Sistemas Operativos

Professor:
José Nuno Panelas Nunes Lau

Problema genérico de gestão de recursos:

Fumadores

Carolina Araújo, 93248
Hugo Paiva, 93195
Igual distribuição de trabalho
entre os dois membros

DETI
Universidade de Aveiro
30-12-2019

Índice

1 Introdução

Este trabalho prático foi desenvolvido com o objetivo de compreender os mecanismos associados à execução de processos e *threads*.

Para empreender este propósito, foi pedido que se solucionasse um problema que envolve várias entidades que terão de colaborar entre si para um bom funcionamento do programa: os fumadores, os watchers e o agente. Dito isto, implementou-se um programa em *C* que simula e soluciona o problema recorrendo a semáforos e a memória partilhada, de modo a sincronizar os vários processos independentes.

2 Contextualização

2.1 Processos e a utilização de múltiplas threads

Uma thread é a unidade básica da utilização do CPU. Um processo que utilize múltiplas threads pode, portanto, realizar mais do que uma tarefa ao mesmo tempo. Isto traz benefícios como o **aumento na capacidade de resposta**, sendo que um programa pode continuar a correr mesmo que algumas threads estejam bloqueadas ou a realizar uma operação mais demorada. Ocorre também **partilha de recursos**, uma vez que os processos podem usá-los através da memória partilhada ou através do envio de mensagens. Há, de facto, um **menor gasto de memória**, uma vez que as múltiplas threads de um mesmo processo partilham entre si memória e dados, tornando-se mais eficiente usufruir das mesmas do que criar diferentes processos para realizar as mais variadas tarefas. Por fim, verifica-se também que em arquiteturas com **multiprocessadores**, as múltiplas threads podem estar a **correr em paralelo** nos diferentes processadores, o que, novamente, aumenta a eficiência de um processo.

O **cancelamento de threads** dá-se quando a tarefa realizada por uma thread é terminada antes de ser completada. A thread cancelada costuma ser referida como **target thread**, podendo terminar de duas maneiras: imediatamente, através de uma outra thread (cancelamento assíncrono) ou verificando periodicamente se deve terminar (cancelamento síncrono). No entanto, isto pode acarretar problemas sendo que uma thread pode ser cancelada enquanto está num ponto vital, especialmente quando se trata do caso assíncrono, o que pode resultar numa recolha não total dos recursos do processo.

2.2 Sincronização

Considere-se agora um conjunto de processos. Caso esses partilhem uma variável que, individualmente, será por eles manipulada, pode acontecer o output acabar por não ser o esperado porque depende da ordem de manipulação dessa mesma variável. Isto é chamado de **condição de corrida**. Para combater algo do género, é necessário garantir que apenas um processo de cada vez pode estar a modificar esse tipo de variáveis partilhadas - ou seja, torna-se mandatório haver algum tipo de **sincronização**.

Cada processo tem um pedaço de código onde irá, possivelmente, alterar uma variável comum a todos os processos. Esta é chamada a **região crítica**. O mais importante, quando se trata de sincronização dos diferentes processos, é garantir que quando um deles entra na região crítica, mais nenhum pode entrar na região crítica que a si lhe compete. No entanto, se nenhum processo se encontra em execução numa destas regiões, caso um deles queira aceder, selecionar qual dos processos pode entrar é uma decisão que tem de ser tomada e não pode ser adiada indefinidamente. Terá, também, de haver um número limite de vezes que um dado processo tem acesso à região, enquanto outro que tenha previamente pedido e cujo acesso ainda não lho foi garantido.

Os **semáforos**, uma solução hardware ao problema das regiões críticas, tornam-se vitais para sincronizar as tarefas realizadas pelas diferentes threads de um mesmo processo, permitindo também uma comunicação eficaz e fulcral entre as mesmas, para um bom funcionamento global. Um semáforo contém uma variável do tipo *integer* que pode ser acedida de através de duas operações default: *down()* e *up()*, para além de quando se dá a sua inicialização. Estas duas operações têm como objetivo, respetivamente, entrar e sair na região crítica, ou esperar por e entregar notificações. Modificações a essa variável através dessas funções são feitas de maneira indivisível. Isto é, quando uma das threads altera este valor, não poderá ser modificado simultaneamente por mais nenhuma. No caso de semáforos do tipo **mutex**, são semáforos binários cujo valor pode apenas ser 0 ou 1, garantido sempre **exclusão mútua**. Existem também semáforos contadores que podem ser utilizados para controlar o número de acessos a regiões críticas num dado processo.

2.3 O problema dos Fumadores

Este problema está relacionado com uma gestão de recursos envolvendo três Fumadores com necessidades distintas para fumar, sendo que cada um deles possui apenas 1 recurso de fonte inesgotável. Para esta gestão existe uma entidade que disponibiliza/gera recursos e outras que necessitam/gastam recursos, sendo que estas necessidades envolvem vários recursos de tipos distintos.

A dificuldade está em como fazer com que as entidades que necessitam dos recursos os usem nas alturas certas, sem que a entidade geradora de recursos faça a notificação direta às entidades gastadoras. Estas notificações vão ocorrer quando os pacotes que as entidades gastadoras necessitam estiverem completos, e sem que tenham de fazer verificações desnecessárias para comprovar se já existem os recursos precisos.

