



deti

universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

TRABALHO 2

JANTAR DE AMIGOS

Sistemas operativos 2022

Professor Nuno Lau e Professor Guilherme Campos

Trabalho realizado por:

João Nuno da Silva Luís (107403) | 50%

Diana Raquel Rodrigues Miranda (107457) | 50%

Índice

Introdução	2
Constantes e Semáforos	4
Ciclo de vida - Client	5
waitFriends()	5
Orderfood()	6
Waitfood()	6
WaitAndPay()	7
Ciclo de vida - Waiter	9
waitForClientorChef()	9
informChef()	10
takeFoodToTable()	11
receivePayment()	11
Ciclo de vida - Chef	12
waitForOrder()	12
processOrder()	12
Resultados	13
Conclusão	15
Bibliografia.....	16

Índice de figuras

Figura 1- Constantes a usar pelo chef.....	4
Figura 2- Constantes a usar pelo waiter	4
Figura 3- Constantes a usar pelo cliente	4
Figura 4- Função waitFriends()	5
Figura 5- Função orderFood().....	6
Figura 6- Função waitFood()	6
Figura 7-Função waitAndPay().....	7
Figura 8- Função waitAndPay().....	8
Figura 9- Função waitAndPay().....	8
Figura 10- Função waitForClientOrChef	10
Figura 11- Função informChef()	10
Figura 12- Função takeFoodToTable.....	11
Figura 13- Função receivePayment()	11
Figura 14- Função waitForOrder()	12
Figura 15- Função processOrder()	12

Introdução

Este trabalho é realizado no âmbito da disciplina Sistemas Operativos do 2º ano da Licenciatura em Engenharia Informática.

O seu objetivo é ter uma melhor compreensão dos mecanismos associados à execução e sincronização de processos e threads, através do adicionamento de código ao código fonte.

O tema deste trabalho é a gestão de 3 identidades num jantar de amigos, num restaurante.

Assim, existem três de entidades: **clientes**, **waiter** e **chef**. Com estas entidades em vista, vamos então usar semáforos para controlar o fluxo de execução e o acesso à região da memória partilhada.

Constantes e Semáforos

Para controlar cada uma das entidades envolvidas, foram usadas algumas constantes, que vão ser redefinidas ao longo do programa, e também impressas como *output* do programa.

Para além disto, foram definidos também alguns semáforos, para controlar o acesso de cada identidade à região de memória partilhada.

```

23  /* Client state constants */
24
25  /** \brief client initial state */
26  #define INIT 1
27  /** \brief client is waiting for friends to arrive at table */
28  #define WAIT_FOR_FRIENDS 2
29  /** \brief client is requesting food to waiter */
30  #define FOOD_REQUEST 3
31  /** \brief client is waiting for food */
32  #define WAIT_FOR_FOOD 4
33  /** \brief client is eating */
34  #define EAT 5
35  /** \brief client is waiting for others to finish */
36  #define WAIT_FOR_OTHERS 6
37  /** \brief client is waiting to complete payment */
38  #define WAIT_FOR_BILL 7
39  /** \brief client finished meal */
40  #define FINISHED 8
41
42  /* Chef state constants */
43
44  /** \brief chef waits for food order */
45  #define WAIT_FOR_ORDER 0
46  /** \brief chef is cooking */
47  #define COOK 1
48  /** \brief chef is resting */
49  #define REST 2
50
51  /* Waiter state constants */
52
53  /** \brief waiter waits for food request */
54  #define WAIT_FOR_REQUEST 0
55  /** \brief waiter takes food request to chef */
56  #define INFORM_CHEF 1
57  /** \brief waiter takes food to table */
58  #define TAKE_TO_TABLE 2
59  /** \brief waiter receives payment */
60  #define RECEIVE_PAYMENT 3

```

Figura 3- Constantes a usar pelo cliente

Figura 1- Constantes a usar pelo chef

Figura 2- Constantes a usar pelo waiter

Também criámos uma tabela para verificar o comportamento de cada entidade, isto é, onde e quando seria necessário ocorrerem **ups** e **downs** de cada semáforo presente no ficheiro *sharedDataSync.h*, e as funções nelas envolvidas.

