# JANTAR DE AMIGOS (RESTAURANT)

## SISTEMAS OPERATIVOS


**Trabalho realizado por:**

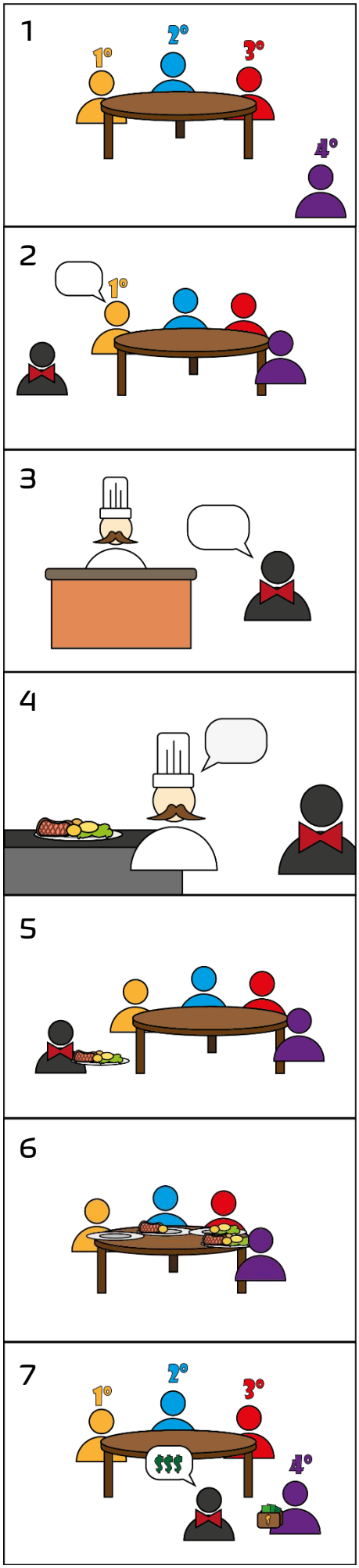
→ Pedro Ramos
> NºMec: 107348

→ Daniel Madureira
> NºMec: 107603


universidade de aveiro

deti universidade de aveiro

# ÍNDICE




# INTRODUÇÃO

## PROPOSIÇÃO DO PROBLEMA

O propósito deste trabalho prático é criar um programa em C que simule um jantar entre amigos. Neste cenário irão interagir:

→ Os vários amigos (*clients*);
→ Um empregado (*waiter*);
→ Um cozinheiro (*chef*);

O jantar pode ser dividido em partes segundo os momentos mais importantes. Assim:

→ Os clientes chegam à mesa no seu próprio tempo, sem ordem pré-definida;
→ Ao chegarem, o primeiro *client* chama o *waiter* e faz o pedido da mesa toda;
→ O *waiter* transmite o pedido ao *chef*;
→ O *chef* prepara o pedido e chama o *waiter* para entregar o pedido;
→ O *waiter* leva o pedido à mesa;
→ Os *clients* comem a refeição, ao seu ritmo, e esperam que todos acabem de comer;
→ O último *client* a chegar à mesa chama o *waiter* e paga a conta.

# METODOLOGIA DO CÓDIGO

Para simularmos, com sucesso, este problema em C precisaremos de três fluxos de código, um para cada interveniente (*client, waiter, chef* ).

Estes fluxos serão simulados em processos diferentes, sendo o objetivo final deste trabalho sincronizar e permitir a comunicação entre estes processos usando semáforos e memória partilhada.

Em suma, o nosso código terá:

- → *n* processos do tipo *client* (para este trabalho vamos considerar *n*=20);
- → Um processo do tipo *waiter*;
- → Um processo do tipo *chef*;
- → Um grupo de variáveis partilhadas contendo:
  - > Os estados de cada processo;
  - > Os semáforos utilizados;
  - > Vários contadores;
  - > Outros números relevantes à sincronização dos processos.

