CONTROLO DO ACESSO A RECURSOS PARTILHADOS USANDO SEMÁFOROS

Sistemas Operativos

Engenharia Informática 2019/2020

Leandro Silva 93446 / Mário Silva 93340

Taxa de contribuição: 50/50

Índice

Introdução

No segundo trabalho prático de Sistemas Operativos foi-nos pedido que completássemos um projeto em linguagem C que simulasse o fluxo de atividades necessárias para que uma entidade conseguisse fumar.

O nosso assunto central será a programação concorrente que se foca na interação e execução de multitarefas. Uma das principais preocupações quando nos deparamos com um algoritmo deste género são os problemas que podem ocorrer na sincronização e na partilha de recursos, assim sendo, torna-se necessário o uso de mecanismos associados à execução e sincronização de processos e *threads*. Um desses são os semáforos.

Espera-se que no decorrer da realização deste projeto enfrentemos alguns *deadlocks*, ou seja, uma situação em que existe um impasse entre dois ou mais processos, causando um bloqueio que impede a boa continuação do algoritmo.

Para resolver este problema, será apenas necessário alterar as funções pertencentes às entidades: *agente*, *watcher* e *smoker*. Estas funções regularizam o fluxo de atividades fazendo uso de semáforos.

Semáforos e Recursos Partilhados

Os semáforos têm como função controlar o acesso a recursos partilhados num ambiente multitarefa.

Quando declaramos um semáforo, o seu valor indica quantos processos (ou *threads*) podem ter acesso ao recurso partilhado. Assim, para partilhar um recurso o leitor deve verificar qual o valor do semáforo para saber qual a ação a executar.

As principais operações nos semáforos são:

semUp() – notificação – decrementa o valor do semáforo. Se o semáforo está com valor zero, o processo é posto em espera. Para receber um sinal, um processo executa o *wait* e bloqueia se o semáforo impedir a continuação da sua execução.

semDown() – espera – se o semáforo estiver com o valor zero e existir algum processo em espera, um processo será acordado. Caso contrário, o valor do semáforo é incrementado. Para enviar um sinal, um processo executa um *signal*.

No total serão utilizados 8 semáforos: o *mutex*, o *ingredient[]*, o *waitCigarette* e o *wait2Ings[]*. Os semáforos em formato *array* têm tamanho 3, para cada ingrediente existente: **TOBACCO**, **MATCHES** e **PAPER**. Isto porque cada *watcher* supervisiona um ingrediente diferente de cada um, assim como cada *smoker* é possuidor de um ingrediente também diferente de cada um. Os ingredientes vão ser usados como Ids para distinguir *watchers* e *smokers*.

Os semáforos estão contidos dentro da estrutura de dados **SHARED_DATA**.

```
typedef struct
{ /** \brief full state of the problem */
  FULL_STAT fSt;

  /* semaphores ids */
  /** \brief identification of critical region protection semaphore - val = 1 */
  unsigned int mutex;
  /** \brief identification of semaphore used by watchers to wait for agent - val = 0 */
  unsigned int ingredient[NUMINGREDIENTS];
  /** \brief identification of semaphore used by agent to wait for smoker to finish rolling - va
l = 0 */
  unsigned int waitCigarette;
```

```
  /** \brief identification of semaphore used by smoker to wait for watchers - val = 0  */
  unsigned int wait2Ings[NUMSMOKERS];

} SHARED_DATA;
```

Na estrutura de dados **FULL_STAT** temos todas as *flags*, *arrays* e variáveis que iremos utilizar para as várias entidades. Sempre que quisermos alterá-las devemos faze-lo dentro das regiões críticas – uma secção de código que começa por um *semDown()* do *mutex* e acaba num *semUp()* do mesmo – uma vez que estas são partilhadas pelos processos.

```
typedef struct
{   /** \brief state of all intervening entities */
    STAT st;

    /** \brief number of ingredients */
    int nIngredients;

    /** \brief number of orders to be performed by agent (each order includes a pack of 2 ingred
ients) */
    int nOrders;

    /** \brief number of smokers */
    int nSmokers;

    /** \brief flag used by agent to close factory */
    bool closing;

    /** \brief inventory of ingredients */
    int ingredients[NUMINGREDIENTS];

    /** \brief number of ingredients already reserved by watcher */
    int reserved[NUMINGREDIENTS];

    /** \brief number of cigarettes each smoker smoked */
    int nCigarettes[NUMSMOKERS];

} FULL_STAT;
```