Para a resolução do problema existem algumas diretrizes:

1. O *Agent* notifica o *Watcher* responsável por cada ingrediente sempre que produz um ingrediente desse tipo.
2. O *Watcher* notifica o *Smoker* sempre que ele puder fumar.
3. Os *Watchers* partilham informação entre si para comprovar que os 2 ingredientes estão disponíveis e poderem notificar o *Smoker* correto.
4. O *Agent* apenas inicia um novo pacote quando o *Smoker* já recolheu os ingredientes do pacote anterior.
5. Após a produção de 5 pacotes, o *Agent* termina a produção de recursos, notificando os *Watchers* para que terminem. Estes, por sua vez, notificam os *Smokers* para terminarem a sua execução.

Na implementação da simulação deste problema foi utilizado o código-fonte disponibilizado pelo docente da disciplina onde já se encontravam definidos vários dados necessários à resolução.

Os recursos, tratados como ingredientes nesta implementação, que estão envolvidos neste processo, são Tabaco, Fósforos e Papel. No ficheiro *probConst.h* é possível encontrar estes ingredientes já definidos, bem como outros parâmetros gerais, úteis à implementação:

```
14  /* Generic parameters */
15
16  /** \brief total number of ingredients */
17  #define  NUMINGREDIENTS   3
18  /** \brief total number of smokers */
19  #define  NUMSMOKERS       3
20
21  /** \brief total number of orders to be generated by agent, each order has 2 different ingredients */
22  #define  NUMORDERS        5
23
24  /** \brief TOBBACO ingredient id */
25  #define  TOBACCO          0
26  /** \brief MATCHES ingredient id */
27  #define  MATCHES          1
28  /** \brief PAPER ingredient id  */
29  #define  PAPER            2
30
31  /** \brief id of smoker that always has TOBACCO */
32  #define  HAVETOBACCO      0
33  /** \brief id of smoker that always has MATCHES */
34  #define  HAVEMATCHES      1
35  /** \brief id of smoker that always has PAPER */
36  #define  HAVEPAPER        2
```

Figure 1: Parâmetros gerais definidos no ficheiro *probConst.h*

Vão existir três entidades com as seguintes funções:

- ***Agent*** - Entidade que produz recursos em pacotes de 2 ingredientes distintos, ou seja, sempre que é produzido um pacote, apenas um *Smoker* pode fumar.
- ***Watcher*** - Entidade responsável por verificar se, após a produção de um novo ingrediente de um pacote do *Agent*, há algum *Smoker* que pode fumar. Existe um watcher por cada ingrediente.
- ***Smoker*** - Entidade que representa um *Smoker* e que tem uma fonte inesgotável de um destes recursos, necessitando apenas dos outros 2.

Cada uma destas entidades têm vários estados associados à tarefa que estão atualmente a executar. Todos estes estados estão definidos, também, no ficheiro *probConst.h*:

```
39  /* Agent state constants */
40
41  /** \brief agent initial state, preparing pack of 2 ingredients */
42  #define  PREPARING        1
43  /** \brief agent waiting for smoker to finish rolling cigarette */
44  #define  WAITING_CIG      2
45  /** \brief agent is closing factory */
46  #define  CLOSING_A        3
47
48  /* Wachers state constants */
49
50  /** \brief watcher waiting for ingredient (each watcher is responsible for a different ingredient) */
51  #define  WAITING_ING      0
52  /** \brief watcher updating reservations */
53  #define  UPDATING         1
54  /** \brief watcher informing smoker that he can start rolling */
55  #define  INFORMING        2
56  /** \brief watcher is closing */
57  #define  CLOSING_W        3
58
59  /* Smokers state constants */
60
61  /** \brief smoker is waiting for the 2 missing ingredients  */
62  #define  WAITING_2ING     0
63  /** \brief smoker is rolling cigarette  */
64  #define  ROLLING          1
65  /** \brief smoker is smoking  */
66  #define  SMOKING          2
67  /** \brief smoker is closing  */
68  #define  CLOSING_S        3
```

Figure 2: Estados das várias entidades referentes ao problema definidos no ficheiro *probConst.h*

O conteúdo da memória partilhada é definido na estrutura *FULL_STAT* e com os 8 *ids* dos semáforos usados durante a implementação, no ficheiro *sharedDataSync.h*:

```
23  /**
24   *  \brief Definition of <em>shared information</em> data type.
25   */
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 watchers to wait for agent - val = 0 */
34            unsigned int ingredient[NUMINGREDIENTS];
35            /** \brief identification of semaphore used by agent to wait for smoker to finish rolling - val = 0 */
36            unsigned int waitCigarette;
37            /** \brief identification of semaphore used by smoker to wait for watchers - val = 0  */
38            unsigned int wait2Ings[NUMSMOKERS];
39
40          } SHARED_DATA;
41
42  /** \brief number of semaphores in the set */
43  #define SEM_NU               ( 2 + NUMINGREDIENTS + NUMSMOKERS )
44
45  #define MUTEX                  1
46  #define WAITCIGARETTE          2
47  #define INGREDIENT             (WAITCIGARETTE + 1)
48  #define WAIT2INGS              (INGREDIENT + NUMINGREDIENTS)
```