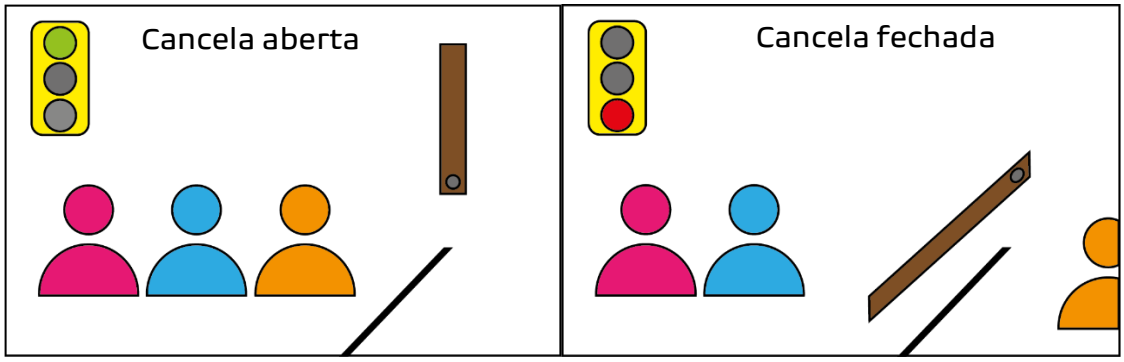
# ESTRUTURAS DE DADOS UTILIZADAS

O nosso programa baseia-se, fundamentalmente, em duas estruturas de dados para gerir e sincronizar processos: semáforos e variáveis partilhadas.

## SEMÁFOROS

Semáforos são estruturas de código que contêm uma variável inteira, métodos de criação e métodos de comparação.

Estas estruturas assemelham-se a uma "cancela". Quando um processo passa pelo semáforo esta fecha, impedindo os outros processos de atravessar até que o semáforo abra novamente.



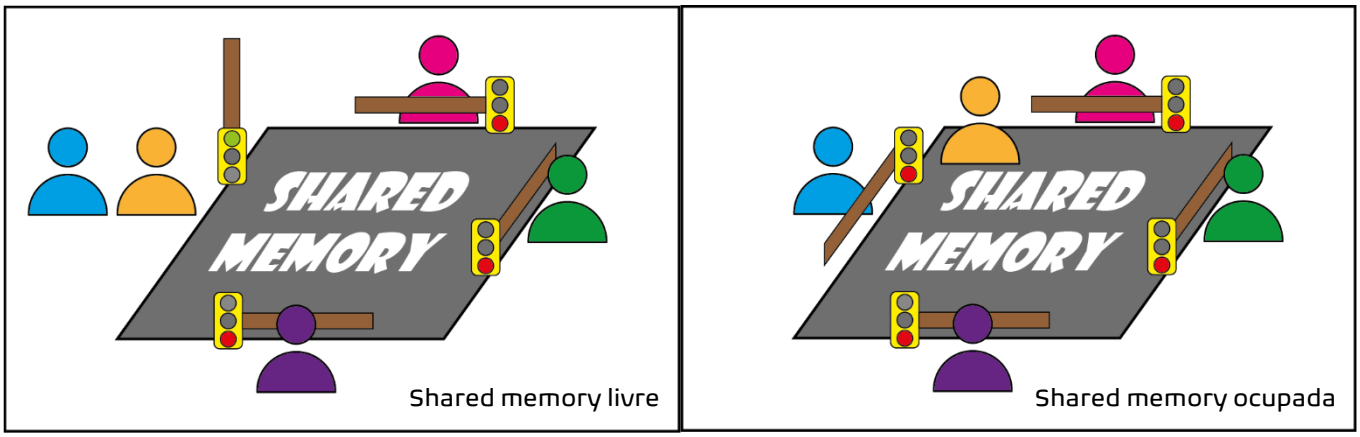

Logicamente, podemos reduzir os semáforos a uma variável inteira que é decrementada conforme um processo passa. Assim, o semáforo bloqueia os processos quando o seu valor for zero (fecha a "cancela") e desbloqueia-os quando o seu valor for maior que zero (abre a "cancela").
Para que a "cancela" abra basta que um processo incremente o valor da variável inteira do semáforo até que este seja maior do que 0.

