Jantar de Amigos (Restaurant)

Sistemas Operativos

Trabalho 2

2022/23

Professor Nuno Lau

Professor Guilherme Campos

Gonçalo Sousa, NºMec: 108133 - 40%

Liliana Ribeiro, NºMec: 108713 - 60%

Índice

Introdução

Este relatório foi elaborado no contexto de expor o trabalho desenvolvido para o segundo trabalho prático da disciplina de Sistemas Operativos. Este trabalho prático consiste em utilizar o código fonte, disponibilizado pelos docentes da disciplina, para desenvolver código na linguagem C que simule processos que ocorrem dentro de um jantar num restaurante. Este possui três entidades intervenientes: o grupo de amigos, o waiter (Empregado de Mesa) e o chef (Cozinheiro).

Estas 3 entidades serão portanto processos independentes criados no início da execução do programa principal, sendo a sua sincronização realizada através de semáforos e acessos a memória partilhada. Estes processos deverão estar ativos apenas quando for necessário, devendo bloquear sempre que têm de esperar por algum evento.

No entanto, devem ser seguidas algumas regras de modo a sincronizar da melhor forma todas as entidades envolvidas, simulando com eficácia o modo de funcionamento de um restaurante no contexto real. As regras são as seguintes:

- O primeiro a chegar faz o pedido da comida para todo o grupo, porém, apenas quando todos chegarem ao restaurante;
- O empregado de mesa deve recolher o pedido e deve levá-lo até ao cozinheiro;
- O empregado de mesa deve levar a comida para mesa quando esta estiver confeccionada;
- O grupo apenas abandona a mesa quando todos terminarem de comer, sendo que o último membro a chegar ao restaurante fica encarregue de pagar a conta;

Assim sendo, o restaurante irá funcionar como o seguinte esquema propõe (**fig. 1**) e iremos partir para a implementação do funcionamento a partir desta estrutura.

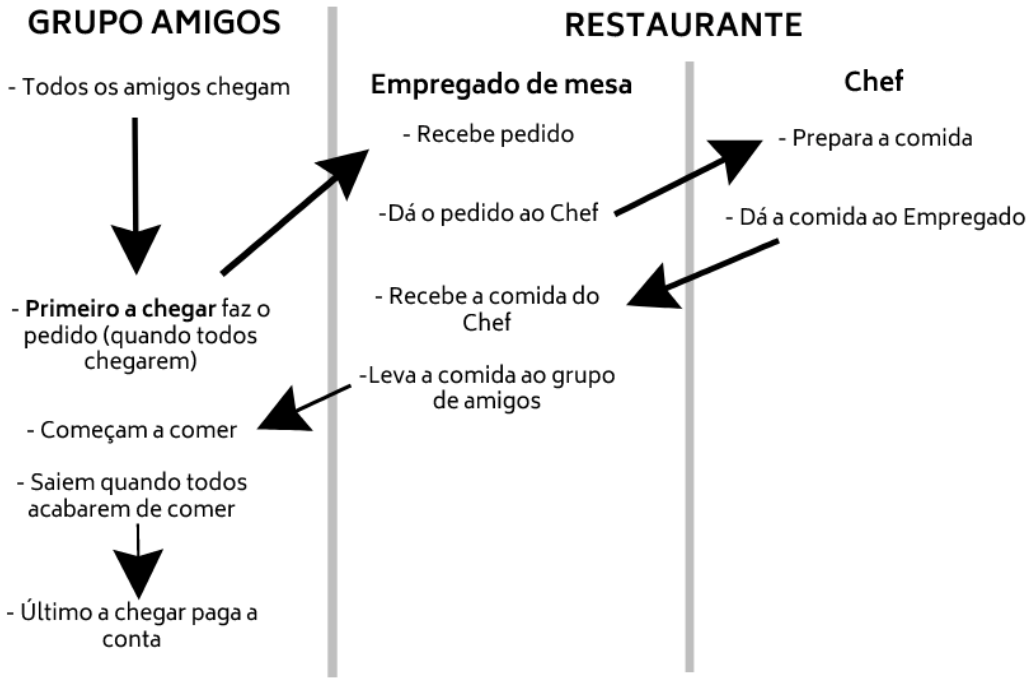

Fig. 1 - Funcionamento do restaurante.

