Sistemas Operativos

Trabalho 2

Restaurante

107283 – Nuno Vieira

100480 – Airton Moreira

Introdução

Foi-nos proposto um problema de um restaurante onde cada processo deve ser sincronizado com a ajuda de semáforos de acesso a memória partilhada entre estes processos. No decorrer deste programa, destacam-se quatro processos concorrentes que disputam acesso à memória partilhada. São estes:

- Chef: Recebe os pedidos que os grupos fazem ao waiter, sendo responsável pela sua confeção, e chama o waiter assim que a comida esteja pronta para que este a leve para as mesas.
- Waiter: É responsável por receber os pedidos de cada grupo, levá-los ao chef para que este os prepare, e, por fim, encarrega-se de levar a comida às mesas.
- Group: Clientes do restaurante que chegam em grupos, pedem mesa, fazem o pedido da comida, esperam pela mesma, comem e pagam.
- Receptionist: Encarrega-se de esperar pela chegada dos grupos, procedendo à atribuição de uma mesa para o respetivo grupo ou, no caso de não haver mesas disponíveis, à colocação do grupo na sala de espera. Por fim, trata de receber os pagamentos.

Chef

O life cycle deste processo consiste em aguardar pelo pedido da parte do group e do waiter e confecionar o pedido.

Funções

waitForOrder

Nesta função, que serve para que o chef aguarde por um pedido da parte do waiter, inicialmente é feito um decremento do semáforo waitOrder, para que o chefe, assim que exista um pedido (um incremento neste semáforo), prossiga para a sua confeção.

De seguida, para conseguir acesso à região crítica de memória partilhada, o processo decrementa o semáforo mutex e atualiza o estado do chef para COOK, atualizando também a flag foodOrder para 0, sendo que o pedido acabou de ser recebido por ele. Por fim, é incrementado o semáforo mutex para sair da região crítica e é também sinalizado o semáforo orderReceived, para indicar ao waiter que deve esperar pela confeção do pedido.

processOrder

Depois de receber o pedido, é simulada uma quantidade de tempo necessária para a preparação do mesmo e, após este intervalo, o processo volta a esperar por acesso à memória partilhada para atualizar o estado do chef para REST, significando que o pedido já foi confecionado, e para modificar a flag waiterRequest, dando a entender ao waiter que o pedido é de FOODREADY e qual o grupo a que a comida “pertence”. Saíndo da região crítica, resta apenas sinalizar o waiter através de um decremento no semáforo waiterRequestPossible seguido de um incremento no semáforo waiterRequest.

```c
static void waitForOrder ()
{

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

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

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

    //TODO insert your code here
    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);
    }


}
```

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

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

    //TODO insert your code here
    sh->fSt.st.chefStat = REST;
    sh->fSt.waiterRequest.reqType = FOODREADY;
    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 (semDown (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the down 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);
    }
}
```

Waiter

O life cycle do empregado de mesa consiste em aguardar por pedidos, sejam eles do chef ou do group, e, para cada um destes, ou informar o chef de que há um pedido a ser preparado, ou levar a comida para a mesa do group em questão.

Funções

waitForClientOrChef

O waiter aguarda por um pedido fazendo semDown no semáforo waiterRequest e, assim que alguém sinalize esse semáforo, efetivamente fazendo um pedido, o waiter vai determinar se é da parte do chef ou do group. No fim, vai sinalizar os outros processos de que já é possível fazer-lhe novos requests.

informChef

No caso de o request ser da parte do group e de se tratar de um FOODREQ, o waiter vai, acedendo à memória partilhada através de um semáforo mutex, atualizar o seu estado para INFORM_CHEF, e vai depois sinalizar o chef de que este tem um pedido para confecionar, efetuando semUp no semáforo waitOrder, semáforo este sob o qual o chef estava "waiting" (tendo feito semDown). Deve também informar a mesa de que o request foi recebido e deve, finalmente, aguardar que o chef confirme o pedido.

takeFoodToTable

No caso de o request vir da parte do chef depois de ter preparado a comida, tratando-se de um FOODREADY, o waiter, acedendo à memória partilhada através de um semáforo mutex, vai atualizar o seu estado para TAKE_TO_TABLE e vai sinalizar o grupo de que a comida já chegou à mesa, efetuando um semUp no semáforo foodArrived. Por último, abandona a memória partilhada.

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

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

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

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

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

    // TODO insert your code here
    if (sh->fSt.waiterRequest.reqType == FOODREADY) {
        req.reqType = FOODREADY;
        req.reqGroup = sh->fSt.waiterRequest.reqGroup;
    }

    else if (sh->fSt.waiterRequest.reqType == FOODREQ){
        req.reqType = FOODREQ;
        req.reqGroup = sh->fSt.waiterRequest.reqGroup;
    }
    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 (semUp (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    return req;

}
```