Figure 3: Definição dos tipos de dados e semáforos da memória partilhada no ficheiro *sharedDataSync.h*

A definição das estruturas *FULL_STAT* e *STAT*, usadas na memória partilhada, com os tipos de dados de todo o problema bem como o estado das entidades, encontram-se no ficheiro *probDataStruct.h*:

```
20  /**
21   *  \brief Definition of <em>state of the intervening entities</em> data type.
22   */
23  typedef struct {
24      /** \brief agent state */
25      unsigned int agentStat;
26      /** \brief watchers state */
27      unsigned int watcherStat[NUMINGREDIENTS];
28      /** \brief smokers state */
29      unsigned int smokerStat[NUMSMOKERS];
30
31  } STAT;
32
33
34  /**
35   *  \brief Definition of <em>full state of the problem</em> data type.
36   */
37  typedef struct
38  {   /** \brief state of all intervening entities */
39      STAT st;
40
41      /** \brief number of ingredients */
42      int nIngredients;
43
44      /** \brief number of orders to be performed by agent (each order includes a pack of 2 ingredients) */
45      int nOrders;
46
47      /** \brief number of smokers */
48      int nSmokers;
49
50      /** \brief flag used by agent to close factory */
51      bool closing;
52
53      /** \brief inventory of ingredients */
54      int ingredients[NUMINGREDIENTS];
55
56      /** \brief number of ingredients already reserved by watcher */
57      int reserved[NUMINGREDIENTS];
58
59      /** \brief number of cigarettes each smoker smoked */
60      int nCigarettes[NUMSMOKERS];
61
62  } FULL_STAT;
```

Figure 4: Definição das estruturas *FULL_STAT* e *STAT* no ficheiro *probDataStruct.h*

3 Implementação

Recorrendo a semáforos e a memória partilhada, de modo a sincronizar os vários processos independentes, foi implementada uma resolução do problema a partir do código-fonte do professor. A explicação da implementação irá, portanto, inserir-se no código feito pelos alunos, nos locais e ficheiros definidos pelo docente da disciplina.

A utilização dos semáforos ao longo da solução serviu principalmente para controlar o acesso à memória partilhada, evitando assim potenciais colisões que as 3 entidades pudessem vir a ter durante a sua execução. As notificações trocadas entre as entidades foram também implementadas usando semáforos, permitindo assim a contínua execução do programa. Para uma melhor interpretação deste funcionamento, foi feita uma tabela que associa cada semáforo à sua função durante a resolução, bem como o contexto em que é utilizado:

	ENTIDADE		FUNÇÃO		SITUAÇÃO	
	UP	DOWN	UP	DOWN	UP	DOWN
mutex	Todas	Todas	Todas	Todas	Ao sair de uma região crítica	Ao entrar numa região crítica
ingredient	- Agente	- Watcher	- (A) prepareIngredients() - (A) closeFactory()	- (W) waitForIngredient()	- (A) Ao haver disponibilidade de 2 ingredientes, os watchers respetivos são notificados - (A) Todos os watchers são notificados de que a fábrica fechou	- (W) Espera por uma notificação do agente para verificar se vai fechar ou para esperar pelo fabrico do ingrediente que lhe corresponde
waitCigarette	- Smoker	- Agente	- (S) rollingCigarette()	- (A) waitForCigarette()	- (S) Notifica o agente que acabou de enrolar	- (A) Espera que o smoker acabe de enrolar
wait2Ings	- Watcher	- Smoker	- (W) waitForIngredient() - (W) informSmoker()	- (S) waitForIngredients()	- (W) Notifica o smoker que a fábrica vai fechar - (W) Notifica o smoker de que pode começar a enrolar	- (S) Espera por uma notificação do watcher para poder começar a enrolar ou para saber se a fábrica fechou

Figure 5: Tabela dos semáforos existentes e a sua utilização

3.1 *Agent*

Começando por implementar a solução da entidade *Agent*, foi necessário alterar o ficheiro *semSharedMemAgent.c* nos locais assinalados. As funções *prepareIngredients()*, *waitForCigarette()* e *closeFactory()* foram modificadas da maneira que se segue.

