Sistemas Operativos
Prof. Nuno Lau

Jantar de Amigos

Rafael Vilaça-107476
Bernardo Figueiredo-108073

2022/2023

Índice

Introdução

O presente trabalho prático, realizado no âmbito da disciplina “Sistemas Operativos”, tem como objetivo uma melhor compreensão dos mecanismos associados à execução e sincronização de processos e threads utilizando semáforos. O seu tema é a gestão de recursos de um jantar de um grupo de amigos. Foi-nos proposto completar o código fonte.

Existem 3 entidades:

Cliente – tem como funções esperar pelos amigos, pedir a comida, esperar a comida, ir para o jantar, comer, esperar e pagar a conta.

Waiter – tem como funções esperar pelo cliente ou pelo chef, informar o chef do pedido, levar a comida para a mesa e receber o pagamento.

Cozinheiro – tem como funções esperar o pedido e cozinhar o pedido.

Existem algumas regras:

1- O primeiro do grupo a chegar faz o pedido da comida, mas apenas quando todos chegarem.
2- O empregado de mesa deve levar o pedido ao cozinheiro e trazer o mesmo quando este estiver pronto.
3- O grupo só deve ir embora quando todos terminarem de comer.
4- O último a chegar ao restaurante é quem paga a conta.

Semáforos	Processos		Ação		Função	
	Up()	Down()	Up()	Down()	Up()	Down()
Mutex	Todos	Todos	Liberta controlo da zona de memória crítica	Toma controlo da zona de memória crítica	Todas	Todas
friendsArrived	Último cliente a chegar (Up() 20*)	Todos os Clientes	Liberta todos os clientes da espera	Espera que resto dos clientes cheguem á mesa	waitFriends()	waitFriends
waiterRequest	Cliente (Primeiro e ultimo) / Cozinheiro	Waiter	Informa o Waiter que existe um pedido	Espera por um pedido do cliente ou do chefe	Cliente: orderFood() e waitAndPay() / Cozinheiro : processOrde()	waitForClientOrChef()
requestReceived	Waiter	Cliente	Informa o cliente que o pedido foi realizado	O cliente aguarde que receba confirmação do seu pedido	informChef() e receivePayment()	orderFood() e waitAndPay()
waitOrder	Waiter	Cozinheiro	Informa o Cozinheiro que existe um pedido para ser realizado	O Cozinheiro aguarda por um pedido	informChef()	waitForOrder()
foodArrived	Waiter (Up() 20*)	Cliente	O Waiter entrega a comida aos 20 clientes	Os clientes aguardam pela comida	takeFoodToTable()	waitFood()
allFinished	Último cliente a acabar de comer (Up() 20*)	Cliente	Informa o resto da mesa que toda a gente já acabou de comer	Espera pelo resto dos clientes para acabarem de comer	waitAndPay()	waitAndPay()

Comportamento de cada entidade

Cliente

Função waitFriends

Função responsável por determinar o primeiro a chegar ao jantar e fazer todos esperarem pelo último.

Nesta função sempre que a mesma é chamada o número de clientes na mesa é incrementado, o estado do cliente é atualizado e se o número de clientes na mesa for igual a um esse cliente é selecionado como o primeiro.

Ao mesmo tempo é feita a verificação se o cliente atual é o último cliente, se isso se verificar é dado sem up ao semáforo friendsArrived vinte vezes, uma por cada cliente, para poderem todos continuar a sua execução.

Caso contrário é dado semDown ao semáforo friendsArrived com o objetivo de fazer o cliente atual esperar pelo último para continuar a sua execução.

```
static bool waitFriends(int id)
{
    bool first = false;

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

    /* insert your code here */

    sh->fSt.tableClients++;
    sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;
    first = sh->fSt.tableClients == 1;
    if (first) sh->fSt.tableFirst = id;

    if (sh->fSt.tableClients == TABLESIZE){
        for (int i = 0; i < TABLESIZE; i++)
        {
            semUp(semgid, sh->friendsArrived);
        }
        sh->fSt.tableLast = id;
    }

    saveState(nFic, &(sh->fSt));

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

    /* insert your code here */

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

    return first;
}
```

Figura 1 - Função waitFriends

Durante esta função são guardados o estado (com a função saveState), o tableFirst e o tableLast para uso posterior.

