CONTROLO DO FLUXO DE UM RESTAURANTE USANDO SEMÁFOROS

Sistemas Operativos

Tomás Costa - 89016
João Carvalho - 89059

Índice

Introdução

Como segundo trabalho prático da unidade curricular de Sistemas Operativos, foi pedido aos alunos que desenvolvessem uma aplicação em linguagem C que simulasse o fluxo de um restaurante com o uso de semáforos.

Um dos principais objetivos deste problema proposto passa pela compreensão dos mecanismos associados à execução e sincronização de processos e threads.

Este projeto tem como base código C fornecido pelo docente, pelo que será preciso apenas alterar funções nas 4 entidades que interagem no restaurante, sendo estas: o chefe, o garçom, o rececionista, e os grupos que chegam ao restaurante.

Antes de Implementar

O pasta fornecida pelo docente contem 3 diretorias:

- Run – contem os executáveis
- Src – contem todo o código usado
- Doc – contem documentação usando o comando doxygen

```
[aluno-2199:semaphore_restaurant Macbook$ ls -R .
doc     run     src

./doc:
Doxyfile

./run:
chef                            filter_log.awk                  receptionist_bin_64
chef_bin_64                     group                           run.sh
clean.sh                        group_bin_64                    waiter
config.txt                      probSemSharedMemRestaurant      waiter_bin_64
filter.sh                       receptionist

./src:
Makefile                        probSemSharedMemRestaurant.c    semaphore.h
config.txt                      semSharedMemChef.c              sharedDataSync.h
logging.c                       semSharedMemGroup.c             sharedMemory.c
logging.h                       semSharedMemReceptionist.c      sharedMemory.h
probConst.h                     semSharedMemWaiter.c
probDataStruct.h                semaphore.c
```

Segundo o enunciado do trabalho estas são a interações que se realizam no ciclo de vida de um restaurante:

```
Um grupo ao chegar dirige se ao recepcionista
Quando o recepcionista permitir, este dirige se a mesa, ou fica a espera
Quando chega a mesa, pede a comida ao waiter
O waiter leva o pedido ao chefe
O chefe cozinha
O chefe entrega a comida ao waiter
O waiter entrega a comida ao grupo
O grupo come
O grupo pede a conta ao recepcionista
O Recepcionista da o checkout ao Grupo

O chef so faz refeicao para 1 grupo de cada vez
```

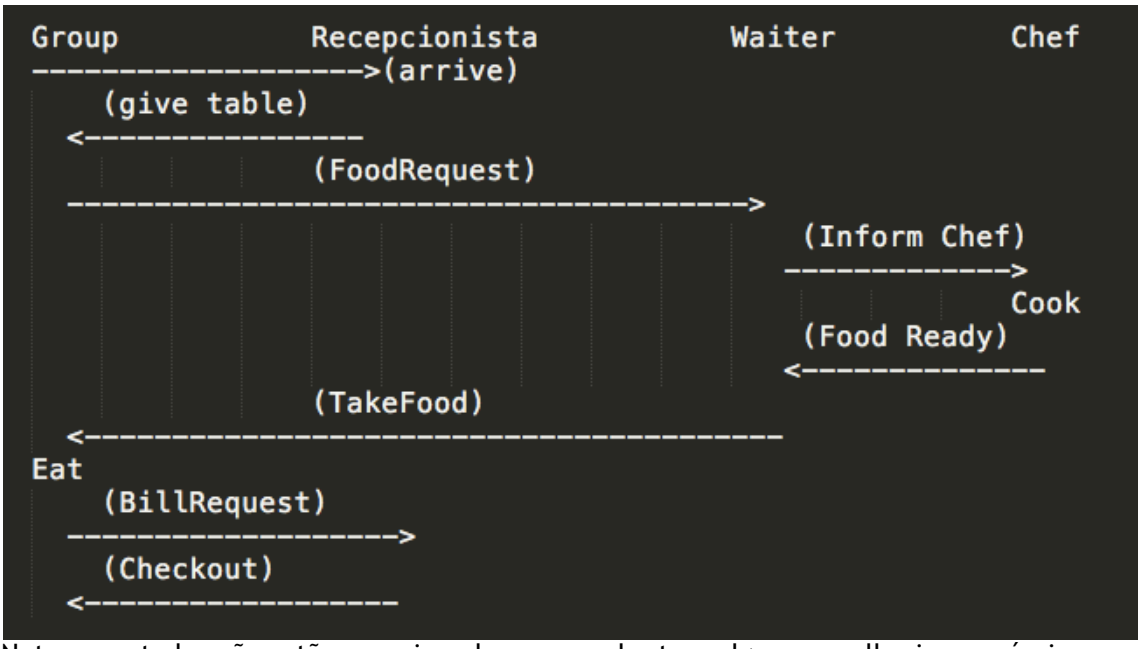
Caso queira testar as entidades do docente basta fazer $make all_bin
Cada entidade pode ser testada individualmente, para isto basta fazer $make <entidade>

