {
162      bool first = false;
163
164      if (semDown (semgid, sh->mutex) == -1) {                                        /* enter critical region */
165          perror ("error on the down operation for semaphore access (CT)");
166          exit (EXIT_FAILURE);
167      }
168
169      /* insert your code here */
170      sh->fSt.tableClients++;  // Incrementa o número de clientes que chegaram à mesa, cada vez que entrar um cliente
171      sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;   // muda o estado para WAIT_FOR_FRIENDS
172
173      // ver se é a primeira pessoa a chegar
174      if(sh->fSt.tableClients == 1)
175      {
176          first = true;
177          sh->fSt.tableFirst = id;  // Guarda o id do primeiro cliente que chegou
178      }
```

Fig.6a – Primeira parte da função waitFriends.

Em termos de código, esta função irá ser utilizada por todos os clientes e começa com a definição de um booleano designado por "first". Ao chegar um cliente, este entra na região crítica, é incrementado o número de clientes que se encontram na mesa e o estado deste muda para WAIT_FOR_FRIENDS. De seguida, através da condição presente na linha 174, é verificado se o cliente que entrou na região crítica é o primeiro e, se for, a variável first é alterada para true. Por isto, é guardado o id do primeiro cliente que chegou à mesa dado que vai ser este que irá efetuar o pedido da comida.

```
180      // ver se é a última pessoa a chegar
181      if(sh->fSt.tableClients == TABLESIZE)
182      {
183          sh->fSt.tableLast = id;    // Guarda o id do último cliente que chegou
184      }
185
186      saveState(nFic,&(sh->fSt));
187
188      if (semUp (semgid, sh->mutex) == -1)                                    /* exit critical region */
189      {
190          perror ("error on the up operation for semaphore access (CT)");
191          exit(EXIT_FAILURE);
192      }
193
194      /* insert your code here */
195      if (id != sh->fSt.tableLast) // Se não for o último cliente
196      {
197          semDown(semgid, sh->friendsArrived); // Adormecer os clientes
198      }
199
200      if (id == sh->fSt.tableLast) // Se for o último cliente
201      {
202          for (int i = 0 ; i < TABLESIZE - 1 ; i++)
203              semUp(semgid, sh->friendsArrived);  // Desbloquear os clientes que já chegaram
204      }
205
206      return first;
207  }
```

Fig.6b – Segunda parte da função waitFriends.

Se não se verificar a condição da verificação do primeiro cliente, avança-se para a próxima condição, em que verifica se o cliente que está na região crítica é o último cliente, através da linha 181. Se for efetivamente verdade, é guardado o ID do cliente em questão, isto porque o último cliente a chegar irá efetuar o pedido da conta e o pagamento da mesma.

Simultaneamente, o estado de cada cliente e as variáveis que foram alteradas serão guardadas através da função saveState (). Guardado o estado do cliente que está a percorrer a função e saindo-se da região crítica, é verificado, desta vez fora da região crítica, se o seu ID não é último. Se não for, o cliente é adormecido com um semDown efetuado no semáforo friendsArrived. Este não pode avançar para a função seguinte até todos os outros clientes chegarem. Se a condição da linha 195 for falsa avança para a próxima condição, em que se é verificado uma última vez se o ID do cliente é o último. Caso seja, os clientes que foram adormecidos são acordados com um semUp no semáforo friendsArrived (efetuado 19 vezes, isto porque é o número de clientes que já chegaram). Assim esta função termina e retorna como verdadeiro o booleano "first", permitindo avançar-se para a próxima função.

3.1.2. orderFood ()

Esta função é utilizada apenas pelo primeiro cliente e tem como objetivo efetuar o pedido da comida. Será novamente necessário guardar o estado do primeiro cliente pois este irá ser alterado.