3.1.1 *prepareIngredients()*

Nesta função o *Agent* prepara 2 ingredientes. Para isso, após entrar na região crítica com o uso da função *semDown()*, é atualizado o seu estado para *PREPARING*, escolhendo-se, posteriormente e de forma aleatória, os 2 ingredientes a fazerem parte do novo pacote. Após esta escolha, o inventário é atualizado com as novas existências, guardando-se estas novas alterações na memória partilhada. Já fora da região crítica, ambos os *Watchers* referentes aos ingredientes gerados são notificados que os recursos destes estão disponíveis. Isto é feito a partir do desbloqueio de dois semáforos *ingredient*, pertencentes a cada ingrediente, com a função *semUp()*.

```
135  static void prepareIngredients()
136  {
137
138      if (semDown(semgid, sh->mutex) == -1)
139      { /* enter critical region */
140          perror("error on the down operation for semaphore access (AG)");
141          exit(EXIT_FAILURE);
142      }
143      //Vou mudar o estado do agente para preparing porque vou começar a preparar os ingredientes
144      sh->fSt.st.agentStat = PREPARING;
145
146      int ing1 = rand() % 3;
147      int ing2 = rand() % 3;
148
149      while (ing1 == ing2)
150      {
151
152          ing2 = rand() % 3;
153      }
154      // guardar os ingredientes fabricados no inventário
155      sh->fSt.ingredients[ing1]++;
156      sh->fSt.ingredients[ing2]++;
157      saveState(nFic, &sh->fSt);
158
159      if (semUp(semgid, sh->mutex) == -1)
160      { /* leave critical region */
161          perror("error on the up operation for semaphore access (AG)");
162          exit(EXIT_FAILURE);
163      }
164
165      if (semUp(semgid, sh->ingredient[ing1]) == -1) //Ingrediente 1 passou a estar disponível. Vou notificar o watcher[ing1]
166      {
167          perror("error on the up operation for semaphore access (AG)");
168          exit(EXIT_FAILURE);
169      }
170
171      if (semUp(semgid, sh->ingredient[ing2]) == -1) //Ingrediente 2 passou a estar disponível. Vou notificar o watcher[ing2]
172      {
173          perror("error on the up operation for semaphore access (AG)");
174          exit(EXIT_FAILURE);
175      }
176  }
```

Figure 6: Função *prepareIngredients()*

3.1.2 *waitForCigarette()*

Ao executar *waitForCigarette()*, o *Agent* vai esperar que o *Smoker* acabe de enrolar o cigarro. Isto é alcançado entrando na região crítica, de modo a que o *Agent* altere o seu estado para *WAITING_CIG*, guardando-o depois na memória partilhada. Fora da região crítica, é feito um *semDown()* para bloquear o semáforo *waitCigarette*, o qual só é liberto quando o *Smoker* acabar de enrolar.

```
184   static void waitForCigarette()
185   {
186       if (semDown(semgid, sh->mutex) == -1)
187       { /* enter critical region */
188           perror("error on the down operation for semaphore access (AG)");
189           exit(EXIT_FAILURE);
190       }
191       //mudar o estado
192       sh->fSt.st.agentStat = WAITING_CIG;
193       saveState(nFic, &sh->fSt);
194
195       if (semUp(semgid, sh->mutex) == -1)
196       { /* leave critical region */
197           perror("error on the up operation for semaphore access (AG)");
198           exit(EXIT_FAILURE);
199       }
200       //O Agent vai ficar à espera que o smoker acabe de enrolar
201       if (semDown(semgid, sh->waitCigarette) == -1)
202       {
203           perror("error on the down operation for semaphore access (AG)");
204           exit(EXIT_FAILURE);
205       }
206   }
```

Figure 7: Função *waitForCigarette()*

3.1.3 *closeFactory()*

Na função *closeFactory()* o *Agent* vai terminar o fabrico de ingredientes, fechando a fábrica. Assim, dentro da região crítica, este atualiza o seu estado para *CLOSING_A* e altera a flag *closing* para *true* guardando estas alterações em memória partilhada. Fora da região crítica, são notificados os 3 *Watchers* desbloqueando os semáforos referentes a cada ingrediente. Desta forma, todos os *Watchers* vão entrar em funcionamento e verificar que a fábrica está a fechar.

```
213  static void closeFactory()
214  {
215      if (semDown(semgid, sh->mutex) == -1)
216      { /* enter critical region */
217          perror("error on the down operation for semaphore access (AG)");
218          exit(EXIT_FAILURE);
219      }
220      //mudar o estado
221      sh->fSt.st.agentStat = CLOSING_A;
222      sh->fSt.closing = true;
223      saveState(nFic, &sh->fSt);
224
225      if (semUp(semgid, sh->mutex) == -1)
226      { /* leave critical region */
227          perror("error on the up operation for semaphore access (AG)");
228          exit(EXIT_FAILURE);
229      }
230
231      //Como os watchers estão à espera de ingredientes, ao notificar que os ingredientes estão disponíveis vou acordá-los de modo a que posteriormente eles também terminem
232
233      if (semUp(semgid, sh->ingredient[0]) == -1)
234      {
235          perror("error on the up operation for semaphore access (AG)");
236          exit(EXIT_FAILURE);
237      }
238
239      if (semUp(semgid, sh->ingredient[1]) == -1)
240      {
241          perror("error on the up operation for semaphore access (AG)");
242          exit(EXIT_FAILURE);
243      }
244
245      if (semUp(semgid, sh->ingredient[2]) == -1)
246      {
247          perror("error on the up operation for semaphore access (AG)");
248          exit(EXIT_FAILURE);
249      }
250  }
```

Figure 8: Função *closeFactory()*

3.2 *Watcher*

Ao implementar a solução da entidade *Watcher* foi necessário alterar o ficheiro *semSharedMemWatcher.c* nos locais assinalados. As funções *waitForIngredient()*, *updateReservations()* e *informSmoker()* foram modificadas da maneira que se segue.

