universidade de aveiro
theoria poiesis praxis

Restaurante

Sistemas Operativos

Prof. António Borges

Tiago Albuquerque, 112901

Guilherme Mesquita, 112957

Índice

Introdução

No âmbito da unidade curricular de Sistemas Operativos, este trabalho tem como objetivo a compreensão dos mecanismos associados à execução e sincronização de processos e threads. O projeto visa a implementação de um simulador de restaurante em linguagem C, mantendo o foco no objetivo enunciado acima. Este simulador representa a dinâmica de um restaurante onde os grupos de clientes interagem com um rececionista, um empregado de mesa e um cozinheiro.

Para a implementação deste projeto, as 4 entidades são representadas como processos independentes que interagem entre si com o uso de semáforos e da memória compartilhada. A sincronização dos processos é assim essencial para simular o fluxo do restaurante de maneira coerente.

Ao longo deste relatório será apresentada a metodologia utilizada na resolução do problema proposto, assim como todos os testes realizados para verificarmos a validade da solução.

Serão ainda apresentadas as diversas etapas de resolução do problema juntamente com uma breve exposição do raciocínio adotado. Surgirão ainda algumas imagens do código fonte e diagramas de modo a facilitar a compreensão de toda a solução.

Abordagem usada para resolver o problema

Com o propósito de simplificar a resolução do problema e diminuir a sua suscetibilidade a erros, necessitamos de dividir a resolução em diferentes etapas, cada uma com o seu objetivo definido.

Etapas de resolução adotadas:

1. Análise e verificação das operações a realizar sobre os semáforos;
2. Resumo do funcionamento/função de cada uma das entidades;
3. Exposição dos resultados obtidos (testes para validar a solução).

A partir deste ponto, será então dado ênfase a cada uma destas etapas do processo de resolução do problema.

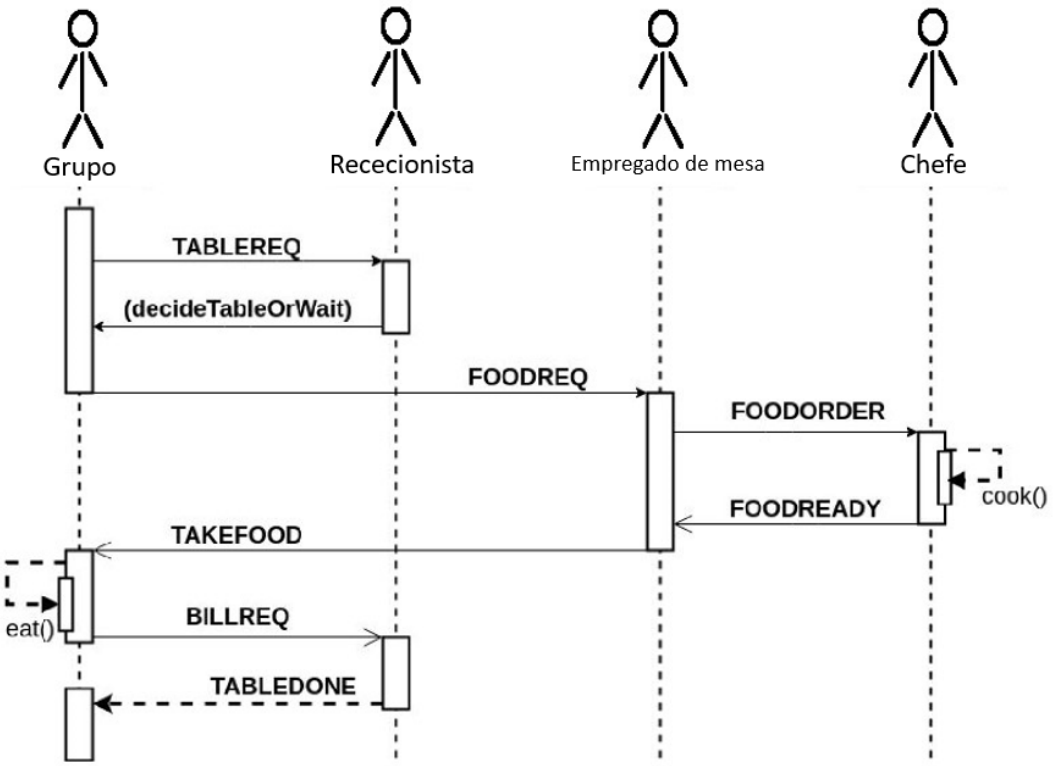
Análise e verificação das operações a realizar sobre os semáforos

Antes de começar com qualquer desenvolvimento das funções referentes a cada entidade, achámos que seria proveitoso fazer uma análise atenta a cada semáforo presente no ficheiro **sharedDataSync.h**, assim como perceber também em que situações é que lhes deveríamos dar 'Up' ou 'Down'. Esta primeira análise facilitou todo o desenvolvimento do projeto.

Para analisar tudo o que foi já enunciado, construímos uma tabela que facilita a interpretação:

Entidades	Semáforos (Up)	Semáforos (Down)
Chefe (Chef)	➔ orderReceived ➔ waiterRequest	➔ waiterRequestPossible ➔ waitOrder
Grupo (Group)	➔ receptionistReq ➔ waiterRequest	➔ receptionistRequestPossible ➔ waiterRequestPossible ➔ waitForTabel[id] - grupo com o 'id' ➔ requestReceived[tableForGroup] - grupo na mesa 'tableForGroup' ➔ foodArrived[tableForGroup] - grupo na mesa 'tableForGroup' ➔ tableDone[tableForGroup] - grupo na mesa 'tableForGroup'
Empregado de Mesa (Waiter)	➔ waiterRequestPossible ➔ waitOrder ➔ requestReceived[tableForGroup] ➔ foodArrived[tableForGroup]	➔ waiterRequest ➔ orderReceived
Recepcionista (Receptionist)	➔ receptionistRequestPossible ➔ waitForTabel[id] ➔ tableDone[tableForGroup]	➔ receptionistReq

Para além da tabela acima, e com o apoio do Professor nas aulas práticas, construímos o seguinte diagrama que ajuda na perceção da ordem dos trabalhos a ser feito por cada uma das 4 entidades:



Consequentemente, e com estas análises feitas antes de começar o desenvolvimento do projeto em si, tornou-se mais percetível o que teríamos de fazer.