```
209 /**
210  *  \brief first client orders food.
211  *
212  *  This function is used only by the first client.
213  *  The first client should update its state, request food to the waiter and
214  *  wait for the waiter to receive the request.
215  *
216  *  The internal state should be saved.
217  *
218  *  \param id client id
219  */
220 static void orderFood (int id)
221 {
222     if (semDown (semgid, sh->mutex) == -1)              /* enter critical region */
223     {
224         perror ("error on the down operation for semaphore access (CT)");
225         exit (EXIT_FAILURE);
226     }
227
228     /* insert your code here */
229     if (id == sh->fSt.tableFirst)  // Verifica se é o id do primeiro cliente
230     {
231         sh->fSt.st.clientStat[id] = FOOD_REQUEST;  // Update the state of the client to Food_Request
232         sh->fSt.foodRequest++;   // Adiciona um pedido
233
234         semUp(semgid, sh->waiterRequest); // Waiter receives the request (wakes up the waiter)
235     }
236
237     saveState(nFic,&(sh->fSt));
238
239     if (semUp (semgid, sh->mutex) == -1)                                        /* exit critical region */
240     {
241         perror ("error on the up operation for semaphore access (CT)");
242         exit (EXIT_FAILURE);
243     }
244
245     /* insert your code here */
246     // semDown(semgid, sh->waiterRequest);
247
248 }
```

Fig.7 – Função orderFood.

Nesta função o cliente começa por entrar na região crítica, onde se é verificado se o ID do cliente é o primeiro. Nesta hipótese, o estado do cliente é alterado para FOOD_REQUEST, é também incrementado o número de pedidos e o empregado é acordado com um semUp no semáforo waiterRequest. No final é guardado o estado interno com a função saveState.

Nas primeiras tentativas de resolução do problema, inserimos depois da região crítica um semDown no semáforo waiterRequest, no entanto, concluímos que se iria combinar uma situação de deadlock, por isso tirámos estas linhas de código.

3.1.3. waitFood ()

Nesta função, com o pedido efetuado, todos os clientes terão de esperar que o empregado traga a comida para a mesa para depois começarem a comer. Os estados dos clientes terão de ser guardados novamente, pois, ao contrário das funções retratadas até aqui, estes terão de ser alterados duas vezes.

```
250 /**
251  *  \brief client waits for food.
252  *
253  *  The client updates its state, and waits until food arrives.
254  *  It should also update state after food arrives.
255  *  The internal state should be saved twice.
256  *
257  */
258 static void waitFood (int id)
259 {
260
261     if (semDown (semgid, sh->mutex) == -1) {                                        /* enter critical region */
262         perror ("error on the down operation for semaphore access (CT)");
263         exit (EXIT_FAILURE);
264     }
265
266     /* insert your code here */
267     sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;  // Update the state of the client to Wait_for_food
268     saveState(nFic,&(sh->fSt));
269
270     if (semUp (semgid, sh->mutex) == -1) {                                      /* exit critical region */
271         perror ("error on the down operation for semaphore access (CT)");
272         exit (EXIT_FAILURE);
273     }
274
275     /* insert your code here */
276     if (semDown(semgid, sh->foodArrived) == -1) // Adormecer os clientes
277     {
278         perror ("error on the down operation for semaphore access (CT)");
279         exit (EXIT_FAILURE);
280     }
281
```

Fig.8a – Primeira parte da função waitFood.

Esta função apresenta duas regiões críticas, visto que o estado de todos os clientes é alterado duas vezes. Na primeira região crítica, o cliente altera o seu estado para WAIT_FOR_FOOD e guarda-o com a função saveState. Ao sair desta região crítica, todos os clientes serão postos a dormir com a execução de um semDown no semáforo foodArrived e só serão acordados quando o empregado colocar a comida na mesa, na função takeFoodToTable () que será explicada mais à frente.

```
282
283     if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
284         perror ("error on the down operation for semaphore access (CT)");
285         exit (EXIT_FAILURE);
286     }
287
288     /* insert your code here */
289     sh->fSt.st.clientStat[id] = EAT;  // Update the state of the client to EAT
290     saveState(nFic,&(sh->fSt));
291
292     if (semUp (semgid, sh->mutex) == -1) {                                      /* exit critical region */
293         perror ("error on the down operation for semaphore access (CT)");
294         exit (EXIT_FAILURE);
295     }
296 }
```

Fig.8b – Segunda parte da função waitFood.

Continuando na função, os clientes são acordados pelo empregado na função takeFoodToTable () e entram na segunda região crítica, em que o estado do cliente é alterado para EAT e, consequentemente, guardado com a função saveState.

3.1.4. waitAndPay ()

Nesta função os clientes que já acabaram de comer terão que esperar que os outros acabem também de comer, visto que só se poderá pedir a conta e efetuar o pagamento no fim de todos terminarem. Uma parte da função só será utilizada pelo último cliente que chegou ao restaurante, o mesmo que pede e efetua o pagamento da conta. Nesta função o estado de todos os clientes é alterado duas vezes à exceção do último cliente que chegou ao restaurante, que será alterado três vezes.