Para este trabalho é apenas necessário modificarmos 4 ficheiros denominados semSharedMem<entidade>.c, e dentro deste ficheiros alterar algumas funções que estão assinaladas com um ToDo.

É relevante ainda mencionar que posteriormente irei mencionar tanto um garçom como um waiter, embora esteja a referir a mesma entidade (há um cruzamento de linguagem).

Análise de Interações

Antes de começar a escrever código, analisei muito bem o código fornecido, principalmente as constantes, os semáforos e a estruturas de dados partilhada. Após esta análise, comecei a desenhar um quadro de interações para melhor entender que entidades interagiam com quais e qual era essa interação.



Nota: os estados não estão mencionados e o quadro tem algumas melhorias possíveis, mas foi uma primeira versão para melhor entender as interações

Após desenhar este quadro de interações e como sugerido pelo professor comecei pela entidade do chefe, pois e a que tem menos interações e menos funcionalidades.

Constantes, Semáforos e Estados

Foi bastante importante anotar todas as constantes de estados, a estrutura de dados chamada FULL_STAT e STAT e também os semáforos que existiam no código fornecido, pois fui capaz de melhor perceber as interações que estavam a ocorrer e onde colocar certos semáforos

Foi desenhado este quadro com os semáforos, quais as entidades que o manipulam, em que funções o fazem e a sua situação no contexto do problema.

	Entidade		Situação		Função	
	up	down	up	down	up	down
-mutex	todas	todas	sair da região critica	entrar na região critica	quase todas	quase todas
-waitOrder	waiter	chef	assinala um pedido ao chef	chefe espera por pedidos	informChef()	waitForOrder()
-waiterRequest	chef grupo	waiter	assinala um pedido ao waiter	waiter espera por pedidos	processOrder() orderFood()	waitForClientOr-Chef()
-request-Received	waiter	grupo	assinala que recebeu o pedido	espera que waiter receba o pedido	informChef()	orderFood()
-foodArrived	waiter	grupo	comida pronta a entregar a mesa	espera pela comida	takeFoodTo-Table()	waitFood()
-receptionist-Req	grupo	receptionist	solicitar o receptionist	espera por pedidos dos grupos	checkOutAt-Reception() checkInAt-Reception()	waitForGroups()
-receptionist-RequestPossible	receptionist	grupo	sinaliza que está disponível	esperar que o receptionist esteja disponível	waitForGroups()	checkOutAt-Reception() checkInAt-Reception()
-waiterRequest-Possible	waiter	grupo chef	sinaliza que está disponível	esperar que o waiter esteja disponível	waitForClientOr-Chef()	orderFood() processOrder()
-orderReceived	chef	waiter	quando o chef termina de cozinhar	espera que o chef cozinhe	waitForOrder()	informChef()
-waitForTable	receptionist	grupo	quando o grupo ja tem uma mesa	espera pela mesa	receivePayment() provideTableOr-WaitingRoom()	checkInAt-Reception()
-tableDone	receptionist	grupo	confirmar que o pagamento foi completado	espera pelo pagamento	receivePayment()	checkOutAt-Reception()