Função orderFood

Função responsável por pedir a comida para todos.

Esta função é executada apenas pelo primeiro a chegar á mesa.

A função apenas atualiza o valor do food request para 1, o estado do clientStat do primeiro cliente e guarda o mesmo usando a função saveState

Por fim é dado semup ao waiterRequest para o waiter tratar do pedido e semdown do requestReceived do client para o mesmo esperar a confirmação do pedido antes de continuar a sua execução.

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

    /* insert your code here */

    // request food to the waiter
    sh->fSt.foodRequest = 1;

    // The first client should update its state
    sh->fSt.st.clientStat[id] = FOOD_REQUEST;

    // The internal state should be saved.
    saveState(nFic, &(sh->fSt));

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

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

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

    return;
}
```

Figura 2 - Função orderFood

Função waitFood

Função responsável por fazer cada cliente esperar pela comida e de seguida quando possível comer.

Nesta função primeiramente é atualizado o estado de cada cliente para WAIT_FOR_FOOD é então salvo o estado do mesmo através da função saveState.

É logo de seguida dado semDown ao semáforo para o cliente não executar mais ações enquanto o waiter não der semUp, o mesmo só o fará no fim de ter trazido a comida para a mesa.

Estando o semáforo up é então realizada o resto da função, que apenas atualiza o estado do cliente para guardar o mesmo de seguida com a função saveState.

```
static void waitFood (int id)
{

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

    /* insert your code here */
    sh->fSt.st.clientStat[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);
    }

    /* insert your code here */
    if(semDown(semgid, sh->foodArrived) == -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);
    }

    /* insert your code here */
    sh->fSt.st.clientStat[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);
    }

    return;
}
```

Figura 3 - Função waitFood

Função waitAndPay

Função responsável por fazer cada cliente <esperar o último acabar de comer e também de o último a chegar para o jantar ser o que paga o jantar.

Nesta função é incrementado o número de clientes que terminou de comer cada vez que a mesma é chamada sendo de seguida atualizado e guardado o estado do cliente para WAIT_FOR_OTHERS, é dado também semDown do semáforo para impedir a continuação da execução.

Chegando o valor da variável tableFinishEat ao tamanho da mesa, ou seja, quando todos os clientes terminaram de comer, o último a acabar dá semup ao semáforo por cada cliente, assim continuando a execução do código em todos os clientes.

```
static void waitAndPay (int id)
{
    bool last=false;

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

    /* insert your code here */
    if (sh->fSt.tableLast == id)
    {
        last = true;
    }

    sh->fSt.tableFinishEat++;
    if (sh->fSt.tableFinishEat == TABLESIZE)
    {
        for (int i = 0; i < TABLESIZE; i++)
        {
            if (semUp (semgid, sh->allFinished) == -1)
            {
                perror ("error on the up operation for semaphore access (CT)");
                exit (EXIT_FAILURE);
            }
        }
    }
    sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
    saveState(nFic, &(sh->fSt));

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

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

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

Figura 4 - Função waitAndPay 1/2

```
    }

    /* insert your code here */
    sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;

    saveState(nFic, &(sh->fSt));

    sh->fSt.paymentRequest = 1;

    if(semUp (semgid, sh->waiterRequest) == -1){
        perror ("error on the up 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);
    }

    /* insert your code here */

    if(semDown (semgid, sh->requestReceived) == -1){
        perror ("error on the up 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);
}

/* insert your code here */
sh->fSt.st.clientStat[id] = FINISHED;
saveState(nFic, &(sh->fSt));

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

return;
}
```

Figura 5 - Função waitAndPay 2/2

Se o cliente atual for o último a chegar ao jantar é então atualizado o seu estado para WAIT_FOR_BILL e guardado, o valor de paymentRequest é atualizado para 1 e é dado semUp ao semáforo waiterRequest para o mesmo ser processado pelo waiter de seguida é dado semDown do semáforo requestReceived para esperar a confirmação do pedido.

