Ano letivo 2022/2023

universidade de aveiro deti departamento de electrónica, telecomunicações e informática

PROJETO 2
JANTAR DE AMIGOS

Sistemas Operativos

Prof : José Lau
Prof : António Campos

Nome: Diogo Almeida N°Mec: 108902
Nome: Joaquim Rosa N°Mec: 109089

Índice

1- Introdução

No âmbito da cadeira de Sistemas Operativos, foi-nos pedido para simular um jantar de 20 amigos (clients) num restaurante. Este mesmo restaurante é constituído apenas por um empregado de mesa (waiter) e um cozinheiro (chef). Foi nos impostas algumas condições: o primeiro a chegar faz o pedido da comida, mas só depois de todos chegarem; o empregado deve levar o pedido ao cozinheiro e trazer a comida quando estiver pronta; os amigos só podem abandonar abandonar a mesa quando todos acabarem de comer; o último client a chegar ao restaurante deve pagar a conta.

Para fazer a sincronização dos três processos (client, wait e chef) com base nas regras que nos foram impostas, utilizamos memória partilhada e semáforos.

Todos os processos terão Regiões Críticas, zona de código onde é manipulado dados que são partilhados pelos processos e que não pode ser executada simultaneamente por mais que um processo. Para que isso aconteça, usamos um Semáforo, ferramenta que garante que a região crítica não será acedida por mais do que um processo (Exclusão Mútua). O semáforo possui um estado interno que é um valor inteiro, caso o valor do semáforo seja maior que zero, o processo pode acessar o código e o valor do semáforo é decrementado, caso seja igual a zero, o processo é bloqueado até que o semáforo seja liberado novamente. Neste trabalho usamos os métodos *semUp* e *semDown* para incrementar e decrementar o valor do semáforo, respetivamente.

Também usamos semáforos para permitir que um processo avisasse outro de que algo tinha acontecido (Signaling). Isto permitiu impor um fluxo de execução entre determinadas secções de código de processos distintos.

	Exclusão Mútua	**Signaling**
Semáforos	*mutex*	*friendsArrived, requestReceived, foodArrived, allFinished, waiterRequest, waiterOrder*

Estes semáforos encontram-se definidos no ficheiro *sharedDataSync.h.* O *mutex* será inicializado com valor 1 e os restantes com valor 0.

Comportamento dos semáforos

Semáforo	Down	Função	NºDown	Up	Função	NºUp
mutex	Client	*waitFriends*	1	Client	*waitFriends*	1
		waitFood	2		*waitFood*	2
	Client (exceto o último)	*waitAndPay*	2	Client (exceto o último)	*waitAndPay*	2
	Último Client	*waitAndPay*	3	Último Client	*waitAndPay*	3
	Primeiro Client	*orderFood*	1	Primeiro Client	*orderFood*	1
	Waiter	*waitForClientOrChef*	6	Waiter	*waitForClientOrChef*	6
		informChef	1		*informChef*	1
		takeFoodToTable	1		*takeFoodToTable*	1
		receivePayment	1		*receivePayment*	1
	Chef	*waitForOrder*	1	Chef	*waitForOrder*	1
		processOrder	1		*processOrder*	1
friendsArrived	Client (exceto o último)	*waitFriends*	1	Último Client	*waitFriends*	19

waiterRequest	Waiter	*waitForClientOrChef*	3	Primeiro Client	*orderFood*	1
				Último Client	*waitAndPay*	1
				Chef	*processOrder*	1
requestReceived	Primeiro Client	*orderFood*	1	Waiter	*informChef*	1
	Último Client	*waitAndPay*	1		*receivePayment*	1
waitOrder	Chef	*waitForOrder*	1	Waiter	*informChef*	1
foodArrived	Client	*waitFood*	1	Waiter	*takeFoodToTable*	20
allFinished	Client	*waitAndPay*	1	Último Client a acabar de comer	*waitAndPay*	20

No ficheiro *probDataStruct.h* podemos encontrar a definição dos estados das entidades do problema, bem como outras variáveis e *flags* que nos vão ajudar na sincronização das mesmas.

```c
/**
 *  \brief Definition of <em>state of the intervening entities</em> data type.
 */
typedef struct {
    /** \brief waiter state */
    unsigned int waiterStat;
    /** \brief chef state */
    unsigned int chefStat;
    /** \brief client state array */
    unsigned int clientStat[TABLESIZE];

} STAT;


/**
 *  \brief Definition of <em>full state of the problem</em> data type.
 */
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;
```

No ficheiro *probConst.h* são definidas as constantes para os estados das entidades intervenientes.

```
/** \brief client initial state */
#define  INIT              1
/** \brie                                    nds to arrive at table */
#define
/** \brie                                    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

/* Chef state constants */

/** \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

/* Waiter state constants */

/** \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
```