```
298 /**
299  *  \brief client waits for others to finish meal, last client to arrive pays the bill.
300  *
301  *  The client updates state and waits for others to finish meal before leaving and update its state.
302  *  Last client to finish meal should inform others that everybody finished.
303  *  Last client to arrive at table should pay the bill by contacting waiter and waiting for waiter to arrive.
304  *  The internal state should be saved twice.
305  *
306  *  \param id client id
307  */
308 static void waitAndPay (int id)
309 {
310     bool last=false;
311
312     if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
313         perror ("error on the down operation for semaphore access (CT)");
314         exit (EXIT_FAILURE);
315     }
316
317     /* insert your code here */
318     if (sh->fSt.tableLast == id)  // Verifica se todos os clientes terminaram de comer
319     {
320         last = true;
321     }
322
323     sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
324     sh->fSt.tableFinishEat++;  // Incrementa o número de clientes que terminaram de comer
325     saveState(nFic,&(sh->fSt));
326
327     if(sh->fSt.tableFinishEat == TABLESIZE)
328     {
329         for (int j = 0 ; j < TABLESIZE ; j++)
330             semUp(semgid, sh->allFinished);  // Acordar os clientes que estão a dormir
331     }
332
333
334     if (semUp (semgid, sh->mutex) == -1) {                                      /* exit critical region */
335         perror ("error on the down operation for semaphore access (CT)");
336         exit (EXIT_FAILURE);
337     }
```

Fig.9a – Segunda parte da função waitAndPay.

Esta função começa com a definição de um booleano last, com valor inicial de false. Um cliente entra na primeira região crítica da função onde encontra uma condição que verifica se o ID é o do último cliente que chegou ao restaurante (isto porque assim conseguimos verificar que todos os clientes já acabaram de comer). Se for verdade, altera o booleano last para true e se não for verdade, altera o estado do cliente em questão para WAIT_FOR_OTHERS, incrementa o número de clientes que já terminaram a sua refeição e é guardado o estado interno de cada cliente com a função saveState.

O último passo que se encontra dentro da região crítica consiste em acordar os clientes que já acabaram de comer através de um "for", para estes poderem continuarem o ciclo de execução do programa, com um semUp no semáforo allFinished 20 vezes.

```
338
339     /* insert your code here */
340     semDown(semgid, sh->allFinished);  // Adormecer os clientes que não são o último
341
342     if(last) {
343         if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
344            perror ("error on the down operation for semaphore access (CT)");
345            exit (EXIT_FAILURE);\
346         }
347
348         /* insert your code here */
349         semUp(semgid, sh->waiterRequest);  // Acordar o waiter
350
351         sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
352         saveState(nFic,&(sh->fSt));
353         sh->fSt.paymentRequest = 1;
354
355         if (semUp (semgid, sh->mutex) == -1) {                                  /* exit critical region */
356             perror ("error on the down operation for semaphore access (CT)");
357             exit (EXIT_FAILURE);
358         }
359
360         /* insert your code here */
361         semDown(semgid, sh->requestReceived);
362
363     }
364
365     if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
366         perror ("error on the down operation for semaphore access (CT)");
367         exit (EXIT_FAILURE);
368     }
```

Fig.9b – Segunda parte da função waitAndPay.

Ao sair-se da região crítica e de modo a trabalhar apenas no último cliente, adormecem-se todos os clientes exceto o último com um semDown no semáforo allFinished. Desta forma, o cliente que colocou o booleano last verdadeiro, entra numa nova região crítica. Aqui, o empregado é acordado com um semUp no semáforo waiterRequest, para que este possa entregar a conta e receber o pagamento. O estado do último cliente é alterado para WAIT_FOR_BILL, o número de pedidos da conta é incrementado e o estado interno deste cliente é guardado com a função saveState. Ao sair da região crítica é feito um semDown no semáforo waiterRequest para adormecer o empregado pois já foi efetuado o pagamento e já não necessitamos mais dele.

```
364
365     if (semDown (semgid, sh->mutex) == -1) {                    /* enter critical region */
366         perror ("error on the down operation for semaphore access (CT)");
367         exit (EXIT_FAILURE);
368     }
369
370     /* insert your code here */
371     sh->fSt.st.clientStat[id] = FINISHED;
372     saveState(nFic,&(sh->fSt));
373
374     if (semUp (semgid, sh->mutex) == -1) {                      /* exit critical region */
375         perror ("error on the down operation for semaphore access (CT)");
376         exit (EXIT_FAILURE);
377     }
378 }
379
```

Fig.9c – Terceira parte da função waitAndPay.

Na última região crítica da função o estado de cada cliente é alterado para FINISHED, guardando-se com a função saveState. Isto conclui a simulação do jantar e o ciclo de vida do cliente.

3.2. Waiter

O Waiter é a segunda mais importante entidade de três. Esta estabelece a relação entre as entidades Client e Chef. O ciclo de vida do Waiter provém de dois inteiros "req" e nReq", variáveis que controlam um switch case que alterna entre as funções informChef (), takeFoodToTable () e receivePayment (), como é possível observar na figura 10. Isso posto, concluímos que o Waiter não tem um ciclo de vida com ordem estipulada e por isso intervém no programa maioritariamente por ordens de terceiros (Client e Chef), através da função waitForClientOrChef.

