Trabalho 2

Restaurante

Trabalho realizado por:

Hugo Sousa 112733

Diogo Guedes 114256

Introdução

Na sequência do programa proposto, este relatório tem como objetivo explicar os raciocínios utilizados para formular o código necessário para a correta sincronização dos processos e threads fornecidas no programa inicial. O trabalho consiste num jantar, onde funcionam em simultâneo 4 identidades, o receptionist, o waiter, o chef e os grupos que tencionam ter uma refeição.

- *Group* : Cada grupo que tenciona jantar no restaurante, após chegar, dirige-se ao recepcionista que lhe irá dizer se há mesas disponíveis das duas ou não. Depois de obter uma mesa, o grupo faz o seu pedido com o empregado, espera que este chegue e começa a comer assim que o empregado o traga para a mesa. No fim, o grupo dirige-se novamente ao rececionista para pagar e sair o estabelecimento.

- *Waiter* : Recebe o pedido do grupo e leva-o ao chef. Quando esta estiver pronta, leva a comida á mesa.

- *Chef* : O chef recebe o pedido que o empregado lhe traz e começa a preparar a comida. Quando estiver pronta, avisa o empregado para levar esta á mesa.

- *Receptionist* : O rececionista recebe o grupo e decide a mesa que lhes irá atribuir. Depois do jantar, trata do pagamento com o grupo.

Chef

Serão apresentadas as funções que desenvolvemos e uma devida explicação, para facilitar a leitura e a compreensão da entidade Chef.

Waitfororder()

```
static void waitForOrder ()
{

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

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

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


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

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

Nesta função o chef espera que seja efetuado um pedido e que o Waiter lho entregue.

- **Aguardar o pedido de comida:** Primeiramente, o chef aguarda um pedido de comida, uma operação ‘down’ do semáforo.

- **Entrar na região critica:** O chefe entra numa região crítica usando um semáforo (sh->mutex). Isso garante acesso exclusivo aos dados compartilhados

- **Atualizar estado e salvar:** O chefe atualiza seu estado interno. Ele salva o último grupo que fez um pedido de comida (lastGroup), define seu estado como cozinhando (COOK), e salva o estado atual do restaurante usando a função saveState.

- **Sair da região critica:** O chefe sai da região crítica, permitindo que outros processos acessem os dados compartilhados.

- **Reconhecer o pedido recebido:** O chefe reconhece o pedido recebido realizando uma operação de up (signal) no semáforo sh->orderReceived.

Processorder()

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

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

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

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

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

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

A função é usada pelo chef no processo de preparação de comida e comunica com o waiter assim que esta estiver pronta:

- **Simulação de tempo de atraso:** Simula o tempo que o pedido demora a ser cozinhado.

- **Esperar pela possibilidade do waiter:** O chef espera pela possibilidade de receber um pedido do waiter. Isso é sincronizado usando o semáforo sh->waiterRequestPossible.

- **Entrar na região critica:** O chef entra em uma região crítica usando um semáforo (sh->mutex). Isso garante acesso exclusivo aos dados compartilhados durante a atualização dos estados internos.

- **Atualizar o estado e salvar:** O chef atualiza o seu estado interno, definindo o estado como WAIT_FOR_ORDER. Ele também prepara a estrutura de pedido (waiterRequest) indicando que a comida está pronta (FOODREADY) e associando o último grupo que fez um pedido.

- **Sair da região critica:** O chef sai da região crítica, permitindo que outros processos acessem os dados compartilhados.

- **Avisar o waiter:** O chef avisa o waiter que a comida está pronta, realizando uma operação de up (signal) no semáforo sh->waiterRequest.

Group

Em seguida está a implementação da identidade group, que é responsável por fazer o pedido ao waiter e depois da sua refeição pagar e sair.

CheckinAtReception()

```c
static void checkInAtReception(int id)
{
    // TODO insert your code here
    // Esperar pela disponibilidade do recepcionista
    if (semDown(semgid, sh->receptionistRequestPossible) == -1) {
        perror("error on the down operation for receptionistRequestPossible semaphore (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);
    }

    // TODO insert your code here

    sh->fSt.st.groupStat[id] = ATRECEPTION;
    saveState(nFic, &sh->fSt);
    // Enviar pedido de mesa ao recepcionista
    sh->fSt.receptionistRequest.reqType = TABLEREQ;
    sh->fSt.receptionistRequest.reqGroup = id;
    //

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

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

    // Esperar pela atribuição de mesa
    if (semDown(semgid, sh->waitForTable[id]) == -1) {
        perror("error on the down operation for waitForTable semaphore (CT)");
        exit(EXIT_FAILURE);
    }
    //

}
```

- **Esperar pela disponibilidade do receptionist:** O grupo espera pela disponibilidade do recepcionista usando o semáforo sh->receptionistRequestPossible.

- **Entrar na região critica:** O grupo entra numa região crítica usando um semáforo (sh->mutex). Isso garante acesso exclusivo aos dados compartilhados durante a atualização dos estados internos.

- **Atualizar estado e sair:** O grupo atualiza seu estado interno, indicando que está na receção (ATRECEPTION). Ele também prepara a estrutura de pedido para o rececionista indicando que deseja uma mesa (TABLEREQ) e associa o ID do grupo ao pedido.