#define FOOD_REQUEST 3
client is requesting food to waiter

2- Fluxo de Execução

Função waitFriends() - Cliente

A função *waitFriends()* é chamada, possuindo como argumento os ids de cada cliente.

```
*/
static bool waitFriends(int id)
{
    bool first = false; //Variável de controlo para saber se o cliente é o primeiro

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

    sh->fSt.tableClients = sh->fSt.tableClients + 1; //Incrementa o número de clientes na mesa

    if(sh->fSt.tableClients == 1){ //Se só houver um cliente na mesa, então é o primeiro

        first = true; //Variável passa a true pois é o primeiro cliente

        sh->fSt.tableFirst = id; //Guarda o id do primeiro cliente na mesa
    }
```

A variável *first* é uma variável de controlo inicializada a *false,* que permite saber se o cliente é o primeiro, para depois fazer o pedido da comida mais tarde.

É efetuado um semDown ao mutex, para que a região crítica seja acedida por um cliente de cada vez.

Dentro da região crítica, cada cliente incrementa o *tableClients,* para guardar o número de clientes da mesa. De seguida é verificado se a mesa possui apenas um cliente, em caso afirmativo, a variável *first* passa a true e o id do primeiro cliente é guardado em *tableFirst.*

```
    if (sh->fSt.tableClients != TABLESIZE){ //Se o número de clientes na mesa for diferente do tamanho da mesa, então não é o último

        sh->fSt.st.clientStat[id] = WAIT_FOR_FRIENDS; //Altera o estado do cliente para WAIT_FOR_FRIENDS(2)

        saveState(nFic, &(sh->fSt)); //Printa uma linha no prompt
    }
       /* insert your code here */


    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 */
    if (sh->fSt.tableClients != TABLESIZE){

        semDown(semgid, sh->friendsArrived); //O cliente fica à espera que os outros clientes cheguem à mesa
    }

    return first;
}
```

É efetuado um semUp ao mutex, fazendo com que a região crítica esteja livre para ser acedida por outro processo.

Enquanto o número de clientes na mesa for diferente do tamanho da mesa, cada cliente irá aguardar pelos restantes, atualizando o seu estado para WAIT_FOR_FRIENDS e imprimindo no terminal.

Fora da região crítica, enquanto a mesa não estiver cheia, cada cliente tenta efetuar um semDown em *friendsArrived*, porém o valor de *friendsArrived* está a 0, fazendo com que estes aguardem que os restantes cheguem à mesa.

```
if(sh->fSt.tableClients == TABLESIZE){ //Se o número de clientes na mesa for igual ao tamanho da mesa, então é o último

    sh->fSt.tableLast = id; //Guarda o id do último cliente na mesa

    sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD; //Altera o estado do último cliente para WAIT_FOR_FOOD(4)

    saveState(nFic, &(sh->fSt)); //Printa uma linha no prompt

    for(int i = 1; i < TABLESIZE; i++){ //Para cada cliente na mesa, exceto o último
        semUp(semgid, sh->friendsArrived); //Acorda o processo do cliente
    }
}
```

Dentro da região crítica, quando a mesa estiver cheia, o programa guarda o id do último cliente a chegar à mesa em *tableLast* e altera o estado do mesmo para ficar a aguardar a comida (WAIT_FOR_FOOD) e imprime no terminal.

De seguida, são efetuados 19 semUps a *friendsArrived,* acordando os 19 processos de *client* que estavam à espera que todos chegassem à mesa.

Função orderFood() - Cliente

A função *orderFood()* é chamada caso *first* seja igual a *true,* logo esta função será apenas executada no processo que corresponde ao primeiro cliente e recebe como argumento o id do mesmo.

```
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);
    }

    /* i  FULL_STAT <unnamed>::fSt
    sh->fSt.foodRequest = 1; //Flag que indica ao waiter que existe um pedido

    semUp(semgid, sh->waiterRequest); //Acorda o waiter pois existe um pedido (waiterRequest = 1)

    sh->fSt.st.clientStat[id] = FOOD_REQUEST; //Altera o estado do cliente para FOOD_REQUEST(3)

    saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados

    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 */
    semDown(semgid, sh->requestReceived); //O cliente fica à espera que o pedido seja recebido pelo waiter
}
```

Dentro da região crítica, a *flag foodRequest* é passada a 1, indicando ao waiter que existe um pedido.

É efetuado um semUp ao semáforo *waiterRequest*, alterando o seu valor para 1, que fará com que acorde o processo do waiter que estava a aguardar um pedido.

O cliente que está a efetuar o pedido atualiza o seu estado para FOOD_REQUEST e imprime no terminal.

Fora da região crítica, o processo do cliente não consegue efetuar o semDown a *requestReceived,* ficando a aguardar que o pedido seja recebido pelo *waiter*.

