UNIVERSIDADE FEDERAL DE RORAIMA
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Sistemas operacionais

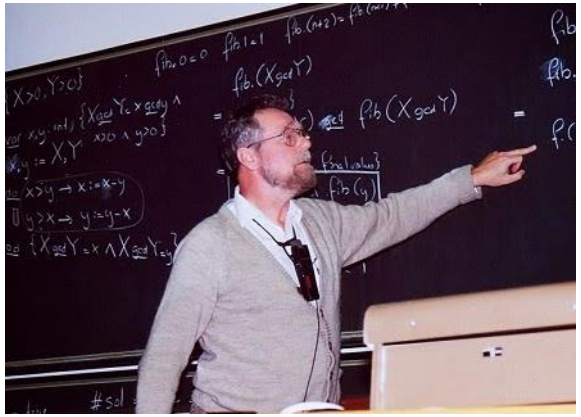
Jantar dos Filósofos

Jorge Siqueira Serrão

William Thiago Almeida Silva

1. A origem do problema:

O Jantar dos filósofos foi proposto por Dijkstra em 1965 como um problema de sincronização. A partir de então todos os algoritmos propostos como soluções de sincronização acabaram sendo relacionados ou testados contra o problema do Jantar dos filósofos.



Edsger Wybe Dijkstra

Jantar entre filósofos

2. O problema:

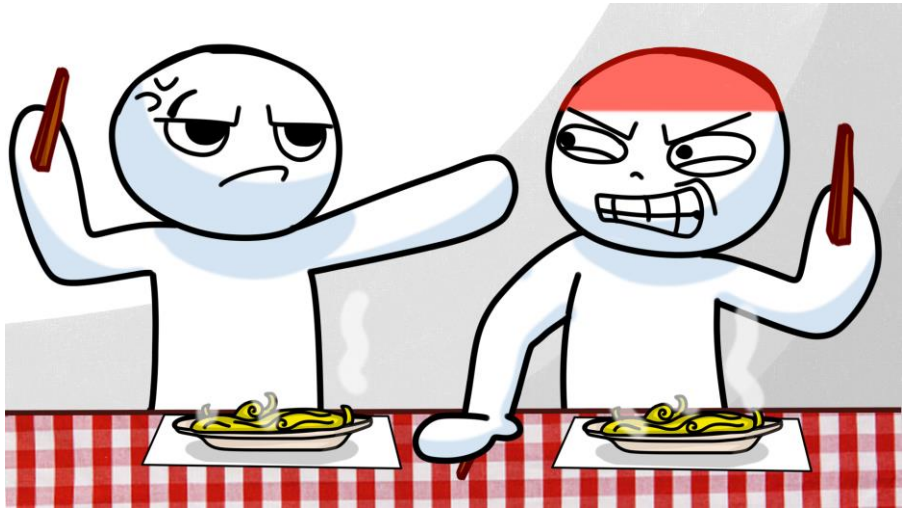
1) Cinco filósofos estão sentados ao redor de uma mesa circular para o jantar.
2) Cada filósofo possui um prato para comer macarrão.
3) Os filósofos dispõem de hashis e e cada um precisa de 2 hashis para comer.
4) Entre cada par de pratos existe apenas um hashi: Hashis precisam ser compartilhados de forma sincronizada.
5) Os filósofos comem e pensam, alternadamente. Eles não se atém a apenas uma das tarefas.
6) Além disso, quando comem, pegam apenas um hashi por vez: Se conseguir pegar os dois come por alguns instantes e depois larga os hashis.

Você é capaz de propor um algoritmo que implemente cada filósofo de modo que ele execute as tarefas de COMER e PENSAR sem nunca ficar travado?

