SO - Trabalho 2

Restaurante

Ano Letivo: 2023/2024

Turma: P1

Guilherme Santos nº113893

Henrique Oliveira nº113585

Índice

1 Introdução **1**

2 Script semSharedMemGroup.c **1**
2.1 Função checkInAtReception 1
2.2 Função orderFood 2
2.3 Função waitFood 2
2.4 Função checkOutAtReception 3

3 Script semSharedMemReceptionist.c **3**
3.1 Função waitForGroup 4
3.2 Função provideTableOrWaitingRoom 4
3.3 Função receivePayment 6

4 Script semSharedMemWaiter.c **6**
4.1 Função waitForClientOrChef 7
4.2 Função informChef 7
4.3 Função takeFoodToTable 8

5 Script semSharedMemChef.c **8**
5.1 Função waitForOrder 9
5.2 Função processOrder 9

6 Testes **9**
6.1 Teste *Default* 9
6.2 Teste Mesas 11
6.3 Teste Grupos 11

1 Introdução

A resolução deste trabalho consistiu no desenvolvimento de determinadas funções dos scripts `semSharedMemGroup.c`, `semSharedMemReceptionist.c`, `semSharedMemWaiter.c` e `semSharedMemChef.c` com o objetivo de simular o funcionamento de um restaurante utilizando semáforos e memória partilhada, na linguagem `C`.
Estes scripts definem os *grupos de clientes*, o *rececionista*, o *empregado* e o *chef* do restaurante, respetivamente. Estando já parcialmente desenvolvidos, o nosso objetivo foi, recorrendo ao nosso conhecimento sobre a gestão de semáforos e de memória partilhada, implementar as funções inacabadas, funções essas que serão explicadas em detalhe nas seguintes secções.

2 Script semSharedMemGroup.c

O script `semSharedMemGroup.c` é responsável por simular o ciclo de vida do processo com *id* = `n` de um grupo. O grupo começa por ir ao restaurante, o que demora um tempo *random* definido em `startTime` no ficheiro `config.txt`. De seguida, dá *check in* na receção e espera que o `waiter` lhe atribua uma mesa. Após ter uma mesa, faz o *request* da sua comida e espera que o `waiter` o entregue ao `chef`, de seguida, espera que a sua comida seja entregue á sua mesa e, quando esta chega, come, o que também demora um certo tempo configurado e, por fim, dá *checkout* na receção, pagando e saindo do restaurante.

```
int main (int argc, char *argv[])
{
    ...
    /* simulation of the life cycle of the group n*/
    goToRestaurant(n);
    checkInAtReception(n);
    orderFood(n);
    waitFood(n);
    eat(n);
    checkOutAtReception(n);
    ...
    return EXIT_SUCCESS;
}
```

2.1 Função checkInAtReception

Esta função é responsável por tratar do *check in* do grupo `id` ao restaurante. Primeiramente, começa por esperar que o rececionista esteja disponível e, para este efeito, dá-se `semDown` ao semáforo `receptionistRequestPossible`. Após o rececionista estar disponível, o estado do grupo é mudado para `ATRECEPTION`, é feito o *request* de uma mesa (`TABLEREQ`) e o grupo avisa o rececionista do seu pedido dando `semUp` ao semáforo `receptionistReq`. Por fim, espera que o rececionista lhe atribua uma mesa com o *down* de `waitForTable[id]`.

```
static void checkInAtReception(int id)
{
    // wait for receptionist to be available
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    // the group asks for a table
    sh->fSt.st.groupStat[id] = ATRECEPTION; // group updates its state
    sh->fSt.receptionistRequest.reqType = TABLEREQ;
    sh->fSt.receptionistRequest.reqGroup = id;
    saveState(nFic,&sh->fSt);
    // signals receptionist of request
    if (semUp (semgid, sh->receptionistReq) == -1) {
        ...
    }
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // wait for receptionist to assign table
    if (semDown (semgid, sh->waitForTable[id]) == -1) {
        ...
    }
}
```