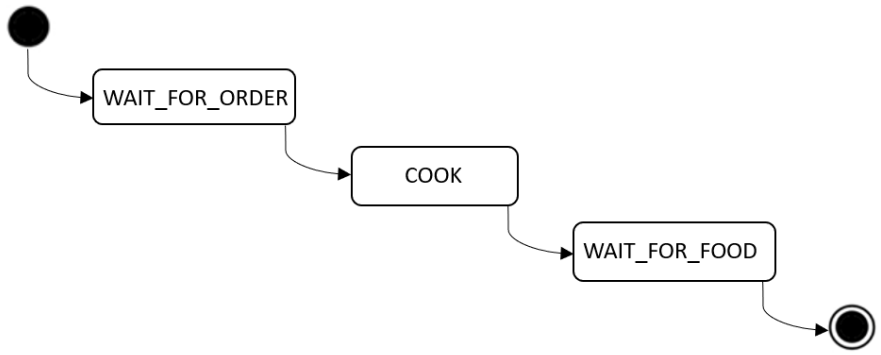
Avancemos agora para um resumo do funcionamento de cada uma das 4 entidades (chefe, empregado de mesa, grupos de clientes, rececionista).

Análise do funcionamento de cada entidade

De seguida, vamos apresentar, com detalhe, as alterações que foram feitas em cada função de cada uma das entidades.

Chefe

Antes mesmo de avançarmos para a análise de cada uma das funções, apresentamos um pequeno diagrama (de estados), que representa toda a transição dos estados correspondente à entidade do Chefe.



Função waitForOrder()

Esta função tem como papel principal, a representação do comportamento do **chefe** enquanto aguarda por pedidos por parte do **empregado de mesa**. Desta feita, fazemos um Down no semáforo 'waitOrder' (figura 1) para garantir que apenas o chefe quer entrar na zona crítica, evitando assim possíveis conflitos e condições de corrida.

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

Figura 1

Após entrar na zona crítica, através de um bloqueio ('Down') no semáforo 'mutex' (figura 2), o chefe fica a saber qual foi o grupo que fez o pedido.

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

lastGroup = sh->fSt.foodGroup;
```

Figura 2

De seguida, e com o recurso a um Down no semáforo ‘mutex’, é atualizado o estado interno do chefe, indicando que o chefe está a cozinhar (estado ‘COOK’). De notar que este estado é guardado (figura 3).

```
sh->fSt.st.chefStat = COOK;
saveState(nFic, &sh->fSt);
```

Figura 3

Antes de sair da zona crítica (libertando também os recursos compartilhados), através de um Up no semáforo mutex, é dado também um Up no semáforo ‘orderReceived’, alertando o empregado de mesa que o pedido foi recebido (figura 4).

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

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

Figura 4

Função processOrder()

Esta função é responsável pela simulação da confeção e da entrega da comida ao empregado de mesa. Consequentemente, e após a execução da função **usleep** (que simula o tempo que o chef demora a cozinhar), é feito um Down no semáforo 'waiterRequestPossible', para que, havendo possibilidade, possa sinalizar ao empregado de mesa que está pronto para entregar a comida (figura 5).

```
usleep((unsigned int)floor((MAXCOOK * random()) / RAND_MAX + 100.0));

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

Figura 5

De seguida, e após realizar a entrada na zona crítica, mais uma vez com o recurso a um Down no semáforo 'mutex', atualizamos o tipo de pedido, 'reqType', com o valor 'FOODREADY', indicando assim que a comida está pronta para ser entregue, pelo empregado de mesa. Atribuímos também o valor do 'lastGroup' ao 'reqGroup' (da estrutura 'waiterRequest'); assim é indicado ao empregado o grupo a que pertence o pedido (figura 6). Seguidamente é alterado, e guardado, o estado do chefe para 'WAIT_FOR_FOOD'.

```
sh->fSt.waiterRequest.reqType = FOODREADY;
sh->fSt.waiterRequest.reqGroup = lastGroup;


sh->fSt.st.chefStat = WAIT_FOR_FOOD;
saveState(nFic, &sh->fSt);
```

Figura 6

Dando um Up no semáforo 'mutex', saímos da zona crítica, libertando também todos os recursos compartilhados (figura 7).

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

Figura 7

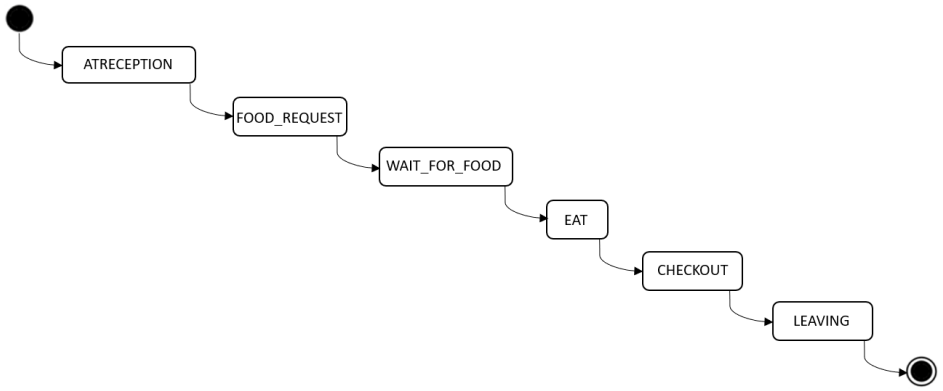
Por fim, através de um Up no semáforo ‘waiterRequest’ avisamos o empregado de mesa que tem um novo pedido (figura 8).