```
109     /* simulation of the life cycle of the waiter */
110     int req, nReq=0;
111     while(nReq<3) {
112         req = waitForClientOrChef();
113         switch(req) {
114             case FOODREQ:
115                 informChef();
116                 break;
117             case FOODREADY:
118                 takeFoodToTable();
119                 break;
120             case BILL:
121                 receivePayment();
122                 break;
123         }
124         nReq++;
125     }
```

Fig.10 – Ciclo de vida da entidade Client.

3.2.1. waitForClientOrChef ()

Nesta primeira função do Waiter, tem-se em conta a variável "ret" que indicará depois (através do retorno da função) a próxima função deste empregado. A variável, por deformidade, tem o valor 0.

Numa primeira parte, o Waiter entra numa região crítica onde espera por ordens ou do Chef ou do Client, alterando-se o seu estado para WAIT_FOR_REQUEST, salvando-se depois este com a função saveState (). Depois, e já fora desta, faz-se um semDown no waiterRequest, "adormecendo-o", fazendo-o esperar por requests.

```
144  static int waitForClientOrChef()
145  {
146      int ret=0;
147
148      if (semDown (semgid, sh->mutex) == -1)  {                       /* enter critical region */
149          perror ("error on the up operation for semaphore access (WT)");
150          exit (EXIT_FAILURE);
151      }
152
153      /* insert your code here */
154      sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
155      saveState(nFic, &(sh->fSt));
156
157      if (semUp (semgid, sh->mutex) == -1)        /* exit critical region */
158      {
159          perror ("error on the down operation for semaphore access (WT)");
160          exit (EXIT_FAILURE);
161      }
162
163      /* insert your code here */
164      if( semDown(semgid, sh->waiterRequest) == -1)  // wait for request from client or chef
165      {
166          perror ("error on the down operation for semaphore access (WT)");
167          exit (EXIT_FAILURE);
168      }
```

Fig.11a – Primeira parte da função waitForClientOrChef ().

Numa segunda parte desta função, e já numa segunda região crítica, verificam-se o valor de três flags possíveis:

- foodRequest, flag do Client para o Waiter para indicar que a mesa está pronta a pedir;
- foodReady, flag do Chef para o Waiter para indicar que o pedido está confecionado;
- paymentRequest, flag do Client para o Waiter para indicar que a toda a mesa já acabou de comer e que estão prontos para pagar e sair.

Cada vez que o Waiter se encontra dentro desta região crítica, espera que uma destas flags seja levantada para poder aceitar pedidos de clientes ou do Chef. Caso isso aconteça, a variável "ret" passa a valer FOODREQ (1), FOODREADY (2) ou BILL (3) consoante a flag levantada. Desta forma no final da função (e já fora da região crítica), retorna-se esta variável permitindo dar continuação ao "switch case".

Nota: para um correto funcionamento do "switch case" e novos requests, após o Waiter identificar a flag que foi levantada, colocámos de volta os valores dessas flags aos valores iniciais – "0".

```
170     if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
171         perror ("error on the up operation for semaphore access (WT)");
172         exit (EXIT_FAILURE);
173     }
174
175     /* insert your code here */
176     if(sh->fSt.foodRequest == 1)   // If the flag foodRequest has been raised (changed from 0 to 1))
177         ret = FOODREQ;
178     else if(sh->fSt.foodReady == 1) //  If the flag foodReady has been raised (changed from 0 to 1))
179         ret = FOODREADY;
180     else if(sh->fSt.paymentRequest == 1) // If the flag paymentRequest has been raised (changed from 0 to 1))
181         ret = BILL;
182
183     // mete tudo a zero -> já leu o pedido, já entregou a comida e já fez tudo - pois é ciclo while
184     sh->fSt.paymentRequest = 0;
185     sh->fSt.foodReady = 0;
186     sh->fSt.foodRequest = 0;
187
188     if (semUp (semgid, sh->mutex) == -1) {                                      /* exit critical region */
189      perror ("error on the down operation for semaphore access (WT)");
190         exit (EXIT_FAILURE);
191     }
192
193     return ret;
194
195 }
```

Fig.11b – Segunda parte da função waitForClientOrChef ().

3.2.2. informChef ()

A função informChef () serve para levar o pedido da mesa ao chef. Esta função conta apenas com uma região crítica que coloca o estado do Waiter para INFORM_CHEF e salva-o com a função saveState (). Depois desta região de código, é levantado o semáforo waitOrder (através de um semUp) para o Waiter acordar o Chef e levar-lhe o pedido.

```
204 static void informChef ()
205 {
206     if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
207         perror ("error on the up operation for semaphore access (WT)");
208         exit (EXIT_FAILURE);
209     }
210
211     /* insert your code here */
212     sh->fSt.st.waiterStat = INFORM_CHEF;
213     saveState(nFic, &(sh->fSt));
214
215     if (semUp (semgid, sh->mutex) == -1)                                        /* exit critical region */
216     { perror ("error on the down operation for semaphore access (WT)");
217         exit (EXIT_FAILURE);
218     }
219
220     /* insert your code here */
221     semUp(semgid, sh->waitOrder); // Take food order to chef
222 }
```

Fig.12 – Função informChef ().

3.2.3. takeFoodToTable ()

A função takeFoodToTable () consiste também em apenas uma região crítica e está desenvolvida para levar os pratos já confecionados desde o Chef até à mesa. Dentro desta região, passa-se o estado do Waiter para

TAKE_TO_TABLE e salva-se com a função saveState (). Após isto, leva a comida cliente a cliente, acordando-os desta forma com o seu pedido (como é possível observar nas linhas 244 e 245 da imagem com o código da função). Ulteriormente, é levantado o semáforo requestReceived do cliente de modo a dar como terminado esta fase do programa onde a mesa dos amigos espera pelos pedidos.