```
// Enviar pedido de mesa ao recepcionista
sh->fSt.receptionistRequest.reqType = TABLEREQ;
sh->fSt.receptionistRequest.reqGroup = id;
//
```

- **Sair da região critica:** O grupo sai da região crítica, permitindo que outros processos acessem os dados compartilhados.

- **Avisar o rececionista:** O grupo sinaliza ao recepcionista que fez um pedido, realizando uma operação de up (signal) no semáforo sh->receptionistReq.

orderFood()

```c
static void orderFood (int id)
{
    // TODO insert your code here
    // Esperar pela disponibilidade do waiter
    if (semDown(semgid, sh->waiterRequestPossible) == -1) {
        perror("error on the down operation for waiterRequestPossible semaphore (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);
    }

    // TODO insert your code here
    sh->fSt.st.groupStat[id] = FOOD_REQUEST;
    saveState(nFic, &sh->fSt);
    // Enviar pedido de comida ao waiter
    sh->fSt.waiterRequest.reqType = FOODREQ;
    sh->fSt.waiterRequest.reqGroup = id;
    //

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

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

    // Esperar pela confirmação do waiter de que o pedido foi recebido
    if (semDown(semgid, sh->requestReceived[sh->fSt.assignedTable[id]]) == -1) {
        perror("error on the down operation for requestReceived semaphore (CT)");
        exit(EXIT_FAILURE);
    }
    //
}
```

- **Esperar pela disponibilidade do waiter:** O grupo espera pela disponibilidade do garçom usando o semáforo sh->waiterRequestPossible.

- **Entrar na região critica:** O grupo entra em uma região crítica usando um semáforo (sh->mutex). Isso garante acesso exclusivo aos dados compartilhados durante a atualização dos estados internos.

- **Atualizar estado e salvar:** O grupo atualiza seu estado interno, indicando que está a fazer um pedido de comida (FOOD_REQUEST). Ele também prepara a estrutura de pedido para o waiter indicando que deseja comida (FOODREQ) e associa o ID do grupo ao pedido.

- **Sair da regiao critica:** O grupo sai da região crítica, permitindo que outros processos acessem os dados compartilhados.

- **Avisar waiter:** O grupo avisa o waiter que fez um pedido, realizando uma operação de up (signal) no semáforo sh->waiterRequest.

- **Esperar pela confirmação do waiter:** O grupo espera pela confirmação do waiter de que o pedido foi recebido usando o semáforo sh->requestReceived associado à mesa atribuída ao grupo.

waitFood()

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

    // TODO insert your code here

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

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

    // TODO insert your code here
    // Esperar pela chegada da comida
    // and waits until food arrives.
    if (semDown(semgid, sh->foodArrived[sh->fSt.assignedTable[id]]) == -1) {
        perror("error on the down operation for foodArrived semaphore (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);
    }

    // TODO insert your code here
    // Atualizar estado para EAT
    // It should also update state after food arrives.
    sh->fSt.st.groupStat[id] = EAT;
    saveState(nFic, &sh->fSt);
    //

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

- **Atualizar e salvar estado na região critica:** O grupo atualiza seu estado interno para indicar que está á espera da comida (WAIT_FOR_FOOD) e salva o estado.

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

// TODO insert your code here

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

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

- **Esperar pela comida:** O grupo espera que a comida chegue usando o semáforo sh->foodArrived associado à mesa atribuída ao grupo.

- **Atualizar e salvar estado na região critica (última vez):** O grupo atualiza o estado interno para indicar que está a comer (EAT) e salva o estado novamente.

CheckOutAtReception()

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

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


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

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

    if (semDown(semgid, sh->tableDone[sh->fSt.assignedTable[id]]) == -1) {
        perror("error on the down operation for tableDone semaphore (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);
    }

    // TODO insert your code here

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

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

}
```

- **Esperar pela disponibilidade do rececionista:** Primeiro,o grupo espera até que o semáforo sh->receptionistRequestPossible esteja disponível, indicando que o recepcionista está pronto para receber solicitações.

- **Entrar na região critica.**

- **Atualizar estado e salvar:** O grupo atualiza o estado interno para indicar que está a realizar o checkout (CHECKOUT) e salva o estado.