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

Figura 8

Grupo

Antes mesmo de avançarmos para a análise de cada uma das funções, apresentamos um pequeno diagrama (de estados), que representa toda a transição dos estados correspondente à entidade do Grupo de clientes.



Função checkInAtReception()

O objetivo desta primeira função é simular a solicitação de uma mesa por parte de cada um dos grupos. Sendo assim, e antes da entrada na zona crítica, é efetuado um Down no semáforo 'receptionistRequestPossible' para fazer uma "reserva" do recepcionista (figura 9). Posto isto, o grupo só prosseguirá se puder realizar com sucesso a operação no semáforo (DOWN).

```
if (semDown(semgid, sh->receptionistRequestPossible) == -1)
{
    perror("error on the down operation for semaphore access (CT)");
    exit(EXIT_FAILURE);
}

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

Figura 9

Logo de seguida, o estado do grupo é alterado e guardado, para o estado 'ATRECEPTION', onde espera na receção ou espera por uma mesa. O grupo indica a solicitação da mesa, definindo o id da solicitação como 'TABLEREQ'. A linha seguinte permite ao rececionista saber qual o grupo que está a fazer a solicitação, passando a conhecer o seu ID (figura 10).

```
sh->fSt.st.groupStat[id] = ATRECEPTION; // Change state to ATRECEPTION
saveState(nFic, &sh->fSt);

sh->fSt.receptionistRequest.reqType = TABLEREQ;
sh->fSt.receptionistRequest.reqGroup = id;
```

Figura 10

Através de um Up no semáforo ‘receptionistReq’, notificamos o rececionista da solicitação. Logo após a saída da zona crítica (Up no semáforo ‘mutex’), libertando também todos os recursos compartilhados, damos um Down no semáforo ‘waitForTable’, com o objetivo principal de aguardar pela disponibilidade de uma mesa (figura 11).

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

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

if (semDown(semgid, sh->waitForTable[id]) == -1) {
    perror("error on the down operation for semaphore receptionist");
    exit(EXIT_FAILURE);
}
```

Figura 11

Função orderFood()

Por sua vez, o papel desta segunda função é a representação da solicitação do pedido de comida, por parte do grupo, ao empregado de mesa.

Começamos por criar uma variável ‘tableForGroup’, que irá representar a mesa atribuída ao grupo (por Id). Antes de entrar na zona crítica, com um Down no semáforo ‘mutex’, e dando também um Down no semáforo ‘waiterRequestPossible’ fazemos uma “reserva” do empregado de mesa. Como já estamos dentro da zona crítica, alteramos o estado do grupo para ‘FOOD_REQUEST’, guardando esse mesmo estado (figura 12).

```
int tableForGroup;      // Table assigned to the group

if (semDown(semgid, sh->waiterRequestPossible) == -1) {
    perror("error on the up operation for semaphore waiter");
    exit(EXIT_FAILURE);
}

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

sh->fSt.st.groupStat[id] = FOOD_REQUEST; // Change state to FOOD_REQUEST
saveState(nFic, &sh->fSt);
```

Figura 12

Logo de seguida, o empregado de mesa recebe, não só o pedido, como também o ID do grupo. Antes de sair da zona crítica, com um Up no semáforo ‘mutex’, damos um Up no semáforo ‘waiterRequest’, onde o grupo comunica ao empregado de mesa o seu novo pedido. Na linha de código seguinte é então atribuída a mesa ao grupo, passando o empregado de mesa a conhecer, através do seu ID, a mesa onde está sentado o mesmo (figura 13).

```
sh->fSt.waiterRequest.reqType = FOODREQ;        // waiter receives request
sh->fSt.waiterRequest.reqGroup = id;            // waiter receives the id of the group

if (semUp(semgid, sh->waiterRequest) == -1) {
    perror("error on the up operation for semaphore waiter");
    exit(EXIT_FAILURE);
}

tableForGroup = sh->fSt.assignedTable[id];      // atribuir a mesa ao grupo

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

Figura 13

Finalizando, é realizado um Down no semáforo ‘requestReceived’, com o intuito de o grupo perceber se o pedido já foi passado ao chefe.

Função waitFood()
O objetivo desta função é simular o comportamento do grupo enquanto espera a chegada da comida.

De modo semelhante à função anterior, começamos por criar uma variável 'tableForGroup', que irá representar a mesa atribuída ao grupo (por Id). Entrando na zona crítica, através de um Down no semáforo 'mutex', alteramos o estado do grupo para 'WAIT_FOR_FOOD', guardando esse mesmo estado. Isto significa que o grupo espera a chegada do seu pedido. Na linha de código seguinte é atribuída a mesa ao grupo, passando o empregado de mesa a conhecer, através do seu ID, a mesa onde está sentado o mesmo. Prontamente, saímos da zona crítica com um Up no semáforo 'mutex' (figura 14).

```
int tableForGroup;

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

sh->fSt.st.groupStat[id] = WAIT_FOR_FOOD; // Change state to WAIT_FOR_FOOD
saveState(nFic, &sh->fSt);

tableForGroup = sh->fSt.assignedTable[id];      // atribuir a mesa ao grupo

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

Figura 14

Antes de voltarmos a entrar na zona crítica, damos um Down no semáforo 'foodArrived' da mesa onde o grupo está sentado. Estando já dentro da zona crítica, e antes de se voltar a sair novamente, alteramos o estado do grupo para 'EAT', guardando essa mesma alteração. Esta mudança é feita aquando da chegada da refeição (figura 15).

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

sh->fSt.st.groupStat[id] = EAT; // Change state to EAT
saveState(nFic, &sh->fSt);


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

Figura 15

Função checkOutAtReception()

O objetivo desta função consiste no check-out do grupo na receção.