3.2.1 *waitForIngredient()*

Nesta função o *Watcher* espera que o seu ingrediente esteja pronto. Para isso, após entrar na região crítica, é atualizado o seu estado para *WAITING_ING*, guardando-o na memória partilhada. Já fora da região crítica, é bloqueado o semáforo referente ao ingrediente desta entidade, ficando esta à espera da disponibilidade desse ingrediente. Quando este semáforo voltar a estar desbloqueado, volta-se a entrar na região crítica, verificando-se se a flag *closing* está a *true*. Nesse caso, é atualizado o estado do *Watcher* para *CLOSING_W*, guardando-se na memória partilhada e mudando o *return* para *false* Notifica-se, ainda, antes de sair da região crítica, o *Smoker* com o mesmo *id* que o *Watcher* em questão, para este também verificar que a fábrica está a fechar. Em caso contrário, não é feito nada.

```
144  static bool waitForIngredient(int id)
145  {
146      bool ret = true;
147
148      if (semDown(semgid, sh->mutex) == -1)
149      { /* enter critical region */
150          perror("error on the down operation for semaphore access (WT)");
151          exit(EXIT_FAILURE);
152      }
153
154      //mudar estado
155      sh->fSt.st.watcherStat[id] = WAITING_ING;
156      saveState(nFic, &sh->fSt);
157
158      if (semUp(semgid, sh->mutex) == -1)
159      { /* exit critical region */
160          perror("error on the up operation for semaphore access (WT)");
161          exit(EXIT_FAILURE);
162      }
163
164      //Watcher vai esperar por uma notificação do Agent
165      if (semDown(semgid, sh->ingredient[id]) == -1)
166      { /* enter critical region */
167          perror("error on the down operation for semaphore access (WT)");
168          exit(EXIT_FAILURE);
169      }
170
171      if (semDown(semgid, sh->mutex) == -1)
172      { /* enter critical region */
173          perror("error on the down operation for semaphore access (WT)");
174          exit(EXIT_FAILURE);
175      }
176
177      //verificar se está a fechar
178      if (sh->fSt.closing == true)
179      {
180          sh->fSt.st.watcherStat[id] = CLOSING_W;
181          saveState(nFic, &sh->fSt);
182          ret = false;
183
184          //Notificar o smoker
185          if (semUp(semgid, sh->wait2Ings[id]) == -1)
186          {
187              perror("error on the up operation for semaphore access (WT)");
188              exit(EXIT_FAILURE);
189          }
190      }
191      if (semUp(semgid, sh->mutex) == -1)
192      { /* exit critical region */
193          perror("error on the up operation for semaphore access (WT)");
194          exit(EXIT_FAILURE);
195      }
196
197      return ret;
198  }
```

Figure 9: Função *waitForIngredient()*

3.2.2 *updateReservations()*

Ao executar *updateReservations()* o *Watcher* atualiza as reservas na memória partilha e verifica se algum *Smoker* pode fumar. Isto é alcançado entrando na região crítica, de modo a que o *Watcher* altere o seu estado para *UPDATING* e incremente 1 à posição correspondente ao seu ingrediente no *array* dos ingredientes reservados, guardando estes dados na memória partilhada. Ainda dentro da região crítica, é verificado se algum *Smoker* pode fazer um cigarro. Como tal, é analisado o *array* de ingredientes reservados e, caso haja dois ingredientes com uma ou mais reservas, a função passa a retornar o *id* do *Smoker* que necessita desses mesmos ingredientes.

```
211  static int updateReservations(int id)
212  {
213      int ret = -1;
214
215      if (semDown(semgid, sh->mutex) == -1)
216      { /* enter critical region */
217          perror("error on the up operation for semaphore access (WT)");
218          exit(EXIT_FAILURE);
219      }
220
221      // mudar o estado
222      sh->fSt.st.watcherStat[id] = UPDATING;
223      // reservar o ingrediente deste watcher
224      sh->fSt.reserved[id]++;
225
226      // Veriicar qual o smoker que pode fazer um cigarro
227      if (sh->fSt.reserved[TOBACCO] > 0 && sh->fSt.reserved[PAPER] > 0)
228          ret = MATCHES;
229      else if (sh->fSt.reserved[TOBACCO] > 0 && sh->fSt.reserved[MATCHES] > 0)
230          ret = PAPER;
231      else if (sh->fSt.reserved[MATCHES] > 0 && sh->fSt.reserved[PAPER] > 0)
232          ret = TOBACCO;
233
234      saveState(nFic, &sh->fSt);
235
236      if (semUp(semgid, sh->mutex) == -1)
237      { /* exit critical region */
238          perror("error on the down operation for semaphore access (WT)");
239          exit(EXIT_FAILURE);
240      }
241
242      return ret;
243  }
```