2.2 Função orderFood

Esta função é responsável por tratar de fazer o *request* da comida do grupo `id` ao `waiter`. Para isso, começa por esperar que o mesmo esteja disponível por meio de se dar `semDown` ao semáforo que o `waiter` usa para avisar que está operacional (`waiterRequestPossible`). Assim que seja possível, o estado do grupo é mudado para `FOOD_REQUEST`, é feito o pedido da comida (`FOODREQ`) e avisa-se o `waiter` que há um *request* por parte do grupo com o `semUp` do semáforo `waiterRequest`. Por fim, o grupo espera que o `waiter` entregue o seu pedido ao `chef`, ficando *down* em `requestReceived[sh->fSt.assignedTable[id]]`.
É usado como argumento de `requestReceived` o valor `sh->fSt.assignedTable[id]` pois, o número deste tipo de semáforo depende da mesa logo, para encontrar a mesa onde o grupo está, é preciso aceder ao argumento passado.

```
static void orderFood(int id)
{
    // wait for waiter to be available
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    sh->fSt.st.groupStat[id] = FOOD_REQUEST; // group updates its state
    // group requests food to waiter
    sh->fSt.waiterRequest.reqType = FOODREQ;
    sh->fSt.waiterRequest.reqGroup = id;
    saveState(nFic,&sh->fSt);
    // signals waiter of request
    if (semUp (semgid, sh->waiterRequest) == -1) {
        ...
    }
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // group waits for waiter to deliver request to chef
    if (semDown (semgid, sh->requestReceived[sh->fSt.assignedTable[id]]) == -1) {
        ...
    }
}
```

2.3 Função waitFood

Esta função é responsável por fazer com que o grupo espere que a sua comida chegue, após saber que o `chef` já recebeu o seu pedido, e, por mudar o seu estado após a comida ter chegado. No início, o estado do grupo é mudado para `WAIT_FOR_FOOD` e, depois, é dado *down* a `foodArrived[sh->fSt.assignedTable[id]]` para que o mesmo fique á espera da sua comida. Mal a esta chegue, o estado do grupo é novamente mudado, ficando no estado `EAT`.

```
static void waitFood(int id)
{
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    // group updates its state
    sh->fSt.st.groupStat[id] = WAIT_FOR_FOOD;
    saveState(nFic,&sh->fSt);
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // group waits for food to arrive
    if (semDown (semgid, sh->foodArrived[sh->fSt.assignedTable[id]]) == -1) {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    // group updates its state
    sh->fSt.st.groupStat[id] = EAT;
    saveState(nFic,&sh->fSt);
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
}
```

2.4 Função checkOutAtReception

Esta função é responsável pelo *checkout* do grupo. Primeiramente, o grupo espera que o rececionista esteja disponível para o atender, usando, novamente, o `semDown` do semáforo `receptionistRequestPossible`, de seguida, o seu estado é mudado para `CHECKOUT`, é feito o pedido de pagamento (`BILLREQ`), a mesa que o grupo acaba de sair é guardada na variável `currentTable` por questões de segurança, pois esse valor vai ser usado logo a seguir, e o grupo sinaliza o rececionista do seu *request*, fazendo `semUp` de `receptionistReq`. Por fim, usando o valor guardado anteriormente `currentTable`, o grupo espera que o rececionista trate do seu pagamento com o *down* do semáforo `tableDone[currentTable]` e o seu estado é mudado para `LEAVING`.

```
static void checkOutAtReception(int id)
{
    // group waits for receptionist to be available
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    // group updates its state
    sh->fSt.st.groupStat[id] = CHECKOUT;
    // group sends payment request to receptionist
    sh->fSt.receptionistRequest.reqType = BILLREQ;
    sh->fSt.receptionistRequest.reqGroup = id;
    int currentTable = sh->fSt.assignedTable[id]; // group saves table it was assigned to
    saveState(nFic,&sh->fSt);
    // signals receptionist of request
    if (semUp (semgid, sh->receptionistReq) == -1) {
        ...
    }
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // group waits for receptionist to acknowledge payment
    if (semDown (semgid, sh->tableDone[currentTable]) == -1) {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    // group updates its state
    sh->fSt.st.groupStat[id] = LEAVING;
    saveState(nFic,&sh->fSt);
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
}
```

3 Script semSharedMemReceptionist.c

O script `semSharedMemReceptionist.c` é responsável por simular o ciclo de vida do rececionista do restaurate. Para isto, é necessário o uso de uma memória interna `groupRecord` que vai servir como auxílio para as funções do script pois, esta memória interna tem como objetivo ajudar na verificação do estado de cada grupo em relação ás mesas disponíveis. Assim, o ciclo de vida do rececionista baseia-se em, primeiramente, esperar que algum grupo lhe faça um pedido, com o uso da função `waitForGroup`, e, caso seja um *request* de uma mesa (`TABLEREQ`), é usada a função `provideTableOrWaitingRoom` para decidir se deve atribuir uma mesa ao grupo que fez o pedido ou se o faz esperar que alguma mesa fique disponível, caso seja um *request* de pagamento (`BILLREQ`), é usada a função `receivePayment` para que o rececionista trate do pagamento do grupo.

```
int main(int argc, char *argv[])
{
    ...
    /* initialize internal receptionist memory */
    int g;
    for (g=0; g < sh->fSt.nGroups; g++) {
       groupRecord[g] = TOARRIVE;
    }
    /* 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);
                   break;
            case BILLREQ:
                   receivePayment(req.reqGroup);
                   break;
        }
        nReq++;
    }
    ...
    return EXIT_SUCCESS;
}
```