Semáforo	Entidade down	Função down	Número de downs	Entidade up	Função up	Número de ups
friendsArrived	Client, exceto o último	waitFriends()	19	Último client	waitFriends()	19
requestReceived	Client	orderFood()	1	Waiter	informChef()	1
		waitAndPay()	1		receivePayment()	1
foodArrived	Client	waitFood()	20	Waiter	receivePayment()	20
allFinished	Client	waitAndPay()	19	Client	waitAndPay()	19
waiterRequest	Waiter	waitForClientOrChef()	3	Client	orderFood()	1
					waitAndPay()	1
				Chef	processOrder()	1
waitOrder	Chef	waitForOrder()	1	Waiter	informChef()	1

Tab.1 Tabela com o comportamento dos semáforos.

Ciclo de vida – Client

A entidade cliente é constituída por 20 clientes (amigos do problema), cada um com id próprio, que agora se vão juntar à mesa.

Para obedecermos ao que é proposto pelo problema, o primeiro amigo a chegar à mesa será o que vai fazer o pedido da comida, depois de todos os outros amigos já terem também chegado. Já o último amigo a chegar à mesa, será o responsável por fazer o pagamento da conta ao waiter.

waitFriends()

Nesta função, os clientes, um a um e de forma aleatória, vão atualizar o seu estado para 2, correspondente ao estado `WAIT_FOR_FRIENDS` e vão incrementar o número de clientes presentes à mesa, na variável `sh->fSt.tableClients`. Depois, vai ser feito o **Down** do semáforo `friendsArrived`, para que os amigos esperem uns pelos outros, sendo depois feito o desbloqueio quando o último amigo chegar.

No caso de ser o primeiro cliente, o id do mesmo vai ser guardado na constante `sh->fSt.tableFirst`, que depois vai ser impresso na tabela, na coluna 1st.

Já o último amigo vai ter também o seu id guardado na variável `sh->fSt.tableLast`, e vai também atualizar o seu estado logo para 4 (`WAIT_FOR_FOOD`), já que, como é o último, não vai fazer o pedido. Por último, dá **Up** a cada um dos amigos que estavam à espera de que todos chegassem.

```

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     /* insert your code here */
169     sh->fSt.tableClients++; // número de clientes soma + 1
170     sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS; // 0 estado é atualizado
171
172     if (sh->fSt.tableClients==1){ // se for o primeiro cliente a chegar
173         first=true;
174         sh->fSt.tableFirst=id;
175     }
176     else if (sh->fSt.tableClients==TABLESIZE){ // se for o último cliente a chegar
177         sh->fSt.tableLast=id; // já guarda o id do último
178         sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD; // o último cliente já pode esperar pelo pedido. Passa do estado 2 para o 4
179         for (int i=1; i<=TABLESIZE; i++){
180             if (semUp (semgid, sh->friendsArrived) == -1) /* unlocks friends (the extra up is so they themselves don't block on the down) */
181             { // este if é para verificar se está bem, mas não dá erro se tirar
182                 perror ("error on the up operation for semaphore access (CT)");
183                 exit (EXIT_FAILURE);
184             }
185         }
186     }
187
188     saveState (nFic, &(sh->fSt)); // Mete-se o & porque é um ponteiro, senão dá erro de FULL_STAT ser diferente de FULL_STAT*
189
190     if (semUp (semgid, sh->mutex) == -1) /* exit critical region */
191     { perror ("error on the up operation for semaphore access (CT)");
192       exit (EXIT_FAILURE);
193     }
194     /* insert your code here */
195     // FAZER SEMPRE o semDown fora da região crítica
196     if (semDown (semgid, sh->friendsArrived) == -1) { /* wait for friends */
197         perror ("error on the down operation for semaphore access (CT)");
198         exit (EXIT_FAILURE);
199     }
200
  
```

Figura 4- Função waitFriends()

orderFood()

Esta função vai ser apenas utilizada pelo primeiro cliente, que é quem faz o pedido da comida.

Nesta, o mesmo vai atualizar o seu estado para 3 (FOOD_REQUEST) e vai ser feito o **Up** no semáforo **waiterRequest**, para que o waiter possa ser desbloqueado, que se encontra sempre bloqueado à espera de um pedido, ou do cliente, ou do chef.