Finalmente é atualizado para FINISHED o estado de cada cliente e guardado usando a função saveState.

Cozinheiro

Função waitForOrder

```
static void waitForOrder ()
{
    /* 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) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

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

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

    return;
}
```

Figura 6 - Função waitForOrder

Função responsável por esperar chegar um pedido para atualizar o estado do cozinheiro.

Nesta função é inicialmente dado semDown no semáforo waitOrder para o restante da função ser executado apenas quando o waiter informar um pedido ao cozinheiro, ou seja, der semUp ao semáforo, verificando-se essa situação muda-se o valor da foodOrder para 0 e atualiza-se o estado do cozinheiro para COOK, de seguida é então gravado o mesmo usando a função saveState.

Função processOrder

Função responsável por cozinhar o pedido.

Nesta função é dado um tempo aleatório para o tempo que demora a cozinhar o pedido, de seguida é atualizado o valor da variável foodReady para 1, assim o waiter leva o pedido á mesa.

Posteriormente, é atualizado o estado do cozinheiro para REST sendo de seguida guardado usando a função saveState.

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

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

    /* insert your code here */
    sh->fSt.foodReady = 1;
    sh->fSt.st.chefStat = REST;
    saveState(nFic, &(sh->fSt));

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

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

    return;
}
```

Figura 7 - Função processOrder

Por fim é dado semUP ao semáforo waiterRequest para o waiter estar preparado para receber um novo pedido

Waiter

Função waitForClientOrChef

```
static int waitForClientOrChef()
{
    int ret=0;
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.st.waiterStat = WAIT_FOR_ORDER;

    saveState(nFic, &(sh->fSt));

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

    /* 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);
    }


    if (sh->fSt.foodRequest == 1) {
        ret = FOODREQ;
        sh->fSt.foodRequest = 0;
    } else if (sh->fSt.foodReady == 1) {
        ret = FOODREADY;
        sh->fSt.foodReady = 0;
    } else if (sh->fSt.paymentRequest == 1) {
        ret = BILL;
        sh->fSt.paymentRequest = 0;
    }

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

    return ret;

}
```

Figura 8 - Função waitForClientOrChef

Função responsável por definir a ação atual do waiter.

Nesta função o estado do waiter enquanto o mesmo não é necessário permanece como WAIT_ FOR_ORDER de seguida é dado semDown ao semáforo waiterRequest que posteriormente é dado semUp no cliente ou no cozinheiro quando for necessário.

Verificando-se o uso do waiter é utilizada uma das variáveis de controlo, sendo as mesmas foodRequest, foodReady, paymentRequest e dependendo da que se encontra em uso no momento o valor da variável ret é atualizado, este pode assumir três valores diferentes FOODREQ, FOODREADY ou BILL.

Por fim logo após a variável ret ser atualizada com o seu novo valor é redefinida a variável de controlo utilizada para 0.

Função informChef

Função responsável por entregar o pedido do cliente para o cozinheiro, quando ret assume o valor FOODREQ.

Nesta função primeiramente é atualizado o estado do waiter para INFORM_CHEF e também o valor da variável foodOrder para um, para o cozinheiro começar a preparar o mesmo.

Finalmente é dado semUp do semáforo requestReceived para informar o cliente que o pedido foi realizado.

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

    /* insert your code here */
    sh->fSt.st.waiterStat = INFORM_CHEF;
    saveState(nFic, &(sh->fSt));
    sh->fSt.foodOrder = 1;

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

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

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

    return;
}
```

Figura 9 - Função informChef

Função takeFoodToTable

Nesta função, realizada, quando ret assume o valor FOODREADY, é atualizado o estado do waiter para TAKE_TO_TABLE e de seguida é dado semUp ao semáforo foodArrived por cada cliente, com o objetivo de voltarem a sua execução.

```
static void takeFoodToTable ()
{

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

    /* insert your code here */
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &(sh->fSt));
    for(int i = 0; i < TABLESIZE; i++){
        if (semUp (semgid, sh->foodArrived))
        {
            perror ("error on the up 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;
}
```

Figura 10 - Função takeFoodToTable

Funçao receivePayment

Nesta função, quando ret assume o valor BILL, é atualizado o estado do waiter para RECEIVE_PAYMENT e é dado semUp ao semáforo requestReceived para informar o cliente que o pedido foi realizado.