Os semáforos são estruturas essenciais para controlar segmentos de memória partilhada, uma vez que impedem múltiplos processos de operar dentro desta paralelamente, evitando ao máximo que haja corrupção de dados sensíveis (ex: se um processo A lê uma variável enquanto um processo B escreve nela, o resultado é imprevisível e, muitas vezes, incorreto).

# VARIÁVEIS PARTILHADAS

As variáveis partilhadas são segmentos de memória que não estão associados a um único processo, podendo assim ser acedidos por vários processos e *threads*.

Dada a natureza destas variáveis, existe o risco de sobreposição ou corrupção de dados sensíveis, o que levaria a comportamentos erráticos e imprevisíveis dos processos que precisam destas variáveis para a sua execução. Para que isto não aconteça usamos semáforos sempre que haja acesso, ou alteração de variáveis partilhadas.

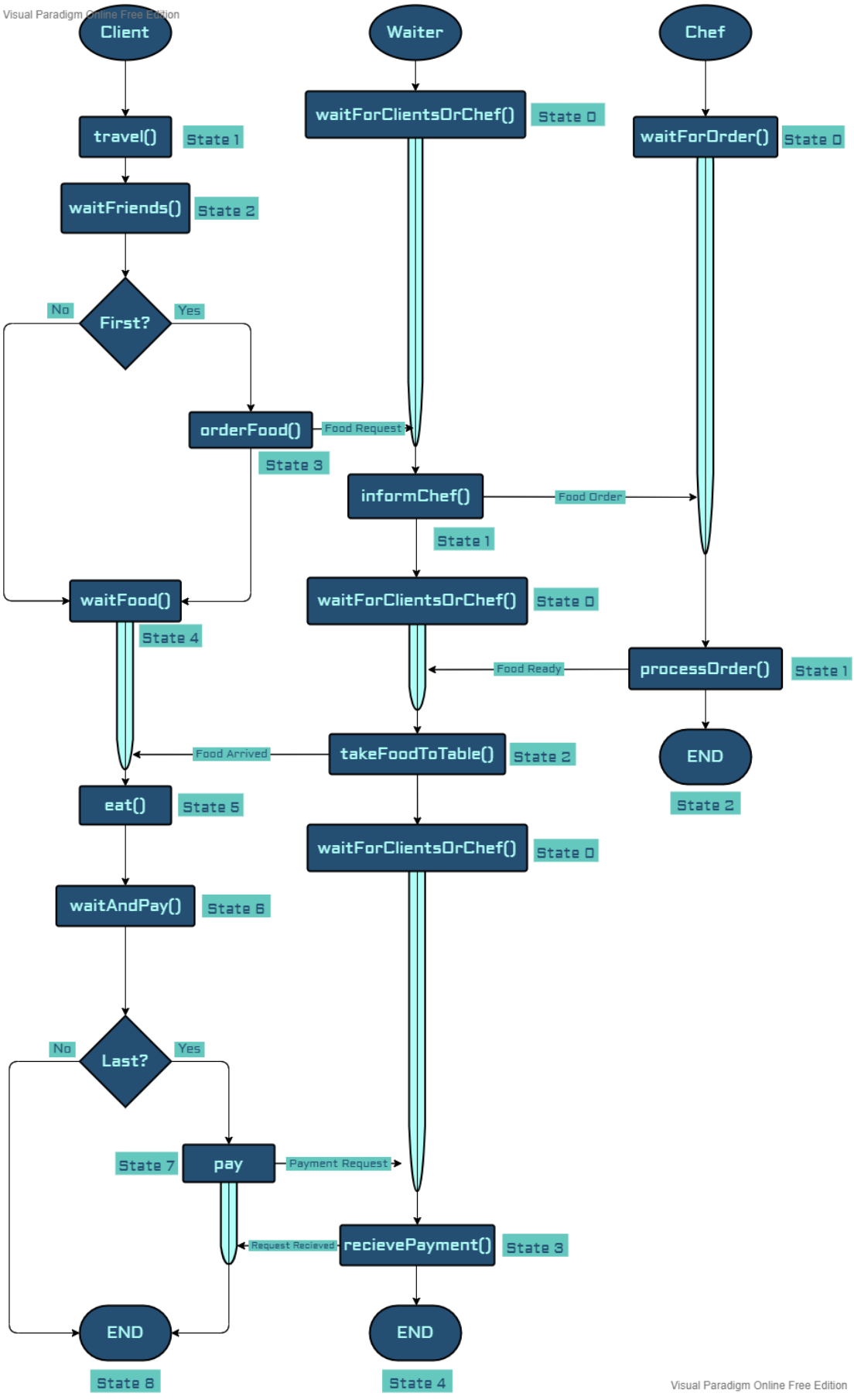



Shared memory livre

Shared memory ocupada

Neste projeto, as variáveis partilhadas são utilizadas para guardar a informação necessária aos diferentes atores e ao cenário geral, como por exemplo, o ProcessID do primeiro e último processo a chegar à mesa, o número de clientes (processos) que já acabou de comer, etc.
Guardamos também na memória partilhada o tipo de pedido que está a ser executado no momento para que os processos incluídos saibam que instrução executar.

# FLUXO DE EXECUÇÃO



| STATE | Client | Waiter | Chef |
|---|---|---|---|
| 0 | --- | Esperar pelo pedido | Esperar pela ordem |
| 1 | INIT | Informar o chef | Preparar o pedido |
| 2 | Esperar pelos outros clientes | Levar a comida para a mesa | END |
| 3 | Pedir a comida | Receber o pagamento | |
| 4 | Esperar pela comida | END | |
| 5 | Comer | | |
| 6 | Esperar que os outros clientes acabem de comer | | |
| 7 | Esperar pela confirmação do pagamento | | |
| 8 | END | | |

# DESCRIÇÃO

Os clientes viajam para a mesa com tempos variados e esperam até que todos cheguem.
Enquanto isso, o empregado e o chef aguardam pelos pedidos.
Quando todos os clientes chegarem, o que chegou primeiro faz o pedido da mesa toda ao empregado, que por sua vez vai informar o chef. Os clientes esperam pela comida e o empregado espera pelo chef.
O chef prepara o pedido e chama o empregado que vai entregar a comida à mesa. O chef não tem mais interações.