Função waitFood() -Cliente

A função *waitFood()* é chamada, possuindo como argumento o id de cada cliente.

```
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 */

    sh->fSt.st.clientStat[id] = WAIT_FOR_FOOD; //Altera o estado do cliente para WAIT_FOR_FOOD(4)

    saveState(nFic, &(sh->fSt));//Printa uma linha no prompt

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

    /* insert your code here */

    semDown(semgid, sh->foodArrived); //O cliente fica à espera que a comida seja entregue pelo waiter
```

Dentro da região crítica, cada processo de *client* altera o seu estado para WAIT_FOR_FOOD e imprime no terminal.

Fora da região crítica, cada processo de *client* é posto a dormir, esperando que a comida chegue.

Função waitForClientOrChef() - Waiter

```c
static int waitForClientOrChef()
{
    int ret=0; //Variavel que vai guardar o tipo de açao que o waiter vai executar

    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 */

    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST; //Alterar o estado do waiter para WAIT_FOR_REQUEST(0) pois ele vai esperar por um pedido

    saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados


    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 */
    semDown (semgid, sh->waiterRequest); //Waiter espera por um pedido (waiterRequest=0)
```

Na função *waitForClientOrChef(),* é inicializada a variável “ret” com valor 0, que irá guardar o tipo de ação que *waiter* vai executar.

O waiter altera o seu estado para WAIT_FOR_RESQUEST, o que significa que ele está à espera de um pedido tanto de um *client* como do *chef*, e imprime o estado no terminal.

Fora da região crítica, o processo do waiter é posto a dormir, ficando a aguardar um pedido. Como foi efetuado um pedido de comida por um client, o processo do *waiter* acorda e continua a correr o código.

```
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 */
if (sh->fSt.foodRequest == 1) { //Se o pedido for de comida (alterado pelo cliente)

    ret = FOODREQ; //ret=1

    sh->fSt.foodRequest = 0; //Volta a por o pedido de comida a 0

} else if (sh->fSt.foodReady == 1) { //Se a comida já está pronta para levar à mesa (alterado pelo chef)

    ret = FOODREADY; //ret=2

    sh->fSt.foodReady = 0; //Volta a chamada do chef a 0

} else if (sh->fSt.paymentRequest == 1) { //Se o pedido for de pagamento (alterado pelo cliente)

    ret = BILL; //ret=3

    sh->fSt.paymentRequest = 0; //Volta a por o pedido de pagamento a 0

}

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

return ret;
```

Dentro da região crítica, o *waiter* verifica que tipo de pedido é que foi feito. Como foi feito um pedido de comida, é aguardado em “ret” o valor de FOODREQ, ou seja 1, e a *flag foodRequest* volta a ter valor 0.

O processo sai da região crítica e a função retorna o valor de “ret”, neste caso 1.

Função *informChef()* - Waiter

```c
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 */
    sh->fSt.foodOrder = 1; //Flag que indica ao chef que existe um pedido de comida

    sh->fSt.st.waiterStat = INFORM_CHEF; //Alterar o estado do waiter para INFORM_CHEF(1) pois ele vai informar o chef do pedido

    saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados

    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 */
    semUp (semgid, sh->requestReceived); //Acorda cliente que realizou o pedido para que ele espere pela comida

    semUp (semgid, sh->waitOrder); //Acorda o chef para que ele pcomece a preparar a comida
}
```

Como o valor de “ret” era FOODREQ, a função a ser executada será *informChef()*.

Entrando na região crítica, a *flag foodOrder* é passada a 1, e indicando depois ao chef que existe um pedido de comida. O *waiter* atualiza o seu estado para INFORM_CHEF, indicando que vai informar o *chef* da existência do pedido, e imprime o mesmo no terminal.

Já fora da região crítica, o *waiter* acorda o cliente que realizou o pedido, ao realizar um semUp ao semáforo *requestReceived,* informando-o de que recebeu o pedido, e o mesmo passa o seu estado para WAIT_FOR_FOOD. O waiter acorda também o *chef* que estava aguardando um pedido de comida, ao realizar um semUp ao semáforo *waitOrder.*

O *waiter* volta a colocar o seu estado a WAIT_FOR_REQUEST, e é novamente posto a dormir, aguardando um pedido.