Figure 10: Função *updateReservations(*

3.2.3 *informSmoker()*

Na função *informSmoker()* o *Watcher* vai informar o *Smoker* que pode usar os ingredientes disponíveis para enrolar um cigarro. Assim, dentro da região crítica, é atualizado o seu estado para *INFORMING* e retirada uma unidade aos ingredientes prestes a ser utilizados pelo *Smoker* do *array* dos ingredientes reservados, guardando estas alterações em memória partilhada. Fora da região crítica, é notificado o *Smoker* que pode enrolar um cigarro através da variável *smokerReady* que foi retornada na última função analisada, aliada ao desbloqueio do semáforo *wait2Ings* referente ao *Smoker* em questão.

```
254  static void informSmoker(int id, int smokerReady)
255  {
256      if (semDown(semgid, sh->mutex) == -1)
257      { /* enter critical region */
258          perror("error on the down operation for semaphore access (WT)");
259          exit(EXIT_FAILURE);
260      }
261
262      // mudar o estado
263      sh->fSt.st.watcherStat[id] = INFORMING;
264
265      // Vou informar o smoker que pode fazer um cigarro mas antes retiro estes ingredientes dos reservados pois irão ser utilizados
266      int other_ing1 = (smokerReady + 1) % 3;
267      int other_ing2 = (smokerReady + 2) % 3;
268      sh->fSt.reserved[other_ing1]--;
269      sh->fSt.reserved[other_ing2]--;
270
271      saveState(nFic, &sh->fSt);
272
273      if (semUp(semgid, sh->mutex) == -1)
274      { /* exit critical region */
275          perror("error on the up operation for semaphore access (WT)");
276          exit(EXIT_FAILURE);
277      }
278
279      // Notificar o smoker que pode fazer um cigarro
280      if (semUp(semgid, sh->wait2Ings[smokerReady]) == -1)
281      { /* exit critical region */
282          perror("error on the up operation for semaphore access (WT)");
283          exit(EXIT_FAILURE);
284      }
285  }
```

Figure 11: Função *informSmoker()*

3.3 *Smoker*

Para implementar a solução da entidade *Smoker* foi necessário alterar o ficheiro *semSharedMemSmoker.c* nos locais assinalados. As funções *waitForIngredients()*, *rollingCigarette()* e *smoke()* foram modificadas da maneira que se segue.

3.3.1 *waitForIngredients()*

Na função *waitForIngredients()* o *Smoker* espera pelos 2 ingredientes que ele não tem. Assim, dentro da região crítica, este atualiza o seu estado para *WAITING_2ING* e guarda em memória partilhada. Fora da região crítica, é bloqueado um semáforo, de modo ao *Smoker* esperar que o *Watcher* o notifique da disponibilidade dos ingredientes. Posteriomente, volta-se a entrar na região crítica verificando-se se a flag *closing* está a *true*. Em caso afirmativo, é alterado o estado para *CLOSING_S* e o retorno da função para *false*, guardando estes dados na memória partilhada. Em caso contrário, o *Smoker* vai retirar os ingredientes que irá usar do *array* dos ingredientes.

```
159  static bool waitForIngredients(int id)
160  {
161      bool ret = true;
162
163      if (semDown(semgid, sh->mutex) == -1)
164      { /* enter critical region */
165          perror("error on the down operation for semaphore access (SM)");
166          exit(EXIT_FAILURE);
167      }
168
169      sh->fSt.st.smokerStat[id] = WAITING_2ING;
170      saveState(nFic, &sh->fSt);
171
172      if (semUp(semgid, sh->mutex) == -1)
173      { /* exit critical region */
174          perror("error on the up operation for semaphore access (SM)");
175          exit(EXIT_FAILURE);
176      }
177
178      // esperar por uma notificação do watcher
179      if (semDown(semgid, sh->wait2Ings[id]) == -1)
180      {
181          perror("error on the down operation for semaphore access (SM)");
182          exit(EXIT_FAILURE);
183      }
184
185      if (semDown(semgid, sh->mutex) == -1)
186      { /* enter critical region */
187          perror("error on the down operation for semaphore access (SM)");
188          exit(EXIT_FAILURE);
189      }
190      if (sh->fSt.closing == true)
191      {
192          sh->fSt.st.smokerStat[id] = CLOSING_S;
193          ret = false;
194      }
195      else
196      { // caso a notificação fosse para ele poder enrolar
197          // descobrir o id dos ingredientes que o não smoker tem sempre
198          int other_ing1 = (id + 1) % 3;
199          int other_ing2 = (id + 2) % 3;
200          // retirar esses ingredientes pois serão usados por este smoker
201          sh->fSt.ingredients[other_ing1]--;
202          sh->fSt.ingredients[other_ing2]--;
203      }
204      saveState(nFic, &sh->fSt);
205
206      if (semUp(semgid, sh->mutex) == -1)
207      { /* exit critical region */
208          perror("error on the up operation for semaphore access (SM)");
209          exit(EXIT_FAILURE);
210      }
211
212      return ret;
213  }
```

Figure 12: Função *waitForIngredients()*

3.3.2 *rollingCigarette()*

Ao executar *rollingCigarette()* o *Smoker* vai enrolar um cigarro. Isto é alcançado entrando na região crítica, de modo a que o *Smoker* altere o seu estado para *ROLLING*, guardando-o depois na memória partilhada. Fora da região crítica, se o tempo para enrolar o cigarro, gerado anteriormente, for maior que 0, o processo é suspenso durante esse tempo através da função *usleep*. Antes da função terminar, é desbloqueado o semáforo *waitCigarette* através do qual é notificado o *Agent* que o *Smoker* acabou de enrolar.

```
223  static void rollingCigarette(int id)
224  {
225      double rollingTime = 100.0 + normalRand(30.0);
226
227      if (semDown(semgid, sh->mutex) == -1)
228      { /* enter critical region */
229          perror("error on the up operation for semaphore access (SM)");
230          exit(EXIT_FAILURE);
231      }
232
233      sh->fSt.st.smokerStat[id] = ROLLING;
234      saveState(nFic, &sh->fSt);
235
236      if (semUp(semgid, sh->mutex) == -1)
237      { /* exit critical region */
238          perror("error on the down operation for semaphore access (SM)");
239          exit(EXIT_FAILURE);
240      }
241
242      if (rollingTime > 0.0)
243      { // vai parar durante o tempo de enrolar
244          usleep(rollingTime);
245      }
246
247      if (semUp(semgid, sh->waitCigarette) == -1)
248      { //Notificar o agente que acabou de enrolar
249          perror("error on the up operation for semaphore access (SM)");
250          exit(EXIT_FAILURE);
251      }
252  }
```