Na estrutura **STAT** temos as variáveis, também partilhadas, que guardam os valores dos estados das 7 entidades – 1 *agent*, 3 *watchers* e 3 *smokers*.

```
typedef struct {
    /** \brief agent state */
    unsigned int agentStat;
    /** \brief watchers state */
    unsigned int watcherStat[NUMINGREDIENTS];
    /** \brief smokers state */
    unsigned int smokerStat[NUMSMOKERS];

} STAT;
```

Sempre que uma variável partilhada ou estado é alterado(a) é feito um *saveStat()*, que fará um output no terminal com as mudanças efetuadas no momento.

Análise de Interações

Para facilitar o desenvolvimento do projeto e seguindo a recomendação do professor, criamos antecipadamente uma tabela com os semáforos, as entidades que os manipulam, a ação que os descreve e as funções que os utilizam.

Esta tabela serviu de grande ajuda para resolver problemas de *deadlock* que surgiram no início do projeto.

Semáforo	**Entidade**		**Ação**		**Função**	
	semUp()	*semDown()*	*semUp*()	*semDown()*	*semUp()*	*semDown()*
mutex	Todas	Todas	Entra/sai da região crítica		Quase todas	Quase todas
Ingredient[]	*Agent*	*Watcher*	No geral, o *agent* informa os *watchers* dos ingredientes que o primeiro produziu.		*prepareIngredients()* *closeFactory()*	*waitForIngredient()*
waitCigarette	*Smoker*	*Agent*	O *smoker* informa o *agent* que acabou de enrolar.		*rollingCigarette()*	*waitForCigarette()*
wait2Ings[]	*Watcher*	*Smoker*	No geral, o *watcher* informa o *smoker* que já pode fumar.		*waitForIngredient()* *informSmoker()*	*waitForIngredients()*

Nota: Os semáforos *ingredient[]* e *wait2Ings[]* também têm são usados para sincronizar o encerramento das entidades.

Entidade *Agent*

O *agent* tem como papel preparar os ingredientes necessários para os *smokers* poderem fumar. É constituído por 3 funções: *prepareIngredients()*, *waitForCigarette()* e *closingFactory()*.

No seu ciclo de vida, o *agent* alterna entre as primeiras duas funções e este termina quando tiver completado no total 5 pedidos. Quando tiver terminado o ciclo, chama a função *closingFactory()*.

```
int nOrders=0;
 while(nOrders < sh->fSt.nOrders) {
    prepareIngredients();
    waitForCigarette();

    nOrders++;
 }
 closeFactory();
```

prepareIngredients()

Esta função simula a preparação de 2 ingredientes aleatórios diferentes pelo *agent*. Começa por mudar o seu estado para **PREPARING** e de seguida escolhe aleatoriamente 2 números diferentes entre 0 e 2 inclusives, que serão utilizados como IDs para ajudar a identificar os ingredientes.

Depois, incrementa os valores da lista *ingredients[]* respeitantes aos IDs para representar a disponibilidade dos mesmos.

No fim, é enviado um sinal (notificação) aos 2 *watchers* que supervisionam tais ingredientes, para que estes tenham conhecimento da sua disponibilidade.

```
static void prepareIngredients ()
{
    if (semDown (semgid, sh->mutex) == -
1) {                                                    /* enter critical region */
        perror ("error on the up operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.agentStat = PREPARING;
    saveState(nFic, &sh->fSt);
```

```
    // randomly selects 2 different ingredients
    int i1 = random() % 3, i2 = random() % 3;
    for (; i1 == i2; i2 = random() % 3);

    // updates inventory
    sh->fSt.ingredients[i1]++; sh->fSt.ingredients[i2]++;

    if (semUp (semgid, sh->mutex) == -
1) {                                                    /* leave critical region */
        perror ("error on the up operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // notifies watcher
    if ((semUp (semgid, sh->ingredient[i1]) == -1) | (semUp (semgid, sh->ingredient[i2]) == -
1)) {
        perror ("error on the up operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }
}
```

waitForCigarette()

Depois do preparo dos ingredientes, esta função é chamada, onde o estado do agente é alterado para **WAITING_CIG**.