Função *waitForOrder()* - Chef

```c
static void waitForOrder ()
{
    /* insert your code here */

    semDown(semgid, sh->waitOrder); //chef espera que o waiter tenha um pedido para ele

    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 */
    sh->fSt.foodOrder = 0; //chef vai fazer o pedido atual por isso coloco o numero de pedidos a 0

    sh->fSt.st.chefStat = COOK; //Alterar o estado do chef para COOK(1), vai preparar o pedido

    saveState(nFic, &sh->fSt); //Printa linha no prompt com os estados atualizados

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

O processo do *chef* que inicialmente estava a dormir, é acordado com o semUp do *waiter a waitOrder*.

Entrando na região crítica, o chef coloca a *flag foodOrder* a 0 outra vez, pois vai começar a fazer o pedido. Este altera o seu estado para COOK, imprime o mesmo no terminal e sai da região crítica.

Função *processOrder()* - Chef

```c
static void processOrder ()
{
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0)); //Tempo que o chef demora a fazer o pedido

    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 */

    sh->fSt.foodReady = 1; //O pedido esta pronto para o waiter levar para a mesa

    sh->fSt.st.chefStat = REST; //Alterar o estado do chef para REST(2), vai descansar porque nao tem pedidos

    saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados


    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 */
    semUp(semgid, sh->waiterRequest); //Acorda o waiter para ele ir buscar o pedido
}
```

Esta função é chamada a seguir à execução da função waitForOrder().

Primeiramente é executado um comando que simula o tempo que o chef demora a cozinhar o pedido.

De seguida, entrando na região crítica, o *chef* informa o *waiter* que o pedido está pronto para levar à mesa, através da alteração do valor da *flag foodReady* para 1, e altera o seu estado para REST e imprime o mesmo no terminal.

Por fim, já fora da região crítica, é realizado um semUp a *waiterResquest*, acordando novamente o *waiter* que estava esperando por um pedido.

Função *waitForClientOrChef()* - 2ª *chamada*

O processo inicial será o mesmo que referido anteriormente. Como a *flag foodReady* tem valor 1, o “ret” receberá o valor de FOODREADY, ou seja 2, e o valor da *flag foodReady* volta a passar a 0.

Função *takeFoodToTable()* - Waiter

```c
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 */
    sh->fSt.st.waiterStat = TAKE_TO_TABLE; //Alterar o estado do waiter para TAKE_TO_TABLE(2) pois ele vai levar a comida à mesa

    saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados

    for (int i = 0; i < sh->fSt.tableClients; i++) { //Para cada cliente na mesa
        semUp (semgid, sh->foodArrived); //Acorda o cliente para que ele comece a comer
    }

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

Como o valor de “ret” era FOODREADY, a função a ser executada será *takeFoodToTable()*.

Entrando na região crítica, o *waiter* altera o seu estado para TAKE_TO_TAKE, indicando que está a levar a comida à mesa, e imprime no terminal. De seguida, são efetuados 20 semUps a *foodArrived,* acordando os processos de *client* que estavam à mesa à espera que a comida chegasse.

Entrando na região crítica, o *waiter* altera o seu estado para TAKE_TO_TAKE, indicando que está a levar a comida à mesa, e imprime no terminal. De seguida, são efetuados 20 semUps a *foodArrived,* acordando os processos de *client* que estavam à mesa à espera que a comida chegasse.
Entrando na região crítica, o *waiter* altera o seu estado para TAKE_TO_TAKE, indicando que está a levar a comida à mesa, e imprime no terminal. De seguida, são efetuados 20 semUps a *foodArrived,* acordando os processos de *client* que estavam à mesa à espera que a comida chegasse.

Função *waitFood()* - Cliente

```c
/* insert your code here */

semDown(semgid, sh->foodArrived); //O cliente fica à espera que a comida seja entregue pelo waiter

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 */

sh->fSt.st.clientStat[id] = EAT; //Altera o estado do cliente para EAT(5) pois a comida foi entregue

saveState(nFic, &(sh->fSt)); //Printa uma linha no prompt

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

Depois de acordar os processos de *client* que estavam à espera da comida, estes por sua vez vão entrando um a um na região crítica e alterando o seu estado para EAT e imprimindo no terminal, indicando que a comida foi entregue e que o *client* pode começar a comer.

Função *waitAndPay()* - Cliente

De seguida, é executada a função *waitAndPay()*, que recebe como argumento o id de cada *client*.

```
static void waitAndPay (int id)
{
    bool last=false; //Variável que indica se o cliente é o último a chegar à mesa

    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){ //Se o cliente for o último a chegar à mesa

        last = true; //Variável para true pois o id atual foi o último a chegar à mesa

    }

    sh->fSt.st.clientStat[id] = WAIT_FOR_OTHERS; //Altera o estado do cliente para WAIT_FOR_OTHERS(6)

    saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados

    sh->fSt.tableFinishEat = sh->fSt.tableFinishEat + 1; //Incrementa o número de clientes que terminaram de comer
```

A variável *last* é uma variável de controlo inicializada a *false,* que permite saber se o cliente foi o último a chegar, para depois fazer o pagamento do jantar.

Dentro da região crítica, se o id do *client* corresponder ao id do último cliente a chegar ao restaurante, a variável *last* assume o valor *true.* Cada cliente vai alterando o seu estado para WAIT_FOR_OTHERS, indicando que já terminou de comer e está à espera que os outros também terminem, e imprime o seu estado no terminal. De seguida, incrementa o valor da *flag tableFinishEat*, que indica quantos clientes é que já acabaram de comer.

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

/* insert your code here */
semDown (semgid, sh->allFinished); //O cliente fica à espera que todos os outros clientes terminem de comer
```