Figure 13: Função *rollingCigarette()*

3.3.3 *smoke()*

Na função *smoke()* o *Smoker* vai fumar. Assim, na região crítica, é alterado o seu estado para *SMOKING* e incrementado 1 à posição correspondente ao seu *id* no *array* dos cigarros fumados, guardando estes dados na memória partilhada. Já fora da região crítica, é gerado um tempo para fumar e se este for mais que 0, o processo é suspenso durante esse tempo através da função *usleep*.

```
262  static void smoke(int id)
263  {
264      if (semDown(semgid, sh->mutex) == -1)
265      { /* enter critical region */
266          perror("error on the down operation for semaphore access (SM)");
267          exit(EXIT_FAILURE);
268      }
269      // alterar o estado e aumentar o número de cigarros fumados
270      sh->fSt.st.smokerStat[id] = SMOKING;
271      sh->fSt.nCigarettes[id]++;
272      saveState(nFic, &sh->fSt);
273
274      if (semUp(semgid, sh->mutex) == -1)
275      { /* exit critical region */
276          perror("error on the up operation for semaphore access (SM)");
277          exit(EXIT_FAILURE);
278      }
279
280      double smokingTime = 100.0 + normalRand(30.0);
281      if (smokingTime > 0.0)
282      {//vai parar durante o tempo de fumar
283          usleep(smokingTime);
284      }
285
286  }
```

Figure 14: Função *smoke()*

4 Resultados

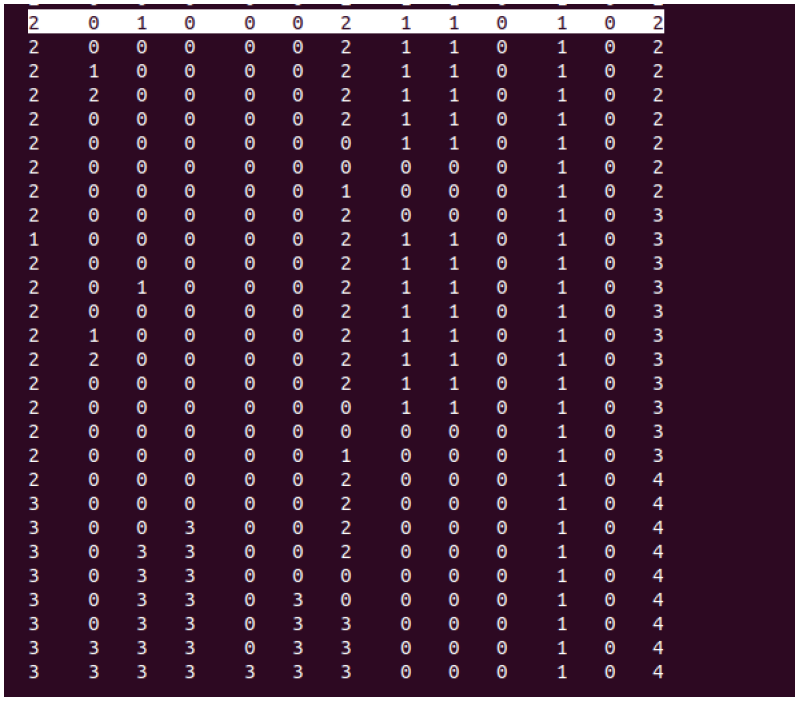
Ao longo da criação de uma solução para este problema foram sendo realizados testes para confirmar que se estava andar na direção certa. Teve-se sempre o cuidado de testar entidade a entidade. Assim que se pensava ter a resolução correta de uma das mesmas, era comparado o output resultante, através de *make sm*, *make ag* e *make wt* na pasta *src*, com o do professor (*make all_bin*) para verificar quaisquer discrepâncias que pudessem ser preocupantes e indicadoras de um ou mais erros.

