



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

Jantar de Amigos

Relatório do segundo projeto realizado no âmbito da disciplina de Sistemas Operativos pelos professores:

Professor José Nuno Panelas Nunes Lau (TP1)
Professor António Guilherme Rocha Campos (P06)

Trabalho realizado pelos alunos:

Tomás Matos (108624)
Gonçalo Ferreira (107853)

Índice

Índice	1
Introdução	1
O problema	2
Primeiras abordagens	2
Tabela de Comportamentos	3
clean.sh	5
filter.sh	5
run.sh	6
Implementação	6
Client	6
Função waitFriends	6
Função orderFood	8
Função waitFood	9
Função waitAndPay	10
Waiter	12
Função waitForClientOrChef	12
Função informChef	13
Função takeFoodToTable	14
Função receivePayment	14
Chef	15
Função waitForOrder	15
Função processOrder	15
Conclusão	20

Introdução

O problema

A temática do trabalho baseia-se num jantar de amigos que foi sediado num restaurante onde havia apenas um cozinheiro e um empregado. O restaurante, por sinal, possui uma quantidade de regras que têm de ser respeitadas:

- O primeiro cliente faz o pedido mas, contudo, só poderá pedir quando todos os membros da mesa tiverem chegado.
- Após isto, o empregado de mesa deve recolher o pedido dos clientes e levar ao cozinheiro, apenas voltando quando o pedido estiver pronto a servir.
- No final, todos os amigos abandonam a mesa sendo que o último a chegar à mesma deve ser encarregue de pagar

Nota: A mesa deve ser ajustável à quantidade de amigos no jantar

Para ser mais fácil para os alunos, foi fornecido na página da disciplina um código suplementar escrito na linguagem C para ajudar a perceber o problema.

Primeiras abordagens

As primeiras abordagens feitas ao tema são de claramente perceber a relação do tema à matéria dada, algo que nos é dito em parte no enunciado

Os clientes, o empregado e o cozinheiro são processos independentes , sendo sincronizados através de semáforos e memória partilhada, tendo em conta que todos os processos são iniciados a quando da inicialização do programa e que estão em execução a partir desse momento, sendo ativos quando necessário e bloqueando quando precisarem de esperar. Para terminar as informações dadas, também existem indicações de que os clientes podem demorar um pouco de tempo e que devemos obter esse tempo através de uma distribuição de probabilidade uniforme com o tempo máximo.

Após isso é preciso analisar o código entregue pelo professor no início, começando pelos ficheiros sh.

Tabela de Comportamentos

Semaphore	Up			Down		
	Entity	Function	Quantity	Entity	Function	Quantity
Mutex	firstClient	waitFriends()	1	firstClient	waitFriends()	1
		orderFood()	1		orderFood()	1
		waitFood()	2		waitFood()	2
		waitAndPay()	2		waitAndPay()	2
	Client	waitAndPay()	2	Client	waitAndPay()	2
		waitFood()	2		waitFood()	2
		waitFriends()	1		waitFriends()	1
	lastClient	waitAndPay()	2	lastClient	waitAndPay()	3
		waitFriends()	1		waitFriends()	1
		waitFood()	2		waitFood()	2
	Waiter	receivePayment()	1	Waiter	receivePayment()	1

		waitForClientOrChef()	2		waitForClientOrChef()	2
		takeFoodForTable()	1		takeFoodForTable()	1
		informChef()	1		informChef()	1
	Chef	waitForOrder()	1	Chef	waitForOrder()	1
		processOrder()	1		processOrder()	1
friendsArrived	lastClient	waitFriends()	1	Client	waitFriends()	1
				firstClient		1
				lastClient		1
requestReceived	Waiter	informChef()	1	firstClient	OrderFood()	1
		receivePayment()	1	lastClient	waitAndPay()	1
foodArrived	Waiter	takeFoodToTable()	TABLESIZE	firstClient	waitFood()	1
				Client		1
				lastClient		1
allFinished	lastClient	waitAndPay()	TABLESIZE	firstClient	waitAndPay()	1
				Client		1
				lastClient		1
waiterRequest	firstClient	orderFood()	1	Waiter	WaitForClientOnChef	1
	lastClient	waitAndPay()	1			
	Chef	processOrder()	1			
waitOrder	waiter	informChef()	1	Chef	waitOrder()	1

Ficheiro run

clean.sh

```
$ clean.sh
1  #!/bin/bash
2
3  rm -f error*
4  rm -f core
5
6  # change 0x61066137 to your semaphore and shared memory key
7  ipcrm -S 0x61066137
8  ipcrm -M 0x61066137
9
```

Os 2 primeiros comandos iniciais fazem referência a remover qualquer ficheiro de erro criado pelo programa ou qualquer ficheiro raiz (-rm).