Em seguida, fica à espera de um sinal de um *smoker* para proceder a um novo pedido.

```
static void waitForCigarette ()
{
    if (semDown (semgid, sh->mutex) == -
1) {                                                /* enter critical region */
        perror ("error on the down operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.agentStat = WAITING_CIG;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -
1) {                                                    /* leave critical region */
        perror ("error on the up operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // waits smoker
    if (semDown (semgid, sh->waitCigarette) == -1)  {
        perror ("error on the down operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }
}
```

closingFactory()

No fim do seu ciclo, é chamada esta função que irá alterar o estado do *agent* para **CLOSING_A** e a variável partilhada *closing* para ***true***. Depois, notifica todos os 3 *watchers* para que eles saibam do "enceramento da fábrica".

```
static void closeFactory ()
{
    if (semDown (semgid, sh->mutex) == -
1) {                                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.agentStat = CLOSING_A;
    saveState(nFic, &sh->fSt);

    // updates factory's state
    sh->fSt.closing = true;

    if (semUp (semgid, sh->mutex) == -
1) {                                                     /* leave critical region */
        perror ("error on the up operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // notifies watchers
    if ((semUp (semgid, sh->ingredient[TOBACCO]) == -1) | (semUp (semgid, sh-
>ingredient[MATCHES]) == -1) | (semUp (semgid, sh->ingredient[PAPER]))) {
        perror ("error on the up operation for semaphore access (AG)");
        exit (EXIT_FAILURE);
    }
}
```

Entidades *Watchers*

Os *watchers* são entidades que servem de mediador entre o *agent* e os *smokers* para alguns casos. São constituídos por 3 funções: *waitForIngredient()*, *updateReservations()* e *informSmoker()*.

Nos seus ciclos de vida, estas entidades alternam entre as primeiras 2 funções enquanto a *waitForIngredient()* retornar ***true***. A função *informSmoker()* só é chamada se, no *updateReservations()*, o *watcher* concluir que existe um *smoker* que já pode enrolar.

```
    int id = n, smokerReady;
    while( waitForIngredient (id) ) {
        smokerReady = updateReservations(id);
        if(smokerReady>=0) informSmoker(id, smokerReady);
    }
```

waitForIngredient()

Nesta função, é alterado o estado de um dos *watchers* para **WAITING_ING** e, de seguida, aguarda por uma notificação do *agent* no respetivo semáforo *ingredient*. Quando recebe a notificação, verifica se o *agent* alterou o estado da variável partilhada *closing* para ***true***.

Em caso afirmativo, altera de novo o seu estado, desta vez para **CLOSING_W**, e muda o valor de retorno para ***false*** (certificando-se de que o ciclo fecha). Depois, envia um sinal ao *smoker* com o mesmo ID para que este saiba do encerramento do fluxo.

Caso contrário, a função termina com valor de retorno ***true***, dando continuidade ao ciclo de vida.

```
static bool waitForIngredient(int id)
{
    bool ret=true;

    if (semDown (semgid, sh->mutex) == -
1)  {                                                   /* enter critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.watcherStat[id] = WAITING_ING;
```

```
    saveState(nFic, &sh->fSt);

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

    /* TODO: insert your code here */

    // waits agent
    if (semDown (semgid, sh->ingredient[id]) == -1)  {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -
1)  {                                                  /* enter critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    if (sh->fSt.closing) {
        // updates state
        sh->fSt.st.watcherStat[id] = CLOSING_W;
        saveState(nFic, &sh->fSt);

        ret = false;
    }

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

    if (sh->fSt.closing) {
        // notifies smoker
        if (semUp (semgid, sh->wait2Ings[id]) == -1)  {
            perror ("error on the up operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
    }
    return ret;
}
```

updateReservations()

Esta função é sempre chamada depois da *waitForIngredients()*. Aqui, o estado de um dos watchers é alterado para **UPDATING** e é reservado o ingrediente com o respetivo ID incrementando o valor de *reserved[id]*.