3.1 Função waitForGroup

Esta função é responsável por fazer com que o rececionista fique á espera que algum grupo lhe faça um *request* e por recebê-lo e retorná-lo. Para este efeito, primeiramente, o seu estado é mudado para `WAIT_FOR_REQUEST`, de seguida, fica a espera de um pedido com o *down* de `receptionistReq` e, assim que seja possível, recebe o pedido `receptionistRequest` e guarda-o na variável `ret` para, posteriormente, retorná-lo. Por fim, avisa a todos os grupos que está disponível para receber mais pedidos usando o `semUp` do semáforo `receptionistRequestPossible` e retorna o *request* lido anteriormente.

```
static request waitForGroup()
{
    request ret;
    if (semDown (semgid, sh->mutex) == -1)  {
        ...
    }
    sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;  // receptionist updates its state
    saveState(nFic, &sh->fSt);
    if (semUp (semgid, sh->mutex) == -1){
        ...
    }
    // receptionist waits for a request
    if (semDown (semgid, sh->receptionistReq) == -1)  {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1)  {
        ...
    }
    ret = sh->fSt.receptionistRequest; // receptionist reads request
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // receptionist signals that new requests are possible
    if (semUp (semgid, sh->receptionistRequestPossible) == -1) {
        ...
    }
    return ret;
}
```

3.2 Função provideTableOrWaitingRoom

Esta função tem como objetivo fazer com que o rececionista decida se deve atribuir diretamente uma mesa ao grupo `n` ou se o mesmo tem de esperar que alguma fique disponível. Assim, primeiro, o estado do rececionista é mudado para `ASSIGNTABLE` e é atribuído á variável `table` o *id* da mesa que vai ser atribuída ao grupo usando a função `decideTableOrWait`.

A função `decideTableOrWait` começa por inicializar uma variável para seguir quantas mesas estão disponíveis (`freeTables`) e uma lista `tables` com tamanho `NUMTABLES` onde todos os valores começam a zero pois, nesta lista, se a posição `x` tem o valor zero, então significa que a mesa `x` está disponível e, caso tenha o valor um, que a mesa está ocupada. Assim, por todos os grupos `nGroups`, é verificado se o grupo `i` está no estado `ATTABLE` na memória interna `groupRecord`, se estiver, então o número de `freeTables` é decrementado e, na lista `tables`, na posição relativa á mesa do grupo `i`, é assinalada como ocupada (1). De seguida, se o número de `freeTables` for 0, então a função retorna -1, que significa que o grupo `n` deve esperar, se não, então por todas as `NUMTABLES` mesas, a primeira mesa i disponível (`tables[i] == 0`) é retornada.

```
static int decideTableOrWait(int n){
    int freeTables = NUMTABLES;  // start by assuming all tables are free
    int* tables = (int*)calloc(NUMTABLES, sizeof(int));
    for (int i = 0; i < sh->fSt.nGroups; i++)
    {
        if (groupRecord[i] == ATTABLE){
            freeTables--;
            tables[sh->fSt.assignedTable[i]] = 1;  // mark table as occupied
        }
    }
    if (freeTables == 0){
        free(tables);
        return -1;
    }
    int i;
    for (i = 0; i < NUMTABLES; i++)
    {
        if (tables[i] == 0){
            break;
        }
    }
    free(tables);
    return i;
}
```

Continuando a análise da função `provideTableOrWaitingRoom`, após obter o valor da função referida anteriormente, caso este valor seja -1, ou seja, se o grupo tem de esperar, como já explicado, a variável `groupsWaiting` da memória partilhada é incrementada e o estado do grupo `n` na memória interna é mudado para `WAIT`, caso contrário, então é atribuído ao grupo a mesa `table` pelo uso da variável `assignedTable[n]` da memória partilhada, de seguida, o estado do grupo, na memória interna, é mudado para `ATTABLE`, caso hajam grupos á espera, a variável `groupsWaiting` é decrementada e, por fim, o rececionista informa o grupo que já tem uma mesa atribuída com o *up* do semáforo `waitForTable[n]`.

```
static void provideTableOrWaitingRoom(int n)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        ...
    }
    sh->fSt.st.receptionistStat = ASSIGNTABLE;  // receptionist updates its state
    int table = decideTableOrWait(n);
    if (table == -1){
        // if there are no free tables, the group must wait
        sh->fSt.groupsWaiting++;
        groupRecord[n] = WAIT;
    } else {
        // if there are free tables, the receptionist must choose the first one available
        sh->fSt.assignedTable[n] = table;
        groupRecord[n] = ATTABLE;
        if (sh->fSt.groupsWaiting > 0){
            sh->fSt.groupsWaiting--;
        }
        // receptionist informs group that it may proceed
        if (semUp (semgid, sh->waitForTable[n]) == -1) {
            ...
        }
    }
    saveState(nFic, &sh->fSt);
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
}
```