Já fora da região crítica, cada processo de *client* é posto a dormir, ao tentar efetuar um semDown ao semáforo *allFinished*, aguardando que todos acabem de comer.

```
if (sh->fSt.tableFinishEat == TABLESIZE){ //Se o número de clientes que terminaram de comer for igual ao número de clientes à mesa

    for (int i = 0; i < TABLESIZE; i++){ //Para cada cliente à mesa
        semUp (semgid, sh->allFinished); //Acorda o cliente
    }
}
```

Quando a região crítica for acessada pelo último cliente a acabar de comer, serão efetuados 19 semUps a *allFinished*, acordando os clientes que estavam à espera que todos acabassem de comer.

```
if(last == true) { //Se o cliente atual foi o último a chegar à mesa
    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 */
    sh->fSt.st.clientStat[id] = WAIT_FOR_BILL; /*Altera o estado do cliente para WAIT_FOR_BILL(7)
    pois o ultimo cliente a chegar à mesa está à espera da conta*/

    saveState(nFic, &sh->fSt); //Printa linha no prompt com os estados atualizados

    sh->fSt.paymentRequest = 1; //Indica ao waiter que o cliente está à espera da conta

    semUp (semgid, sh->waiterRequest); //Acorda o waiter pois o cliente está à espera da conta

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

    /* insert your code here */
    semDown(semgid, sh->requestReceived); //O cliente fica à espera que o waiter receba o pedido de conta

}
```

Caso *last* seja igual a *true,* ou seja, caso o cliente seja o último que chegou à mesa, este entra na região crítica, e muda o seu estado para WAIT_FOR_BILL, indicando que está a espera para pagar a conta, e imprime o seu estado no terminal. Este muda também a *flag paymentRequest* para 1, que irá indicar ao *waiter* que tipo de pedido vai receber, e faz um semUp a *waiterRequest,* acordando o *waiter* que estava a aguardar um pedido.

Fora da região crítica, o cliente é posto a dormir ao tentar fazer um semDown a *requestReceived*, indicando que está à espera que o *waiter* receba o seu pedido.

Função *waitForClientOrChef()* – 3ª *chamada*

O processo inicial será o mesmo que referido anteriormente. Como a *flag paymentRequest* tem valor 1, o “ret” receberá o valor de BILL, ou seja 3, e o valor da *flag paymentRequest* volta a passar a 0.

Função *receivePayment()* - Waiter

Como o valor de “ret” era BILL, a função a ser executada será *receivePayment()*.

```
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 */
    sh->fSt.st.waiterStat=RECEIVE_PAYMENT; //Alterar o estado do waiter para RECEIVE_PAYMENT(3) pois ele vai receber o pagamento

    saveState (nFic, &sh->fSt); //Printa linha no prompt com os estados atualizados

    semUp (semgid, sh->requestReceived); //Acorda o cliente para que ele saia da mesa pois o pagamento foi recebido

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


}
```