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

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

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

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

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

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

```
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[n]) == -1)  {
        perror ("error on the up 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);
    }
}
```

Group

O life cycle do group é o mais linear de todos: o group desloca-se até ao restaurante, apresenta-se à receção, depois de ter mesa faz o seu pedido, aguarda que o o pedido chegue, come e dá checkout.

Funções

checkInAtReception

O group, depois de se deslocar até ao restaurante, apresenta-se na receção e, se ele estiver disponível, sinaliza o receptionist de que está a aguardar para ser atendido. Acedendo à memória partilhada, atualiza o seu estado para ATRECEPTION e é atualizado também o groupsWaiting para facilitar o trabalho do receptionist do seu lado. Por fim, o grupo aguarda com o semáforo waitForTable, esperando pela atribuição de mesa.

OrderFood

Assim que lhe é atribuído mesa, o group, acedendo à memória partilhada, volta a atualizar o groupsWaiting, reconhecendo que já não está a espera de mesa, atualiza a flag foodOrder para 1, atualiza a flag foodGroup para o seu respetivo id de grupo, atualiza o seu estado para FOOD_REQUEST, e, por fim, atualiza a flag waiterRequest passando-lhe o tipo de request (FOODREQ neste caso) e o id do seu grupo, responsável pelo request. De seguida, verificando se há disponibilidade por parte do waiter, sinaliza-o e aguarda que este receba o

pedido, incrementando o semáforo waiterRequest e decrementando os semáforos waiterRequestPossible e requestReceived.

waitFood

Enquanto aguarda pela comida, o group acede à memória partilhada e atualiza o seu estado para WAIT_FOR_FOOD. Saíndo da região crítica, é efetuado um semDown no semáforo foodArrived, através do qual o group aguarda que a comida chegue à mesa. Assim que o waiter sinalize este semáforo, o group começa a comer, acedendo novamente à região crítica e atualizando o estado para EAT.

checkOutAtReception

O group, ao certificar-se que o receptionist está disponível para receber o pagamento, vai sinalizá-lo através da operação semDown sobre o semáforo receptionistRequestPossible seguido de um semUp sobre receptionistReq. De seguida, acedendo à memória partilhada, o grupo atualiza o seu estado para CHECKOUT. Por fim, aguarda com o semáforo tableDone, que, assim que sinalizado pelo receptionist, reconhece que o pagamento foi efetuado com sucesso e o group pode então sair, sendo atualizado o seu estado para LEAVING.

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

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

    // TODO insert your code here
    sh->fSt.st.groupStat[id] = ATRECEPTION;
    sh->fSt.groupsWaiting = sh->fSt.groupsWaiting + 1;
    sh->fSt.receptionistRequest.reqType = TABLEREQ;
    sh->fSt.receptionistRequest.reqGroup = id;
    saveState(nFic, &sh->fSt);

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

    // 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);
    }
}
```

```c
static void orderFood (int id)
{
    // TODO insert your code here
    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.groupsWaiting = sh->fSt.groupsWaiting - 1;
    sh->fSt.st.groupStat[id] = FOOD_REQUEST;
    sh->fSt.waiterRequest.reqType = FOODREQ;
    sh->fSt.waiterRequest.reqGroup = id;
    saveState(nFic, &sh->fSt);

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

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

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

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

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

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

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

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

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

    // TODO insert your code here
    sh->fSt.st.groupStat[id] = CHECKOUT;
    sh->fSt.receptionistRequest.reqType = BILLREQ;
    sh->fSt.receptionistRequest.reqGroup = id;
    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
    if (semDown (semgid, sh->tableDone[id]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }


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

    // 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);
    }
```

Receptionist

O life cycle do receptionist consiste em aguardar por pedidos, sejam eles dos groups que chegam ao restaurante ou dos que pretendem pagar e sair, e, para cada um destes, ou decidir a atribuição de mesa/sala de espera, ou receber o pagamento.

Funções

waitForGroup

O recepcionist faz um semDown ao mutex para poder aceder à região crítica e atualiza o seu estado para WAIT_FOR_REQUEST, ficando disponível para receber pedidos através de um semDown a receptionistReq. Assim que

recebe um pedido, volta a aceder à região crítica e analisa o pedido. No caso de ser um TABLEREQ (pedido para atribuição de mesa), vai passar à atribuição de uma mesa para o grupo, e no caso de ser BILLREQ (pedido para pagar), vai processar o pagamento do grupo. No fim desta desta leitura de pedido, sinaliza o semáforo receptionistRequestPossible, permitindo que novos pedidos sejam efetuados.

ProvideTableOrWaitingRoom

Sendo o pedido um TABLEREQ, o receptionist, acedendo a memória partilhada, atualiza o seu estado para ASSIGNTABLE e, recorrendo a uma função auxiliar decideTableOrWait, que avalia o estado das mesas e conclui se existe alguma disponível ou não, ele vai atribuir ao grupo ou a mesa 1, ou a mesa 2, ou nenhuma, deixando o grupo à espera e não sinalizando o semáforo waitForTable.

receivePayment

Assim que um grupo acaba de comer e sinaliza o receptionist que quer pagar, com um request do type BILLREQ, o receptionist, acedendo à memória partilhada, vai primeiro guardar a table que foi vagada numa variável auxiliar e, de seguida, vai atualizar o seu estado para RECVPAY e remover a variável assignedTable do grupo que vai abandonar. Por fim, sinaliza o semáforo tableDone, transmitindo ao grupo que o pagamento foi recebido com sucesso e, se existirem grupos há espera por mesa, através da função auxiliar decideNextGroup, vai decidir quem deve ocupar aquela mesa com base na ordem de chegada.

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

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

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

    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 down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

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

    // TODO insert your code here
    if (sh->fSt.receptionistRequest.reqType == TABLEREQ) {
        ret.reqType = TABLEREQ;
        ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
    }

    else if (sh->fSt.receptionistRequest.reqType == BILLREQ){
        ret.reqType = BILLREQ;
        ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
    }
    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 (semUp (semgid, sh->receptionistRequestPossible) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    return ret;

}
```

```
static void provideTableOrWaitingRoom (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.receptionistStat = ASSIGNTABLE;
    if (decideTableOrWait(n) == 1) {
        sh->fSt.assignedTable[n] = 1;
        if (semUp (semgid, sh->waitForTable[n]) == -1) {                                        /* exit critical region */
            perror ("error on the up operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
    }
    else if (decideTableOrWait(n) == 2) {
        sh->fSt.assignedTable[n] = 2;
        if (semUp (semgid, sh->waitForTable[n]) == -1) {                                        /* exit critical region */
            perror ("error on the up operation for semaphore access (WT)");
            exit (EXIT_FAILURE);
        }
    }
    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);
    }

}
```

```c
static void receivePayment (int n)
{
    int table;
    int nextGroup;
    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
    if (sh->fSt.assignedTable[n] == 1) {
        table = 1;
    }
    else {
        table = 2;
    }
    sh->fSt.st.receptionistStat = RECVPAY;
    sh->fSt.assignedTable[n] = 0;
    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 (semUp (semgid, sh->tableDone[n]) == -1)  {
     perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    if (sh->groupsWaiting > 0) {
        nextGroup = decideNextGroup();
        sh->fSt.assignedTable[nextGroup] = table;
    }
}
```

Conclusão

Com este trabalho, conseguimos pôr em prática os conceitos e conhecimentos que abordamos nas aulas teóricas e nos guiões práticos relativos a processos, threads e sincronização através de semáforos.