Análise do Código fornecido

Os ficheiros que nos foram disponibilizados de modo a completar o problema dado continham três diretórios distintos:

- **semaphore_restaurant/doc** - Apresenta a documentação sobre a aplicação
- **semaphore_restaurant/run** - Apresenta todos os executáveis da aplicação
- **semaphore_restaurant/src** - Apresenta todo o código fonte disponibilizado

Ficheiro probConst.h

No ficheiro **probConst.h** encontramos definidas variáveis e respectivos estados (em conjunto com o número respetivo a cada estado) de cada entidade que a utilizar para o desenvolvimento do programa:

VARIÁVEIS GERAIS

```
/** \brief table capacity, equal to number of clients */
#define  TABLESIZE        20
/** \brief controls time taken to eat */
#define  MAXEAT       500000
/** \brief controls time taken to cook */
#define  MAXCOOK     3000000
```

ESTADOS POSSÍVEIS DO CLIENTE

```
/** \brief client initial state */
#define  INIT              1
/** \brief client is waiting for friends to arrive at table */
#define  WAIT_FOR_FRIENDS  2
/** \brief client is requesting food to waiter */
#define  FOOD_REQUEST      3
/** \brief client is waiting for food */
#define  WAIT_FOR_FOOD     4
/** \brief client is eating */
#define  EAT               5
/** \brief client is waiting for others to finish */
#define  WAIT_FOR_OTHERS   6
/** \brief client is waiting to complete payment */
#define  WAIT_FOR_BILL     7
/** \brief client finished meal */
#define  FINISHED          8
```

ESTADOS POSSÍVEIS DO CHEF

```
/** \brief chef waits for food order */
#define  WAIT_FOR_ORDER    0
/** \brief chef is cooking */
#define  COOK              1
/** \brief chef is resting */
#define  REST              2
```

ESTADOS POSSÍVEIS DO EMPREGADO

```
/** \brief waiter waits for food request */
#define  WAIT_FOR_REQUEST   0
/** \brief waiter takes food request to chef */
#define  INFORM_CHEF        1
/** \brief waiter takes food to table */
#define  TAKE_TO_TABLE      2
/** \brief waiter reiceives payment */
#define  RECEIVE_PAYMENT    3
```

Ficheiro ProbDataStruct.h

O ficheiro **ProbDataStruct.h** contém a definição de todas as estruturas de dados internas que especificam os estados das entidades intervenientes, sendo que este possui as estruturas STAT (variáveis para aceder aos estados das entidades) e FULLSTAT (definição de variáveis gerais e flags). É de salientar que as flags: foodRequest, foodOrder e foodReady vão ser usadas pelo waiter para saber o tipo de request que recebeu.

ESTADOS ENTIDADES

```
typedef struct {
   /** \brief waiter state */
   unsigned int waiterStat;
   /** \brief chef state */
   unsigned int chefStat;
```

```
    /** \brief client state array */
    unsigned int clientStat[TABLESIZE];

} STAT;
```

VARIÁVEIS GERAIS E FLAGS

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

    /** \brief number of clients at table */
    int tableClients;
    /** \brief number of clients that finished eating */
    int tableFinishEat;

    /** \brief flag of food request from client to waiter */
    int foodRequest;
    /** \brief flag of food order from waiter to chef */
    int foodOrder;
    /** \brief flag of food ready from chef to waiter */
    int foodReady;
    /** \brief flag of payment request from client to waiter */
    int paymentRequest;

    /** \brief id of first client to arrive */
    int tableLast;
    /** \brief id of last client to arrive */
    int tableFirst;

} FULL_STAT;
```

Ficheiro sharedDataSync.h

No ficheiro **sharedDataSync.h** estão estabelecidos os formatos de memória partilhada e a identificação dos diferentes semáforos, que irão ser responsáveis pela sincronização entre entidades intervenientes na aplicação.