Semelhante às duas funções anteriores, começamos por criar uma variável 'tableForGroup', que irá representar a mesa do grupo (por Id), que por sua vez ficará vazia visto que o grupo vai sair. De seguida, e antes da entrada na zona crítica, é efetuado um Down no semáforo 'receptionistRequestPossible' para fazer uma "reserva" do recepcionista. Posto isto, o grupo só prosseguirá se puder realizar com sucesso a operação no semáforo (DOWN). No passo seguinte alteramos o estado do grupo para o estado de 'CHECKOUT', guardando essa mesma alteração. Prontamente, o rececionista recebe a fatura com os gastos do grupo, assim como o ID do mesmo para que o possa identificar. Depois damos um Up no semáforo 'receptionistReq', notificando o rececionista sobre o novo pedido (figura 16).

```
int tableForGroup;

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

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

sh->fSt.st.groupStat[id] = CHECKOUT; // Change state to CHECKOUT
saveState(nFic, &sh->fSt);

sh->fSt.receptionistRequest.reqType = BILLREQ;      // receptionist receives the bill request
sh->fSt.receptionistRequest.reqGroup = id;          // and receptionist receives the id of the group

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

tableForGroup = sh->fSt.assignedTable[id];      // retirar o id da mesa que vai ficar livre, pois o grupo vai sair

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

Figura 16

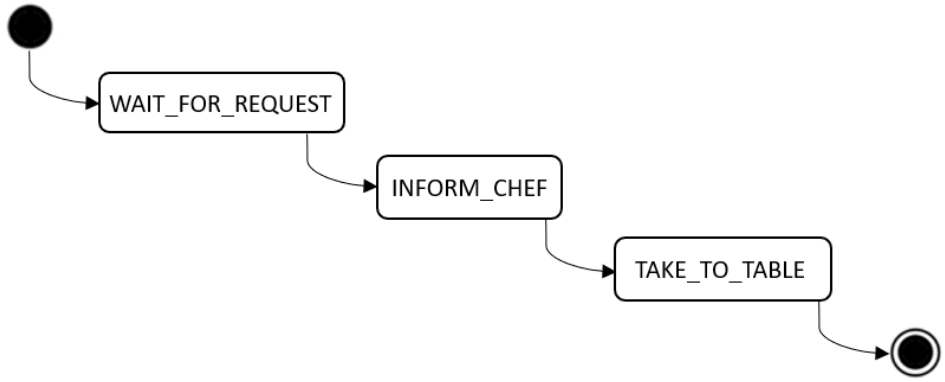
Após a saída da zona crítica, e antes de voltarmos a entrar, damos um Down no semáforo ‘requestReceived’ e outro Down no semáforo ‘tableDone’, ambos na mesa onde o Grupo está sentado. O segundo Down faz com que o grupo aguarde enquanto o rececionista resolve o pedido. Após a resolução do mesmo, e antes de sairmos da zona crítica, alteramos o estado do grupo para ‘LEAVING’ (figura 17).

```
int sem_reqReceived = sh->requestReceived[tableForGroup];

if (semDown(semgid, sem_reqReceived) == -1) {
    perror("error on the down operation for semaphore receptionist");
    exit(EXIT_FAILURE);
}

int sem_tableDone = sh->tableDone[tableForGroup];

if (semDown(semgid, sem_tableDone) == -1) {
    perror("error on the down operation for semaphore receptionist");
    exit(EXIT_FAILURE);
}

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

sh->fSt.st.groupStat[id] = LEAVING; // Change state to LEAVING
saveState(nFic, &sh->fSt);

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

Figura 17

Empregado de mesa

Antes mesmo de avançarmos para a análise de cada uma das funções, apresentamos um pequeno diagrama (de estados), que representa toda a transição dos estados correspondente à entidade do Empregado de mesa.



Função waitForClientOrChef()

Esta função é responsável por manter o empregado à espera de um pedido, quer por parte de um grupo, quer por parte do chefe.

Começamos por dar DOWN ao semáforo associado ao mutex, garantindo que apenas este processo possa acessar a zona crítica e definimos o estado do waiter como "WAIT_FOR_REQUEST" no estado compartilhado “sh->fSt”, e no final guardamos o estado alterado (figura 18).

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

sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
saveState(nFic, &sh->fSt);
```

Figura 18

Damos UP ao semáforo do mutex, permitindo que outros processos acedam à zona crítica, aguardamos por uma solicitação (request) do cliente ou do chef e damos DOWN novamente ao semáforo mutex para garantir o acesso exclusivo à zona crítica (figura 19).

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

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

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

Figura 19

Começamos por atribuir o grupo da solicitação atual e depois o tipo de solicitação atual para a variável req. Damos UP novamente ao semáforo mutex e sinalizamos que o waiter pode receber outra solicitação (figura 20).

```
req.reqGroup = sh->fSt.waiterRequest.reqGroup;
req.reqType = sh->fSt.waiterRequest.reqType;

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

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

Figura 20

Função informChef()

Esta função representa a ação do empregado de mesa ao informar o chefe sobre um pedido realizado por um grupo.

Damos DOWN ao semáforo associado ao mutex, garantindo que apenas este processo possa aceder à zona crítica, atualizamos depois o estado do waiter para "INFORM_CHEF" e indicamos que o waiter informa o chef sobre o pedido de comida. Concluindo guardando o estado atualizado na memória compartilhada (figura 21).

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

// Update the waiter state to reflect that it is taking food order to chef
sh->fSt.st.waiterStat = INFORM_CHEF;
saveState(nFic, &sh->fSt);
```

Figura 21

Definimos o grupo que fez o pedido de comida (n) e indicamos que um pedido de comida foi feito (foodOrder = 1). Fazemos um Up no semáforo 'waitOrder', sinalizando ao chef que um pedido foi feito e que pode começar a preparar a comida. No final identificamos a mesa atribuída ao grupo n (figura 22).