Anteriormente, apenas reservávamos o ingrediente caso este estivesse disponível em *ingredients[]*, mas reparamos que este passo era redundante, pois um *watcher* só chamaria esta função quando fosse notificado na função *waitForIngredient()*. E só seria notificado se o respetivo ingrediente tivesse sido produzido. Assim, era certo que o mesmo estava disponível para ser reservado.

Continuando, o *watcher*, de seguida, examina se existe um *smoker* que possa começar a fumar. Para isso, nesta função é feito um conjunto de *if statements* que verificam se existem dois ingredientes diferentes reservados. Se houver, estes são decrementados e o valor de retorno passa a ser o valor do ID do *smoker* que pode fumar. Senão, o valor de retorno é **-1**.

```
static bool waitForIngredient(int id)
{
    bool ret=true;

    if (semDown (semgid, sh->mutex) == -
1)  {                                                   /* enter critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.watcherStat[id] = WAITING_ING;
    saveState(nFic, &sh->fSt);

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

    /* TODO: insert your code here */

    // waits agent
    if (semDown (semgid, sh->ingredient[id]) == -1)  {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -
1)  {                                                   /* enter critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    if (sh->fSt.closing) {
        // updates state
        sh->fSt.st.watcherStat[id] = CLOSING_W;
        saveState(nFic, &sh->fSt);

        ret = false;
    }

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

    if (sh->fSt.closing) {
        // notifies smoker
        if (semUp (semgid, sh->wait2Ings[id]) == -1)  {
            perror ("error on the up operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
    }
    return ret;
}
```

informSmoker()

Se o valor de retorno de *updateReservation()* for maior ou igual a **0**, esta função é chamada usando este valor como parâmetro, *smokerReady*. Aqui, o estado do *watcher* é alterado para **INFORMING** e no fim é notificado *smoker* com ID igual ao *smokerReady*, fazendo uso do semáforo *wait2Ings*.

```
static void informSmoker (int id, int smokerReady)
{
    if (semDown (semgid, sh->mutex) == -
1)  {                                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.watcherStat[id] = INFORMING;

    saveState(nFic, &sh->fSt);

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

    /* TODO: insert your code here */

    // notifies smoker
    if (semUp (semgid, sh->wait2Ings[smokerReady]) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
}
```

Entidade *Smoker*

Os *smokers* são as entidades que fazem uso dos ingredientes preparados pelo *agent*. São constituídos por 3 funções: *waitForIngredients()*, *rollingCigarette()* e *smoke()*.

Nos seus ciclos de vida chamam as funções acima mencionadas pela ordem que foram escritas, enquanto o valor de retorno de *waitForIngredients()* for ***true***.

```
    while(waitForIngredients(n)) {
        rollingCigarette(n);
        smoke(n);
    }
```

waitForIngredients()

Nesta função, o estado de um dos *smoker* é alterado para **WAITING_2ING** ficando, de seguida, à espera de uma notificação do *watcher* através do semáforo *wait2Ings* com o respetivo ID.

Quando este *smoker* é notificado, verifica se o *agent* alterou a variável partilhada *closing* para ***true***. Se se verificar, a função altera de novo o estado do *smoker* para **CLOSING_S** e o valor de retorno da função passa a ***false***. Senão, é decrementado os ingredientes providos do *agent* que o *smoker* precisar na lista *ingredients[]* e o valor de retorno mantém-se a ***true***, continuando o ciclo de vida.

```
static bool waitForIngredients (int id)
{
    bool ret = true;

    if (semDown (semgid, sh->mutex) == -
1)  {                                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.smokerStat[id] = WAITING_2ING;
    saveState(nFic, &sh->fSt);


    if (semUp (semgid, sh->mutex) == -
1) {                                                     /* exit critical region */
        perror ("error on the up operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
```

```
    }

    /* TODO: insert your code here */

    // waits watcher
    if (semDown (semgid, sh->wait2Ings[id]) == -1)  {
        perror ("error on the down operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -
1)  {                                                   /* enter critical region */
        perror ("error on the down operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    if (sh->fSt.closing) {
        // updates state
        sh->fSt.st.smokerStat[id] = CLOSING_S;
        saveState(nFic, &sh->fSt);

        ret = false;
    }
    else {
        // updates inventory
        switch (id) {
            case HAVETOBACCO:
                sh->fSt.ingredients[MATCHES]--;
                sh->fSt.ingredients[PAPER]--;
                break;
            case HAVEMATCHES:
                sh->fSt.ingredients[TOBACCO]--;
                sh->fSt.ingredients[PAPER]--;
                break;
            case HAVEPAPER:
                sh->fSt.ingredients[TOBACCO]--;
                sh->fSt.ingredients[MATCHES]--;
        }
    }

    if (semUp (semgid, sh->mutex) == -
1) {                                                  /* exit critical region */
        perror ("error on the up operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    return ret;
}
```

rollingCigarette()