Nesta estrutura de dados estão presentes todas as flags, arrays e variáveis que iremos usar para as várias entidades:

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

    /** \brief number of groups */
    int nGroups;
    /** \brief number of groups waiting for table */
    int groupsWaiting;

    /** \brief estimated start time of groups */
    int startTime[MAXGROUPS];
    /** \brief estimated eat time of groups */
    int eatTime[MAXGROUPS];

    /** \brief saves the table that is being used by each group */
    int assignedTable[MAXGROUPS];

    /** \brief flag of food request from waiter to chef */
    int foodOrder;
    /** \brief group associated to food request from waiter to chef */
    int foodGroup;


    /** \brief used by groups to store request to receptionist */
    request receptionistRequest;

    /** \brief used by groups and chef to store request to waiter */
    request waiterRequest;


} FULL_STAT;
```

```
typedef struct {
    /** \brief receptionist state */
    unsigned int receptionistStat;
    /** \brief waiter state */
    unsigned int waiterStat;
    /** \brief chef state */
    unsigned int chefStat;
    /** \brief group state array */
    unsigned int groupStat[MAXGROUPS];

} STAT;
```

Esta imagem contem todos os estados que as diferentes entidades podem assumir:

```
Estados:
    Chef-
        WAIT_FOR_ORDER
        COOK
        REST

    Waiter-
        WAIT_FOR_REQUEST
        INFORM_CHEF
        TAKE_TO_TABLE

    Receptionist-
        ASSIGN_TABLE
        RECVPAY

    Groups-
        GOTOREST
        ATRECEPTION
        FOOD_REQUEST
        WAIT_FOR_FOOD
        EAT
        CHECKOUT
        LEAVING
```

Entidade Chef

O chefe interage apenas com o garçom e tem apenas duas funções (waitForOrder e processOrder) pelo que e uma entidade muito simples, como vemos aqui demonstrado pelo seu ciclo de vida:

```
/* simulation of the life cycle of the chef */

int nOrders=0;
while(nOrders < sh->fSt.nGroups) {
    waitForOrder();
    processOrder();

    nOrders++;
}
```

waitForOrder()

Para esta função e necessário o Chef dar down ao semáforo waitOrder para poder esperar que chegue um pedido. Depois disso atribuímos valor de 0 a foodOrder pois e uma flag que server para mostrar que o chefe já esta a tratar de um pedido, alteramos o estado do chefe para COOK e atualizamos o valor de lastGroup para o valor em memória partilhada do grupo que pediu a comida e guardamos o estado, antes de terminarmos damos Up ao semáforo orderReceived, que o garçom deu Down para esperar que o Chef cozinhe.

```
static void waitForOrder ()
{

    if (semDown (semgid, sh->waitOrder) == -1) {
        perror ("error on the down operation for semaphore access (PT)");
            exit (EXIT_FAILURE);
    }
        //TODO insert your code here
    if (semDown (semgid, sh->mutex) == -1) {
        //aqui nao e down operation?
        perror ("error on the down operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

        //TODO insert your code here
    sh->fSt.foodOrder = 0;
    sh->fSt.st.chefStat = COOK;
    lastGroup = sh->fSt.foodGroup;
    saveState(nFic,&sh->fSt);

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

processOrder()