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

    /* insert your code here */
    sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
    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) == -1){
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
    return;
}
```

Figura 11 - Função receivePayment

Testes realizados para validar a Solução

Primeiramente programa foi executado 100 vezes e nunca foi verificado qualquer tipo de erro nos ficheiros de erro que o mesmo disponibiliza logo assumimos que o mesmo corre sem problemas

Seguidamente, verificamos se os resultados obtidos tinham lógica face ao objetivo do projeto, se de facto as entidades funcionam como é pretendido. Este teste foi realizado manualmente, verificando linha a linha se as atualizações e mudanças de estado fazem sentido, isto foi feito para vários casos individuais, sendo que aqui demonstramos apenas um.

Run n.º 5

Restaurant - Description of the internal state

CH	WT	C00	C01	C02	C03	C04	C05	C06	C07	C08	C09	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	ATT	FIE	1st	las
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	-1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1	0	13	-1
0	0	1	1	2	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	2	0	13	-1
0	0	1	1	2	1	1	1	1	1	1	2	1	1	1	2	1	1	1	1	1	1	3	0	13	-1
0	0	1	1	2	1	1	2	1	1	1	2	1	1	1	2	1	1	1	1	1	1	4	0	13	-1
0	0	1	1	2	1	1	2	1	1	1	2	1	1	2	2	1	1	1	1	1	1	5	0	13	-1
0	0	1	1	2	1	1	2	1	1	1	2	1	1	2	2	1	1	1	1	1	2	6	0	13	-1
0	0	1	2	2	1	1	2	1	1	1	2	1	1	2	2	1	1	1	1	1	2	7	0	13	-1
0	0	1	2	2	1	1	2	1	1	1	2	2	1	2	2	1	1	1	1	1	2	8	0	13	-1
0	0	1	2	2	1	1	2	2	1	1	2	2	1	2	2	1	1	1	1	1	2	9	0	13	-1
0	0	1	2	2	1	1	2	2	1	1	2	2	1	2	2	1	1	2	1	1	2	10	0	13	-1
0	0	1	2	2	1	1	2	2	1	2	2	2	1	2	2	1	1	2	1	1	2	11	0	13	-1
0	0	1	2	2	1	1	2	2	1	2	2	2	1	2	2	1	2	2	1	1	2	12	0	13	-1
0	0	1	2	2	1	2	2	2	1	2	2	2	1	2	2	1	2	2	1	1	2	13	0	13	-1