No final da função, é feito o **Down** do semáforo **requestReceived**, e assim, o cliente que faz o pedido, vai ficar bloqueado até receber uma resposta.

```

215 static void orderFood (int id)
216 {
217     if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
218         perror ("error on the down operation for semaphore access (CT)");
219         exit (EXIT_FAILURE);
220     }
221
222     /* insert your code here */
223     sh->fSt.st.clientStat[id] = FOOD_REQUEST; // 0 estado é atualizado para o estado 3
224
225     // 0 pedido é feito ao waiter
226     if (semUp (semgid, sh->waiterRequest) == -1) { // dar up porque ele aqui está a esperar
227         perror ("error on the down operation for semaphore access (WT)");
228         exit (EXIT_FAILURE);
229     }
230
231     saveState (nFic, &(sh->fSt));
232     sh->fSt.foodRequest=1;
233     if (semUp (semgid, sh->mutex) == -1) /* exit critical region */
234     { perror ("error on the up operation for semaphore access (CT)");
235       exit (EXIT_FAILURE);
236     }
237     /* insert your code here */
238     if (semDown (semgid, sh->requestReceived) == -1) {
239         perror ("error on the down operation for semaphore access (CT)");
240         exit (EXIT_FAILURE);
241     }
242 }

```

Figura 5- Função orderFood()

waitFood()

Nesta função, todos os clientes vão esperar que o pedido da comida seja feito pelo 1º amigo, atualizando o seu estado para 4 (WAIT_FOR_FOOD). Após isso, ficam bloqueados até que a comida chegue, através de um **down** no semáforo **foodArrived**.

Quando a comida chegar após o aviso do waiter, os clientes vão então atualizar o seu estado para 5 (EAT).

```

253 static void waitFood (int id)
254 {
255     if (semDown (semgid, sh->mutex) == -1) {
256         perror ("error on the down operation for semaphore access (CT)");
257         exit (EXIT_FAILURE);
258     }
259
260     /* insert your code here */
261     sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD; // 0 estado é atualizado 4
262     saveState (nFic, &(sh->fSt));
263
264     if (semUp (semgid, sh->mutex) == -1) {
265         perror ("error on the down operation for semaphore access (CT)");
266         exit (EXIT_FAILURE);
267     }
268
269     /* insert your code here */
270     if (semDown (semgid, sh->foodArrived) == -1) {
271         perror ("error on the down operation for semaphore access (WT)");
272         exit (EXIT_FAILURE);
273     }
274
275     if (semDown (semgid, sh->mutex) == -1) {
276         perror ("error on the down operation for semaphore access (CT)");
277         exit (EXIT_FAILURE);
278     }
279
280     /* insert your code here */
281     sh->fSt.st.clientStat[id] = EAT; // 0 estado é atualizado para 5
282     saveState (nFic, &(sh->fSt));
283
284     if (semUp (semgid, sh->mutex) == -1) {
285         perror ("error on the down operation for semaphore access (CT)");
286         exit (EXIT_FAILURE);
287     }
288
289 }

```

Figura 6- Função waitFood()

waitAndPay()

Esta função começa por atualizar o estado dos clientes que, depois de terem comido, vão atualizar para o estado 6 (WAIT_FOR_OTHERS), e incrementa o número de clientes que terminaram de comer, número que vai ser impresso no terminal na coluna FIE.

É também feita a atualização da variável **last**, e feito **Down** do semáforo **allFinished**, que faz com que os amigos esperem pelos restantes. Quando todos chegarem ao estado 6, vai ser feito o **Up** do semáforo **allFinished** para cada cliente, desbloqueando-os.

```

301 static void waitAndPay(int id)
302 {
303     bool last = false;
304
305     if (semDown(semgid, sh->mutex) == -1)
306     { /* enter critical region */
307         perror("error on the down operation for semaphore access (CT)");
308         exit(EXIT_FAILURE);
309     }
310     /* insert your code here */
311     if (sh->fSt.tableLast == id)
312     { // entra aqui se for o ultimo a chegar à mesa
313         last = true;
314     }
315
316     sh->fSt.clientStat[id] = WAIT_FOR_OTHERS; // O estado é atualizado para 6
317     sh->fSt.tableFinishEat++; // incrementa o numero de clientes que terminaram de comer
318     saveState(nFic, &(sh->fSt));
319
320     if (sh->fSt.tableFinishEat == TABLESIZE)
321     { // quando acabarem todos de comer
322         for (int i = 0; i < TABLESIZE; i++)
323         {
324             semUp(semgid, sh->allFinished);
325         }
326     }
327     if (semUp(semgid, sh->mutex) == -1)
328     { /* enter critical region */
329         perror("error on the down operation for semaphore access (CT)");
330         exit(EXIT_FAILURE);
331     }
332
333     /* insert your code here */
334     semDown(semgid, sh->allFinished); // Para esperar que terminem de comer, ou seja que passem pelo menos, ao estado 6
  