O Chef gasta algum tempo para cozinhar, de seguida espera que o garçom fique disponível, depois disso atualizamos o estado do Chef e atualizamos os valores do pedido com o número do ultimo grupo a pedir e com FOODREADY para sinalizar que a comida esta pronta a levar, para terminar damos Up ao semáforo waiterRequest que o waiter da Down para esperar por pedidos.

```
static void processOrder (){
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));

    //this may only happen when waiter is available
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
        //TODO insert your code here
    if (semDown (semgid, sh->mutex) == -1) {
        //aqui nao e down operation?
        perror ("error on the down operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

        //TODO insert your code here
    sh->fSt.st.chefStat = WAIT_FOR_ORDER;
    sh->fSt.waiterRequest.reqGroup=lastGroup;
    sh->fSt.waiterRequest.reqType=FOODREADY;
    saveState(nFic,&sh->fSt);

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

Entidade Waiter

O waiter interage com o Chef e com os grupos e tem três funções, aqui está demonstrado o seu ciclo de vida.

```
/* simulation of the life cycle of the waiter */
int nReq=0;
request req;
while( nReq < sh->fSt.nGroups*2 ) {
    req = waitForClientOrChef();
    switch(req.reqType) {
        case FOODREQ:
                informChef(req.reqGroup);
                break;
        case FOODREADY:
                takeFoodToTable(req.reqGroup);
                break;
    }
    nReq++;
}
```

Daqui retiramos que a função waitForClientOrChef() funciona para ambos os pedidos e conforme o reqType muda que função vai ser executada posteriormente, pelo que temos que mudar o valor de req.reqType dentro da primeira função.

waitForClientOrChef()

Primeiro começamos por guardar o estado do garçom como a espera de pedidos e guardamos o seu estado.
Seguidamente fazemos o Down ao semáforo waiterRequest para esperar por pedidos, após este semáforo atualizamos o valor do pedido req com o valor do pedido guardado na estrutura de dados partilhada fSt.waiterRequest. Para terminar fazemos o Up ao semáforo waiterRequestPossible pois o waiter já está disponível para ser requisitado.

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

    // change state and save
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
    saveState(nFic,&sh->fSt);

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

    // TODO insert your code here

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

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

  //verificar se e chef ou nao, mais qulquer coisa
  // semdown e semup aqui dentro, mudar valor de request req
  req = sh->fSt.waiterRequest;

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

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

informChef()

Para informamos o chefe, começamos por mudar o estado do waiter, vamos buscar o ID da mesa com o array assignedTable do groupID passado como argumento, atualizamos a foodOrder para 1 para sinalizar que foi feito um pedido para o chefe e inserimos o valor do groupID(n) no foodGroup.
De seguida e feito o Up do requestReceived[tableID] pois e preciso sinalizar que o pedido dessa mesa foi feito e também Up a waitOrder para sinalizarmos o Chef que estava a espera de pedidos.
Para finalizar e feito o Down a orderReceived para o waiter poder esperar que o chefe cozinhe.

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

    // change and save state
    int tableId = sh->fSt.assignedTable[n];
    sh->fSt.st.waiterStat = INFORM_CHEF;
    sh->fSt.foodOrder = 1;
    sh->fSt.foodGroup = n;
    saveState(nFic,&sh->fSt);

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

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

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

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

}
```

takeFoodToTable()

A função takeFoodTable e bastante simples, basta mudarmos o estado do waiter, atualizarmos o foodGroup para o groupID passado como argumento, e no final dar Up ao semáforo foodArrived[tableID] (vamos buscar tableID da mesma forma que explicamos la em cima), para sinalizar chegada da comida a mesa, assim o grupo para de esperar.