Depois da função anterior, é chamada esta função, onde o estado do *smoker* é alterado para **ROLLING**. Depois, é suspendida a execução através do *usleep()* para simular o tempo que o *smoker* demora a enrolar. Este tempo, *rollingTime*, é aleatório e segue uma distribuição normal de média 100 e desvio padrão 30. É necessário confirmar com um *if statement* se o valor de *rollingTime* é positivo, pois há a possibilidade de este acabar por ser negativo, o que faria a execução parar.

Após a suspensão, o *agent* é notificado através do semáforo *waitCigarette* para que este possa fazer outro pedido.

```
static void rollingCigarette (int id)
{
    double rollingTime = 100.0 + normalRand(30.0);

    if (semDown (semgid, sh->mutex) == -
1)  {                                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.smokerStat[id] = ROLLING;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -
1) {                                                      /* exit critical region */
        perror ("error on the up operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // takes time
    if (rollingTime > 0)
        usleep(rollingTime);

    // notifies agent
    if (semUp (semgid, sh->waitCigarette) == -1) {
        perror ("error on the up operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }
}
```

smoke()

No fim, esta função é chamada para simular o tempo que *smoker* demora a fumar. O estado deste é alterado para **SMOKING**, o número de cigarros fumados por este *smoker* incrementa, *nCigarettes[id]*, e a execução é suspensa durante um tempo aleatório, *smokingTime*, com a mesma distribuição do tempo em *rollingTime*.

```
static void rollingCigarette (int id)
{
    double rollingTime = 100.0 + normalRand(30.0);

    if (semDown (semgid, sh->mutex) == -
1)  {                                                    /* enter critical region */
        perror ("error on the down operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // updates state
    sh->fSt.st.smokerStat[id] = ROLLING;
    saveState(nFic, &sh->fSt);
```

```
    if (semUp (semgid, sh->mutex) == -
1) {                                                    /* exit critical region */
        perror ("error on the up operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }

    /* TODO: insert your code here */

    // takes time
    if (rollingTime > 0)
        usleep(rollingTime);

    // notifies agent
    if (semUp (semgid, sh->waitCigarette) == -1) {
        perror ("error on the up operation for semaphore access (SM)");
        exit (EXIT_FAILURE);
    }
}
```