Os clientes comem ao seu ritmo e esperam até todos terminarem a refeição. Enquanto isso, o empregado espera.
Após todos acabarem de comer, todos os clientes, exceto o último a chegar, acabam as suas interações. O último cliente chama o empregado e paga pelo pedido. O empregado recebe o pagamento e acaba as suas interações. Ao receber o comprovativo de pagamento, o último cliente acaba também as suas interações.

# SEMÁFOROS USADOS

Neste projeto, para controlar os acessos à memória partilhada usamos os seguintes semáforos:

| Semáforo | Quem espera? (down()) | Função | Quem o levanta? (up()) | Onde | Número de *ups* |
|---|---|---|---|---|---|
| mutex | Todos | Garantir a fidelidade da memória partilhada | Todos | Todas as funções | --- |
| friendsArrived | *Clients* | Esperar que todos os *clients* cheguem à mesa | Último *client* e depois todos os outros *clients* | waitFriends() | N.º de *clients* |
| requestRecieved | *Clients* | Esperar a confirmação do *waiter* | *Waiter* | waitForChef() ou recievePayment() | 4 |
| foodArrived | *Clients* | Esperar que a comida chegue à mesa | *Waiter* | takeFoodToTable() | N.º de *clients* |
| allFinished | *Clients* | Esperar que todos os *clients* acabem de comer | Último *client* que acaba de comer e depois todos os outros *clients* | waitAndPay() | N.º de *clientes* |
| waiterRequest | *Waiter* | Identificar que um pedido foi enviado ao *waiter* | Primeiro e último *client* a chegar à mesa e Chef | orderFood(), processOrder(), waitAndPay() | 3 |
| waitOrder | *Chef* | Identificar que um pedido foi enviado ao *chef* | *Waiter* | informChef() | 1 |

# FUNÇÕES

O código que nos foi fornecido como base contempla funções incompletas para gerir os semáforos de cada '*role* ' dada aos processos (*client, waiter, chef* ). Assim, existem funções exclusivas a cada *role* :

## CLIENT

### travel()

Nesta função é calculado um número pseudoaleatório que será o tempo que o cliente demora a chegar à mesa.

```
static void travel (int id)
{
    usleep((unsigned int) floor ((1000000 * random ()) / RAND_MAX + 1000));
}
```

### waitFriends()

Esta função entra na região crítica (memória partilhada), altera o seu estado para WAIT_FOR_FRIENDS e incrementa o número de *clients* na mesa.
A função também observa o número de *clients* que já se encontram na mesa para determinar o primeiro e o último *client* a chegar.