```
typedef struct
        { /** \brief full state of the problem */
          FULL_STAT fSt;

          /* semaphores ids */
          /** \brief identification of critical region protection semaphore
- val = 1 */
          unsigned int mutex;
          /** \brief identification of semaphore used by clients to wait
for friends to arrive - val = 0 */
          unsigned int friendsArrived;
          /** \brief identification of semaphore used by client to wait for
waiter after a request - val = 0 */
          unsigned int requestReceived;
          /** \brief identification of semaphore used by clients to wait
for food - val = 0 */
          unsigned int foodArrived;
          /** \brief identification of semaphore used by clients to wait
for friends to finish eating - val = 0  */
          unsigned int allFinished;
          /** \brief identification of semaphore used by waiter to wait for
requests - val = 0  */
          unsigned int waiterRequest;
          /** \brief identification of semaphore used by chef to wait for
order - val = 0  */
          unsigned int waitOrder;

        } SHARED_DATA;

#define SEM_NU                (7)

#define MUTEX                   1
#define FRIENDSARRIVED          2
#define REQUESTRECEIVED         3
#define FOODARRIVED             4
#define ALLFINISHED             5
#define WAITERREQUEST           6
#define WAITORDER               7
```

Implementação da Solução

Para implementar a solução tivemos por base semáforos e memória partilhada. Foram implementados, no total, cerca de 7 semáforos, como mostrado anteriormente, sendo que um deles funciona em exclusão mútua (**mutex**), de modo a ser possível aceder à memória partilhada. Como possuímos 3 entidades com necessidade de partilhar informações entre si, a utilização de semáforos torna-se essencial, visto que é necessário garantir que não existem conflitos entre as mudanças de estado das mesmas.

Concretamente, podemos controlar quem acede à memória partilhada, garantindo que apenas uma entidade se encontra de cada vez na mesma.

Assim sendo, existem no diretório **semaphore_restaurant/src** 3 ficheiros incompletos, cada um definindo as operações que cada uma das 3 entidades realiza. Sendo estes:

- semSharedMemChef.c
- semSharedMemWaiter.c
- semSharedMemClient.c

Sempre que se quiser aceder a uma zona de memória partilhada, é necessário fazer primeiro a operação semDown() do semáforo **mutex** e no final a operação semUp(), imprimindo o programa uma mensagem de erro se algo correr mal.

Semáforo	Entidade		Ação		Função	
	Down()	Up()	Down()	Up()	Down()	Up()
Mutex	Todas	Todas	Entrar na região crítica	Sair da região crítica	Quase todas	Quase todas
FriendsArrived	Client	Client	Os clientes esperam todos os	O último cliente a chegar	waitFriends(int id)	waitFriends(int id)

			amigos chegarem	desbloqueia este sinal		
RequestReceived	Client	Waiter	O cliente espera o chef receber o pedido	O waiter avisa o cliente que o chef já recebeu o pedido	orderFood(int id)	informChef()
FoodArrived	Client	Waiter	O cliente espera receber a comida	O waiter entrega a comida aos clientes	waitFood(int id)	takeFoodToTable()
AllFinished	Client	Client	Os clientes esperam todos os amigos acabarem de comer	O último cliente a acabar de comer avisa os outros	waitAndPay(int id)	waitAndPay(int id)
WaiterRequest	Waiter	Client	O waiter espera ser requesitado	O cliente requesita o waiter para receber o pedido.	waitForClientOrChef()	orderFood(int id)
		Chef		O chef requesita o waiter para levar a comida à mesa.		processOrder()
		Client		O cliente requesita o waiter para receber o pagamento.		waitAndPay(int id)
WaitOrder	Chef	Waiter	O chef espera receber um pedido	O waiter dá o pedido do cliente ao chef	waitForOrder()	informChef()

Tabela 1 - Semáforos existentes e operações correspondentes

Chef - Ficheiro semSharedMemChef.c

- Life cycle - main()

O life cicle do chef caracteriza-se em **esperar a comida** e quando sinalizado pelo waiter que existe um pedido para fazer o **processo do pedido**.

```
waitForOrder();
processOrder();
```

- Função static void waitForOrder()

Nesta função o chef vai esperar que o waiter lhe leve um pedido através da operação **Down** do semáforo **waitOrder.**

Ao receber o pedido vai então entrar na **região crítica,** protegida pelo semáforo **mutex**.

Aqui dentro o chef irá tirar a ordem do waiter atualizando a flag **foodOrder** para o valor **0** e começa a cozinhar, atualizando o seu status, **chefStat**, para **COOK**.

No final guarda todas estas atualizações.