Resultados

Resultado obtido com a seguinte configuração:

```
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/run$ cd ../src/
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/src$ make
echo clean
clean
rm -f *.o
gcc -Wall   -c -o semSharedMemAgent.o semSharedMemAgent.c
gcc -Wall   -c -o sharedMemory.o sharedMemory.c
gcc -Wall   -c -o semaphore.o semaphore.c
gcc -Wall   -c -o logging.o logging.c
gcc -o ../run/agent semSharedMemAgent.o sharedMemory.o semaphore.o logging.o -lm
gcc -Wall   -c -o semSharedMemWatcher.o semSharedMemWatcher.c
gcc -o ../run/watcher semSharedMemWatcher.o sharedMemory.o semaphore.o logging.o
gcc -Wall   -c -o semSharedMemSmoker.o semSharedMemSmoker.c
gcc -o ../run/smoker semSharedMemSmoker.o sharedMemory.o semaphore.o logging.o -
lm
gcc -Wall   -c -o probSemSharedMemSmokers.o probSemSharedMemSmokers.c
gcc -o ../run/probSemSharedMemSmokers probSemSharedMemSmokers.o sharedMemory.o s
emaphore.o logging.o -lm
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/src$ cd ../run
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/run$ ./run.sh 1
```

```
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   1   1    0   0   0
  1    0   0   0    0   0   0    0   1   1    0   0   0
  1    0   0   0    0   0   0    0   1   1    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   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   2   0    1   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    0   0   0
  1    0   0   0    2   0   0    0   0   0    1   0   0
  2    0   0   0    2   0   0    1   0   1    1   0   0
  2    0   0   1    2   0   0    1   0   1    1   0   0
  2    0   0   0    2   0   0    1   0   1    1   0   0
  2    1   0   0    2   0   0    1   0   1    1   0   0
  2    2   0   0    2   0   0    1   0   1    1   0   0
  2    0   0   0    2   0   0    1   0   1    1   0   0
  2    0   0   0    2   1   0    0   0   0    1   0   0
  2    0   0   0    0   1   0    0   0   0    1   0   0
  1    0   0   0    0   1   0    0   0   0    1   0   0
  2    0   0   0    0   1   0    0   1   1    1   0   0
  2    0   0   1    0   1   0    0   1   1    1   0   0
  2    0   0   0    0   1   0    0   1   1    1   0   0
  2    0   1   0    0   1   0    0   1   1    1   0   0
  2    0   1   0    0   2   0    0   1   1    1   0   0
  2    0   2   0    0   2   0    0   1   1    1   1   0
  2    0   2   0    1   2   0    0   0   0    1   1   0
  2    0   0   0    1   2   0    0   0   0    1   1   0
  2    0   0   0    1   0   0    0   0   0    1   1   0
  2    0   0   0    2   0   0    0   0   0    1   1   0
```

```
 1   0   0   0   2   0   0   0   0   0   2   1   0
 2   0   0   0   2   0   0   1   0   1   2   1   0
 2   1   0   0   2   0   0   1   0   1   2   1   0
 2   0   0   0   2   0   0   1   0   1   2   1   0
 2   0   0   1   2   0   0   1   0   1   2   1   0
 2   0   0   2   2   0   0   1   0   1   2   1   0
 2   0   0   0   2   0   0   1   0   1   2   1   0
 2   0   0   0   0   0   0   1   0   1   2   1   0
 2   0   0   0   0   1   0   0   0   0   2   1   0
 2   0   0   0   0   2   0   0   0   0   2   1   0
 1   0   0   0   0   2   0   0   0   0   2   2   0
 2   0   0   0   0   2   0   1   1   0   2   2   0
 2   1   0   0   0   2   0   1   1   0   2   2   0
 2   0   0   0   0   2   0   1   1   0   2   2   0
 2   0   1   0   0   2   0   1   1   0   2   2   0
 2   0   2   0   0   2   0   1   1   0   2   2   0
 2   0   0   0   0   2   0   1   1   0   2   2   0
 2   0   0   0   0   2   1   0   0   0   2   2   0
 2   0   0   0   0   0   1   0   0   0   2   2   0
 2   0   0   0   0   0   2   0   0   0   2   2   0
 3   0   0   0   0   0   2   0   0   0   2   2   1
 3   0   0   3   0   0   2   0   0   0   2   2   1
 3   0   0   3   0   0   0   0   0   0   2   2   1
 3   0   0   3   0   0   3   0   0   0   2   2   1
 3   3   0   3   0   0   3   0   0   0   2   2   1
 3   3   3   3   0   0   3   0   0   0   2   2   1
 3   3   3   3   0   3   3   0   0   0   2   2   1
 3   3   3   3   3   3   3   0   0   0   2   2   1
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/run$ ./clean.sh
```