- **Enviar pedido de pagamento ao rececionista:** O grupo informa o sistema que deseja realizar o pagamento ao definir o tipo de solicitação como BILLREQ e especificando o grupo associado.

```
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] = CHECKOUT;
saveState(nFic, &sh->fSt);
// Enviar pedido de pagamento ao recepcionista
sh->fSt.receptionistRequest.reqType = BILLREQ;
sh->fSt.receptionistRequest.reqGroup = id;
//


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

- **Avisar o Rececionista e aguardar a Confirmação do Pagamento:** O grupo avisa o rececionista que está pronto para pagar e aguarda a confirmação de pagamento usando o semáforo sh->tableDone associado à mesa atribuída ao grupo.

- **Atualizar o estado e salvar, na região critica:** Após entrar na região critica, o grupo atualiza seu estado interno para indicar que está a sair(LEAVING) e salva o estado novamente.

Waiter

De seguida é apresentado o código de implementação do Waiter, sendo este responsável por receber o pedido de comida do grupo, entrega-o ao chef e quando a comida estiver pronta leva-a á mesa.

waitForClientOrChef()

- **Atualizar estado e salvar dentro da região critica:** O waiter atualiza o estado interno para indicar que está á espera de um pedido (WAIT_FOR_REQUEST) e salva o estado.

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

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

- **Aguardar pedido:** O waiter aguarda até que um pedido seja enviado usando o semáforo sh->waiterRequest.

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

- **Ler pedido na região critica:** O waiter lê o pedido do grupo ou do chef que está armazenado no estado interno.

```
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
req=sh->fSt.waiterRequest;
//

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

- **Avisar que novos pedidos são possiveis:** O waiter avisa que está pronto para receber novos pedidos usando o semáforo sh->waiterRequestPossible.

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

- **Retorna o pedido:** Por último, a função retorna o pedido.

informchef()

```c
static void informChef (int n)
{
    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.waiterStat=INFORM_CHEF;
    saveState(nFic,&(sh->fSt));
    sh->fSt.foodGroup=n;
    if (semUp (semgid, sh->requestReceived[sh->fSt.assignedTable[n]]) == -1)   {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    //

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


    // TODO insert your code here
    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);
    }
    //
```

- **Atualizar estado e salvar, na região critica:** O waiter atualiza seu estado interno para indicar que está a informar o chef (INFORM_CHEF) e salva o estado. Além disso, sinaliza ao grupo que o pedido foi recebido usando o semáforo sh->requestReceived.

- **Avisar o chef:** O waiter avisa ao chef usando o semáforo sh->waitOrder que está a levar um pedido de comida. Em seguida, aguarda o sinal do chef indicando que o pedido foi recebido usando o semáforo sh->orderReceived.

- **Retorna o pedido.**

TakeFoodToTable()

```c
static void takeFoodToTable (int n)
{
    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.waiterStat=TAKE_TO_TABLE;
    saveState(nFic,&(sh->fSt));
    if (semUp (semgid, sh->foodArrived[sh->fSt.assignedTable[n]]) == -1)  {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    //

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

- **Entrar na região critica:** O waiter entra na região crítica a usar um semáforo (sh->mutex). Isso garante acesso exclusivo aos dados compartilhados durante a atualização do estado interno.

- **Atualizar o estado e salvar:** O waiter atualiza o seu estado interno para indicar que está a levar a comida até à mesa (TAKE_TO_TABLE) e guarda o estado. Além disso, sinaliza ao grupo que a comida está disponível usando o semáforo sh->foodArrived.

- **Sair da região critica:** O empregado de mesa sai da região crítica, permitindo que outros processos acedam aos dados partilhados.

Receptionist

DecideTableOrWait()

```
static int decideTableOrWait(int n)
{
    //TODO insert your code here
    int tableFree[NUMTABLES];
    for (int i=0; i<NUMTABLES; i++){
       tableFree[i]=1;
    }
    for (int i=0; i<MAXGROUPS; i++){
       if (groupRecord[i]==ATTABLE){
           tableFree[sh->fSt.assignedTable[i]]=0;
       }
    }
    for (int i=0; i<NUMTABLES; i++){
       if (tableFree[i]==1) return i;
    }
    //

    return -1;
}
```

- Primeiramente, O array tableFree é inicializado e são postos todos os valores a 1, significando que as mesas estão livres. O segundo ciclo ‘for’, itera sobre os grupos existentes (MAXGROUPS) e, se um grupo estiver atualmente numa mesa (ATTABLE), marca a mesa correspondente como ocupada. De seguida, é itera sobre as mesas disponíveis (NUMTABLES) e retorna o identificador da primeira mesa livre encontrada. Por último, se não for encontrada nenhuma mesa livre, a função retorna -1 para indicar que o grupo deve esperar.

DecideNextGroup()