```
// Inform the chef of the group that ordered food
sh->fSt.foodGroup = n;
sh->fSt.foodOrder = 1;

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

tableForGroup = sh->fSt.assignedTable[n];      //identify the table assigned to the group
```

Figura 22

Aguarda até que a ordem seja recebida pelo chef e depois o semáforo é usado para sincronizar o waiter com o chef, indicando que o pedido foi recebido pelo chef. Sinalizamos para a mesa específica que o pedido foi recebido pelo mesmo e está em preparação. Seguidamente, damos um Up no semáforo 'mutex', permitindo que outros processos acedam à zona crítica (figura 23).

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

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

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

Figura 23

Função takeFoodToTable()
Já esta função representa a ação do empregado de mesa ao levar a comida para a mesa do grupo.

Primeiramente começamos por dar DOWN ao semáforo associado ao mutex, garantindo que apenas este processo possa aceder à zona crítica. De seguida, o segundo comando atualiza o estado do waiter para "TAKE_TO_TABLE" no estado compartilhado sh->fSt, indicando que o waiter está a levar comida para a mesa. No final salvamos o estado atualizado (figura 24).

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

sh->fSt.st.waiterStat = TAKE_TO_TABLE;
saveState(nFic, &sh->fSt);
```

Figura 24

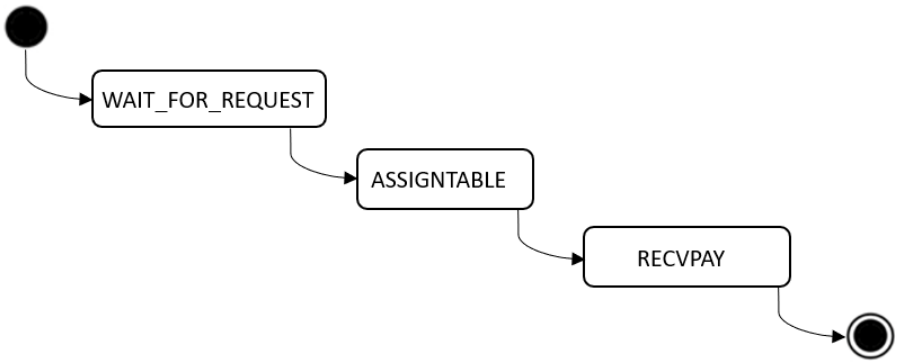
Começamos por dar um Up no semáforo ‘foodArrived’, sinalizando, ao grupo da mesa específica, que a comida está disponível para ser servida.
Seguidamente, damos um Up no semáforo mutex, permitindo que outros processos acedam à zona crítica (figura 25).

```
// Inform the group that the food is available
if (semUp(semgid, *sh->foodArrived) == -1) {
    perror("error on the up operation for semaphore access");
    exit(EXIT_FAILURE);
}

// TODO Insert your code here

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

Figura 25

Rececionista

Antes mesmo de avançarmos para a análise de cada uma das funções, apresentamos um pequeno diagrama (de estados), que representa toda a transição dos estados correspondente à entidade do Rececionista.



Função decideTabelOrWait()

Este método é responsável por tomar uma decisão para um grupo específico sobre se deve esperar por uma mesa disponível no restaurante ou se pode ser imediatamente acomodado.

```
static int decideTableOrWait(int n)
{
    //TODO insert your code here

    if(sh->fSt.st.groupStat[n] == ATRECEPTION && sh->fSt.assignedTable[n] == -1){
        int occupiedTables = 0;
        for(int num = 0; num < sh->fSt.nGroups; num++)
            if(sh->fSt.assignedTable[num] != -1)
                occupiedTables += 1 + sh->fSt.assignedTable[num];

        if(occupiedTables <= 1)
            return 1;
        if(occupiedTables == 2)
            return 0;
    }

    return -1;
}
```

Figura 26

A primeira condição verifica se o grupo identificado por n está atualmente na recepção (ATRECEPTION) e não possui uma mesa atribuída (sh->fSt.assignedTable[n] == -1). Isso indica que o grupo está pronto para ser acomodado e ainda não foi designado a uma mesa (figura 26).

A segunda condição verifica se o grupo não possui uma mesa atribuída e depois incrementa o contador de mesas ocupadas por cada mesa atribuída aos grupos.

O valor 1 + sh->fSt.assignedTable[num] representa o número real da mesa atribuída (é adicionado 1 para compensar o índice 0).

Se a condição (occupiedTables <= 1) for verdadeira (ou seja, há uma ou nenhuma mesa ocupada), o método retorna 1, indicando que o grupo pode ocupar a mesa 1.

Se a condição (occupiedTables == 2) for verdadeira (ou seja, exatamente duas mesas ocupadas), o método retorna 0, indicando assim que o grupo pode ocupar a mesa 0.

Se nenhuma destas condições for atendida, é retornado -1, indicando que o grupo tem que esperar.

Função decideNextGroup()

Esta função tem como objetivo encontrar o próximo grupo que está a aguardar por uma mesa e que pode ser acomodado imediatamente, ou seja, procura pelo próximo grupo que está na fila de espera.

A função itera sobre todos os grupos (num) existentes no restaurante, verificando cada um deles. Para cada grupo (num), chama a função decideTableOrWait(num). Verifica se a decisão retornada não é igual a -1 (ou seja, se o grupo pode ser imediatamente acomodado) e se o estado do grupo (groupRecord[num]) é igual a WAIT, indicando que ele está aguardando. Se encontrar um grupo que atenda a essas condições, retorna o identificador desse grupo (num), indicando que é o próximo grupo que deve ser acomodado, caso contrário, se nenhum grupo atender às condições especificadas, a função retorna -1, indicando que não há nenhum grupo imediatamente disponível para ser acomodado (figura 27).