```
static void waitForOrder()
{
   /* insert your code here */
   // proceeds when receive an order
   if (semDown(semgid, sh->waitOrder) == -1)
   {
       perror("error on the down operation for waitOrder semaphore
access (CH)");
       exit(EXIT_FAILURE);
   }
   /* end code */

   if (semDown(semgid, sh->mutex) == -1)
   { /* enter critical region */
```

```
            perror("error on the up operation for semaphore access
(PT)");
            exit(EXIT_FAILURE);
        }

        /* insert your code here */
        // take order from waiter and star cooking
        sh->fSt.foodOrder = 0;
        sh->fSt.st.chefStat = COOK;
        saveState(nFic, &sh->fSt);
        /* end code */

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

- Função static void processOrder()

A função começa por chamar a função **usleep** com um número aleatório de microssegundos como argumento. Isto faz com que o programa pare por um período aleatório de tempo, simulando o tempo que leva para cozinhar um prato.

Em seguida, a função entra numa **região crítica** através do semáforo **mutex**.

Dentro da região crítica, a função atualiza a flag **foodReady** para **1** e o status do chef, **chefStat,** para **REST** e guarda todas as atualizações.

Depois da região crítica, a função dá um sinal ao empregado de mesa para levar a comida à mesa dando **Up** ao semáforo **waiterRequest**.

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

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

            /* insert your code here */
            // signal waiter food is ready, rest, wait for an order
            sh->fSt.foodReady = 1;
            sh->fSt.st.chefStat = REST;
            saveState(nFic, &sh->fSt);
            /* end code */

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

            /* insert your code here */
            // give signal to waiter to stop waiting the request --> food
ready
            if (semUp(semgid, sh->waiterRequest) == -1)
            {
                perror("error on the up operation for waiterRequest semaphore
access (CH)");
                exit(EXIT_FAILURE);
            }
            /* end code */
        }
```

Empregado - Ficheiro semSharedMemWaiter.c

- Life cycle - main()

Aqui iremos usar duas variáveis e um ciclo while para controlar os requests feitos ao waiter:

- **Req** para saber qual tipo de request é feito, assim permitindo que posteriormente consigamos aceder às funções correspondentes a cada tipo de request.
- **NReq** para que consigamos controlar a quantidade de requests já feitos e para que quando NReq seja igual a 3 o Waiter acabe o seu life cycle.

```
int req, nReq = 0;
   while (nReq < 3)
   {
        req = waitForClientOrChef();
        switch (req)
        {
        case FOODREQ:
            informChef();
            break;
        case FOODREADY:
            takeFoodToTable();
            break;
        case BILL:
            receivePayment();
            break;
        }
        nReq++;
   }
```

- Função static void waitForClientOrChef()

Nesta função o waiter irá esperar por um request vindo do Client ou do Chef.

Ele começa por inicializar a variável **ret** com o valor **0**, este valor irá ser atualizado, em seguida, para o tipo de request feito ao waiter.

Dentro da **região crítica**, o estado do empregado, **waiterStat,** é atualizado para **WAIT_FOR_REQUEST** e o estado é salvo.

A seguir, o waiter espera por um request usando o semáforo **waiterRequest** no estado **Down**.

Quando o waiter recebe um request, a função entra novamente numa **região crítica** para ler qual o tipo de request foi feito ao waiter. Iremos obter

estes valores através das flags: **foodRequest**, **foodOrder**, **foodReady** e uma condição if que irá confirmar qual destas flags tem valor 1:

- **foodRequest:** Se o request for para o empregado levar comida ao cliente, o valor de **ret** será definido como **FOODREQ**.
- **foodOrder:** Se a requisição for para o empregado levar a conta ao cliente, **ret** será definida como **BILL**.
- **foodReady:** Se a requisição for para avisar que a comida está pronta, **ret** será definida como **FOODREADY**.

Após dar **Up** ao semáforo **mutex** a função dá return ao valor de ret.

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

    /* insert your code here */
    // waiter updates state
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
    saveState(nFic, &sh->fSt);
    /* end code */

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

    /* insert your code here */
    // waits for request
    if (semDown(semgid, sh->waiterRequest) == -1)
    {
        perror("error on the down operation for waiterRequest
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);
    }

    /* insert your code here */
    // read request
    // save the type of request in ret from falgs (see each falg if
one was value 1 than we save the corresponding value in ret and put
value 0 to the flag)
    if (sh->fSt.foodRequest)
    {
        ret = FOODREQ;
    }
    if (sh->fSt.foodReady)
    {
        ret = FOODREADY;
    }
    if (sh->fSt.paymentRequest)
    {
        ret = BILL;
    }
    /* end code */

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

    return ret;
}
```

- Função static void informChef()

Nesta função o waiter informa o chef do pedido feito pelo cliente.