3.3 Função receivePayment

Esta função é responsável por tratar do pagamento do grupo `n`, de dar *free* á mesa que este acabou de sair e de atribuir a mesma a um novo grupo caso haja algum á espera de mesa. A função começa por mudar o estado do rececionista para `RECVPAY`, de guardar a mesa que acaba de ficar vaga na variável `vacantTable`, de desassociar a mesa ao grupo, atribuindo -1 á variável `assignedTable[n]` e de mudar o estado do mesmo na memória interna para `DONE`. De seguida, caso haja algum grupo á espera de mesa, guarda o valor do próximo grupo na variável `group` utilizando a função `decideNextGroup`.

```
static int decideNextGroup()
{
    int groupId = -1;
    for (int i = 0; i < sh->fSt.nGroups; i++)
    {
        if (groupRecord[i] == WAIT){
            groupId = i;
            break;
        }
    }
    return groupId;
}
```

A função `decideNextGroup` é usada para devolver o próximo grupo que deve receber uma mesa. Em princípio só é chamada quando há grupos á espera e existe uma mesa vaga, no entanto, a função começa por atribuir a `groupId` o valor de -1 como forma de precaução, caso a função tenha sido chamada pelas condições erradas, como forma de dizer que se deve ficar á espera que algum grupo chegue. De seguida, percorre todos os `nGroups` e, o primeiro que encontre que tenha como estado `WAIT` na memória interna, quebra o ciclo e retorna o seu *id*.

Assim que seja atribuído um *id* a `group`, caso este seja diferente de -1, o estado do rececionista muda para `ASSIGNTABLE` e, de seguida, é atribuída ao `group` a mesa que antes estava vaga (`vacantTable`), o número de `groupsWaiting` é decrementado, o estado de `group`, na memória interna, é atualizado para `ATTABLE` e o mesmo é avisado que já tem uma mesa atribuída pelo `semUp` de `waitForTable[group]`. Por fim, o rececionista sinaliza ao grupo `n` que está a pagar, de que o seu pagamento foi tratado com o *up* do semáforo `tableDone[vacantTable]`.

```
static void receivePayment(int n)
{
    if (semDown (semgid, sh->mutex) == -1)  {
        ...
    }
    sh->fSt.st.receptionistStat = RECVPAY;  // receptionist updates its state
    int vacantTable = sh->fSt.assignedTable[n];
    sh->fSt.assignedTable[n] = -1; // group leaves table
    groupRecord[n] = DONE;
    saveState(nFic, &sh->fSt);
    if (sh->fSt.groupsWaiting > 0){
        int group = decideNextGroup();
        if (group != -1){
            sh->fSt.st.receptionistStat = ASSIGNTABLE;  // receptionist updates its state
            sh->fSt.assignedTable[group] = vacantTable;
            sh->fSt.groupsWaiting--;
            groupRecord[group] = ATTABLE;
            saveState(nFic, &sh->fSt);
            if (semUp (semgid, sh->waitForTable[group]) == -1) {
                ...
            }
        }
    }
    if (semUp (semgid, sh->mutex) == -1)  {
        ...
    }
    // receptionist signals to vacant table's group that payment was received
    if (semUp (semgid, sh->tableDone[vacantTable]) == -1) {
        ...
    }
}
```

4 Script semSharedMemWaiter.c

O script `semSharedMemWaiter.c` tem como objetivo simular o ciclo de vida do empregado do restaurante, sendo este responsável pela comunicação entre os grupos de clientes e o chefe da cozinha. Para este efeito, o seu ciclo consiste em esperar pelo *request* de um grupo ou do chefe com o uso da função `waitForClientOrChef` e, de seguida, caso o pedido seja feito por um grupo (`FOODREQ`), o `waiter` transmite-o ao `chef` via a função `informChef`. Caso seja um pedido feito pelo chefe (`FOODREADY`), o empregado leva a comida ao grupo via a função `takeFoodToTable`.

```
int main(int argc, char *argv[])
{
    ...
    /* 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++;
    }
    ...
    return EXIT_SUCCESS;
}
```