```
static int decideNextGroup()
{
    //TODO insert your code here
    for(int num = 0; num < sh->fSt.nGroups; num++)
       if(decideTableOrWait(num) != -1 && groupRecord[num] == WAIT)
           return num;

    return -1;
}
```

Figura 27

Função waitForGroup()

Esta função tem como principal objetivo permitir que o recepcionista aguarde uma solicitação de um grupo.

Damos DOWN ao semáforo ‘mutex’, permitindo que o recepcionista entre na zona crítica e depois atualizamos o seu estado para "WAIT_FOR_REQUEST", indicando que aguarda por uma solicitação. No final guardamos o estado atualizado (figura 28).

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

// TODO insert your code here
sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;
saveState(nFic, &sh->fSt);
```

Figura 28

Damos Up ao semáforo ‘mutex’, saindo da zona crítica. O rececionista aguarda a chegada da solicitação do grupo através do semáforo receptionistReq. De seguida, entramos novamente na zona crítica (figura 29).

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

if (semDown (semgid, sh->receptionistReq) == -1)  {
    perror ("error on the up operation for semaphore access");
    exit (EXIT_FAILURE);
}

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

Figura 29

Atribuímos a solicitação do grupo à variável ret. Damos UP ao semáforo do mutex, saindo da zona crítica novamente. Seguidamente, damos também UP ao semáforo receptionistRequestPossible para indicar que o recepcionista está pronto para receber novas solicitações e no final retornamos a solicitação do grupo como resultado da função (figura 30).

```
ret = sh->fSt.receptionistRequest;

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

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

// TODO insert your code here

return ret;
```

Figura 30

Função provideTableOrWaitingRoom()

Esta função representa a ação do rececionista ao decidir se o grupo deve ocupar uma mesa ou aguardar numa sala de espera.

A função começa por dar um Down no semáforo 'mutex' para garantir acesso exclusivo à zona crítica (figura 31).

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

Figura 31

Depois alteramos o estado do recepcionista para ASSIGNTABLE. Esse estado é depois guardado (figura 32).

```
// TODO insert your code here
sh->fSt.st.receptionistStat = ASSIGNTABLE;
saveState(nFic, &sh->fSt);
```

Figura 32

Chamamos a função decideTableOrWait(n) para decidir se o grupo deve ser alocado para uma mesa ou colocado em espera.

O resultado é armazenado na variável choiceTable (figura 33).

```
int choiceTable = decideTableOrWait(n);
```

Figura 33

Se choiceTable for diferente de -1 (ou seja, uma mesa foi escolhida) libera o semáforo correspondente à mesa escolhida (sh->waitForTable[n]), atualizamos o estado do grupo para ATTABLE e atribui a mesa escolhida ao grupo n no estado geral (sh->fSt.assignedTable[n]).

Se choiceTable for -1 (indica que nenhuma está mesa disponível), atualizamos o estado do grupo para WAIT e incrementa o contador de grupos em espera (sh->fSt.groupsWaiting).

É depois dado um Up no semáforo ‘mutex’ para sair da zona crítica (figura 34).

```
if(choiceTable != -1){
    if (semUp (semgid, sh->waitForTable[n]) == -1) {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    groupRecord[n] = ATTABLE;

    sh->fSt.assignedTable[n] = choiceTable;

} else{

    groupRecord[n] = WAIT;
    sh->fSt.groupsWaiting++;
}

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

Figura 34

Função receivePayment()
Esta função simula a ação do rececionista a receber o pagamento de um grupo que já terminou a sua refeição neste restaurante.

Damos um Down no semáforo ‘mutex’ para acedermos à zona crítica (figura 35).

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

Figura 35

Seguidamente, recorremos à declaração e à inicialização das variáveis emptyTable (mesa ocupada pelo grupo n) e newTableForGroup (para a possível nova atribuição). Posteriormente, alteramos o estado do recepcionista para RECVPAY, guardando o estado atual (figura 36).

```
// TODO insert your code here
int emptyTable = sh->fSt.assignedTable[n];
int newTableForGroup;

sh->fSt.st.receptionistStat = RECVPAY;
saveState(nFic, &sh->fSt);
```

Figura 36

Libertamos a mesa ocupada pelo grupo n através de um Up no semáforo correspondente (tableDone[emptyTable]).
Atualização do registo do grupo n para DONE e remoção da associação da mesa com este grupo (figura 37).

```
if (semUp (semgid, sh->tableDone[emptyTable]) == -1)  {
    perror ("error on the down operation for semaphore access");
    exit (EXIT_FAILURE);
}

groupRecord[n] = DONE;
sh->fSt.assignedTable[n] = -1;
```

Figura 37

De seguida, verificamos se há grupos à espera (sh->fSt.groupsWaiting > 0).
Se existirem grupos à espera definimos o estado do recepcionista para ASSIGNTABLE e guardamos o estado atual.
Depois decide qual é o próximo grupo a ser atendido usando decideNextGroup().
Se houver um próximo grupo a ser atendido associa a mesa vazia (emptyTable) ao novo grupo (newTableForGroup), atualiza o estado desse novo grupo para ATTABLE, dá UP ao semáforo correspondente à mesa para o novo grupo.
No final decrementa o contador de grupos em espera (figura 38).