Depois disto, todos os *clients,* menos o último, ficam à espera que os restantes *clients* cheguem à mesa usando o semáforo *friendsArrived* para bloquear todos os processos.

O último *client* ignora o semáforo e incrementa o valor do semáforo para que um outro processo *client* possa passar. Os processos vão passando e abrem lugar para que o próximo passe ao incrementar o valor do semáforo.

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

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

    /* Check if this Client is the first (happens if no other client is at the table) */
    if (sh->fSt.tableClients == 0) {
        first = true;
        sh->fSt.tableFirst = id;
    }

    /* Set Client state as waiting for the other Clients to arrive and add him to the table
*/
    sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS;
    sh->fSt.tableClients++;

    /* Check if the Client is the last one and pass him directly to waiting for food and
set him as the last one */
    if (sh->fSt.tableClients == TABLESIZE) {
        sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
        sh->fSt.tableLast = id;
    }

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

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

    /* Stop all the clients except the last one to arrive */
    if (sh->fSt.tableClients != TABLESIZE) {
        if (semDown (semgid, sh->friendsArrived) == -1)
        { perror ("error on the down operation for 'friendsArrived' semaphore access
(CT)");
            exit (EXIT_FAILURE);
        }
    }

    /* Give enough space for the next client to pass the semaphore*/
    if (semUp (semgid, sh->friendsArrived) == -1)
    { perror ("error on the up operation for 'friendsArrived' semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    return first;
}
```

# orderFood()

Esta função é executada, exclusivamente, pelo primeiro processo *client* a chegar à mesa. O processo altera o seu estado para FOOD_REQUEST e muda a *flag* partilhada de foodRequest para *true* (sempre dentro da região crítica).

Após fazer isto, o processo espera que todos os restantes *clients* cheguem à mesa e faz o pedido ao *waiter* , esperando que este responda antes de prosseguir.

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

    /* Update its state */
    sh->fSt.st.clientStat[id] = FOOD_REQUEST;
    sh->fSt.foodRequest = true;
    saveState(nFic, &(sh->fSt));

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

    /* Wait for the everyone to arrive at the table before calling the waiter */
    if (semDown (semgid, sh->friendsArrived) == -1) {
        perror ("error on the down operation for 'friendsArrived' semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    /* Wait for the waiter to receive the request */
    if (semDown (semgid, sh->requestReceived) == -1) {
        perror ("error on the down operation for 'requestReceived' semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}
```

# waitFood()

Nesta função o processo *client* muda o seu estado para WAIT_FOR_FOOD e espera num semáforo foodArrived pela comida.
Neste caso, os processos *client* só passam pelo semáforo quando o processo *waiter* os deixa passar.

Após passarem pelo semáforo os processos *client* mudam o seu estado para EAT.

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

    /* Update its state */
    sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD;
    saveState(nFic, &(sh->fSt));

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

    /* Wait for the food */
    if (semDown (semgid, sh->foodArrived) == -1) {
        perror ("error on the down operation for 'foodArrived' semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

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

    /* Update its state */
    sh->fSt.st.clientStat[id] = EAT;
    saveState(nFic, &(sh->fSt));

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

## eat()

Nesta função é calculado um número pseudoaleatório que será o tempo que o *client* demora a comer. Este tempo é aplicado ao fazer o processo dormir por essa duração calculada.

```
static void eat (int id)
{
    usleep((unsigned int) floor ((MAXEAT * random ()) / RAND_MAX + 1000));
}
```

## waitAndPay()

Aqui os processos *client* mudam o seu estado para WAIT_FOR_OTHERS , incrementam o número de *clients* que já acabaram de comer e esperam que os restantes *clients* acabem a refeição.