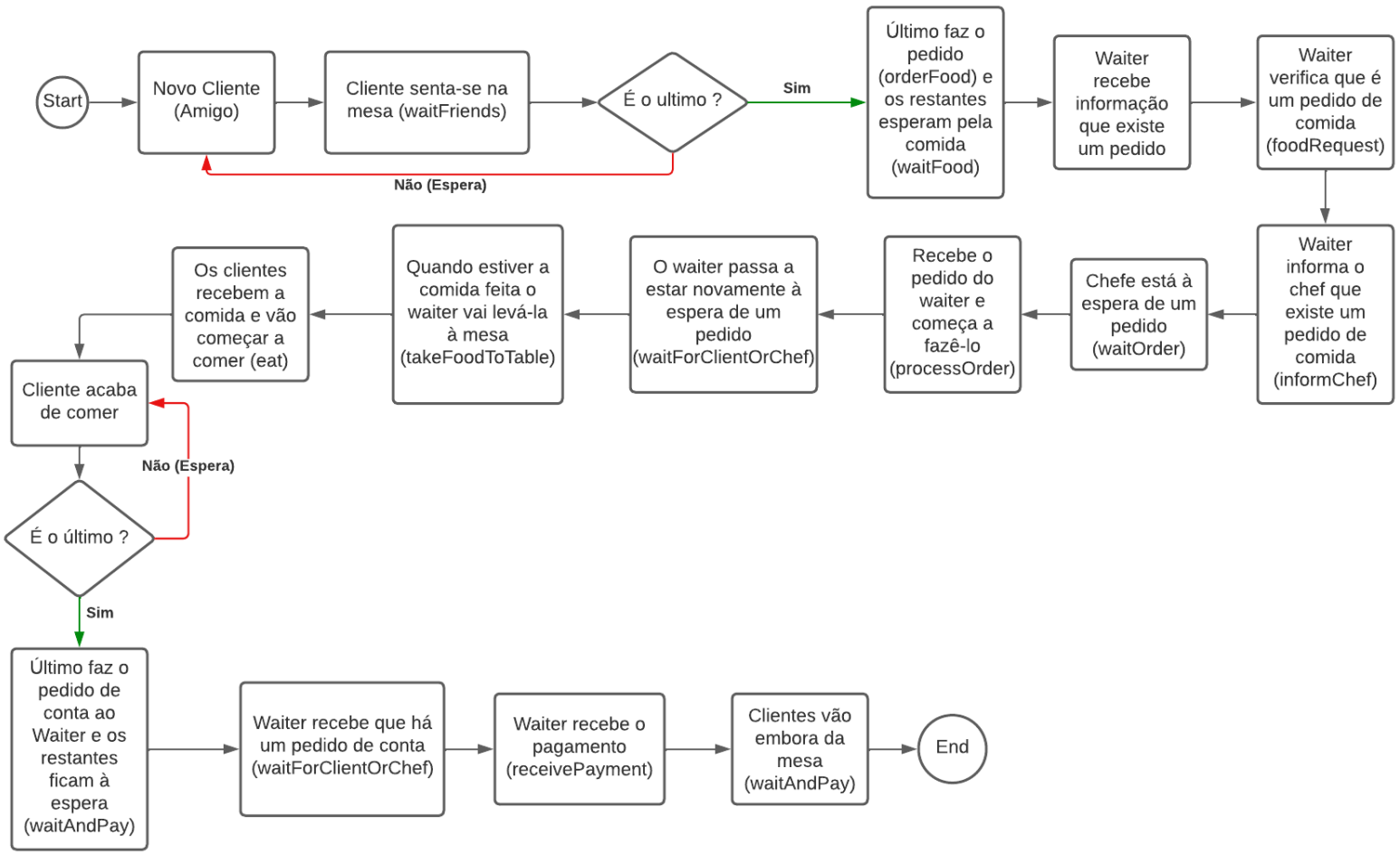
Entrando na região crítica, o *waiter* altera o seu estado para RECEIVE_PAYMENT, indicando que vai receber o pagamento, e imprime no terminal. De seguida, o *waiter* acorda o cliente que realizou o pedido, ao realizar um semUp ao semáforo *requestReceived,* informando-o de que recebeu o pedido, e sai da região crítica.