A função começa por entrar na **região crítica**, aqui será atualizado o estado do empregado, **waiterStat,** para **INFORM_CHEF**, bem como o valor das flags: **foodRequest** para **0** e **foodOrder** para **1** e guarda-se todas as atualizações.

Depois disso, sai da região crítica e informa o cozinheiro sobre o pedido de comida, dando **Up** ao semáforo **waitOrder**. Também avisa os clientes de que o pedido de comida foi recebido, dando **Up** ao semáforo **requestReceived**.

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

   /* insert your code here */
   // save status, signs that have a foodOrder
   sh->fSt.st.waiterStat = INFORM_CHEF;
   sh->fSt.foodRequest = 0;
   sh->fSt.foodOrder = 1;
   saveState(nFic, &sh->fSt);
   /*end code*/

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

   /* insert your code here */
   // take food request to chef
   if (semUp(semgid, sh->waitOrder) == -1)
   {
       perror("error on the up operation for waitOrder semaphore access
(WT)");
```

```
        exit(EXIT_FAILURE);
    }
    // let clients know food request was received
    if (semUp(semgid, sh->requestReceived) == -1)
    {
        perror("error on the up operation for requestReceived semaphore
access (WT)");
        exit(EXIT_FAILURE);
    }
    /* end code */
}
```

- Função static void takeFoodToTable()

A função começa por entrar na **região crítica**, onde atualiza o estado do empregado, **waiterStat**, para **TAKE_TO_TABLE** e coloca a flag **foodReady** a **0**.
Depois disso, o código sai da **região crítica**, onde o waiter vai levar a comida para a mesa, deixando assim os clientes começar a comer. Isso é feito com um loop que itera sobre os valores de i = 0 a TABLESIZE, dando **Up** ao semáforo **foodArrived** de todos os clientes.

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

        /* insert your code here */
        // update state
        sh->fSt.st.waiterStat = TAKE_TO_TABLE;
        sh->fSt.foodReady = 0;
        saveState(nFic, &sh->fSt);
        /* end code */

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

```
            perror("error on the down operation for semaphore access
    (WT)");
            exit(EXIT_FAILURE);
        }

        /* students code */
        // take food to table allow meal to start
        for (int i = 0; i < TABLESIZE; i++)
            if (semUp(semgid, sh->foodArrived) == -1)
            {
                perror("error on the up operation for foodArrived
    semaphore access (WT)");
                exit(EXIT_FAILURE);
            }
        /* end code */
    }
```

- Função static void receivePayment()

A função começa por entrar na **região crítica**, com o semáforo mutex. Dentro da região crítica, é atualizado o estado do empregado, **chefStat**, para **RECEIVE_PAYMENT** e atualiza-se o estado do último cliente a chegar (o cliente que faz o pagamento) para **FINISHED**. Também é colocada a flag **paymentRequest** a **0**. No final, as atualizações são guardadas e sai-se da **região crítica**.

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

    /* insert your code here */
    // update status and receive payment
    sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
    sh->fSt.st.clientStat[sh->fSt.tableLast] = FINISHED;
```

```
    sh->fSt.paymentRequest = 0;
    saveState(nFic, &sh->fSt);
    /* end code */

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

}
```

Cliente - Ficheiro semSharedMemClient.c

- Life cycle - main()

O cliente começa por realizar a viagem (**travel**). Em seguida, espera pelos amigos chegarem (**waitFriends**). Posteriormente, o primeiro cliente a chegar irá pedir a comida (**orderFood**). Depois todos os clientes vão esperar a comida chegar (**waitFood**), comer (**eat**) e por último esperar que todos acabem de comer (**waitAndPay**). Apenas o último cliente irá fazer o pagamento (**waitAndPay**).

```
travel(n);
bool first = waitFriends(n);
if (first)
    orderFood(n);
waitFood(n);
eat(n);
waitAndPay(n);
```