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

    // change and save state
    int tableId = sh->fSt.assignedTable[n];
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    sh->fSt.foodGroup = n;
    saveState(nFic,&sh->fSt);

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

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

}
```

Entidade Receptionist

O Receptionist interage apenas com o grupo e aqui em baixo está o seu ciclo de vida:

```
/* simulation of the life cycle of the receptionist */
int nReq=0;
request req;
while( nReq < sh->fSt.nGroups*2 ) {
    req = waitForGroup();
    switch(req.reqType) {
        case TABLEREQ:
              provideTableOrWaitingRoom(req.reqGroup); //TODO param should be groupid
              break;
        case BILLREQ:
              receivePayment(req.reqGroup);
              break;
    }
    nReq++;
}
```

Aqui está presente a função waitForGroup() que funciona para ambos os pedidos e conforme o tipo de pedido muda que função vai ser executada posteriormente, pelo que temos que mudar o valor de req.reqType dentro da primeira função.

decideTableOrWait()

Esta função verifica se existe mesas disponíveis.
Para isso temos que percorrer o array das mesas e se mais de MAXTABLES estiverem usadas, return -1, se não return tableid.

```
static int decideTableOrWait(int n)
{
    //percorrer array mesas, se mais de maxtables estiverem usadas, return -1, else return tableid
    //ver se o grupo ja comeu pq pode estar assigned e dps acabar de comer
    //TODO insert your code here
    int flagMesa0 = 0, flagMesa1 = 0;
    for (int i = 0; i <= sh->fSt.nGroups; i++)
    {
        if (sh->fSt.assignedTable[i] == 0)
            flagMesa0 = 1;
        else if(sh->fSt.assignedTable[i] == 1)
            flagMesa1 = 1;
    }

    if(flagMesa0 == 1 && flagMesa1 == 1)
        return -1;
    if(flagMesa0 == 1)
        return 1;
    return 0;
    //Estamos a retornar o numero de mesas disponiveis e nao o id da mesa

}
```

decideNextGroup()

Esta função vai escolher um grupo que esta em espera para ocupar uma mesa.
Para isso basta devolver o primeiro groupRecord que está em WAIT.

```
static int decideNextGroup()
{

    for (int i = 0; i < sh->fSt.nGroups; i++)
    {
        if(groupRecord[i] == WAIT)
            return i;
    }
    return -1;
    //os grupos entram no groupRec por ordem logo o primeiro em WAIT e quem esta a mais tempo

}
```

waitForGroup()

Primeiro começamos por guardar o estado do Receptionist como a espera de pedidos e guardamos o seu estado.
Seguidamente fazemos o Down ao semáforo recepcionistReq para esperar por pedidos, após este semáforo atualizamos o valor do pedido ret com o valor do pedido guardado na estrutura de dados partilhada fSt.recepcionistRequest e atribuímos o valor de WAIT ao groupRecord do grupo desse pedido.
Para terminar fazemos o Up ao semáforo recepcionistRequestPossible pois o rececionista já se encontra disponível a ser requisitado.

```
static request waitForGroup()
{
    request ret;

    //rc entrar save state e sair, 2 sem down e 2 sem up

    if (semDown (semgid, sh->mutex) == -1)  {
        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);

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

    // TODO insert your code here
    if (semDown (semgid, sh->receptionistReq) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

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

    ret = sh->fSt.receptionistRequest;
    groupRecord[ret.reqGroup] = WAIT;

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

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

    return ret;

}
```

provideTableOrWaitingRoom()

Para arranjar uma mesa ao grupo, começamos por mudar o estado do Receptionist.

De seguida chamamos a função decideTableOrWait() que verifica se há mesas disponíveis, se houver (return != -1) o Receptionist entrega uma mesa ao grupo se não (return == -1) o grupo tem que esperar e incrementamos 1 nos grupos a espera.

No caso de atribuir uma mesa ao grupo faz um Up ao waitForTable para o grupo saber que já tem uma mesa. Por fim guarda-se na estrutura de dados partilhada fSt.assignedTable o id da mesa(return) associadada ao grupo passado como argumento.

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

    //save state, decide table or wait
    //manda esperar se mesas estiverem cheias
    // TODO insert your code here
    sh->fSt.st.receptionistStat = ASSIGNTABLE;
    saveState(nFic,&sh->fSt);

    int fl = decideTableOrWait(n);
    if (fl == -1)
    {
        //groupRecord[n]=WAIT;
        sh->fSt.groupsWaiting++;

    }
    else
    {
        //return fl tem o id da mesa
        groupRecord[n]=ATTABLE;
        if (semUp (semgid, sh->waitForTable[n]) == -1) {
            perror ("error on the down operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
        sh->fSt.assignedTable[n] = fl;

    }


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

}
```

receivePayment()