3. Entendendo o problema:

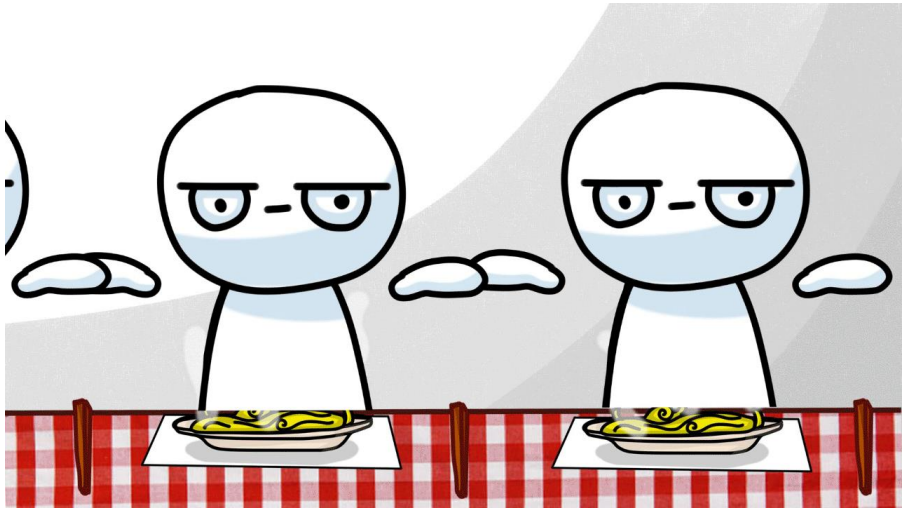
- O que devemos evitar?

Deadlock



cada entidade pertencente ao conjunto está bloqueada esperando por um evento (ou recurso) que somente outra entidade no mesmo conjunto pode gerar (ou liberar)

Starvation

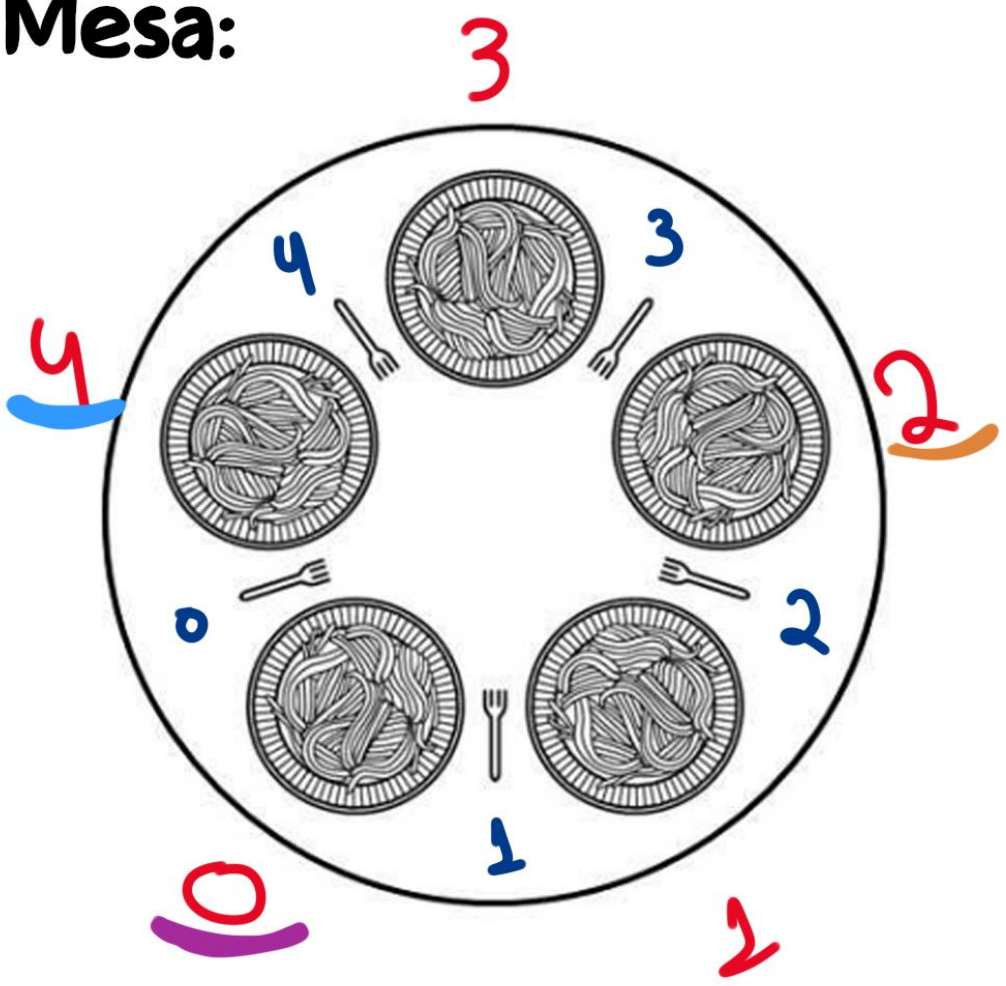


quando ocorrem negativas de acesso a um determinado recurso perpetuamente, impedindo que execute o restante das tarefas, sem que a entidade esteja bloqueada.