```
231  static void takeFoodToTable ()
232  {
233
234      if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
235          perror ("error on the up operation for semaphore access (WT)");
236          exit (EXIT_FAILURE);
237      }
238
239      /* insert your code here */
240
241      sh->fSt.st.waiterStat = TAKE_TO_TABLE;
242      saveState(nFic, &(sh->fSt));
243
244      for (int i = 0 ; i < TABLESIZE ; i++)
245          semUp(semgid, sh->foodArrived);  // Acorda os clientes que estao a espera de comida
246
247      semUp(semgid, sh->requestReceived);
248
249      if (semUp (semgid, sh->mutex) == -1)  {                                  /* exit critical region */
250          perror ("error on the down operation for semaphore access (WT)");
251          exit (EXIT_FAILURE);
252      }
253  }
```

Fig.13 – Função takeFoodToTable.

3.2.4. receivePayment ()

Por último, temos a função receivePayment (). Tal como as duas funções anteriores, apenas possui uma região crítica, responsável pela receção do pagamento. Mais uma vez dentro da região crítica mudamos o estado o Waiter para RECEIVE_PAYMENT e salvamos esta operação através da função saveState (). Finalmente, fazemos semUp do semáforo requestReceived que permite notificar as clientes que o Waiter deu entrada no pedido.

```
262  static void receivePayment ()
263  {
264      if (semDown (semgid, sh->mutex) == -1)  {                                   /* enter critical region */
265          perror ("error on the up operation for semaphore access (WT)");
266          exit (EXIT_FAILURE);
267      }
268
269      /* insert your code here */
270      sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
271      saveState(nFic, &(sh->fSt));
272
273      semUp(semgid, sh->requestReceived);
274
275      if (semUp (semgid, sh->mutex) == -1)  {                                  /* exit critical region */
276          perror ("error on the down operation for semaphore access (WT)");
277          exit (EXIT_FAILURE);
278      }
279  }
280
```

Fig.14 – Função receivePayment ().

3.3. Chef

O chef é a última de 3 entidades que falta referir. Esta entidade apenas "conversa" com o Waiter, ou seja, não tem comunicação direta com a entidade Client. O seu ciclo de vida no programa é relativamente simples, como é possível observar na imagem 15. Possui apenas 2 funções: waitForOrder e processOrder. Ao contrário do que se assume, estas funções não fazem necessariamente o que indicam no seu nome, como iremos explicar mais adiante.

3.3.1. waitForOrder ()

O nome desta função pode induzir ao erro já que "wait for order" traduz-se para "esperar por pedido". De facto, o Chef já possui como estado inicial o "WAIT_FOR_ORDER" (0) e como o estado seguinte possível é "COOK", esta função irá tocar apenas nestes (estados). Por isto, o único código em falta antes da região crítica da função é a realização de um semDown do semáforo waitOrder, levantado pelo Waiter na função informChef ().

Já dentro da região crítica desta função colocamos o estado do chef para COOK, ou seja, é aqui que o Chef começa o processo de confeção dos pedidos dos clientes. Na linha seguinte 131, salva-se este novo estado com a função saveState ().

```
116  static void waitForOrder ()
117  {
118      /* insert your code here */
119      if (semDown (semgid, sh->waitOrder) == -1) {
120          perror ("error on the down operation for semaphore access (PT)");
121          exit (EXIT_FAILURE);
122      }
123
124      if (semDown (semgid, sh->mutex) == -1) {                    /* enter critical region */
125          perror ("error on the up operation for semaphore access (PT)");
126          exit (EXIT_FAILURE);
127      }
128
129      /* insert your code here */
130      sh->fSt.st.chefStat = COOK; // Changes the internal state of the chef to COOK
131      saveState(nFic,&(sh->fSt));
132
133      if (semUp (semgid, sh->mutex) == -1) {                    /* exit critical region */
134          perror ("error on the up operation for semaphore access (PT)");
135          exit (EXIT_FAILURE);
136      }
137  }
```

Fig.15 – Função waitForOrder ().

3.3.2. processOrder ()

Na função processOrder, o chef cozinha e, aqui, através de um sleep correspondente ao tempo que o Chef demora a cozinhar o pedido, demora-se um certo tempo até a comida estar pronta. Quando a comida ficar pronta, o chef irá entrar numa região crítica na qual o seu estado muda para REST e é levantada uma flag que indica que a comida está pronta. Salva-se posteriormente este estado. Ao sair da região crítica, o Chef vai acordar o Waiter com um semUp do waiterRequest para este levar a comida até à mesa, como é possível verificar na imagem 16.