4.1 Função waitForClientOrChef

Esta função é responsável por fazer com que o empregado fique á espera de que algum grupo ou o `chef` lhe faça um *request*, por recebê-lo e retorná-lo. Para este efeito, primeiramente, o seu estado é mudado para `WAIT_FOR_REQUEST`, de seguida, fica a espera de um pedido com o *down* de `waiterRequest` e, assim que seja possível, recebe o pedido `waiterRequest` e guarda-o na variável `req` para, posteriormente, retorná-lo. Por fim, avisa a todos os grupos e ao chefe de que está disponível para receber mais pedidos usando o `semUp` do semáforo `waiterRequestPossible` e retorna o *request* lido anteriormente.

```
static request waitForClientOrChef()
{
    request req;
    if (semDown (semgid, sh->mutex) == -1)  {
        ...
    }
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;  // waiter updates its state
    saveState(nFic, &sh->fSt);
    if (semUp (semgid, sh->mutex) == -1){
        ...
    }
    // waiter waits for a request
    if (semDown (semgid, sh->waiterRequest) == -1)  {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1)  {
        ...
    }
    req = sh->fSt.waiterRequest; // waiter reads request
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // waiter signals that new requests are possible
    if (semUp (semgid, sh->waiterRequestPossible) == -1) {
        ...
    }
    return req;
}
```

4.2 Função informChef

Esta função é responsável por fazer com que o `waiter` transmita o `request` do grupo `n` ao chefe de cozinha. Para isso, a função começa por mudar o estado do empregado para `INFORM_CHEF` e guarda o grupo que fez o pedido na variável `foodGroup` que vai ser posteriormente utilizada pelo chefe para conseguir identificar o grupo autor do `request`. De seguida, o `waiter` sinaliza ao grupo que o seu pedido foi tratado pelo *up* de `requestReceived[sh->fSt.assignedTable[n]]` e também avisa o `chef` de que tem um pedido usando o `semUp` do semáforo de `waitOrder`. Por fim, o empregado espera que o chefe dê *acknowledge* do pedido que ele lhe entregou com o *down* do semáforo `orderReceived`.

```
static void informChef(int n)
{
    if (semDown (semgid, sh->mutex) == -1){
        ...
    }
    sh->fSt.st.waiterStat = INFORM_CHEF;  // waiter updates its state
    sh->fSt.foodGroup = n;            // waiter takes food request to chef
    saveState(nFic, &sh->fSt);
    // waiter informs group that request is received
    if (semUp (semgid, sh->requestReceived[sh->fSt.assignedTable[n]]) == -1){
        ...
    }
    // waiter signals chef that he has a request
    if (semUp (semgid, sh->waitOrder) == -1)  {
        ...
    }
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // waiter waits for chef receiving request
    if (semDown (semgid, sh->orderReceived) == -1){
        ...
    }
}
```

4.3 Função takeFoodToTable

Esta função é responsável por fazer com que o `waiter` leve o prato, do chefe, para o grupo `n`. Para tal, primeiramente, o estado do empregado é mudado para `TAKE_TO_TABLE` e, por fim, o grupo é informado de que a sua comida chegou á sua mesa (`assignedTable[n]`) com o *up* de `foodArrived[sh->fSt.assignedTable[n]]`.

```
static void takeFoodToTable (int n)
{
    if (semDown (semgid, sh->mutex) == -1){
        ...
    }
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;  // waiter updates its state
    saveState(nFic, &sh->fSt);
    // waiter informs group that food is available
    if (semUp (semgid, sh->foodArrived[sh->fSt.assignedTable[n]]) == -1){
        ...
    }
    if (semUp (semgid, sh->mutex) == -1){
        ...
    }
}
```

5 Script semSharedMemChef.c

O script `semSharedMemChef.c` é responsável por simular o ciclo de vida do chef do restaurate, que é responsável por receber e processar os pedidos feitos pelos grupos. Para tal, é inicializada uma variável `nOrders = 0`, para contabilizar o nº de pedidos processados. Assim, enquanto `nOrders` for inferior ao nº de grupos total (enquanto não tiverem sido concluídos todos os pedidos), o chef vai executar as funções `waitForOrder`, através da qual o chef espera por um novo pedido, assinalando depois que o recebeu, e `processOrder`, através da qual, após um compasso de espera aleatório (equivalente ao tempo que o chef demora a preparar o pedido), vai sinalizar ao empregado que o pedido está pronto para ser entregue ao grupo correspondente.