O último *client* a terminar a refeição ignora o semáforo que bloqueou os outros processos e incrementa o valor do semáforo permitindo que o outro processo *client* passe, muda o seu estado para FINISHED e incremente o valor do semáforo, que por sua vez deixa passar outro processo *client*, muda o seu estado para FINISHED, ... , repetindo-se esta situação até que todos os *clients* passem pelo semáforo.

O último processo *client* que chegou à mesa é o último a passar pelo semáforo, visto que ainda tem de pagar. Assim, este processo *client* muda o seu estado para WAIT_FOR_BILL e notifica o processo *waiter* através do semáforo *waiterRequest* e da *flag paymentRequest*.
Depois disto, o *client* espera pela resposta do *waiter* e continua para o próximo estado.

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

    /* Update its state */
    sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS;
    sh->fSt.tableFinishEat++;
    saveState(nFic, &(sh->fSt));
```

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


    /* If the client was not the last to stop eating, wait for all to do so */
    if (sh->fSt.tableFinishEat != TABLESIZE) {
        /* wait for all the Clients to finish eating */
        if (semDown (semgid, sh->allFinished) == -1)
        { perror ("error on the down operation for 'allFinished' semaphore access (CT)");
            exit (EXIT_FAILURE);
        }
    }

    /* Give enough space for the next client to pass the semaphore*/
    if (semUp (semgid, sh->allFinished) == -1)
    { perror ("error on the up operation for 'allFinished' semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    /* Make the last client to arrive at the table pay the bill */
    if(sh->fSt.tableLast == id) {

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

        /* Update its state */
        sh->fSt.st.clientStat[id] = WAIT_FOR_BILL;
        sh->fSt.paymentRequest = true;
        saveState(nFic, &(sh->fSt));

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

        /* Wait for everyone to be finished before paying */
        if (semDown (semgid, sh->allFinished) == -1)
        { perror ("error on the down operation for 'allFinished' semaphore access (CT)");
            exit (EXIT_FAILURE);
        }

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

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

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

    /* Update its state */
    sh->fSt.st.clientStat[id] = FINISHED;
    saveState(nFic, &(sh->fSt));

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

    /* let the next Clients stop eating */
    if (semUp (semgid, sh->allFinished) == -1)
    { perror ("error on the up operation for 'allFinished' semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}
```

# WAITER

## waitForClientOrChef()

Nesta função, o *waiter* muda o seu estado para WAIT_FOR_REQUEST e aguarda por um pedido através do semáforo *waiterRequest*.

Após algum processo (*client* ou *chef* ) aumentar o semáforo, o processo *waiter* lê qual *flag* tem o valor *true* para determinar qual foi o pedido realizado. Estas operações são sempre executadas dentro do *mutex*.

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

    /* Update its state */
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
    saveState(nFic, &(sh->fSt));

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

    /* Wait for a request */
    if (semDown (semgid, sh->waiterRequest) == -1)  {
        perror ("error on the down operation for 'waiterRequest' semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

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

    if(sh->fSt.foodRequest) {
        ret = FOODREQ;
        sh->fSt.foodRequest = false;
```

```
        /* Tell Client that ordered the food that the request was received */
        if (semUp (semgid, sh->requestReceived) == -1) {
            perror ("error on the up operation for 'requestReceived' semaphore access
(WT)");
            exit (EXIT_FAILURE);
        }
    }
    else if (sh->fSt.foodReady) {
        ret = FOODREADY;
        sh->fSt.foodReady = false;
        /* Tell the Chef that the request was received */
        if (semUp (semgid, sh->requestReceived) == -1) {
            perror ("error on the up operation for 'requestReceived' semaphore access
(WT)");
            exit (EXIT_FAILURE);
        }
    }
    else if (sh->fSt.paymentRequest) {
        ret = BILL;
        sh->fSt.paymentRequest = false;
        /* Tell Client that paid the bill that the request was received */
        if (semUp (semgid, sh->requestReceived) == -1) {
            perror ("error on the up operation for 'requestReceived' semaphore access
(WT)");
            exit (EXIT_FAILURE);
        }
    }
    else {
        perror ("error of invalid waiter request (WT)");
        exit (EXIT_FAILURE);
    }