Os outros 2 dirigem-se a qualquer a retirar qualquer tipo de comunicação inter-processos. É necessário introduzir passar uma “shared memory key” e “uma semaphore key” para que esta comunicação seja interrompida. Então, é preciso trocar os valores inseridos à frente dos argumentos -S e -M.

```
try ipcs -help for more information.
goncalo@goncalomf:~$ ipcs -s
----- Semaphore Arrays -----
key          semid      owner      perms      nsems
goncalo@goncalomf:~$ ipcs -m
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000  98306      goncalo    600        2097152    2          dest
0x00000000  98307      goncalo    600        4194304    2          dest
0x00000000   6         goncalo    600        524288     2          dest
0x00000000  98312      goncalo    600        524288     2          dest
0x00000000  65554      goncalo    600        524288     2          dest
0x00000000   19        goncalo    600        524288     2          dest
0x00000000  65557      goncalo    600        4194304    2          dest
0x00000000  65560      goncalo    606        12015360   2          dest
0x00000000  65561      goncalo    606        12015360   2          dest
0x00000000  65562      goncalo    606        2880000    2          dest
0x00000000  65563      goncalo    606        2880000    2          dest
0x00000000  65572      goncalo    600        524288     2          dest
0x00000000  32815      goncalo    600        524288     2          dest
0x00000000  65594      goncalo    600        4194304    2          dest
0x00000000  65595      goncalo    600        134217728  2          dest
0x00000000   61        goncalo    600        524288     2          dest
```

filter.sh

```
1  #!/bin/bash
2
3  ./probSemSharedMemRestaurant | awk -f filter_log.awk
4
5
```

Esta linha de código remete à utilização executável “probSemSharedMemRestaurant”, que a quando de ser chamado deve correr a “bash cheatsheet” filter_log.awk, que possui código remetente ao filtro de resultados obtidos obtidos pelo programa no terminal.

run.sh

```
1  #!/bin/bash
2
3  case $# in
4      0) n=1000;;
5      1) n=$1;;
6      *) echo "USAGE: $0 «number-of-runs»"; exit;;
7  esac
8
9  if ! [ $n -gt 0 ] 2>/dev/null; then
10     echo "Wrong argument value (\"$n\"). Aborting."
11     exit 1
12 fi
13
14 for i in $(seq 1 $n)
15 do
16     echo -e "\n\e[34;1mRun n.º $i\e[0m"
17     ./probSemSharedMemRestaurant
18 done
19
```

Neste ficheiro podemos ver a utilização de case onde ele utiliza \$# (número de argumentos passados na chamada do programa) e verifica os casos específicos da passagem de argumentos, como definir o número de “runs” do programa em 1000 quando não é passado nenhum argumento na função, ou quando é passado apenas 1 argumento, o número de “runs” vai ser o passado nesse argumento. Qualquer outro valor vai ser um erro, onde foi.

Implementação

Client

Função waitFriends

```

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++;
171     if (sh->fSt.tableClients == 1){
172         first = true;
173         sh->fSt.tableFirst = id;
174     }
175     if (sh->fSt.tableClients == TABLESIZE){
176         sh->fSt.tableLast = id;
177         for(int i = 0; i < TABLESIZE - 1; i++){
178             semUp(semgid, sh->friendsArrived);
179         }
180     }
181     if (id == sh->fSt.tableLast)
182     {
183         sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
184     }else{
185         sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;
186     }
187     saveState(nFic, &(sh->fSt));
188
189     if (semUp (semgid, sh->mutex) == -1)                                   /* exit critical region */
190     { perror ("error on the up operation for semaphore access (CT)");
191       exit (EXIT_FAILURE);
192     }
193     /* insert your code here */
194     if (sh->fSt.tableClients != TABLESIZE){
195         semDown(semgid, sh->friendsArrived);
196     }
197     return first;
198 }
199

```

Esta função tal como o nome indica faz com que os clientes esperem uns pelos outros até à próxima ação, esta recebe como argumento o id dos clientes que vão chegando. Primeiramente entramos na zona crítica e aí incrementamos a variável `sh->fSt.tableClients` que posteriormente vai ser escrita no output na coluna do ATT (at the table). Se apenas estiver uma pessoa na mesa, este é o primeiro cliente a chegar, então o booleano `first` passa para verdadeiro e a variável `sh->fSt.tableFirst` é atualizada com o respetivo id do cliente.