```
145  static void processOrder ()
146  {
147      usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));
148
149      if (semDown (semgid, sh->mutex) == -1) {                                    /* enter critical region */
150          perror ("error on the up operation for semaphore access (PT)");
151          exit (EXIT_FAILURE);
152      }
153
154      /* insert your code here */
155      sh->fSt.st.chefStat = REST; // Changes the internal state of the chef to COOK
156      sh->fSt.foodReady = 1;  // Increments the number of food ready
157      saveState(nFic,&(sh->fSt));
158
159
160      if (semUp (semgid, sh->mutex) == -1) {                                    /* exit critical region */
161          perror ("error on the up operation for semaphore access (PT)");
162          exit (EXIT_FAILURE);
163      }
164
165      /* insert your code here */
166      semUp(semgid, sh->waiterRequest); // Wakes up waiter to take food to table
167
168  }
```

Fig.16 – Função processOrder ().

4. Resultados

Durante a implementação da nossa solução, fizemos compilámos, com o comando make, e corremos o executável probSemSharedMemRestaurant, que resultou da compilação do ficheiro proSemSharedMemRestaurant.c. Compilámos a solução que o professor disponibilizou usando o comando make_allbin e corremos o executável para podermos comparar os nossos resultados com os do professor.

Neste ponto do relatório, irão ser avaliados os resultados obtidos, para isso, corremos o script run.sh que simula 1000 jantares. O objetivo de correr este script é para sabermos se existem situações de deadlock na nossa implementação.

Escolhemos a milésima e última vez que o programa correu para analisarmos e reparamos que não existem situações de deadlock e todas as regras descritas no enunciado do problema são cumpridas, logo podemos concluir que a nossa implementação funciona corretamente.

Run n.º 1000

Restaurant - Description of the internal state

CH	WT	C00	C01	C02	C03	C04	C05	C06	C07	C08	C09	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	ATT	FIE	1st	las
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	-1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	0	15	-1
0	0	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	2	0	15	-1
0	0	1	1	1	1	2	1	1	1	1	1	1	2	1	1	1	2	1	1	1	1	3	0	15	-1
0	0	2	1	1	1	2	1	1	1	1	1	1	2	1	1	1	2	1	1	1	1	4	0	15	-1
0	0	2	1	1	1	2	1	1	2	1	1	1	2	1	1	1	2	1	1	1	1	5	0	15	-1
0	0	2	1	1	2	2	1	1	2	1	1	1	2	1	1	1	2	1	1	1	1	6	0	15	-1
0	0	2	1	1	2	2	1	1	2	1	1	2	2	1	1	1	2	1	1	1	1	7	0	15	-1
0	0	2	1	1	2	2	1	1	2	1	1	2	2	1	1	1	2	1	2	1	1	8	0	15	-1
0	0	2	1	1	2	2	1	2	2	1	1	2	2	1	1	1	2	1	2	1	1	9	0	15	-1
0	0	2	1	1	2	2	1	2	2	1	1	2	2	1	2	1	2	1	2	1	1	10	0	15	-1
0	0	2	1	1	2	2	1	2	2	1	1	2	2	1	2	1	2	2	2	1	1	11	0	15	-1
0	0	2	1	1	2	2	2	2	2	1	1	2	2	1	2	1	2	2	2	1	1	12	0	15	-1
0	0	2	1	1	2	2	2	2	2	1	1	2	2	1	2	1	2	2	2	1	2	13	0	15	-1
0	0	2	2	1	2	2	2	2	2	1	1	2	2	1	2	1	2	2	2	1	2	14	0	15	-1
0	0	2	2	1	2	2	2	2	2	2	1	2	2	1	2	1	2	2	2	1	2	15	0	15	-1
0	0	2	2	1	2	2	2	2	2	2	1	2	2	1	2	1	2	2	2	2	2	16	0	15	-1
0	0	2	2	1	2	2	2	2	2	2	1	2	2	1	2	2	2	2	2	2	2	17	0	15	-1
0	0	2	2	2	2	2	2	2	2	2	1	2	2	1	2	2	2	2	2	2	2	18	0	15	-1
0	0	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2	2	19	0	15	-1
0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	20	0	15	12