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


    return ret;

}
```

## informChef()

Nesta função, o processo *waiter* muda o seu estado para INFORM_CHEF e muda o valor da *flag foodOrder* para *true*. Após isto o processo alerta o *Chef* da presença de um pedido.

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

    /* Update its state */
    sh->fSt.st.waiterStat = INFORM_CHEF;
    sh->fSt.foodOrder = true;
    saveState(nFic, &(sh->fSt));

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

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

## takeFoodToTable()

Aqui o *waiter* muda o seu estado para TAKE_TO_TABLE e envia o sinal aos processos *client* ao aumentar o valor do semáforo *foodArrived* para que estes passem.

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

    /* Update its state */
    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
```

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

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

    /* Up the foodArrived semaphore so all the clients can start eating */
    for (int x = 0; x < TABLESIZE; x++) {
        if (semUp (semgid, sh->foodArrived) == -1)  {
            perror ("error on the up operation for 'foodArrived' semaphore access (CT)");
            exit (EXIT_FAILURE);
        }
    }
}
```

## recievePayment()

Nesta função, o *waiter* muda o seu estado para RECIEVE_PAYMENT e indica ao *client* que o pagamento foi efetuado com sucesso com o uso do semáforo *requestRecieved*.

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

    /* Update its state */
    sh->fSt.st.waiterStat = RECEIVE_PAYMENT;
    saveState(nFic, &(sh->fSt));

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

    /* Tell the Client that is paying that the payment was received */
    if (semUp (semgid, sh->requestReceived) == -1)  {
     perror ("error on the up operation for 'requestReceived' semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
}
```

# CHEF

## waitForOrder()

Aqui o processo *chef* muda o seu estado para WAIT_FOR_ORDER e aguarda que um outro processo levante o semáforo *waitOrder*.

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

    /* Update its state */
    sh->fSt.st.chefStat = WAIT_FOR_ORDER;