Por outro lado, se estiverem já os 20 clientes na mesa, o id do último que chegou vai ser armazenado na variável `sh->fSt.tableLast` e os semáforos `sh->friendsArrived` são libertados para todos os outros clientes que estavam à espera por amigos na mesa.

Seguidamente, se ainda faltarem clientes chegar alteramos o estado do cliente em questão para `WAIT_FOR_FRIENDS` (2), o último cliente a chegar nunca toma este estado, passa diretamente para `WAIT_FOR_FOOD` (4).

Posteriormente executamos a função *saveState* que imprime os estados no terminal e saímos da região crítica.

A última condição bloqueia a execução do programa através do semáforo *sh->friendsArrived* se ainda não tiverem chegado os 20 amigos, quando todos os amigos chegarem, o semáforo será libertado e a execução continua. Esta função vai retornar um boolean que indica se o cliente em questão é o primeiro a chegar ou não.

Função *orderFood*

```

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.foodRequest = 1;
224     semUp(semgid, sh->waiterRequest);
225
226     sh->fSt.st.clientStat[id] = FOOD_REQUEST;
227     saveState(nFic, &(sh->fSt));
228
229     if (semUp (semgid, sh->mutex) == -1)                                    /* exit critical region */
230     {
231         perror ("error on the up operation for semaphore access (CT)");
232         exit (EXIT_FAILURE);
233     }
234
235     /* insert your code here */
236     if (semDown(semgid, sh->requestReceived) == -1) {
237         perror("error on the up operation for semaphore access(GL)");
238         exit(EXIT_FAILURE);
239     }
240
241 }
242
243 }

```

Esta função está definida apenas para o primeiro cliente que chegou à mesa. Tal como anteriormente vamos entrar na zona crítica e só aí atualizar as variáveis, *sh->fSt.foodRequest* toma o valor 1 referente a um pedido de comida e libertamos o semáforo *sh->waiterRequest* para que o Waiter possa atender este mesmo pedido.

Atualizamos o estado deste cliente para *FOOD_REQUEST* (3), imprimimos os estados e saímos da região crítica.

Finalmente vamos bloquear a execução do programa até que o Waiter responda ao pedido, quando este responder o programa continuará.

Função waitFood

```

255 static void waitFood (int id)
256 {
257
258     if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
259         perror ("error on the down operation for semaphore access (CT)");
260         exit (EXIT_FAILURE);
261     }
262
263     /* insert your code here */
264
265     if (id != sh->fSt.tableLast)
266     {
267         sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
268         saveState(nFic, &(sh->fSt));
269     }
270
271
272     if (semUp (semgid, sh->mutex) == -1) {                                    /* exit critical region */
273         perror ("error on the down operation for semaphore access (CT)");
274         exit (EXIT_FAILURE);
275     }
276
277     /* insert your code here */
278
279     semDown(semgid, sh->foodArrived);
280
281     if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
282         perror ("error on the down operation for semaphore access (CT)");
283         exit (EXIT_FAILURE);
284     }
285
286     /* insert your code here */
287
288     sh->fSt.st.clientStat[id] = EAT;
289     saveState(nFic, &(sh->fSt));
290
291     if (semUp (semgid, sh->mutex) == -1) {                                    /* exit critical region */
292         perror ("error on the down operation for semaphore access (CT)");
293         exit (EXIT_FAILURE);
294     }
295 }

```

Esta função também é referente aos clientes enquanto estes esperam pelas suas refeições e começam a refeição. Começamos novamente por entrar numa zona crítica e deparamos-nos com uma condição onde vamos mudar o estado do cliente atual para WAIT_FOR_FOOD (4), o último cliente que chegou não entra nesta condição uma vez que o estado já foi alterado para a espera, saímos da zona crítica.

Seguidamente, utilizamos o semáforo *sh->foodArrived* que vai bloquear o programa até que a comida chegue, após isto a execução continuará.