0	0	1	2	2	1	2	2	2	1	2	2	2	2	2	2	1	2	2	1	1	2	14	0	13	-1
0	0	1	2	2	1	2	2	2	1	2	2	2	2	2	2	1	2	2	1	2	2	15	0	13	-1
0	0	2	2	2	1	2	2	2	1	2	2	2	2	2	2	1	2	2	1	2	2	16	0	13	-1
0	0	2	2	2	1	2	2	2	2	2	2	2	2	2	2	1	2	2	1	2	2	17	0	13	-1
0	0	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	18	0	13	-1
0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	19	0	13	-1
0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	20	0	13	17
0	0	2	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	2	20	0	13	17
0	0	2	2	2	2	2	2	2	2	2	2	2	2	4	3	2	2	2	2	2	2	20	0	13	17
0	0	2	2	2	2	2	2	2	2	2	4	2	2	4	3	2	2	2	2	2	2	20	0	13	17
0	0	2	2	2	2	2	2	2	2	2	4	2	2	4	3	2	2	2	2	2	4	20	0	13	17
0	0	2	2	4	2	2	2	2	2	2	4	2	2	4	3	2	2	2	2	2	4	20	0	13	17
0	0	2	2	4	2	2	2	2	2	2	4	2	2	4	3	2	2	4	2	2	4	20	0	13	17
0	0	2	2	4	2	2	4	2	2	2	4	2	2	4	3	2	2	4	2	2	4	20	0	13	17
0	0	2	4	4	2	2	4	2	2	2	4	2	2	4	3	2	2	4	2	2	4	20	0	13	17
0	0	2	4	4	2	4	4	2	2	2	4	2	2	4	3	2	2	4	2	2	4	20	0	13	17
0	0	2	4	4	2	4	4	2	2	2	4	2	2	4	3	2	4	4	2	2	4	20	0	13	17
0	0	2	4	4	2	4	4	2	2	4	4	2	2	4	3	2	4	4	2	2	4	20	0	13	17
0	0	2	4	4	4	4	4	2	2	4	4	2	2	4	3	2	4	4	2	2	4	20	0	13	17
0	0	2	4	4	4	4	4	2	2	4	4	2	4	4	3	2	4	4	2	2	4	20	0	13	17
0	0	2	4	4	4	4	4	4	2	4	4	2	4	4	3	2	4	4	2	2	4	20	0	13	17
0	0	2	4	4	4	4	4	4	4	4	4	2	4	4	3	2	4	4	2	2	4	20	0	13	17
0	0	2	4	4	4	4	4	4	4	4	4	4	4	4	3	2	4	4	2	2	4	20	0	13	17
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	3	2	4	4	2	2	4	20	0	13	17
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	3	2	4	4	2	4	4	20	0	13	17
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	3	4	4	4	2	4	4	20	0	13	17
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	3	4	4	4	4	4	4	20	0	13	17
0	1	4	4	4	4	4	4	4	4	4	4	4	4	4	3	4	4	4	4	4	4	20	0	13	17

Figura 12 - Resultados para uma execução 1/3

Fig.11 Teste de funcionamento

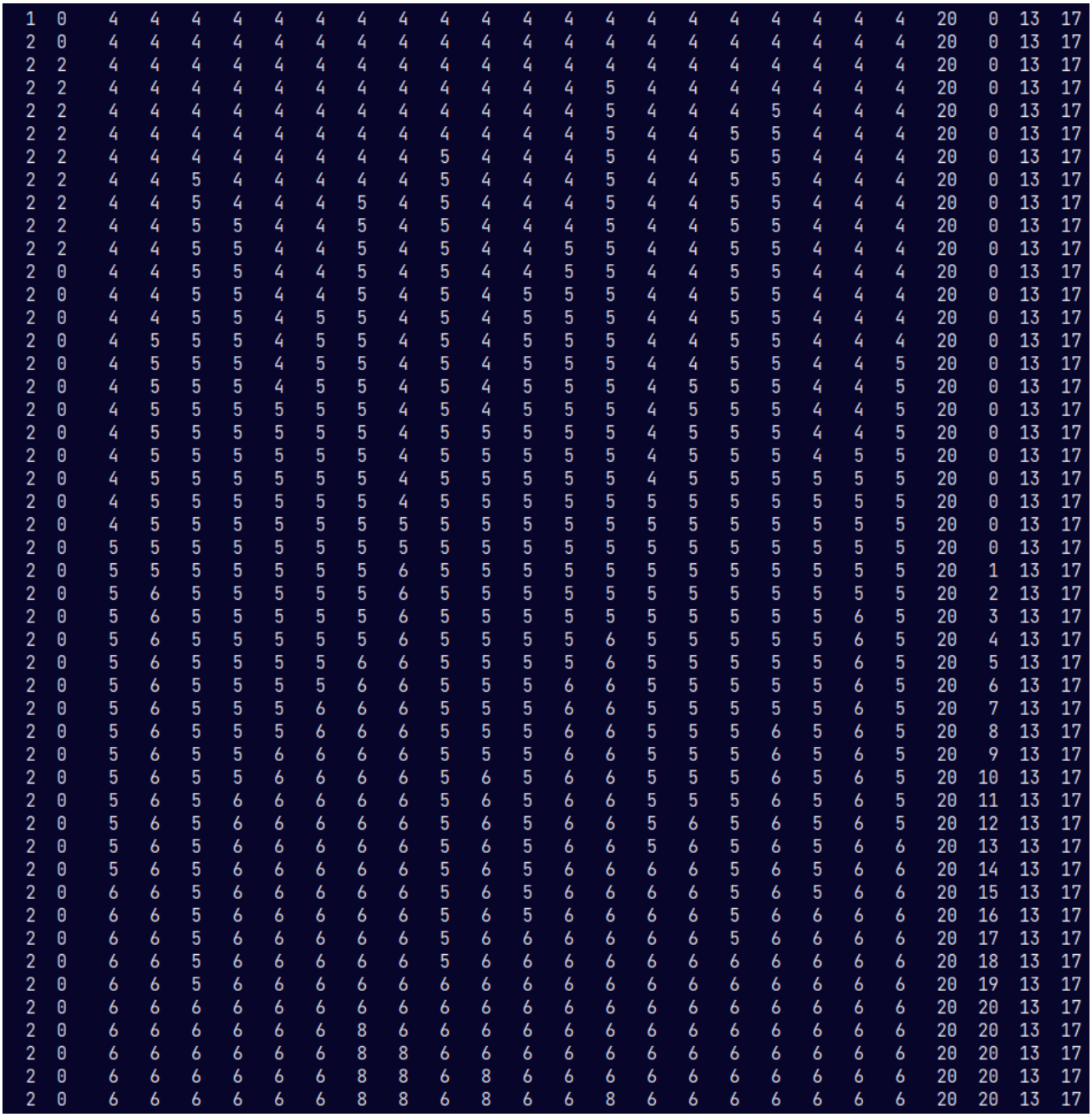

Figura 13 - Resultados para uma execução 2/3