Resultado obtido com o filter.sh:

```
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/run$ ./filter.sh
                    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
  .    .   .   .    .   .   .    0   0   0    0   0   0
  .    .   .   .    .   .   .    0   0   0    0   0   0
  2    .   .   .    .   .   .    1   0   1    0   0   0
  .    1   .   .    .   .   .    1   0   1    0   0   0
  .    0   .   .    .   .   .    1   0   1    0   0   0
  .    .   .   .    .   .   .    1   0   1    0   0   0
  .    .   .   .    .   .   .    1   0   1    0   0   0
  .    .   .   .    .   .   .    1   0   1    0   0   0
  .    .   .   .    .   .   .    1   0   1    0   0   0
  .    .   .   1    .   .   .    1   0   1    0   0   0
  .    .   .   2    .   .   .    1   0   1    0   0   0
  .    .   .   0    .   .   .    1   0   1    0   0   0
  .    .   .   .    .   .   .    1   0   1    0   0   0
  .    .   .   .    .   1   .    0   0   0    0   0   0
  .    .   .   .    .   2   .    0   0   0    0   0   0
  1    .   .   .    .   .   .    0   0   0    0   1   0
  2    .   .   .    .   .   .    1   0   1    0   1   0
  .    1   .   .    .   .   .    1   0   1    0   1   0
  .    0   .   .    .   .   .    1   0   1    0   1   0
  .    .   .   1    .   .   .    1   0   1    0   1   0
  .    .   .   2    .   .   .    1   0   1    0   1   0
  .    .   .   0    .   .   .    1   0   1    0   1   0
  .    .   .   .    .   0   .    1   0   1    0   1   0
  .    .   .   .    .   1   .    0   0   0    0   1   0
  1    .   .   .    .   .   .    0   0   0    0   1   0
  2    .   .   .    .   .   .    1   0   1    0   1   0
  .    1   .   .    .   .   .    1   0   1    0   1   0
  .    0   .   .    .   .   .    1   0   1    0   1   0
  .    .   .   .    .   2   .    1   0   1    0   1   0
  .    .   .   1    .   .   .    1   0   1    0   2   0
  .    .   .   2    .   .   .    1   0   1    0   2   0
  .    .   .   0    .   .   .    1   0   1    0   2   0
  .    .   .   .    .   0   .    1   0   1    0   2   0
  .    .   .   .    .   1   .    0   0   0    0   2   0
  1    .   .   .    .   .   .    0   0   0    0   2   0
  2    .   .   .    .   .   .    0   1   1    0   2   0
  .    .   .   .    .   2   .    0   1   1    0   2   0
  .    .   1   .    .   .   .    0   1   1    0   3   0
  .    .   0   .    .   .   .    0   1   1    0   3   0
  .    .   .   1    .   .   .    0   1   1    0   3   0
  .    .   .   2    .   .   .    0   1   1    0   3   0
  .    .   .   0    .   .   .    0   1   1    0   3   0
  .    .   .   .    1   .   .    0   0   0    0   3   0
  .    .   .   .    .   0   .    0   0   0    0   3   0
  .    .   .   .    2   .   .    0   0   0    0   3   0
  1    .   .   .    .   .   .    0   0   0    1   3   0
  2    .   .   .    .   .   .    1   0   1    1   3   0
  .    1   .   .    .   .   .    1   0   1    1   3   0
  .    0   .   .    .   .   .    1   0   1    1   3   0
  .    .   .   1    .   .   .    1   0   1    1   3   0
  .    .   .   2    .   .   .    1   0   1    1   3   0
  .    .   .   0    .   .   .    1   0   1    1   3   0
  .    .   .   .    .   1   .    0   0   0    1   3   0
  3    .   .   .    .   .   .    0   0   0    1   3   0
  .    3   .   .    .   .   .    0   0   0    1   3   0
  .    .   .   .    .   2   .    0   0   0    1   3   0
  .    .   .   3    .   .   .    0   0   0    1   4   0
  .    .   .   .    .   .   3    0   0   0    1   4   0
  .    .   .   .    .   0   .    0   0   0    1   4   0
  .    .   3   .    .   .   .    0   0   0    1   4   0
  .    .   .   .    .   3   .    0   0   0    1   4   0
  .    .   .   .    0   .   .    0   0   0    1   4   0
  .    .   .   .    3   .   .    0   0   0    1   4   0
aluno@asus:~/Desktop/2ano/SO/TP2/semaphore_smokers/run$
```

Conclusão

Como o acesso a dados e recursos partilhados de forma simultânea pode criar uma situação de inconsistência, percebemos do quão é importante e necessário fazer um bom uso dos mecanismos de sincronização que assegurem a execução ordenada e correta dos processos e *threads*.

Referências

Revista Programar, THREADS, SEMÁFOROS E DEADLOCKS – O JANTAR DOS FILÓSOFOS – https://www.revista-programar.info/artigos/threads-semaforos-e-deadlocks-o-jantar-dos-filosofos/

Slides da Unidade Curricular de Sistemas Operativos

Documentação fornecida pelo Doxyfile