```
if(sh->fSt.groupsWaiting > 0){
    sh->fSt.st.receptionistStat = ASSIGNTABLE;
    saveState(nFic, &sh->fSt);

    newTableForGroup = decideNextGroup();

    if(newTableForGroup != -1){
        sh->fSt.assignedTable[newTableForGroup] = emptyTable;
        groupRecord[newTableForGroup] = ATTABLE;

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

        sh->fSt.groupsWaiting--;
    }
}
```

Figura 38

Finalmente, damos um Up no semáforo ‘mutex para permitir o acesso de outros processos à zona crítica (figura 39).

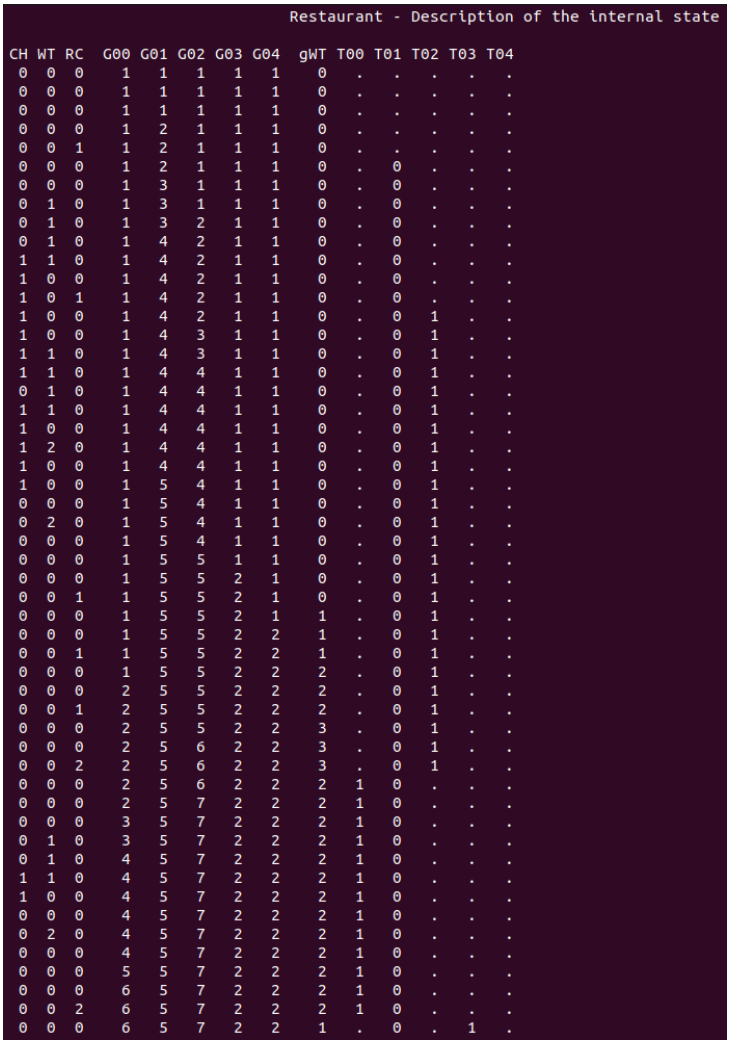
```
if (semUp (semgid, sh->mutex) == -1)  {
    perror ("error on the down operation for semaphore access (WT)");
    exit (EXIT_FAILURE);
}
```

Figura 39

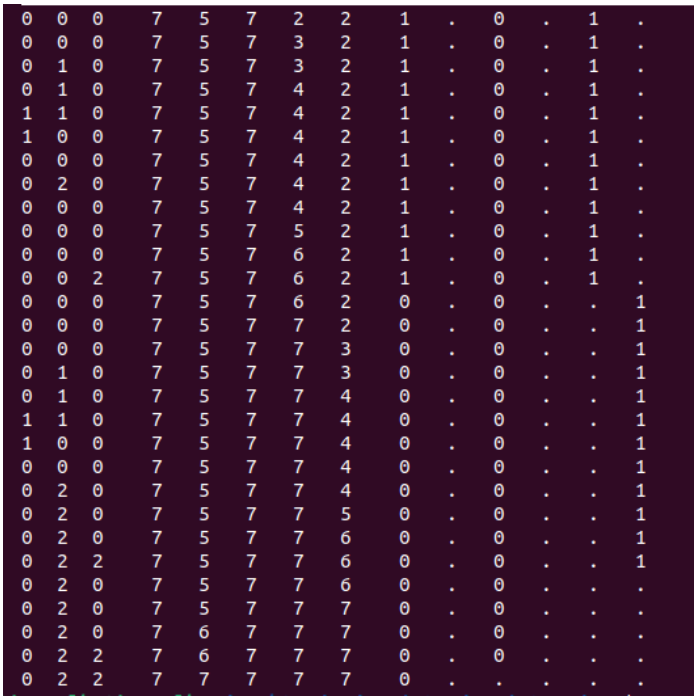
Exposição dos resultados obtidos

Focando-nos agora na parte final do relatório, vamos analisar o output gerado correndo, primeiramente, o teste presente no enunciado deste projeto:

1. Na diretoria ‘src’, corremos o comando “make all_bin”;
2. Na diretoria ‘run’, corremos o comando “./probSemSharedMemRestaurant”



```
                                Restaurant - Description of the internal state