Função *waitAndPay*() - Cliente

```c
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 */
sh->fSt.st.clientStat[id] = FINISHED;//Altera o estado do cliente para FINISHED(8) pois vai abandonar o restaurante

saveState(nFic, &(sh->fSt)); //Printa linha no prompt com os estados atualizados

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

Enquanto o último cliente que chegou ao restaurante pede a conta, os restantes clientes vão entrando um por um na região crítica, alterando o seu estado para FINISHED, que indica que estão prontos para ir embora, e imprime o mesmo no terminal. Por fim, depois de efetuar o pagamento do jantar, o último cliente também passa o seu estado para FINISHED.

Fluxograma

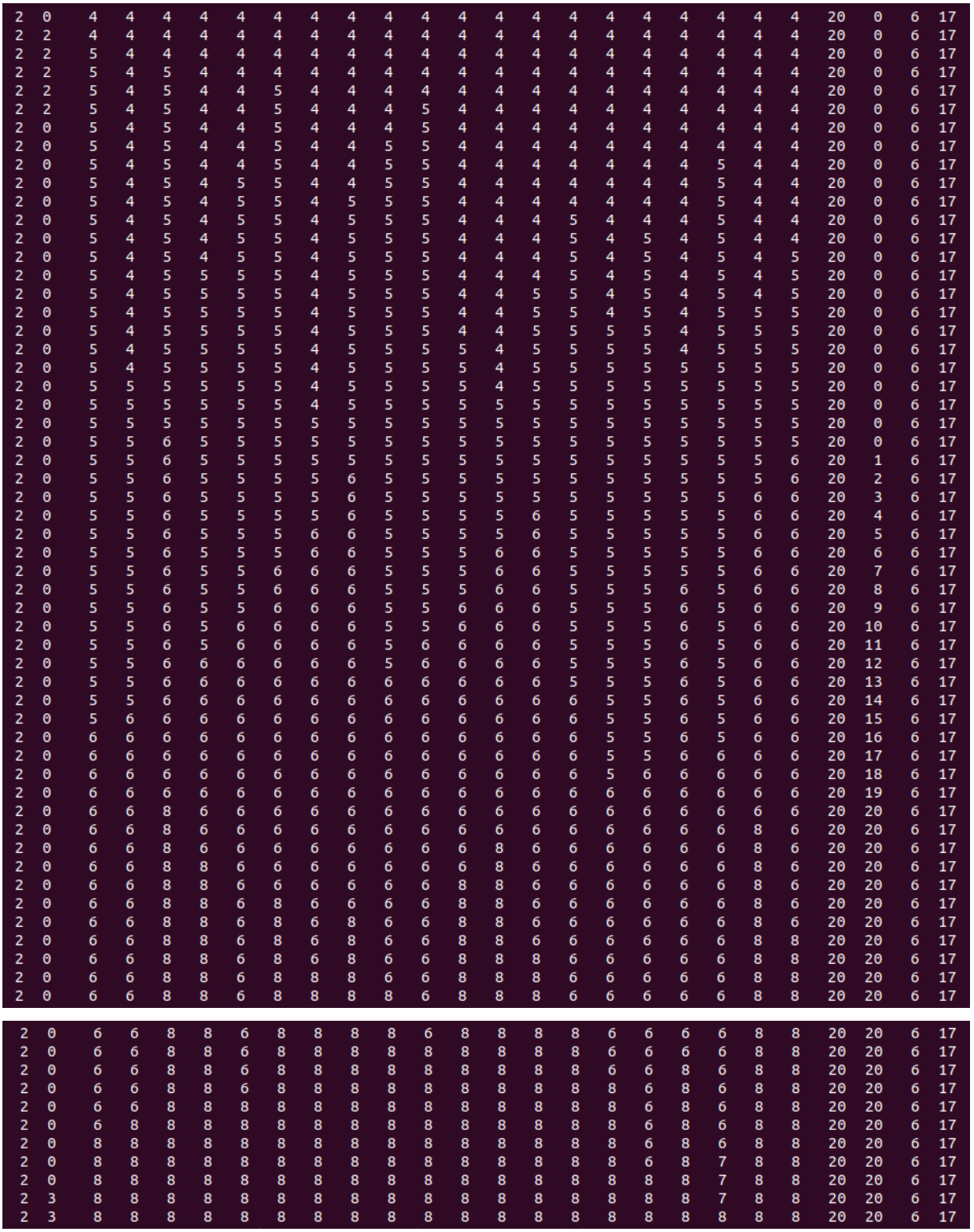

Start
Novo Cliente (Amigo)
Cliente senta-se na mesa (waitFriends)
É o ultimo ?
Sim
Não (Espera)
Último faz o pedido (orderFood) e os restantes esperam pela comida (waitFood)
Waiter recebe informação que existe um pedido
Waiter verifica que é um pedido de comida (foodRequest)
Waiter informa o chef que existe um pedido de comida (informChef)
Chefe está à espera de um pedido (waitOrder)
Recebe o pedido do waiter e começa a fazê-lo (processOrder)
O waiter passa a estar novamente à espera de um pedido (waitForClientOrChef)
Quando estiver a comida feita o waiter vai levá-la à mesa (takeFoodToTable)
Os clientes recebem a comida e vão começar a comer (eat)
Cliente acaba de comer
Não (Espera)
É o último ?
Sim
Último faz o pedido de conta ao Waiter e os restantes ficam à espera (waitAndPay)
Waiter recebe que há um pedido de conta (waitForClientOrChef)
Waiter recebe o pagamento (receivePayment)
Clientes vão embora da mesa (waitAndPay)
End

3- Resultados

Ao longo da elaboração do trabalho, fomos comparando com o pré-compilado fornecido pelo professor, e chegámos a um resultado semelhante ao que era o resultado do professor.

Por fim, deixamos a correr o *run.sh* que simula o jantar de amigos 1000 vezes, para assegurar de que não haveria deadlocks na nossa implementação.