```

Figura 7-Função waitAndPay()

No caso de o cliente ser o último, cuja *flag* foi atualizada anteriormente, este vai atualizar o seu estado para 7 (WAIT_FOR_BILL), e ativar a flag **sh->fSt.paymentRequest**, que vai ser utilizada na função *WaitForClientOrChef()* do waiter, para permitir a atualização da variável **ret**, que depois vai levar à chamada da função *receivePayment()*, também do waiter. Por fim, vai ser feito o **Up** do semáforo **waiterRequest**, que vai sinalizar o waiter para um pedido de pagamento. Após o waiter receber esse pedido, o último cliente vai também ser desbloqueado.


```

335  ✓   if (last)
336      {
337  ✓   { if (semDown(semgid, sh->mutex) == -1)
338      { /* enter critical region */
339      { perror("error on the down operation for semaphore access (CT)");
340      { exit(EXIT_FAILURE);
341      { }
342      { /* insert your code here */
343      { sh->fSt.st.clientStat[id] = WAIT_FOR_BILL; // 0 estado é atualizado para 7
344      { saveState(nFic, &(sh->fSt));
345      { sh->fSt.paymentRequest = 1; // flag que vai ser usada para chamar a função receivePayment() do waiter no switch case
346  ✓   { if (semUp(semgid, sh->waiterRequest) == -1)
347      { // Para ir chamar a função do Wait que depois chama a função receivePayment() do waiter
348      { perror("error on the down operation for semaphore access (WT)");
349      { exit(EXIT_FAILURE);
350      { }
351      { }
352  ✓   { if (semUp(semgid, sh->mutex) == -1)
353      { /* exits critical region */
354      { perror("error on the down operation for semaphore access (CT)");
355      { exit(EXIT_FAILURE);
356      { }
357      { /* insert your code here */
358  ✓   { if (semDown(semgid, sh->requestReceived) == -1)
359      {
360      { perror("error on the down operation for semaphore access (WT)");
361      { exit(EXIT_FAILURE);
362      { }
363      { }
  
```

Figura 8- Função waitAndPay()

Por fim, todos os clientes atualizam uma última vez o seu estado para 8 (FINISHED), concluindo-se assim o ciclo de vida do client, tendo todos os amigos acabado a sua refeição e tendo sido feito o pagamento.

```

364      {
365      { if (semDown(semgid, sh->mutex) == -1)
366      { { /* enter critical region */
367      { { perror("error on the down operation for semaphore access (CT)");
368      { { exit(EXIT_FAILURE);
369      { { }
370      { { }
371      { { /* insert your code here */
372      { { sh->fSt.st.clientStat[id] = FINISHED; // 0 estado é atualizado para 8
373      { { saveState(nFic, &(sh->fSt));
374      { { }
375      { { if (semUp(semgid, sh->mutex) == -1)
376      { { { /* enter critical region */
377      { { { perror("error on the down operation for semaphore access (CT)");
378      { { { exit(EXIT_FAILURE);
379      { { { }
380      { { }
  
```

Figura 9- Função waitAndPay()

Ciclo de vida – Waiter

A entidade waiter vai ser responsável por receber os pedidos de feitos pelo primeiro cliente, levar esses pedidos ao chefe, trazer a comida pelo chefe cozinhada, e, no fim, receber o pagamento feito pelo último cliente.

waitForClientOrChef()

Esta função está dentro de um *while loop*, que a vai assim chamar 3 vezes, e serve para o waiter esperar por um pedido, ou do cliente, ou do chef.

Assim, sendo o waiter começa por ter o seu estado inicializado a 0 (WAIT_FOR_REQUEST). De seguida, o semáforo **waiterRequest** vai dar **Down**, para bloquear o waiter, enquanto não houverem pedidos.