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

       nOrders++;
    }
    ...
    return EXIT_SUCCESS;
}
```

5.1 Função waitForOrder

Esta função é responsável por fazer com que o chef fique á espera que o waiter lhe passe um *request* feito por um grupo e sinalizar que o recebeu e que está a trabalhar nele. Para tal, o chef fica à espera que o empregado lhe dê um pedido com o *down* de `waitOrder` e, mal seja possível, recebe o pedido do empregado, mudando posteriormente o seu estado para `COOK` e guardando o *id* do grupo que fez o pedido na variável interna `lastGroup`, salvando depois o seu estado. Finalmente, o chef sinaliza que recebeu o pedido com o *up* do `orderReceived`.

```
static void waitForOrder()
{
    // chef waits for a food request
    if (semDown (semgid, sh->waitOrder) == -1) {
        ...
    }
    if (semDown (semgid, sh->mutex) == -1) {
        ...
    }
    sh->fSt.st.chefStat = COOK;  // chef updates state
    lastGroup = sh->fSt.foodGroup;  // chef saves group that requested food
    saveState(nFic, &sh->fSt);
    if (semUp (semgid, sh->mutex) == -1) {
        ...
    }
    // chef acknowledges the order
    if (semUp (semgid, sh->orderReceived) == -1) {
        ...
    }
}
```

5.2 Função processOrder

Esta função é responsável por fazer com que o `chef` sofra um compasso de espera correspondente ao tempo (aleatório) do processamento de um pedido, esperando pelo empregado para levar o pedido através do *down* do `waiterRequestPossible`. Depois, é alterado o tipo de *request* do empregado para `FOODREADY` e o `reqGroup` correspondente é alterado para o *id* do grupo guardado em `lastGroup`. De seguida, o estado de `chef` passa a `WAIT_FOR_ORDER` e, por fim, o `chef` avisa o `waiter` de que tem um pedido feito com o *up* do semáforo `waiterRequest`.