Quando a comida chegar alteramos o estado do cliente para EAT (5), imprimimos os estados e saímos da zona crítica.

Função waitAndPay

```

307 static void waitAndPay (int id)
308 {
309     bool last=false;
310
311     if (semDown (semgid, sh->mutex) == -1) {                               /* enter critical region */
312         perror ("error on the down operation for semaphore access (CT)");
313         exit (EXIT_FAILURE);
314     }
315
316     /* insert your code here */
317     if (sh->fSt.tableLast == id){
318         last = true;
319     }else{
320         last = false;
321     }
322
323     // last = (sh->fSt.tableLast == id);
324
325     sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
326     sh->fSt.tableFinishEat++;
327     saveState(nFic, &(sh->fSt));
328
329
330
331     if (sh->fSt.tableFinishEat == TABLESIZE){
332         for (int i = 0; i < TABLESIZE; i++){
333             if (semUp (semgid, sh->allFinished) == -1) {
334                 perror ("error on the down operation for semaphore access (CT)");
335                 exit (EXIT_FAILURE);
336             }
337         }
338     }
339
340
341
342     if (semUp (semgid, sh->mutex) == -1) {                               /* exit critical region */
343         perror ("error on the down operation for semaphore access (CT)");
344         exit (EXIT_FAILURE);
345     }

```

No início desta função, depois de entrar na zona crítica vamos verificar se o cliente que estamos a executar é o último ou não e alteramos o estado deste cliente para `WAIT_FOR_OTHERS` (6) uma vez que já acabou a sua refeição e está à espera dos restantes, além disso incrementamos a variável `sh->fSt.tableFinishEat` que vai ser imprimida no terminal na coluna FIE (Finish Eating).

Quando todos os clientes acabarem de comer vamos libertar os processos que estão bloqueados pelo semáforo `sh->allFinished` uma vez que todos acabaram a refeição, por fim saímos da zona crítica.

```

346
347 /* insert your code here */
348 if (semDown (semgid, sh->allFinished) == -1) {
349     perror ("error on the down operation for semaphore access (CT)");
350     exit (EXIT_FAILURE);
351 }
352
353 if(last) {
354     if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
355         perror ("error on the down operation for semaphore access (CT)");
356         exit (EXIT_FAILURE);
357     }
358
359     /* insert your code here */
360     sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
361     saveState(nFic, &sh->fSt);
362     sh->fSt.paymentRequest = 1;
363
364     if (semUp (semgid, sh->waiterRequest) == -1) {
365         perror ("error on the down operation for semaphore access (CT)");
366         exit (EXIT_FAILURE);
367     }
368
369     if (semUp (semgid, sh->mutex) == -1) {                                /* exit critical region */
370         perror ("error on the down operation for semaphore access (CT)");
371         exit (EXIT_FAILURE);
372     }
373
374     /* insert your code here */
375     if (semDown (semgid, sh->requestReceived) == -1) {
376         perror ("error on the down operation for semaphore access (CT)");
377         exit (EXIT_FAILURE);
378     }
379 }

```

Após sair da primeira zona crítica, entramos numa zona de espera, o programa só vai prosseguir quando todos os clientes acabarem de comer.

Seguidamente, entramos numa secção apenas referente a quando o cliente atual é o último, aqui entramos novamente numa zona crítica, atualizamos o estado deste cliente para `WAIT_FOR_BILL` (7) , colocamos a variável `sh->fSt.paymentRequest` a 1 e solicitamos que o Waiter venha receber a conta a partir do semáforo `sh->waiterRequest`.

Após sair desta zona crítica, vamos usar o semáforo `sh->requestReceived` para bloquear a execução até o Waiter receber o pagamento, estando tudo pago a execução continua.

```

380
381 if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
382     perror ("error on the down operation for semaphore access (CT)");
383     exit (EXIT_FAILURE);
384 }
385
386 /* insert your code here */
387 // sh->fSt.tableClients--;
388 sh->fSt.st.clientStat[id] = FINISHED;
389 saveState(nFic, &(sh->fSt));
390
391 if (semUp (semgid, sh->mutex) == -1) {                                /* exit critical region */
392     perror ("error on the down operation for semaphore access (CT)");
393     exit (EXIT_FAILURE);
394 }
395 }

```

Por último, entramos novamente numa zona crítica, mudamos os estados dos clientes para `FINISHED` (8) , imprimimos os estados e saímos desta zona.