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

    /* Wait for an order to be received */
    if (semDown (semgid, sh->waitOrder) == -1) {
        perror ("error on the down operation for 'waitOrder' semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
}
```

## processOrder()

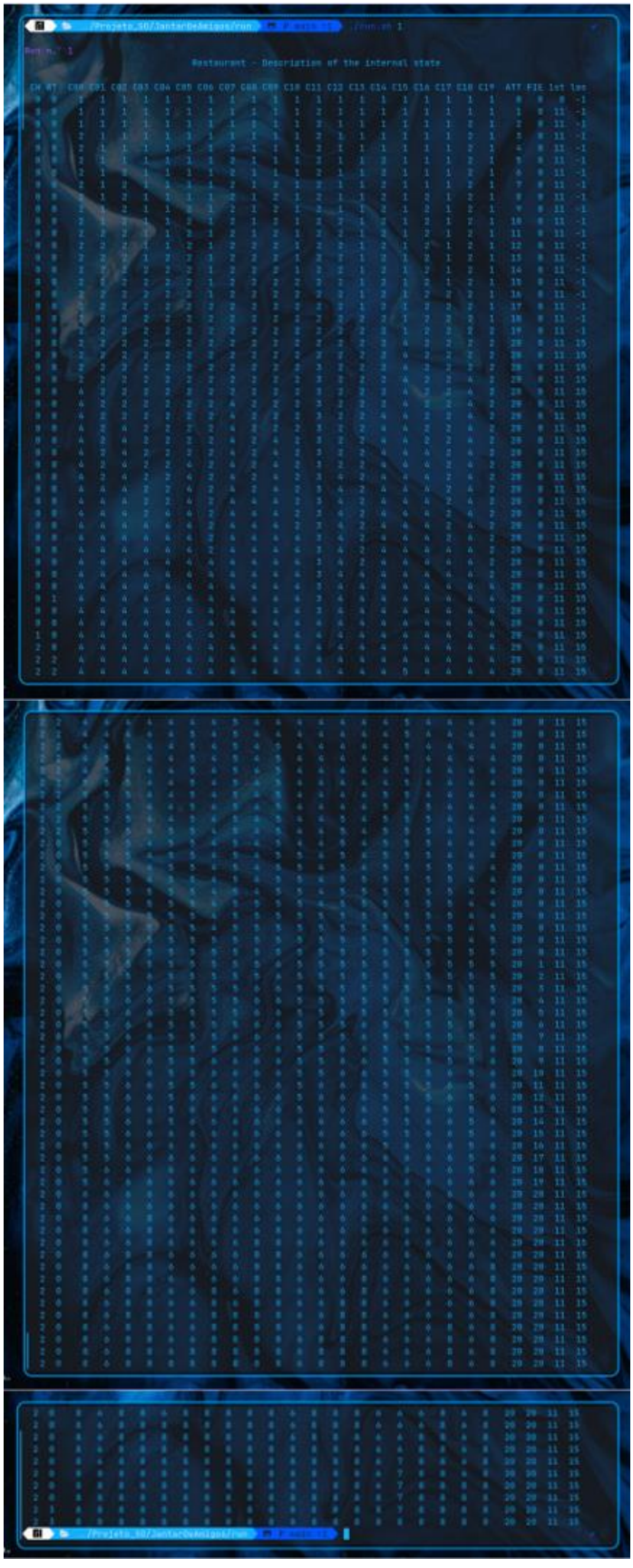
Nesta função o processo *chef* muda o seu estado para COOK e é calculado um número pseudoaleatório que será o tempo de preparo do pedido.

Após o tempo de preparo, o *chef* muda o seu estado para REST, altera a *flag foodReady* para *true* e notifica o *waiter* com o semáforo *waiterRequest*. Antes de terminar, o *chef* aguarda pela confirmação de que o *waiter* recebeu o pedido com sucesso através do semáforo requestRecieved.

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

    /* Update its state */
    sh->fSt.st.chefStat = COOK;
    sh->fSt.foodReady = true;
    saveState(nFic, &(sh->fSt));

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

    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));

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

    /* Update its state */
    sh->fSt.st.chefStat = REST;
    saveState(nFic, &(sh->fSt));

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

    /* Signal the waiter that the cook is finished */
    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for 'waiterRequest' semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    /* Wait for waiter to confirm request */
    if (semDown (semgid, sh->requestReceived) == -1) {
        perror ("error on the down operation for 'requestReceived' semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
}
```

# RESULTADOS

Após compilar e executar o código, obtivemos os seguintes resultados:



Estes resultados são bastante similares aos resultados esperados para este projeto (contemplados ao executar os ficheiros de referência fornecidos no início do projeto). As únicas diferenças entre os nossos resultados e os resultados esperados são a ordem pelo que os processos *client* passam pelos semáforos e os processos *client* que chegam em primeiro e último lugar, visto que a ordem de chegada inicial é (pseudo)aleatória.

Assim, podemos afirmar que os resultados obtidos estão de acordo com o esperado para este projeto.

# CONCLUSÃO

Após correrrmos o código, e compararmos os resultados obtidos e os resultados esperados, pudemos concluir que todas as operações foram implementadas com sucesso.

Com este projeto tivemos uma excelente oportunidade de perceber melhor o funcionamento da memória partilhada e da sincronização de processos, tal como a utilização de semáforos para controlar acessos à memória partilhada e envio de sinais, o que nos permite desenvolver programas mais complexos (envolvendo múltiplas threads/processos e operando de forma previsível). Pudemos também aprofundar os nossos conhecimentos em C, principalmente o âmbito do próprio projeto – manipulação de processos e uso de memória partilhada.

Face aos dados obtidos, podemos concluir que este trabalho se tornou possível graças ao conhecimento e prática desenvolvida nas aulas teóricas e práticas, tanto de *C* como de manipulação de processos e memória partilhada. Valorizamos também as muitas pesquisas que fizemos na web de modo a concluirmos este projeto com sucesso.

# WEBGRAFIA


- https://askubuntu.com/
- https://linuxhint.com/
- https://online.visual-paradigm.com/diagrams/templates/?category=flowchart
- https://stackoverflow.com/
- https://superuser.com/
- https://unix.stackexchange.com/
- https://www.tutorialkart.com/