Para receber o pagamento, começámos por alterar o estado do Receptionist.
Depois damos Up à tableDone para confirmar que o pagamento da mesa(tableid) foi completo, e os clientes que estavam nessa mesa já podem sair.
A seguir chamamos a função decideNextGroup() para escolher outro grupo(grid) que está à espera para ocupar a mesa, se houver algum grupo à espera fazemos um Up ao waitForTable para o novo grupo saber que já tem uma mesa.
Seguidamente guarda-se na estrutura fSt.assignedTable a mesa(tableid) associada ao novo grupo e diminuímos em 1 os grupos a espera.
Por fim atribuímos -1 a mesa atribuída ao grupo que acabou a refeição.

```
static void receivePayment (int n)
{   //contem n do grupo
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // TODO insert your code here
    //descobrir qual a mesa do grupo
    //no do prof o id do grupo da reset, ou seja volta a 0 se o 0 comer e acabar
    sh->fSt.st.receptionistStat = RECVPAY;
    saveState(nFic,&sh->fSt);

    int tableId = sh->fSt.assignedTable[n];

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

    groupRecord[n]= DONE;
   int grid = decideNextGroup();
   //so entramos aqui se houver algum grupo a espera, mas mesmo que nao haja a mesa fica livre
   if (grid != -1)
   {

       if (semUp (semgid, sh->waitForTable[grid]) == -1) {
           perror ("error on the down operation for semaphore access (WT)");
           exit (EXIT_FAILURE);
       }
       //o n tem o grupo que estava na mesa que vai ficar livre
       groupRecord[grid] = ATTABLE;
       sh->fSt.assignedTable[grid] = tableId;

       sh->fSt.groupsWaiting--;

   }

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

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

}
```

Entidade Grupo

O Grupo interage com o Receptionist e com o waiter e aqui em baixo está o seu ciclo de vida:

```
/* simulation of the life cycle of the group */
goToRestaurant(n);
checkInAtReception(n);
orderFood(n);
waitFood(n);
eat(n);
checkOutAtReception(n);
```

goToRestaurant()
  Esta função apresenta o tempo que o grupo demora a chegar ao restaurante.
eat()
  Esta função apresenta o tempo que o grupo demora a chegar a comer.

checkInAtReception()

Esta função começa por solicitar o Receptionist, e espera que ele esta disponível, para poder pedir uma mesa, para isso fazemos Down ao receptionistRequestPossible.
Depois mudamos o estado do groupRecord para ATRECEPTION, o tipo de pedido do receptionist(reqType) para o pedido de mesa(TABLEREQ) e ainda atribuímos o ID do grupo ao req.reqGroup, além disso fazemos Up ao receptionistReq para sinalizar o Receptionist que estava a espera de grupos. Por fim damos Down ao waitForTable para o grupo esperar por uma mesa.

```
static void checkInAtReception(int id)
{
    // TODO insert your code here
    //SemDown
    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) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // TODO insert your code here
    //saveState
    sh->fSt.st.groupStat[id] = ATRECEPTION;
    sh->fSt.receptionistRequest.reqGroup = id;
    sh->fSt.receptionistRequest.reqType = TABLEREQ;
    saveState(nFic, &sh->fSt);
    //nFOODREQ-nBILLREQ da o numero de mesas sentadas

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

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

    // TODO insert your code here
        if (semDown (semgid, sh->waitForTable[id]) == -1) {
            perror ("error on the down operation for semaphore access (CT)");
            exit (EXIT_FAILURE);
        }

}
```

orderFood()