0	0	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	20	0	15	12
0	0	4	2	2	2	2	2	2	4	2	2	2	2	2	2	2	2	2	2	2	2	20	0	15	12
0	0	4	2	2	2	2	2	2	4	2	2	2	2	2	2	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	2	2	2	2	4	2	2	4	2	2	2	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	2	2	2	2	4	2	2	4	4	2	2	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	2	2	2	4	4	2	2	4	4	2	2	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	2	4	2	4	4	2	2	4	4	2	2	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	4	4	2	4	4	2	2	4	4	2	2	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	4	4	2	4	4	2	2	4	4	2	4	2	2	2	4	2	2	20	0	15	12
0	0	4	2	2	4	4	2	4	4	2	2	4	4	2	4	2	2	2	4	2	4	20	0	15	12
0	0	4	2	2	4	4	2	4	4	2	2	4	4	2	4	2	2	4	4	2	4	20	0	15	12
0	0	4	2	2	4	4	2	4	4	2	2	4	4	2	4	2	3	4	4	2	4	20	0	15	12
0	0	4	4	2	4	4	2	4	4	2	2	4	4	2	4	2	3	4	4	2	4	20	0	15	12
0	0	4	4	2	4	4	2	4	4	2	2	4	4	2	4	2	3	4	4	4	4	20	0	15	12
0	0	4	4	2	4	4	2	4	4	4	2	4	4	2	4	2	3	4	4	4	4	20	0	15	12
0	0	4	4	2	4	4	4	4	4	4	2	4	4	2	4	2	3	4	4	4	4	20	0	15	12
0	0	4	4	2	4	4	4	4	4	4	2	4	4	4	4	2	3	4	4	4	4	20	0	15	12
0	0	4	4	4	4	4	4	4	4	4	2	4	4	4	4	2	3	4	4	4	4	20	0	15	12
0	0	4	4	4	4	4	4	4	4	4	2	4	4	4	4	4	3	4	4	4	4	20	0	15	12
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	4	4	4	4	20	0	15	12

0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
0	1	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
1	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
2	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
2	2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	4	4	4	4	4	4	4	5	4	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	4	4	4	5	4	4	4	5	4	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	4	4	4	5	4	4	4	5	5	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	5	4	4	5	4	4	4	5	5	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	5	5	4	5	4	4	4	5	5	4	4	4	4	4	4	4	20	0	15	12
2	2	5	4	4	5	5	4	5	4	4	4	5	5	4	4	4	4	4	4	4	5	20	0	15	12
2	2	5	4	4	5	5	4	5	4	4	4	5	5	4	4	4	4	4	5	4	5	20	0	15	12
2	2	5	4	4	5	5	4	5	5	4	4	5	5	4	4	4	4	4	5	4	5	20	0	15	12
2	2	5	4	4	5	5	4	5	5	4	4	5	5	4	5	4	4	4	5	4	5	20	0	15	12
2	2	5	5	4	5	5	4	5	5	4	4	5	5	4	5	4	4	4	5	4	5	20	0	15	12
2	2	5	5	4	5	5	4	5	5	4	4	5	5	4	5	4	4	5	5	4	5	20	0	15	12
2	2	5	5	4	5	5	4	5	5	5	4	5	5	4	5	4	4	5	5	4	5	20	0	15	12
2	2	5	5	4	5	5	5	5	5	5	4	5	5	4	5	4	4	5	5	4	5	20	0	15	12
2	2	5	5	4	5	5	5	5	5	5	4	5	5	4	5	4	4	5	5	5	5	20	0	15	12
2	2	5	5	5	5	5	5	5	5	5	4	5	5	5	5	4	4	5	5	5	5	20	0	15	12
2	0	5	5	5	5	5	5	5	5	5	4	5	5	5	5	4	4	5	5	5	5	20	0	15	12
2	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	4	5	5	5	5	20	0	15	12
2	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	20	0	15	12
2	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	20	0	15	12
2	0	5	5	5	5	5	5	5	5	5	5	5	5	6	5	5	5	5	5	5	5	20	1	15	12
2	0	5	5	5	5	5	5	6	5	5	5	5	5	6	5	5	5	5	5	5	5	20	2	15	12
2	0	6	5	5	5	5	5	6	5	5	5	5	5	6	5	5	5	5	5	5	5	20	3	15	12
2	0	6	5	5	5	5	5	6	5	5	5	5	6	6	5	5	5	5	5	5	5	20	4	15	12
2	0	6	5	5	5	6	5	6	5	5	5	5	6	6	5	5	5	5	5	5	5	20	5	15	12
2	0	6	5	5	5	6	5	6	5	6	5	5	6	6	5	5	5	5	5	5	5	20	6	15	12
2	0	6	5	5	5	6	5	6	5	6	5	5	6	6	5	5	5	5	5	5	6	20	7	15	12
2	0	6	5	5	5	6	5	6	5	6	5	5	6	6	6	5	5	5	5	5	6	20	8	15	12
2	0	6	6	5	5	6	5	6	5	6	5	5	6	6	6	5	5	5	5	5	6	20	9	15	12
2	0	6	6	5	5	6	5	6	5	6	5	5	6	6	6	5	5	5	6	5	6	20	10	15	12
2	0	6	6	5	5	6	6	6	5	6	5	5	6	6	6	5	5	5	6	5	6	20	11	15	12