```
2 0 4 5 5 5 5 5 5 4 5 5 5 5 5 4 5 5 5 4 4 5 20 0 13 17
2 0 4 5 5 5 5 5 5 4 5 5 5 5 5 4 5 5 5 4 5 5 20 0 13 17
2 0 4 5 5 5 5 5 5 4 5 5 5 5 5 4 5 5 5 5 5 5 20 0 13 17
2 0 4 5 5 5 5 5 5 4 5 5 5 5 5 5 5 5 5 5 5 5 20 0 13 17
2 0 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 20 0 13 17
2 0 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 20 0 13 17
2 0 5 5 5 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 5 5 20 1 13 17
2 0 5 6 5 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 5 5 20 2 13 17
2 0 5 6 5 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 6 5 20 3 13 17
2 0 5 6 5 5 5 5 5 6 5 5 5 5 6 5 5 5 5 5 6 5 20 4 13 17
2 0 5 6 5 5 5 5 6 6 5 5 5 5 6 5 5 5 5 5 6 5 20 5 13 17
2 0 5 6 5 5 5 5 6 6 5 5 5 6 6 5 5 5 5 5 6 5 20 6 13 17
2 0 5 6 5 5 5 6 6 6 5 5 5 6 6 5 5 5 5 5 6 5 20 7 13 17
2 0 5 6 5 5 5 6 6 6 5 5 5 6 6 5 5 5 6 5 6 5 20 8 13 17
2 0 5 6 5 5 6 6 6 6 5 5 5 6 6 5 5 5 6 5 6 5 20 9 13 17
2 0 5 6 5 5 6 6 6 6 5 6 5 6 6 5 5 5 6 5 6 5 20 10 13 17
2 0 5 6 5 6 6 6 6 6 5 6 5 6 6 5 5 5 6 5 6 5 20 11 13 17
2 0 5 6 5 6 6 6 6 6 5 6 5 6 6 5 6 5 6 5 6 5 20 12 13 17
2 0 5 6 5 6 6 6 6 6 5 6 5 6 6 5 6 5 6 5 6 6 20 13 13 17
2 0 5 6 5 6 6 6 6 6 5 6 5 6 6 6 6 5 6 5 6 6 20 14 13 17
2 0 6 6 5 6 6 6 6 6 5 6 5 6 6 6 6 5 6 5 6 6 20 15 13 17
2 0 6 6 5 6 6 6 6 6 5 6 5 6 6 6 6 5 6 6 6 6 20 16 13 17
2 0 6 6 5 6 6 6 6 6 5 6 6 6 6 6 6 5 6 6 6 6 20 17 13 17
2 0 6 6 5 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 20 18 13 17
2 0 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 20 19 13 17
2 0 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 13 17
2 0 6 6 6 6 6 6 8 6 6 6 6 6 6 6 6 6 6 6 6 6 20 20 13 17
2 0 6 6 6 6 6 6 8 8 6 6 6 6 6 6 6 6 6 6 6 6 20 20 13 17
2 0 6 6 6 6 6 6 8 8 6 8 6 6 6 6 6 6 6 6 6 6 20 20 13 17