De seguida, são feitos 3 *if's*, para que, havendo um pedido, o waiter possa saber para quem o encaminhar, ou seja, decidir que função chama a seguir. As *flags* usadas nos *if's* vão sendo atualizadas ao longo dos ciclos de vida do cliente ou chef, e, para não haver problemas em posteriores chamadas da função, estas *flags* são colocadas novamente a 0.

```

145 static int waitForClientOrChef()
146 {
147     int ret=0;
148
149     if (semDown (semgid, sh->mutex) == -1) {
150         perror ("error on the up operation for semaphore access (WT)");
151         exit (EXIT_FAILURE);
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) {
158         perror ("error on the down operation for semaphore access (WT)");
159         exit (EXIT_FAILURE);
160     }
161     /* insert your code here */
162
163     if (semDown (semgid, sh->waiterRequest) == -1) {
164         perror ("error on the down operation for semaphore access (WT)");
165         exit (EXIT_FAILURE);
166     }
167
168     if (semDown (semgid, sh->mutex) == -1) {
169         perror ("error on the up operation for semaphore access (WT)");
170         exit (EXIT_FAILURE);
171     }
172     /* insert your code here */
173     if(sh->fSt.foodRequest== 1){
174         ret= FOODREQ; // vai chamar depois a outra função informChef
175     }
176
177     if(sh->fSt.foodReady== 1){
178         ret= FOODREADY; // vai chamar depois a outra função takeFoodToTable();
179     }

```

```

180     if(sh->fSt.paymentRequest== 1){
181         ret= BILL; // vai chamar depois a outra função ReceivePayment();
182     }
183     sh->fSt.foodRequest=0;
184     sh->fSt.foodReady=0;
185     sh->fSt.paymentRequest=0;
186
187     if (semUp (semgid, sh->mutex) == -1) {
188         perror ("error on the down operation for semaphore access (WT)");
189         exit (EXIT_FAILURE);
190     }
191
192     return ret;
193 }
  
```

Figura 10- Função waitForClientOrChef

informChef()

Esta função vai ser executada pelo waiter quando recebe um pedido do cliente.

Assim, o waiter atualiza o seu estado para 1 (INFORM_CHEF).

Para além disso e através do uso dos semáforos, é feito o desbloqueio do cliente que fez o pedido e o bloqueio do waiter, que espera um pedido vindo do waiter.

Agora, o chef irá proceder à preparação da comida, depois de ser chamada novamente a função *waitForClientOrChef()*, que vai bloquear novamente o waiter.

```

static void informChef ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    /* insert your code here */
    sh->fSt.waiterStat = INFORM_CHEF;
    saveState (nFic, &(sh->fSt));
    sh->fSt.foodOrder=1; // flag of food order from waiter to chef

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

    /* insert your code here */
    if(semUp(semgid,sh->requestReceived) == -1)
    {
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }
    if (semUp (semgid, sh->waitOrder) == -1) {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
}
  
```

Figura 11- Função informChef()

takeFoodToTable()

A função começa por desbloquear os clientes que estavam à espera que a comida chegasse, já que esta função é executada após o chef cozinhar a comida e desbloquear o próprio waiter.

Como o waiter vai entregar a comida à mesa, vai também atualizar o seu estado para 2 (TAKE_TO_TABLE).

```

238 static void takeFoodToTable()
239 {
240     if (semDown(semgid, sh->mutex) == -1)
241     { /* enter critical region */
242         perror("error on the up operation for semaphore access (WT)");
243         exit(EXIT_FAILURE);
244     }
245
246     /* insert your code here */
247     for (int i = 0; i < TABLESIZE; i++)
248     {
249         if (semUp(semgid, sh->foodArrived) == -1)
250         {
251             perror("error on the down operation for semaphore access (WT)");
252             exit(EXIT_FAILURE);
253         }
254     }
255     sh->fSt.st.waiterStat = TAKE_TO_TABLE;
256     saveState(nFic, &(sh->fSt));
257
258     if (semUp(semgid, sh->mutex) == -1)
259     { /* exit critical region */
260         perror("error on the down operation for semaphore access (WT)");
261         exit(EXIT_FAILURE);
262     }
263 }
  
```

Figura 12- Função takeFoodToTable()

receivePayment()

A última função do waiter recebe o pagamento da conta feito pelo último cliente a chegar ao restaurante, depois de este acabar a sua refeição. O waiter sinaliza o cliente que recebeu o pedido com sucesso, desbloqueando-o, através do **Up** do semáforo **requestReceived**.