2	0	6	6	5	5	6	6	6	5	6	5	5	6	6	6	5	5	6	6	5	6	20	12	15	12
2	0	6	6	5	5	6	6	6	5	6	5	6	6	6	6	5	5	6	6	5	6	20	13	15	12
2	0	6	6	5	6	6	6	6	5	6	6	6	6	6	6	6	6	6	6	5	6	20	17	15	12
2	0	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	6	6	6	5	6	20	18	15	12
2	0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	20	19	15	12
2	0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	20	20	15	12
2	0	6	8	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	20	20	15	12
2	0	6	8	6	6	6	6	6	6	8	6	6	6	6	6	6	6	6	6	6	6	20	20	15	12
2	0	6	8	6	6	6	8	6	6	8	6	6	6	6	6	6	6	6	6	6	6	20	20	15	12
2	0	6	8	6	6	6	8	6	6	8	6	8	6	6	6	6	6	6	6	6	6	20	20	15	12
2	0	6	8	6	6	6	8	6	6	8	8	8	6	6	6	6	6	6	6	6	6	20	20	15	12
2	0	6	8	6	6	6	8	6	6	8	8	8	6	6	6	6	6	6	6	8	6	20	20	15	12
2	0	6	8	6	8	6	8	6	6	8	8	8	6	6	6	6	6	6	6	8	6	20	20	15	12
2	0	6	8	6	8	6	8	6	6	8	8	8	6	6	6	6	6	8	6	8	6	20	20	15	12
2	0	6	8	8	8	6	8	6	6	8	8	8	6	6	6	6	6	8	6	8	6	20	20	15	12
2	0	6	8	8	8	6	8	6	8	8	8	8	6	6	6	6	6	8	6	8	6	20	20	15	12
2	0	6	8	8	8	6	8	6	8	8	8	8	6	6	6	8	6	8	6	8	6	20	20	15	12
2	0	6	8	8	8	6	8	6	8	8	8	8	8	6	6	8	6	8	6	8	6	20	20	15	12
2	0	6	8	8	8	6	8	6	8	8	8	8	8	6	6	8	6	8	6	8	8	20	20	15	12
2	0	6	8	8	8	8	8	6	8	8	8	8	8	6	6	8	6	8	6	8	8	20	20	15	12
2	0	8	8	8	8	8	8	8	8	8	8	8	8	6	6	8	6	8	6	8	8	20	20	15	12
2	0	8	8	8	8	8	8	8	8	8	8	8	8	6	8	8	6	8	6	8	8	20	20	15	12
2	0	8	8	8	8	8	8	8	8	8	8	8	8	7	8	8	6	8	6	8	8	20	20	15	12
2	0	8	8	8	8	8	8	8	8	8	8	8	8	7	8	8	6	8	8	8	8	20	20	15	12
2	0	8	8	8	8	8	8	8	8	8	8	8	8	7	8	8	8	8	8	8	8	20	20	15	12
2	0	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	20	20	15	12
2	3	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	20	20	15	12

Fig.17 – Output da solução implementada por nós.

Para melhor compreender e verificar a existência de erros contruímos a seguinte tabela com informações relativas aos semáforos, como qual a entidade que faz up ou down no semáforo, o número de down's e up's feitos e a função onde se faz up ou down num semáforo.

Semáforo	Entidade que faz down	Função onde se faz down	Número de down's	Entidade que faz up	Função onde se faz up	Número de up's
friendsArrived	Clientes	waitFriends ()	19	LC	waitFriends ()	19
requestReceived	LC;	waitAndPay ()	1	Empregado	takeFoodToTable () receivePayment ()	1 1
foodArrived	Clientes	waitFood ()	20	Empregado	takeFoodToTable ()	20
allFinished	Clientes	waitAndPay ()	20	Clientes	waitAndPay ()	20
waiterRequest	Empregado	waitForClientOrChef ()	1	FC; LC; Chefe	orderFood () waitAndPay () processOrder ()	1 1 1
waitOrder	Chefe	waitForOrder ()	1	Empregado	informChef ()	1

Legenda:

- LC - Last Client (último cliente a chegar ao restaurante);
- FC - First Client (primeiro cliente a chegar ao restaurante).

5. Conclusão

Em conclusão, apresentámos através deste relatório uma noção de semáforos e regiões críticas e relacionámos estes conteúdos com o segundo projeto da cadeira de Sistemas Operativos. Completámos o código fornecido incidindo principalmente nas entidades Client, Waiter e Chef, processos independentes cuja sincronização e comunicação é feita através de vários semáforos e de memória partilhada. Conseguimos também ultrapassar situações de deadlock (onde dois ou mais processos – que sejam da mesma entidade ou não – estejam bloqueados e, por fim, adquirimos conhecimentos sobre programação de semáforos em C.