Waiter

Função waitForClientOrChef

Passando agora para o Waiter a função waitFotClientOrChef é necessária para retornar solicitações do Cliente ou do Chef.

```

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

```

Inicialmente, tal como em funções anteriores entramos numa zona crítica, nesta situação colocamos o estado do Waiter para WAIT_FOR_REQUEST (0) que tal como o nome do estado indica, o Waiter vai esperar por alguma solicitação.

Saindo da zona crítica, vamos bloquear a execução do programa até que o Waiter tenha alguma tarefa, após este momento avançamos.

```

166
167     if (semDown (semgid, sh->mutex) == -1) {                               /* enter critical region */
168         perror ("error on the up operation for semaphore access (WT)");
169         exit (EXIT_FAILURE);
170     }
171
172     /* insert your code here */
173     if (sh->fSt.foodRequest == 1) {
174         ret = FOODREQ;
175         sh->fSt.foodRequest = 0;
176     }
177     if (sh->fSt.foodReady == 1) {
178         sh->fSt.foodReady = 0;
179         ret = FOODREADY;
180     }
181     if (sh->fSt.paymentRequest == 1) {
182         sh->fSt.paymentRequest = 0;
183         ret = BILL;
184     }
185     if (semUp (semgid, sh->mutex) == -1) {                                   /* exit critical region */
186         perror ("error on the down operation for semaphore access (WT)");
187         exit (EXIT_FAILURE);
188     }
189
190     return ret;
191 }

```

Por último, entramos novamente numa zona crítica, aqui vamos verificar que tipo de solicitações foi feita ao Waiter: Um pedido de comida feita pelo Cliente, uma notificação que a comida está pronta feita pelo Chef e um pedido da conta feito pelo Cliente. Sabendo qual o tipo de solicitação vamos guardar na variável `ret` o novo estado do Waiter e retorná-la.

Função `informChef`

```

200 static void informChef ()
201 {
202     if (semDown (semgid, sh->mutex) == -1) {                                /* enter critical region */
203         perror ("error on the up operation for semaphore access (WT)");
204         exit (EXIT_FAILURE);
205     }
206
207     /* insert your code here */
208     sh->fSt.foodOrder = 1;
209     sh->fSt.st.waiterStat = INFORM_CHEF;
210     saveState(nFic, &(sh->fSt));
211
212     if (semUp (semgid, sh->mutex) == -1)                                    /* exit critical region */
213     { perror ("error on the down operation for semaphore access (WT)");
214       exit (EXIT_FAILURE);
215     }
216
217     /* insert your code here */
218     if (semUp (semgid, sh->requestReceived) == -1) {
219         perror ("error on the up operation for semaphore access (WT)");
220         exit (EXIT_FAILURE);
221     }
222     if (semUp (semgid, sh->waitOrder) == -1) {
223         perror ("error on the up operation for semaphore access (WT)");
224         exit (EXIT_FAILURE);
225     }
226 }
227

```

Esta função, tal como o nome indica, está a ser usada para informar o Chef de uma refeição pendente.

Primeiramente, vamos entrar numa região crítica e atualizar as variáveis `sh->fSt.foodOrder` que vai ser 1, `sh->fSt.st.waiterStat` passa a `INFORM_CHEF` (1) e imprimimos estes novos estados.

Posteriormente, usamos os semáforos `sh->requestReceived` e `sh->waitOrder` para acordar os processos, informando o Chef que tem uma solicitação pendente e que o Waiter está pronto e à espera para entregar o pedido.

Função takeFoodToTable

A função takeFoodToTable é executada pelo Waiter para este levar a comida para a mesa e informar os clientes que a comida está pronta.

```

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

```

Inicialmente, entramos numa zona crítica, atualizamos `sh->fSt.st.waiterStat` para `TAKE_TO_TABLE` e imprimimos os novos estados. O ciclo for que procede está a sinalizar todos os clientes que estavam à espera pela comida para acordarem, uma vez que o Waiter está a levar a comida, utilizamos o semáforo `sh->foodArrived`.