```
Run n.º 1000
                                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   1   1   1   1   2   1   1   1    3   0   6  -1
  0  0    1   1   1   1   1   2   2   1   1   2   1   1   1   1   1   1   2   1   1   1    4   0   6  -1
  0  0    1   1   1   1   1   2   2   1   1   2   1   1   2   1   1   1   2   1   1   1    5   0   6  -1
  0  0    1   1   1   1   1   2   2   1   1   2   1   1   2   1   1   1   2   1   1   2    6   0   6  -1
  0  0    1   1   1   1   1   2   2   1   1   2   1   1   2   1   1   2   2   1   1   2    7   0   6  -1
  0  0    1   1   1   1   1   2   2   1   1   2   1   1   2   1   1   2   2   1   2   2    8   0   6  -1
  0  0    2   1   1   1   1   2   2   1   1   2   1   1   2   1   1   2   2   1   2   2    9   0   6  -1
  0  0    2   1   1   1   1   2   2   2   1   2   1   1   2   1   1   2   2   1   2   2   10   0   6  -1
  0  0    2   1   1   1   1   2   2   2   1   2   1   1   2   1   2   2   2   1   2   2   11   0   6  -1
  0  0    2   1   1   1   1   2   2   2   1   2   2   1   2   1   2   2   2   1   2   2   12   0   6  -1
  0  0    2   1   1   1   1   2   2   2   1   2   2   1   2   2   2   2   2   1   2   2   13   0   6  -1
  0  0    2   1   2   1   1   2   2   2   1   2   2   1   2   2   2   2   2   1   2   2   14   0   6  -1
  0  0    2   2   2   1   1   2   2   2   1   2   2   1   2   2   2   2   2   1   2   2   15   0   6  -1
  0  0    2   2   2   1   1   2   2   2   2   2   2   1   2   2   2   2   2   1   2   2   16   0   6  -1
  0  0    2   2   2   1   2   2   2   2   2   2   2   1   2   2   2   2   2   1   2   2   17   0   6  -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   6  -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   6  -1
  0  0    2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   4   2   2   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   2   2   2   2   2   2   2   2   4   2   2   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   2   2   2   2   2   2   2   4   4   2   2   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   2   2   2   2   2   2   2   4   4   2   4   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   4   2   2   2   2   2   2   4   4   2   4   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   4   2   2   4   2   2   4   4   4   2   4   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   4   2   2   4   2   4   4   4   4   2   4   20   0   6  17
  0  0    2   2   2   2   2   2   3   2   2   4   4   2   4   2   4   4   4   4   2   4   20   0   6  17
  0  0    4   2   2   2   2   2   3   2   2   4   4   2   4   2   4   4   4   4   2   4   20   0   6  17
  0  0    4   2   2   2   2   4   3   2   2   4   4   2   4   2   4   4   4   4   2   4   20   0   6  17
  0  0    4   2   2   2   2   4   3   2   2   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   2   4   2   2   4   3   2   2   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   2   4   2   2   4   3   2   2   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   2   4   2   4   4   3   2   2   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   2   4   2   4   4   3   4   2   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   2   4   2   4   4   3   4   4   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   4   4   2   4   4   3   4   4   4   4   2   4   2   4   4   4   4   4   4   20   0   6  17
  0  0    4   4   4   2   4   4   3   4   4   4   4   2   4   4   4   4   4   4   4   4   20   0   6  17
  0  0    4   4   4   2   4   4   3   4   4   4   4   4   4   4   4   4   4   4   4   4   20   0   6  17
  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  17
  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  17
  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  17
  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  17
  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  17
```



A figura anterior é o resultado de um teste que implementa as funções anteriores. O CH representa os estados do *Chef*, o WT representa os estados do *Waiter* e os C's representam os estados dos 20 *Clients*. Nas colunas ATT (At Table) e FIE (Finished Eating) atualizam o número de pessoas que estão à mesa e que acabaram de comer, respetivamente. As colunas 1st (first) e las

(last) mostram os IDs do primeiro e do último cliente a chegar à mesa, respetivamente.

Como podemos observar na imagem, o *run.sh* conseguiu simular o jantar 1000 vezes, logo à partida não existem deadlocks. Fazendo uma análise ao código apresentando, também conseguimos observar que o mesmo satisfaz todas as condições referidas na **Introdução**.

4- Conclusão

Com a realização deste projeto conseguimos aprender o funcionamento de memória partilhada bem como a utilização de semáforos de forma a controlarmos a ordem de execução de processos.

Estas são ferramentas muito úteis e muito importantes para o enriquecimento do conhecimento em sistemas operativos.

5- Webgrafia

Fluxograma : https://www.lucidchart.com/pages/pt/landing?utm_source=google&utm_medium=cpc&utm_campaign=_chart_pt_allcountries_mixed_search_brand_exact_&km_CPC_CampaignId=1500131167&km_CPC_AdGroupID=59412157138&km_CPC_Keyword=lucidchart&km_CPC_MatchType=e&km_CPC_ExtensionID=&km_CPC_Network=g&km_CPC_AdPosition=&km_CPC_Creative=294337318298&km_CPC_TargetID=kwd-33511936169&km_CPC_Country=20873&km_CPC_Device=c&km_CPC_placement=&km_CPC_target=&gclid=CjwKCAiAqt-dBhBcEiwATw-ggNKzRCh-BSdSYH8YzbeL4x32beHw1vCxwcEtZLPOzLFQ-4buOx8-5RoCQ3QQAvD_BwE

Semaphores in C : https://greenteapress.com/thinkos/html/thinkos013.html

Language C manual : https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html