```
static void processOrder()
{
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));
    // chef waits for waiter to be available
    if (semDown (semgid, sh->waiterRequestPossible) == -1){
        ...
    }
    if (semDown (semgid, sh->mutex) == -1){
        ...
    }
    sh->fSt.waiterRequest.reqType = FOODREADY;
    sh->fSt.waiterRequest.reqGroup = lastGroup;
    sh->fSt.st.chefStat = WAIT_FOR_ORDER;  // chef updates state
    saveState(nFic, &sh->fSt);
    if (semUp (semgid, sh->mutex) == -1){
        ...
    }
    // chef signals waiter that he has a request
    if (semUp (semgid, sh->waiterRequest) == -1){
        ...
    }
}
```

6 Testes

6.1 Teste *Default*

Este teste consiste no *run* do material em *src* com 2 mesas, 4 grupos e configuração *default*, tal como o exemplo pré-compilado fornecido pelo professor. Como podemos ver na seguinte parte do *output* do teste, ele mostra o funcionamento adequado de todos os tipos de processo desenvolvidos. Primeiramente, há um grupo que chega ao restaurante, neste caso o `G02`, que muda o seu estado para 2 (`ATRECEPTION`), espera que o rececionista chegue e é lhe atribuído a mesa 0, de seguida, em paralelo com o `G02` a fazer o seu `FOOD_REQUEST` (estado 3), o `G01` também chega

ao restaurante e é lhe atribuído a segunda mesa disponível. No decorrer dos pedidos é possível ver tudo a funcionar bem, um grupo espera pelo `waiter`, muda de estado (`FOOD_REQUEST`, estado 3), faz o seu pedido, o mesmo chega ao `chef` que, por sua vez, também muda de estado (`COOK`, estado 2), o grupo muda de estado outra vez (`WAIT_FOR_FOOD`, estado 4), o `chef` espera pelo `waiter` para lhe entregar o pedido, o `waiter` entrega ao grupo (`TAKE_TO_TABLE`, estado 2), o grupo come (`EAT`, estado 5) e, por fim, vai ao rececionista pagar (`CHECKOUT`, estado 6) e sai, acabando no estado 7 (`LEAVING`).
Enquanto isto, vão se acumulando grupos que chegam ao restaurante no campo `gWT` (*Waiting Room*), pois não têm mesa vaga, até um grupo pagar e sair do restaurante, deixando uma mesa vaga, que é logo atribuída ao grupo de menor *id* que está na *Waiting Room*. Neste caso foi o grupo da mesa 0 o primeiro a acabar.
Vale apena salientar que, o chefe consegue receber um pedido de uma mesa, enquanto tem o pedido de outra para entregar, sendo possível observar quando há 2 grupos, ambos com o estado 4 (`WAIT_FOR_FOOD`).

Run n.º 10

Restaurant - Description o

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

Figure 1: OutPut Teste Default

6.2 Teste Mesas

No decorrer dos testes, acabámos por descobrir que, apesar do problema ser destinado para um restaurante com apenas 2 mesas, se os tempos dos grupos forem todos diferentes, é possível adicionar mais mesas e o sistema funciona, mesmo sendo executado várias vezes.

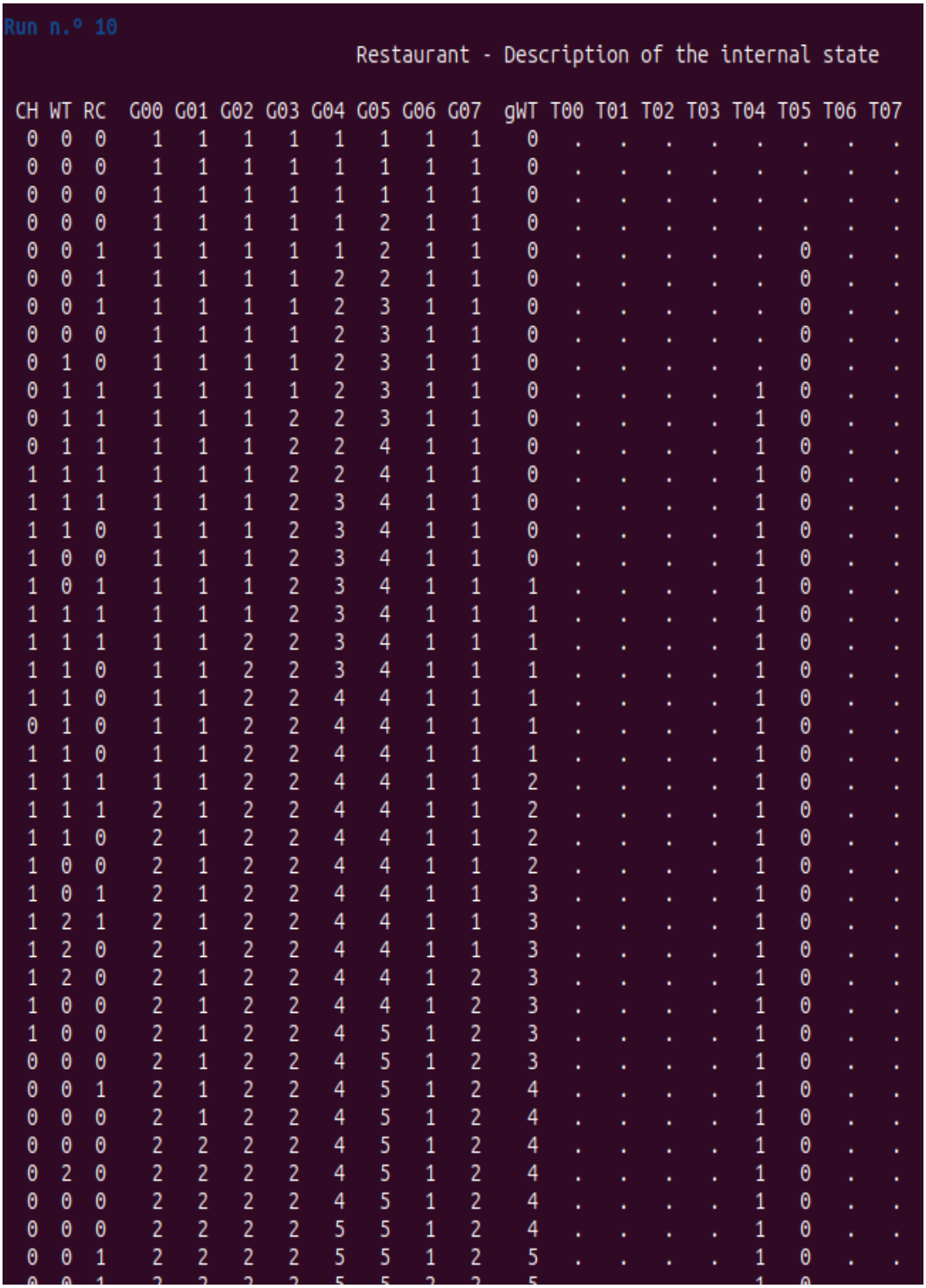
```
Run n.º 10
                                        Restaurant - Description o

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

Figure 2: OutPut Teste Mesas

6.3 Teste Grupos

Ainda no âmbito dos testes, o sistema também aceita a adição de mais grupos. No entanto, nós reparámos que, ao acrescentar um grupo, no âmbito do funcionamento normal do restaurante, ou seja, só com 2 mesas, os valores do grupo adicionado podem ser valores completamente aleatórios, no entanto, caso se deseje colocar mais mesas é preciso ter especial atenção pois, todos os grupos precisam de valores diferentes.