Assim, é terminado o ciclo de vida do waiter.

```

272 static void receivePayment ()
273 {
274     if (semDown (semgid, sh->mutex) == -1) {
275         perror ("error on the up operation for semaphore access (WT)");
276         exit (EXIT_FAILURE);
277     }
278
279     /* insert your code here */
280     sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
281     saveState (nFic, &(sh->fSt));
282     //printf("Waiter: Payment received\n");recebe o pagamento e na proxima linha o last passa a 8
283     if (semUp (semgid, sh->requestReceived) == -1) {
284         perror ("error on the down operation for semaphore access (WT)");
285         exit (EXIT_FAILURE);
286     }
287
288     if (semUp (semgid, sh->mutex) == -1) {
289         perror ("error on the down operation for semaphore access (WT)");
290         exit (EXIT_FAILURE);
291     }
292 }
  
```

Figura 13- Função receivePayment()

Ciclo de vida - Chef

A entidade chef vai ser responsável por cozinhar o pedido do último cliente entregue pelo waiter.

waitForOrder()

Esta função faz o chef esperar por um pedido, encontrando-se bloqueado.

Quando for o momento de cozinhar, a *flag* **sh->fSt.foodOrder** vai ter o valor 1, proveniente do waiter, o que provoca o atualizar do estado do chefe para 1 (COOK).

```

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 (WT)");
121         exit (EXIT_FAILURE);
122     }
123     if (semDown (semgid, sh->mutex) == -1) {                               /* enter critical region */
124         perror ("error on the up operation for semaphore access (PT)");
125         exit (EXIT_FAILURE);
126     }
127     /* insert your code here */
128     if( sh->fSt.foodOrder==1){
129         sh->fSt.st.chefStat = COOK;
130         saveState (nFic, &(sh->fSt));
131     }
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 }
  
```

Figura 14- Função waitForOrder()

processOrder()

A última função do chef vai cozinhar o pedido e depois sinalizar o waiter que o mesmo está pronto e que o pode levar para a mesa.

O chef cozinha o pedido durante um tempo aleatório, calculado na função *usleep* (linha 147).

Depois de cozinhar o pedido, o chef altera o valor da *flag* **sh->fSt.foodReady** para 1 e altera o seu estado para 2 (REST). Por fim, desbloqueia o waiter que se encontrava à espera de um pedido, fazendo **Up** do semáforo **waiterRequest**. Assim, é terminado o ciclo de vida do chef.

```

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     /* insert your code here */
154     sh->fSt.foodReady=1; // aqui o chef diz que a comida está pronta
155
156     sh->fSt.st.chefStat = REST;
157     saveState (nFic, &(sh->fSt));
158
159     if (semUp (semgid, sh->mutex) == -1) {                               /* exit critical region */
160         perror ("error on the up operation for semaphore access (PT)");
161         exit (EXIT_FAILURE);
162     }
163     /* insert your code here */
164     if (semUp (semgid, sh->waiterRequest) == -1) { // dar up porque ele aqui está a esperar
165         perror ("error on the down operation for semaphore access (WT)");
166         exit (EXIT_FAILURE);
167     }
168 }
  
```

Figura 15- Função processOrder()

Resultados

Para verificarmos o *output* do programa e se este estava correto, utilizámos partes de código pré-compilado fornecido pelo professor, através de comandos como *make ct* e *make ct_wt*. Para verificarmos se a nossa solução acompanhava o expectável, fizemos também *make all_bin*, para testar exclusivamente a solução fornecida.

Depois de serem resolvidos todos os problemas, fizemos *make all* e obtivemos o seguinte:

[illegible]

[illegible]

Conclusão

Com este trabalho, pudemos aprofundar os nossos conhecimentos sobre o uso de semáforos e *flags* associadas. Percebemos melhor como usá-los e qual a sua relação com a área crítica.

As nossas maiores dificuldades foram perceber todas as funções, semáforos e *flags* necessárias, visto que todo o código teria de ter uma estrutura e um funcionamento correto. Para além disso, algumas vezes tivemos dificuldades em perceber quando bloquear ou desativar certas identidades com os semáforos, mas com a criação da Tabela acima descrita foi mais fácil desenvolver o trabalho.

Assim, concluímos que chegámos ao objetivo do trabalho, cumprindo as orientações do problema fornecido com sucesso.

Bibliografia

- Slides teóricos
- Aula prática nº 10