Para o grupo pedir a comida, fazemos Down ao waiterRequestPossible para o grupo esperar que o waiter esteja disponível, de seguida mudamos o estado do grupo e o tipo de pedido do waiter(reqType) para o pedido de comida(FOODREQ), assim como o reqGroup que recebe o valor do ID do grupo. Depois damos Up ao waiterRequest para o waiter parar de esperar por um pedido. Finalmente temos que fazer Down do requestReceived para que o grupo espere que o seu pedido seja reconhecido pelo waiter. Como este semáforo requestReceived tem um tamanho de NUMTABLES e preciso indicar o pedido vem da mesa e não do grupo, para isso vamos buscar o ID da mesa atribuída a esse grupo com a array assignedTable.

```
static void orderFood (int id)
{
    // TODO insert your code here
    //SemDown
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    int idTable = sh->fSt.assignedTable[id];

    sh->fSt.waiterRequest.reqGroup = id;
    sh->fSt.waiterRequest.reqType = FOODREQ;
    sh->fSt.st.groupStat[id]=FOOD_REQUEST;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    // TODO insert your code here

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

    //SemDown
    // FALTA AQUI O NUMERO DA MESA como arg do request
    if (semDown (semgid, sh->requestReceived[idTable]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

}
```

waitFood()

Nesta função alteramos o estado do grupo para WAIT_FOR_FOOD e fazemos um Down ao foodArrived[mesa] para que a mesa espere pela comida e o grupo possa comer. Após isto mudamos de novo o estado do grupo mas agora para EAT, pois já vai estar a comer.

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

    // TODO insert your code here
    //SaveState
    int idTable = sh->fSt.assignedTable[id];
    sh->fSt.st.groupStat[id]= WAIT_FOR_FOOD;
    saveState(nFic, &sh->fSt);

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

    //IR BUSCAR ID DA MESA
    if (semDown (semgid, sh->foodArrived[idTable]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    // TODO insert your code here
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    // TODO insert your code here
    //SaveState
    sh->fSt.st.groupStat[id]=EAT;
    saveState(nFic, &sh->fSt);

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

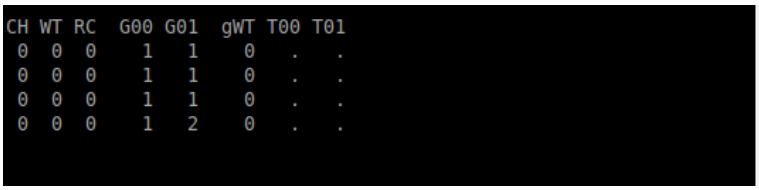
checkOutAtReception()

Nesta função o grupo faz o check out na receção, para isso faz-se Down no receptionistRequestPossible para que o grupo espere que o receptionist esteja disponível para fazer o pedido, depois mudamos o reqType do receptionistRequest e também o estado do grupo.
Fazemos Up ao receptionistReq para que o receptionist pare de esperar pelo pedido do grupo, além disso fazemos Down ao tableDone[mesa] para que o grupo espere pelo pagamento. Quando o pagamento e efetuado, alteramos o estado do grupo para LEAVING.

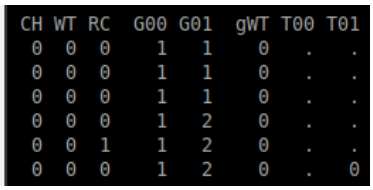
```
static void checkOutAtReception (int id)
{
    // TODO insert your code here
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    int idTable = sh->fSt.assignedTable[id];

    // TODO insert your code here
    sh->fSt.receptionistRequest.reqGroup = id;
    sh->fSt.receptionistRequest.reqType = BILLREQ;
    sh->fSt.st.groupStat[id]=CHECKOUT;
    saveState(nFic, &sh->fSt);

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

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    // TODO insert your code here
    if (semDown (semgid, sh->tableDone[idTable]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    // TODO insert your code here
    sh->fSt.st.groupStat[id] = LEAVING;
    saveState(nFic, &sh->fSt);

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

}
```