Run n.º 10

Restaurant - Description of the internal state

CH	WT	RC	G00	G01	G02	G03	G04	G05	G06	G07	gWT	T00	T01	T02	T03	T04	T05	T06	T07
0	0	0	1	1	1	1	1	1	1	1	0	.	.	.	.	.	.	.	.
0	0	0	1	1	1	1	1	1	1	1	0	.	.	.	.	.	.	.	.
0	0	0	1	1	1	1	1	1	1	1	0	.	.	.	.	.	.	.	.
0	0	0	1	1	1	1	1	2	1	1	0	.	.	.	.	.	.	.	.
0	0	1	1	1	1	1	1	2	1	1	0	.	.	.	.	.	0	.	.
0	0	1	1	1	1	1	2	2	1	1	0	.	.	.	.	.	0	.	.
0	0	1	1	1	1	1	2	3	1	1	0	.	.	.	.	.	0	.	.
0	0	0	1	1	1	1	2	3	1	1	0	.	.	.	.	.	0	.	.
0	1	0	1	1	1	1	2	3	1	1	0	.	.	.	.	.	0	.	.
0	1	1	1	1	1	1	2	3	1	1	0	.	.	.	.	1	0	.	.
0	1	1	1	1	1	2	2	3	1	1	0	.	.	.	.	1	0	.	.
0	1	1	1	1	1	2	2	4	1	1	0	.	.	.	.	1	0	.	.
1	1	1	1	1	1	2	2	4	1	1	0	.	.	.	.	1	0	.	.
1	1	1	1	1	1	2	3	4	1	1	0	.	.	.	.	1	0	.	.
1	1	0	1	1	1	2	3	4	1	1	0	.	.	.	.	1	0	.	.
1	0	0	1	1	1	2	3	4	1	1	0	.	.	.	.	1	0	.	.
1	0	1	1	1	1	2	3	4	1	1	1	.	.	.	.	1	0	.	.
1	1	1	1	1	1	2	3	4	1	1	1	.	.	.	.	1	0	.	.
1	1	1	1	1	2	2	3	4	1	1	1	.	.	.	.	1	0	.	.
1	1	0	1	1	2	2	3	4	1	1	1	.	.	.	.	1	0	.	.
1	1	0	1	1	2	2	4	4	1	1	1	.	.	.	.	1	0	.	.
0	1	0	1	1	2	2	4	4	1	1	1	.	.	.	.	1	0	.	.
1	1	0	1	1	2	2	4	4	1	1	1	.	.	.	.	1	0	.	.
1	1	1	1	1	2	2	4	4	1	1	2	.	.	.	.	1	0	.	.
1	1	1	2	1	2	2	4	4	1	1	2	.	.	.	.	1	0	.	.
1	1	0	2	1	2	2	4	4	1	1	2	.	.	.	.	1	0	.	.
1	0	0	2	1	2	2	4	4	1	1	2	.	.	.	.	1	0	.	.
1	0	1	2	1	2	2	4	4	1	1	3	.	.	.	.	1	0	.	.
1	2	1	2	1	2	2	4	4	1	1	3	.	.	.	.	1	0	.	.
1	2	0	2	1	2	2	4	4	1	1	3	.	.	.	.	1	0	.	.
1	2	0	2	1	2	2	4	4	1	2	3	.	.	.	.	1	0	.	.
1	0	0	2	1	2	2	4	4	1	2	3	.	.	.	.	1	0	.	.
1	0	0	2	1	2	2	4	5	1	2	3	.	.	.	.	1	0	.	.
0	0	0	2	1	2	2	4	5	1	2	3	.	.	.	.	1	0	.	.
0	0	1	2	1	2	2	4	5	1	2	4	.	.	.	.	1	0	.	.
0	0	0	2	1	2	2	4	5	1	2	4	.	.	.	.	1	0	.	.
0	0	0	2	2	2	2	4	5	1	2	4	.	.	.	.	1	0	.	.
0	2	0	2	2	2	2	4	5	1	2	4	.	.	.	.	1	0	.	.
0	0	0	2	2	2	2	4	5	1	2	4	.	.	.	.	1	0	.	.
0	0	0	2	2	2	2	5	5	1	2	4	.	.	.	.	1	0	.	.
0	0	1	2	2	2	2	5	5	1	2	5	.	.	.	.	1	0	.	.

Figure 3: OutPut Teste Grupos