CH WT RC  G00 G01 G02 G03 G04  gWT T00 T01 T02 T03 T04
 0  0  0    1   1   1   1   1    0   .   .   .   .   .
 0  0  0    1   1   1   1   1    0   .   .   .   .   .
 0  0  0    1   1   1   1   1    0   .   .   .   .   .
 0  0  0    1   2   1   1   1    0   .   .   .   .   .
 0  0  1    1   2   1   1   1    0   .   .   .   .   .
 0  0  0    1   2   1   1   1    0   .   0   .   .   .
 0  0  0    1   3   1   1   1    0   .   0   .   .   .
 0  1  0    1   3   1   1   1    0   .   0   .   .   .
 0  1  0    1   3   2   1   1    0   .   0   .   .   .
 0  1  0    1   4   2   1   1    0   .   0   .   .   .
 1  1  0    1   4   2   1   1    0   .   0   .   .   .
 1  0  0    1   4   2   1   1    0   .   0   .   .   .
 1  0  1    1   4   2   1   1    0   .   0   .   .   .
 1  0  0    1   4   2   1   1    0   .   0   1   .   .
 1  0  0    1   4   3   1   1    0   .   0   1   .   .
 1  1  0    1   4   3   1   1    0   .   0   1   .   .
 1  1  0    1   4   4   1   1    0   .   0   1   .   .
 0  1  0    1   4   4   1   1    0   .   0   1   .   .
 1  1  0    1   4   4   1   1    0   .   0   1   .   .
 1  0  0    1   4   4   1   1    0   .   0   1   .   .
 1  2  0    1   4   4   1   1    0   .   0   1   .   .
 1  0  0    1   4   4   1   1    0   .   0   1   .   .
 1  0  0    1   5   4   1   1    0   .   0   1   .   .
 0  0  0    1   5   4   1   1    0   .   0   1   .   .
 0  2  0    1   5   4   1   1    0   .   0   1   .   .
 0  0  0    1   5   4   1   1    0   .   0   1   .   .
 0  0  0    1   5   5   1   1    0   .   0   1   .   .
 0  0  0    1   5   5   2   1    0   .   0   1   .   .
 0  0  1    1   5   5   2   1    0   .   0   1   .   .
 0  0  0    1   5   5   2   1    1   .   0   1   .   .
 0  0  0    1   5   5   2   2    1   .   0   1   .   .
 0  0  1    1   5   5   2   2    1   .   0   1   .   .
 0  0  0    1   5   5   2   2    2   .   0   1   .   .
 0  0  0    2   5   5   2   2    2   .   0   1   .   .
 0  0  1    2   5   5   2   2    2   .   0   1   .   .
 0  0  0    2   5   5   2   2    3   .   0   1   .   .
 0  0  0    2   5   6   2   2    3   .   0   1   .   .
 0  0  2    2   5   6   2   2    3   .   0   1   .   .
 0  0  0    2   5   6   2   2    2   1   0   .   .   .
 0  0  0    2   5   7   2   2    2   1   0   .   .   .
 0  0  0    3   5   7   2   2    2   1   0   .   .   .
 0  1  0    3   5   7   2   2    2   1   0   .   .   .
 0  1  0    4   5   7   2   2    2   1   0   .   .   .
 1  1  0    4   5   7   2   2    2   1   0   .   .   .
 1  0  0    4   5   7   2   2    2   1   0   .   .   .
 0  0  0    4   5   7   2   2    2   1   0   .   .   .
 0  2  0    4   5   7   2   2    2   1   0   .   .   .
 0  0  0    4   5   7   2   2    2   1   0   .   .   .
 0  0  0    5   5   7   2   2    2   1   0   .   .   .
 0  0  0    6   5   7   2   2    2   1   0   .   .   .
 0  0  2    6   5   7   2   2    2   1   0   .   .   .
 0  0  0    6   5   7   2   2    1   .   0   .   1   .
```



```
0  0  0    7   5   7   2   2    1   .   0   .   1   .
0  0  0    7   5   7   3   2    1   .   0   .   1   .
0  1  0    7   5   7   3   2    1   .   0   .   1   .
0  1  0    7   5   7   4   2    1   .   0   .   1   .
1  1  0    7   5   7   4   2    1   .   0   .   1   .
1  0  0    7   5   7   4   2    1   .   0   .   1   .
0  0  0    7   5   7   4   2    1   .   0   .   1   .
0  2  0    7   5   7   4   2    1   .   0   .   1   .
0  0  0    7   5   7   4   2    1   .   0   .   1   .
0  0  0    7   5   7   5   2    1   .   0   .   1   .
0  0  0    7   5   7   6   2    1   .   0   .   1   .
0  0  2    7   5   7   6   2    1   .   0   .   1   .
0  0  0    7   5   7   6   2    0   .   0   .   .   1
0  0  0    7   5   7   7   2    0   .   0   .   .   1
0  0  0    7   5   7   7   3    0   .   0   .   .   1
0  1  0    7   5   7   7   3    0   .   0   .   .   1
0  1  0    7   5   7   7   4    0   .   0   .   .   1
1  1  0    7   5   7   7   4    0   .   0   .   .   1
1  0  0    7   5   7   7   4    0   .   0   .   .   1
0  0  0    7   5   7   7   4    0   .   0   .   .   1
0  2  0    7   5   7   7   4    0   .   0   .   .   1
0  2  0    7   5   7   7   5    0   .   0   .   .   1
0  2  0    7   5   7   7   6    0   .   0   .   .   1
0  2  2    7   5   7   7   6    0   .   0   .   .   1
0  2  0    7   5   7   7   6    0   .   0   .   .   .
0  2  0    7   5   7   7   7    0   .   0   .   .   .
0  2  0    7   6   7   7   7    0   .   0   .   .   .
0  2  2    7   6   7   7   7    0   .   0   .   .   .
0  2  2    7   7   7   7   7    0   .   .   .   .   .
```

Seguidamente, e para verificarmos possíveis existências de deadlocks, executamos o script disponibilizado pelo Professor - **run.sh**. O código foi executado 1000 vezes e pareceu não haver presença de deadlocks durante toda a execução.

Uma outra análise efetuada, foi a verificação dos outputs obtidos, após a execução do primeiro teste efetuado. Percebemos assim que ia de encontro à lógica pedida para o problema proposto.

Conclusão

Por fim, achamos que este segundo projeto nos possibilitou um desenvolvimento dos nossos conhecimentos em relação à execução e sincronização de processos e threads. Posto isto, achamos que o trabalho foi desenvolvido cumprindo com as diretrizes apresentadas, visto que os objetivos propostos foram alcançados.

Inicialmente deparamo-nos com algumas dificuldades, nomeadamente:
• Entender quais as operações a efetuar sobre os semáforos;
• Entender a organização das funções e dos ciclos de vida para as diferentes entidades.

Porém, e com a análise apresentada no início do relatório, foi nos mais fácil combater essas dificuldades iniciais.

Bibliografia

https://stackoverflow.com

Material do E-learning (guiões práticos)

http://www.ece.ufrgs.br/~fetter/ele213/sem.pdf

https://www.geeksforgeeks.org/

https://greenteapress.com/thinkos/html/thinkos013.html