- Função static bool waitFriends(int id)

A função começa por fazer **down** no semáforo **mutex** para entrar na **região crítica**. Dentro da região crítica, o estado do cliente, **clientStat**, é atualizado para **WAIT_FOR_FRIENDS** e o número de clientes na mesa (**tableClients**) é incrementado. Se este for o primeiro cliente a sentar-se à mesa, a variável **first** é definida como **True** e o **ID** deste cliente é salvo na

variável **tableFirst**. Se for o último cliente a sentar-se à mesa, o **ID** deste cliente é salvo na variável **tableLast**. Por fim, o estado atual é salvo.

Depois de sair da região crítica, os clientes (exceto o último a chegar) irão esperar todos os amigos chegarem recorrendo à operação **Down** do semáforo **friendsArrived**. O último cliente a chegar irá avisar que os outros clientes que já chegaram todos, dando **Up** ao semáforo **friendsArrived**.

Por fim, a função retorna o valor de first, que indica se este cliente foi o primeiro a sentar-se à mesa.

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

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

   /* insert your code here */
   // update state, add client to table
   sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;
   sh->fSt.tableClients += 1;
   // first and last save id (first return true)
   if (sh->fSt.tableClients == 1)
   {
       sh->fSt.tableFirst = id;
       first = true;
   }
   if (sh->fSt.tableClients == TABLESIZE)
   {
       sh->fSt.tableLast = id;
   }
   saveState(nFic, &sh->fSt);
   /* end code */

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

```
            perror("error on the up operation for semaphore access
(CT)");
            exit(EXIT_FAILURE);
        }

        /* insert your code here */
        // wait for friends (last let others know)
        if (id == sh->fSt.tableLast)
        {
            for (int i = 1; i < TABLESIZE; i++)
                if (semUp(semgid, sh->friendsArrived) == -1)
                {
                    perror("error on the up operation for friendsArrived
semaphore access (GR)");
                    exit(EXIT_FAILURE);
                }
        }
        else
        {
            if (semDown(semgid, sh->friendsArrived) == -1)
            {
                perror("error on the down operation for friendsArrived
semaphore access (GR)");
                exit(EXIT_FAILURE);
            }
        }
        /* end code */

        return first;
}
```

- Função static void orderFood(int id)

Esta é função é usada pelo último cliente para fazer a solicitação de comida ao empregado. A função começa por fazer **down** no semáforo **mutex** para entrar na **região crítica.** Dentro da região crítica, o estado do cliente, **clientStat** é atualizado para **FOOD_REQUEST** e a flag **foodRequest** é definida como 1. O estado atual é então salvo. Depois de sair da região crítica, a função faz **Up** no semáforo **waiterRequest** para notificar o

empregado da solicitação e espera pelo sinal de que a solicitação foi recebida pelo empregado, fazendo **down** no semáforo **requestReceived**.

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

         /* insert your code here */
         // update status and request food to waiter
         sh->fSt.st.clientStat[id] = FOOD_REQUEST;
         sh->fSt.foodRequest = 1;
         saveState(nFic, &sh->fSt);
         /*end code*/

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

         /* insert your code here */
         // request food to waiter and wait waiter receive request
         if (semUp(semgid, sh->waiterRequest) == -1)
         {
             perror("error on the up operation for waiterRequest semaphore
access (WT)");
             exit(EXIT_FAILURE);
         }

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

```
}
```

- Função static void waitFood(int id)

Esta é uma função que é chamada pelos clientes quando estão a esperar pela comida que solicitaram. A função começa por fazer **down** no semáforo **mutex** para entrar na **região crítica**. Dentro da região crítica, o estado do cliente, **ClientStat**, é atualizado para **WAIT_FOR_FOOD** e o estado atual é salvo. Depois de sair da região crítica, os clientes esperam pelo sinal de que a comida chegou fazendo **down** no semáforo **foodArrived.**