2 0 6 6 6 6 6 6 8 8 6 8 6 6 8 6 6 6 6 6 6 6 20 20 13 17
2 0 6 6 6 6 8 6 8 8 6 8 6 6 8 6 6 6 6 6 6 6 20 20 13 17
2 0 6 8 6 6 8 6 8 8 6 8 6 6 8 6 6 6 6 6 6 6 20 20 13 17
2 0 6 8 6 8 8 6 8 8 6 8 6 6 8 6 6 6 6 6 6 6 20 20 13 17
2 0 6 8 6 8 8 6 8 8 6 8 6 8 8 6 6 6 6 6 6 6 20 20 13 17
2 0 6 8 6 8 8 6 8 8 6 8 6 8 8 6 6 6 8 6 6 6 20 20 13 17
2 0 6 8 6 8 8 8 8 8 6 8 6 8 8 6 6 6 8 6 6 6 20 20 13 17
2 0 6 8 6 8 8 8 8 8 6 8 6 8 8 6 6 6 8 6 6 8 20 20 13 17
2 0 6 8 6 8 8 8 8 8 6 8 6 8 8 6 8 6 8 6 6 8 20 20 13 17
2 0 6 8 6 8 8 8 8 8 6 8 6 8 8 6 8 6 8 6 8 8 20 20 13 17
2 0 8 8 6 8 8 8 8 8 6 8 6 8 8 6 8 6 8 6 8 8 20 20 13 17
2 0 8 8 6 8 8 8 8 8 6 8 6 8 8 6 8 6 8 7 8 8 20 20 13 17
2 0 8 8 6 8 8 8 8 8 8 8 6 8 8 6 8 6 8 7 8 8 20 20 13 17
2 0 8 8 6 8 8 8 8 8 8 8 8 8 8 6 8 6 8 7 8 8 20 20 13 17
2 0 8 8 6 8 8 8 8 8 8 8 8 8 8 8 8 6 8 7 8 8 20 20 13 17
2 0 8 8 6 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7 8 8 20 20 13 17
2 0 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7 8 8 20 20 13 17
2 3 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7 8 8 20 20 13 17
2 3 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 20 20 13 17
```

Figura 14 - Resultados para uma execução 3/3

Conclusão

A realização deste projeto permitiu-nos aprofundar os nossos conhecimentos sobre os mecanismos associados à execução e sincronização de processos e threads.

Este trabalho foi feito com base na análise do código fornecido, sendo que grande parte do conhecimento do mesmo foi adquirido nas aulas teóricas e práticas previamente.

De modo geral, o maior desafio foi perceber todas as funções necessárias para a implementação do projeto, visto que todo o código teria que ter uma estrutura e um funcionamento correto.

Finalmente, todos os testes realizados foram bem sucedidos.