```
static int decideNextGroup()
{
    //TODO insert your code here
    for (int i=0; i<MAXGROUPS; i++){
       if (groupRecord[i]==WAIT) return i;
    }
    //

    return -1;
}
```

- Esta função percorre os grupos (MAXGROUPS) e verifica se um grupo está atualmente em estado de espera (WAIT).Se encontrar um grupo à espera, a função retorna imediatamente o identificador desse grupo. Se a função percorrer todos os grupos e não encontrar nenhum grupo à espera, retorna -1 para indicar que não há grupos para serem atendidos.

waitForGroup()

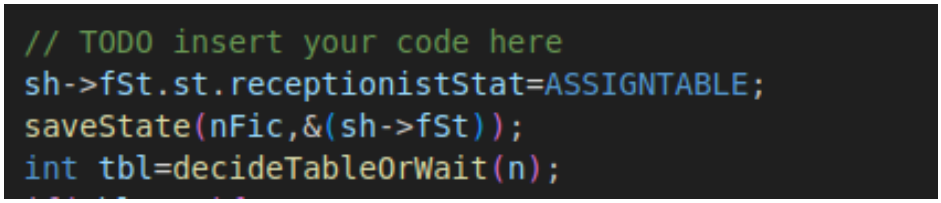
- **Atualização do estado e espera pelo pedido:** O rececionista atualiza o seu estado para indicar que está à espera de um pedido (WAIT_FOR_REQUEST). Salva o estado interno. Aguarda pelo pedido de um grupo, bloqueando-se no semáforo sh->receptionistReq.

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

if (semUp (semgid, sh->mutex) == -1)      {                                          /* exit critical region */
    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);
}
//
```

- **Leitura do pedido de sinalização de disponibilidade:** Após receber o pedido, o rececionista guarda o pedido (sh->fSt.receptionistRequest). Sinaliza que está disponível para receber novos pedidos, incrementando o semáforo sh->receptionistRequestPossible.

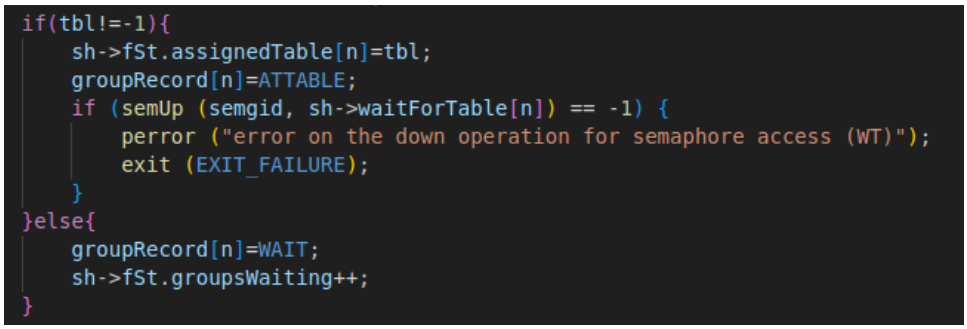
ProvideTableOrWaitingRoom()

- **Atualização do estado e decisão:** O rececionista atualiza o seu estado para indicar que está atribuindo uma mesa (ASSIGNTABLE). Salva o estado interno. Chama a função decideTableOrWait para decidir se o grupo n deve ocupar uma mesa ou esperar. A função retorna o número da mesa se o grupo deve ocupar uma mesa, ou -1 se o grupo deve esperar.



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

- **Atribuição de mesa ou espera:** Se a função decideTableOrWait retornar um número de mesa (tbl != -1), o grupo é atribuído a essa mesa. O grupo é marcado como ATTABLE no groupRecord. O semáforo sh->waitForTable[n] é incrementado para sinalizar que o grupo pode prosseguir. Se tbl for -1, o grupo é marcado como WAIT e o contador de grupos esperando (sh->fSt.groupsWaiting) é incrementado.



```
if(tbl!=-1){
    sh->fSt.assignedTable[n]=tbl;
    groupRecord[n]=ATTABLE;
    if (semUp (semgid, sh->waitForTable[n]) == -1) {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
}else{
    groupRecord[n]=WAIT;
    sh->fSt.groupsWaiting++;
}
```

- **Saída da região critica:** A função termina incrementando o semáforo sh->mutex para sair da região crítica.

Conclusão

Este trabalho possibilitou-nos desenvolver conhecimentos sobre os mecanismos associados à execução e sincronização de processos e threads. Posto isto, o trabalho foi desenvolvido de forma adequada e de acordo com todos os requisitos propostos, visto que os objetivos propostos foram alcançados.