Função receivePayment

```

269 static void receivePayment ()
270 {
271     if (semDown (semgid, sh->mutex) == -1) {                               /* enter critical region */
272         perror ("error on the up operation for semaphore access (WT)");
273         exit (EXIT_FAILURE);
274     }
275
276     /* insert your code here */
277
278     sh->fSt.st.waiterStat=RECEIVE_PAYMENT;
279     saveState (nFic, &sh->fSt);
280
281     if (semUp (semgid, sh->mutex) == -1) {                                   /* exit critical region */
282         perror ("error on the down operation for semaphore access (WT)");
283         exit (EXIT_FAILURE);
284     }
285
286     if (semUp (semgid, sh->requestReceived) == -1) {
287         perror ("error on the up operation for semaphore access (WT)");
288         exit (EXIT_FAILURE);
289     }
290 }

```

Nesta função, tal como em anteriores, vamos atualizar variáveis, neste caso `sh->fSt.st.waiterStat` será `RECEIVE_PAYMENT` (3). Usamos também um semáforo para acordar o Cliente e sinalizar que este recebeu o pedido.

Chef

Função waitForOrder

```

116 static void waitForOrder ()
117 {
118     /* insert your code here */
119     if (semDown(semgid, sh->waitOrder) == -1) {                                /* enter critical region */
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.foodOrder = 0;
131     sh->fSt.st.chefStat = COOK;
132     saveState(nFic, &sh->fSt);
133
134     if (semUp(semgid, sh->mutex) == -1) {                                    /* exit critical region */
135         perror("error on the up operation for semaphore access (PT)");
136         exit(EXIT_FAILURE);
137     }
138 }

```

Nesta função, começamos por utilizar o semáforo `sh->waitOrder` que vai fazer o Chef esperar que tenha uma solicitação, após a espera entramos numa zona crítica e colocamos o `sh->fSt.foodOrder` para 0, uma vez que já não vão haver pedidos pendentes e atualizamos o estado do Chef para COOK (1), imprimimos os estados e saímos desta zona.

Função processOrder

```

146 static void processOrder ()
147 {
148     usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));
149
150     if (semDown(semgid, sh->mutex) == -1) {                                    /* enter critical region */
151         perror("error on the up operation for semaphore access (PT)");
152         exit(EXIT_FAILURE);
153     }
154
155     /* insert your code here */
156     sh->fSt.foodReady = 1;
157     sh->fSt.st.chefStat = REST;
158     saveState(nFic, &(sh->fSt));
159
160
161     if (semUp(semgid, sh->mutex) == -1) {                                    /* exit critical region */
162         perror("error on the up operation for semaphore access (PT)");
163         exit(EXIT_FAILURE);
164     }
165
166     /* insert your code here */
167     if (semUp(semgid, sh->waiterRequest) == -1) {
168         perror("error on the up operation for semaphore access (PT)");
169         exit(EXIT_FAILURE);
170     }
171 }

```

Seguidamente, esta função vai começar por colocar o processo em espera a partir da primeira linha, que se vai referir ao cozinheiro a executar a refeição. Posteriormente entramos na zona crítica onde `sh->fSt.foodReady` vai tomar o valor 1, uma vez que a comida está pronta, e alteramos o estado do Chef para REST (2) pois o seu trabalho está finalizado.

Por último, imprimimos os estados e utilizamos o semáforo `sh->waiterRequest` para informar o Waiter que a refeição está pronta, acordando este processo.

Resultados

Durante a implementação do código guiamos-nos pelas diferentes diferentes formas de correr o programa utilizando código pré-compilado pelo professor. Os comandos `make ct` e `make ct_wt` ajudaram-nos a identificar soluções para a existência de *deadlocks* que surgiram na realização do código.

Finalmente, após diversas execuções do código apresentado, provamos que não existem *deadlocks*. Usando o comando `./run.sh` para 100 restaurantes concluímos isso mesmo. O restaurante 100 vai ter o seguinte output:

Run n.º 100		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	1	1	1	2	1	1	0	18	-1		
0	0	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	2	0	18	-1		
0	0	2	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	2	1	3	0	18	-1		
0	0	2	1	1	2	1	1	1	2	1	1	1	1	1	1	1	1	1	1	2	1	4	0	18	-1		
0	0	2	1	2	2	1	1	1	2	1	1	1	1	1	1	1	1	1	1	2	1	5	0	18	-1		
0	0	2	1	2	2	1	1	1	2	1	2	1	1	1	1	1	1	1	1	2	1	6	0	18	-1		
0	0	2	1	2	2	1	1	1	2	1	2	1	1	2	1	1	1	1	1	2	1	7	0	18	-1		
0	0	2	2	2	2	1	1	1	2	1	2	1	1	2	1	1	1	1	1	2	1	8	0	18	-1		
0	0	2	2	2	2	1	1	1	2	1	2	1	1	2	1	1	2	1	1	2	1	9	0	18	-1		
0	0	2	2	2	2	1	1	1	2	1	2	1	1	2	1	2	2	1	1	2	1	10	0	18	-1		
0	0	2	2	2	2	1	1	1	2	1	2	2	1	2	1	2	2	1	1	2	1	11	0	18	-1		
0	0	2	2	2	2	1	1	1	2	1	2	2	1	2	1	2	2	1	2	2	1	12	0	18	-1		
0	0	2	2	2	2	1	1	1	2	2	1	2	2	1	2	1	2	2	1	2	1	13	0	18	-1		
0	0	2	2	2	2	1	1	1	2	2	1	2	2	1	2	2	2	1	2	2	1	14	0	18	-1		
0	0	2	2	2	2	1	1	1	2	2	1	2	2	1	2	2	2	2	2	2	1	15	0	18	-1		
0	0	2	2	2	2	1	1	1	2	2	1	2	2	1	2	2	2	2	2	2	1	16	0	18	-1		
0	0	2	2	2	2	1	2	2	2	1	2	2	1	2	2	2	2	2	2	2	17	0	18	-1			
0	0	2	2	2	2	1	2	2	2	2	2	2	1	2	2	2	2	2	2	2	18	0	18	-1			
0	0	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2	19	0	18	-1			
0	0	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4	2	2	2	2	2	2	2	2	4	4	2	2	2	2	2	2	20	0	18	11			
0	0	2	2	4																							

```
tomas@tomas-HP-Pavilion-Gaming-Laptop-16-a0xxx:~/Desktop/S0/PROJ2/run$
```

Como forma de esclarecer melhor o output que conseguimos e verificar o sentido das diversas mudanças de estado, decidimos imprimir um simples esclarecimento de algumas mudanças de estado. Nesta execução usamos o comando `./filter.sh 1` para excluirmos repetições de estados e aparecer apenas um restaurante:

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
.	.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	-1
.	2	1	0	11	-1
.	2	.	.	2	0	11	-1
.	.	.	.	2	2	.	3	0	11	-1
.	2	4	0	11	-1
.	2	.	.	.	5	0	11	-1
.	2	6	0	11	-1
.	2	7	0	11	-1
.	2	8	0	11	-1
.	2	9	0	11	-1
.	2	.	10	0	11	-1
.	.	2	11	0	11	-1
.	2	12	0	11	-1
.	2	13	0	11	-1
.	2	14	0	11	-1
.	2	15	0	11	-1
.	2	16	0	11	-1
.	2	17	0	11	-1
.	2	18	0	11	-1
.	.	.	2	19	0	11	-1
Todos os amigos chegaram Vão esperar pela comida(4)																									
.	4	20	0	11	8
.	4	0	11	8
.	.	4	0	11	8
O primeiro cliente C11 vai pedir as refeições(3)																									
.	3	0	11	8
.	4	0	11	8
.	4	0	11	8
.	4	0	11	8
.	4	0	11	8
.	4	.	.	.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.	0	11	8
.</																					

Os clientes que acabarem mais cedo progressivamente vão esperando pelos outros(6)

[illegible][illegible][illegible][illegible]

```
tomas@tomas-HP-Pavilion-Gaming-Laptop-16-a0xxx:~/Desktop/SO/PROJ2/run$
```

Conclusão

Para concluir, os semáforos são ferramentas úteis de sincronização e são usados para controlar o acesso a recursos compartilhados em um ambiente multi-threaded. Eles permitem garantir que apenas um processo possa acessar um recurso de cada vez, evitando condições de corrida e outros problemas de sincronização. Embora os semáforos possam ser complexos de implementar, eles são uma ferramenta poderosa para gerenciar processos concorrentes e garantir a integridade de recursos compartilhados no bash.

Com este trabalho conseguimos concluir que ficamos com maior conhecimento acerca de semáforos e sobre as suas utilizações, e agora perceber como os usar uma quantidade diferente de situações.