Quando a comida chega, a função entra novamente na região crítica para atualizar o estado do cliente **ClientStat** para **EAT** e salvar o estado atual.

```
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);
    }

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

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

    /* insert your code here */
    // wait food arrive
    if (semDown(semgid, sh->foodArrived) == -1)
```

```
        { /* enter critical region */
            perror("error on the down operation for foodArived semaphore
access (CT)");
            exit(EXIT_FAILURE);
        }
        /* end code */

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

        /* insert your code here */
        // start eating (update state)
        sh->fSt.st.clientStat[id] = EAT;
        saveState(nFic, &sh->fSt);
        /* end code */

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

- Função static void waitAndPay(int id)

Este código representa a função que o cliente chama após ter terminado de comer. A função começa por entrar na **região crítica** pelo semáforo **mutex** e atualiza o estado do cliente, **ClientStat,** para **WAIT_FOR_OTHERS** e aumenta o número de clientes que terminaram de comer, **tableFinishedEat**. Em seguida, a função verifica se o cliente é o último a terminar de comer. Se for o último, atualiza o valor de **last** para 1. Ao sair da região crítica, é usado uma condição if para saber se o cliente é o último. Se não for, o cliente fica à espera que todos acabem de comer utilizando a operação **Down** do semáforo **allFinished**. Se for o último cliente, este avisa os

outros que todos já terminaram de comer, fazendo a operação **Up** no semáforo **allFInished** para todos os clientes. Em seguida, o último cliente a acabar de comer atualiza o estado do último cliente a chegar para **WAIT_FOR_BILL** e colocando a flag **paymentRequest** com valor **1**. A função sai da região crítica e avisa o waiter que tem um request a dar **Up** ao semáforo **waiterRequest**. Em seguida, para todos os clientes dentro do semáforo **mutex,** a função atualiza o estado de todos os clientes para **FINISHED**.

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

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

         /* insert your code here */
         // update status, num persons that finished eating
         sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
         sh->fSt.tableFinishEat += 1;
         last = (TABLESIZE == sh->fSt.tableFinishEat);
         saveState(nFic, &sh->fSt);
         /* end code */

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

         /* insert your code here */
         // wait everyone eating (if is the last one let other know)
         if (last)
         {
             for (int i = 1; i < TABLESIZE; i++)
```

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

        /* end code */

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

            /* insert your code here */
            // update status and flag payment
            sh->fSt.st.clientStat[sh->fSt.tableLast] = WAIT_FOR_BILL;
            sh->fSt.paymentRequest = 1;
            saveState(nFic, &sh->fSt);
            /* end code */

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

            /* insert your code here */
```

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

            /* end code */
        }

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

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

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

Execução do programa

Para testar as funções desenvolvidas, é necessário compilar cada entidade individualmente e testá-las com os demais arquivos binários fornecidos. Isso pode ser feito usando as regras já existentes no arquivo Makefile no diretório **semaphore_restaurant/src**.

No final, após termos todas as entidades completas, era necessário fazer o comando “$make all” no terminal. Para executar o programa, é necessário ir até à pasta semaphore_restaurant/run através da linha de comandos do terminal e fazer o comando “$./probSemSharedMemRestaurant”. No entanto, poderíamos também correr o programa 1000 vezes seguidas utilizando o script em BASH ./run.sh.

O principal objetivo com estes testes é de verificar se ocorre ou não uma situação de deadlock, bem como verificar se os estados das diferentes entidades estão semelhantes aos do output do programa quando se faz o comando “$make all_bin”. Todos os testes efetuados foram gravados em ficheiros com a extensão “.txt” através do comando “$./run.sh > ../tests/test_entidade.txt”, disponíveis na pasta semaphore_restaurant/tests.

De seguida, segue-se um exemplo ao correr em “$make all”, para 1 restaurante.

```
goncalo@goncalo-VivoBook-ASUS:~/SO/ProjetoSO2/semaphore_restaurant/run$ ./run 1

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

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

Conclusão

Com este trabalho conseguimos fortalecer os seus conhecimentos em diversos tópicos como por exemplo a compreensão dos mecanismos associados à execução e sincronização de processos e threads, utilização de semáforos e memória partilhada.

Referências

Para a realização deste trabalho recorremos ao material disponibilizado na disciplina, aos guiões práticos e slides teóricos. Face a dificuldades em algum assunto em específico recorremos também a sites web, como por exemplo o stackoverflow.com