Método Análise e Solução de Deadlocks

A analise deste código foi feita individualmente, o que significa que cada entidade foi testada com os binários que o professor forneceu.
Durante o trabalho fomos encontrando varias situações de Deadlock, quando nos deparávamos com uma situação destas analisávamos onde e que o processo parava e comparávamos o output do código com deadlock com o output da solução.
Através disto conseguimos perceber melhor onde e que tínhamos que trabalhar para solucionar o deadlock. Como vemos neste exemplo o programa origina um deadlock quando o Rececionista não atualiza o seu primeiro estado.



CH	WT	RC	G00	G01	gWT	T00	T01
0	0	0	1	1	0	.	.
0	0	0	1	1	0	.	.
0	0	0	1	1	0	.	.
0	0	0	1	2	0	.	.



CH	WT	RC	G00	G01	gWT	T00	T01
0	0	0	1	1	0	.	.
0	0	0	1	1	0	.	.
0	0	0	1	1	0	.	.
0	0	0	1	2	0	.	.
0	0	1	1	2	0	.	.
0	0	0	1	2	0	.	0

Resultados Obtidos

Resultados obtidos com esta configuração:

```
#ngroups
2
#startTime timeToEat
15 20
25 10
```

CH	WT	RC	G00	G01	gWT	T00	T01
0	0	0	1	1	0	.	.
0	0	0	1	1	0	.	.
0	0	0	1	1	0	.	.
0	0	0	1	2	0	.	.
0	0	1	1	2	0	.	.
0	0	0	1	2	0	.	0
0	0	0	1	3	0	.	0
0	1	0	1	3	0	.	0
1	1	0	1	3	0	.	0
1	0	0	1	3	0	.	0
1	0	0	1	4	0	.	0
1	0	0	2	4	0	.	0
1	0	1	2	4	0	.	0
1	0	0	2	4	0	1	0
1	0	0	3	4	0	1	0
1	1	0	3	4	0	1	0
0	1	0	3	4	0	1	0
1	1	0	3	4	0	1	0
1	0	0	3	4	0	1	0
1	2	0	3	4	0	1	0
1	0	0	3	4	0	1	0
1	0	0	3	5	0	1	0
1	0	0	4	5	0	1	0
0	0	0	4	5	0	1	0
0	2	0	4	5	0	1	0
0	2	0	4	6	0	1	0
0	2	2	4	6	0	1	0
0	2	0	4	6	0	1	.
0	2	0	4	7	0	1	.
0	2	0	5	7	0	1	.
0	2	0	6	7	0	1	.
0	2	2	6	7	0	1	.
0	2	2	7	7	0	.	.

(Página Seguinte) (Ligeiramente deslocado pois não e uma única imagem)

```
#ngroups
5
#startTime timeToEat
25 30
100 30
100 20
200 10
25 10
```

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

Conclusões

Concluindo o trabalho, é importante mencionar que o código não foi desenvolvido com o intuito de obter uma melhor otimização do mesmo, mas sim com o principal objetivo do funcionamento de todas as funcionalidades requeridas.

No entanto, muitas das funções desenvolvidas foram feitas de modo a adaptarem a mudanças de constantes definidas em probConst.c (exceto caso o numero de mesas mude, mas neste projeto essa constante não e alterada).

É também importante salientar que este trabalho e importante nesta altura pois é uma boa revisão dos mecanismos associados à execução e sincronização de processos e threads .

Bibliografia

Slides da Unidade Curricular de Sistemas Operativos

https://www.geeksforgeeks.org/operating-system-dining-philosopher-problem-using-semaphores/

Documentação fornecida por Doxyfile