Foram feitas 1000 execuções do programa, através do script *run.sh* disponibilizado pelo professor, de modo a assegurar as condições referidas em cima. A imagem que se segue mostra o resultado da primeira execução, sendo que todas as execuções podem ser encontradas no ficheiro *resultados.txt* na raiz da entrega.

```
paiva@ubuntu:~/Projetos/so_p2/run$ ./run.sh > resultados.txt
paiva@ubuntu:~/Projetos/so_p2/run$ head -75 resultados.txt

Run n.º 1
                    Smokers - Description of the internal state

 AG  W00 W01 W02  S00 S01 S02  I00 I01 I02  C00 C01 C02
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   0   0    0   0   0
  1    0   0   0    0   0   0    0   1   1    0   0   0
  2    0   0   0    0   0   0    0   1   1    0   0   0
  2    0   0   1    0   0   0    0   1   1    0   0   0
  2    0   0   0    0   0   0    0   1   1    0   0   0
  2    0   1   0    0   0   0    0   1   1    0   0   0
  2    0   2   0    0   0   0    0   1   1    0   0   0
  2    0   0   0    0   0   0    0   1   1    0   0   0
  2    0   0   0    0   0   0    0   0   0    0   0   0
  2    0   0   0    1   0   0    0   0   0    0   0   0
  2    0   0   0    2   0   0    0   0   0    1   0   0
  1    0   0   0    2   0   0    1   1   0    1   0   0
  2    0   0   0    2   0   0    1   1   0    1   0   0
  2    0   1   0    2   0   0    1   1   0    1   0   0
  2    0   0   0    2   0   0    1   1   0    1   0   0
  2    1   0   0    2   0   0    1   1   0    1   0   0
  2    2   0   0    2   0   0    1   1   0    1   0   0
  2    0   0   0    2   0   0    1   1   0    1   0   0
  2    0   0   0    2   0   0    0   0   0    1   0   0
  2    0   0   0    2   0   1    0   0   0    1   0   0
  2    0   0   0    0   0   1    0   0   0    1   0   0
  2    0   0   0    0   0   2    0   0   0    1   0   1
  1    0   0   0    0   0   2    1   1   0    1   0   1
  2    0   0   0    0   0   2    1   1   0    1   0   1
  2    0   1   0    0   0   2    1   1   0    1   0   1
  2    0   0   0    0   0   2    1   1   0    1   0   1
  2    1   0   0    0   0   2    1   1   0    1   0   1
  2    2   0   0    0   0   2    1   1   0    1   0   1
  2    0   0   0    0   0   2    1   1   0    1   0   1
  2    0   0   0    0   0   0    1   1   0    1   0   1
  2    0   0   0    0   0   0    0   0   0    1   0   1
  2    0   0   0    0   0   1    0   0   0    1   0   1
  2    0   0   0    0   0   2    0   0   0    1   0   2
  1    0   0   0    0   0   2    1   1   0    1   0   2
  2    0   0   0    0   0   2    1   1   0    1   0   2
  2    0   1   0    0   0   2    1   1   0    1   0   2
```



2	0	1	0	0	0	2	1	1	0	1	0	2
2	0	0	0	0	0	2	1	1	0	1	0	2
2	1	0	0	0	0	2	1	1	0	1	0	2
2	2	0	0	0	0	2	1	1	0	1	0	2
2	0	0	0	0	0	2	1	1	0	1	0	2
2	0	0	0	0	0	0	1	1	0	1	0	2
2	0	0	0	0	0	0	0	0	0	1	0	2
2	0	0	0	0	0	1	0	0	0	1	0	2
2	0	0	0	0	0	2	0	0	0	1	0	3
1	0	0	0	0	0	2	1	1	0	1	0	3
2	0	0	0	0	0	2	1	1	0	1	0	3
2	0	1	0	0	0	2	1	1	0	1	0	3
2	0	0	0	0	0	2	1	1	0	1	0	3
2	1	0	0	0	0	2	1	1	0	1	0	3
2	2	0	0	0	0	2	1	1	0	1	0	3
2	0	0	0	0	0	2	1	1	0	1	0	3
2	0	0	0	0	0	0	1	1	0	1	0	3
2	0	0	0	0	0	0	0	0	0	1	0	3
2	0	0	0	0	0	1	0	0	0	1	0	3
2	0	0	0	0	0	2	0	0	0	1	0	4
3	0	0	0	0	0	2	0	0	0	1	0	4
3	0	0	3	0	0	2	0	0	0	1	0	4
3	0	3	3	0	0	2	0	0	0	1	0	4
3	0	3	3	0	0	0	0	0	0	1	0	4
3	0	3	3	0	3	0	0	0	0	1	0	4
3	0	3	3	0	3	3	0	0	0	1	0	4
3	3	3	3	0	3	3	0	0	0	1	0	4
3	3	3	3	3	3	3	0	0	0	1	0	4

Figure 15: Resultados obtidos

5 Conclusão

Terminando, pensa-se que, de acordo com as metas estabelecidas pelo docente, o trabalho foi bem sucedido. Uma das principais dificuldades sentidas esteve relacionada com a primeira análise e compreensão de todo o código previamente escrito pelo professor. No entanto, assim que se finalizou uma das entidades, percebendo-se bem a sua implementação, facilmente se desenvolveu o resto do trabalho.

Aprofundaram-se os conhecimentos sobre o funcionamento de semáforos e sincronização de *threads*, sendo mais simples interiorizar certos pormenores a partir de um projeto como este, da mesma maneira que através da realização dos guiões práticos propostos sobre estes temas, que serviram como base para a realização do projeto.

Assim sendo, como os resultados obtidos podem ser considerados semelhantes aos do docente, é concluído que esta poderá ser uma das possíveis soluções.

6 Bibliografia

[1] http://index-of.es/Java/Operating%20System%20Concepts%20with%20Java%208th.pdf