4. O algoritmo (Solução 1):

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Mesa:



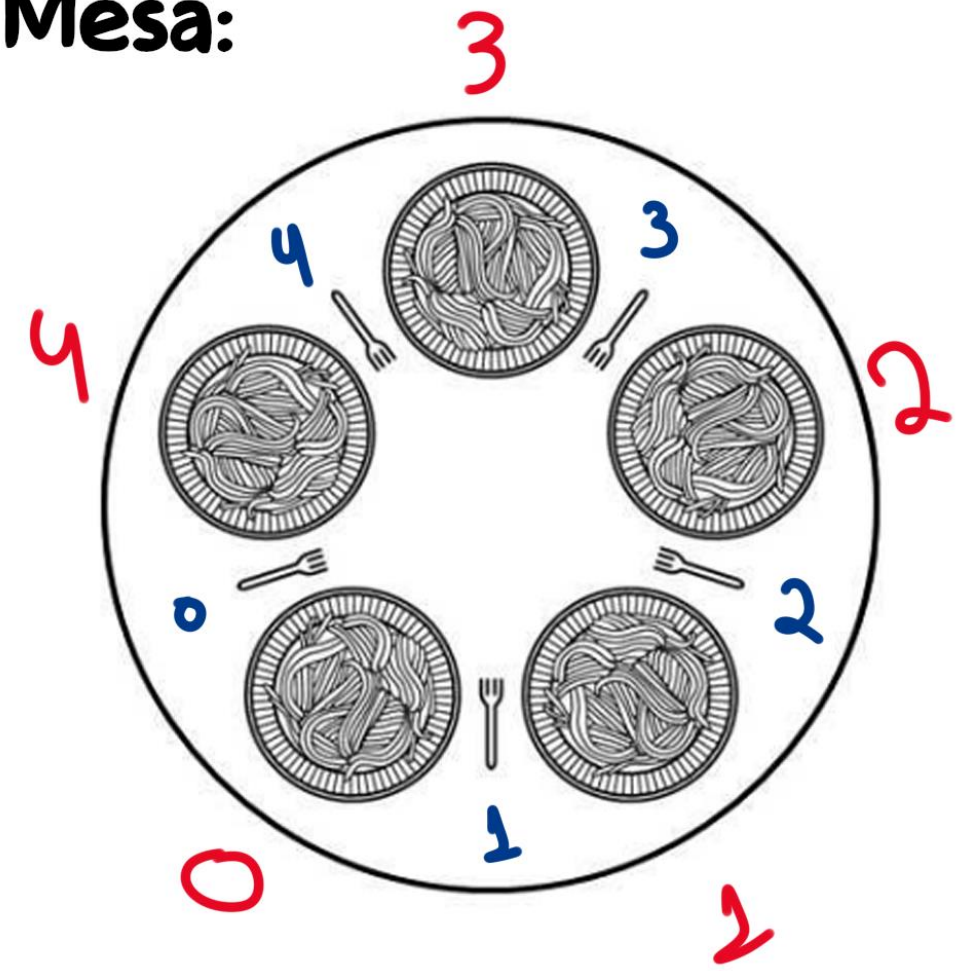

```
take_fork(i);
take_fork((i+1) % N);
```

Ex(2):
(i+1)%.N=(2+1)%.5=3//

Ex(4):
(4+1)%.5=5%.5=0//

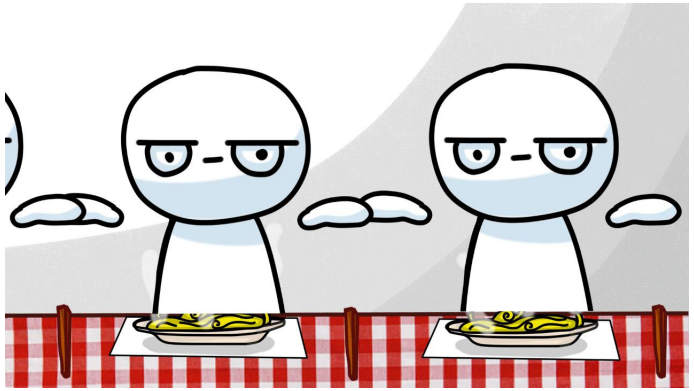
Ex(0):
(0+1)%.5=1%.5=1//

Mesa:



Problema:

take_fork(i);

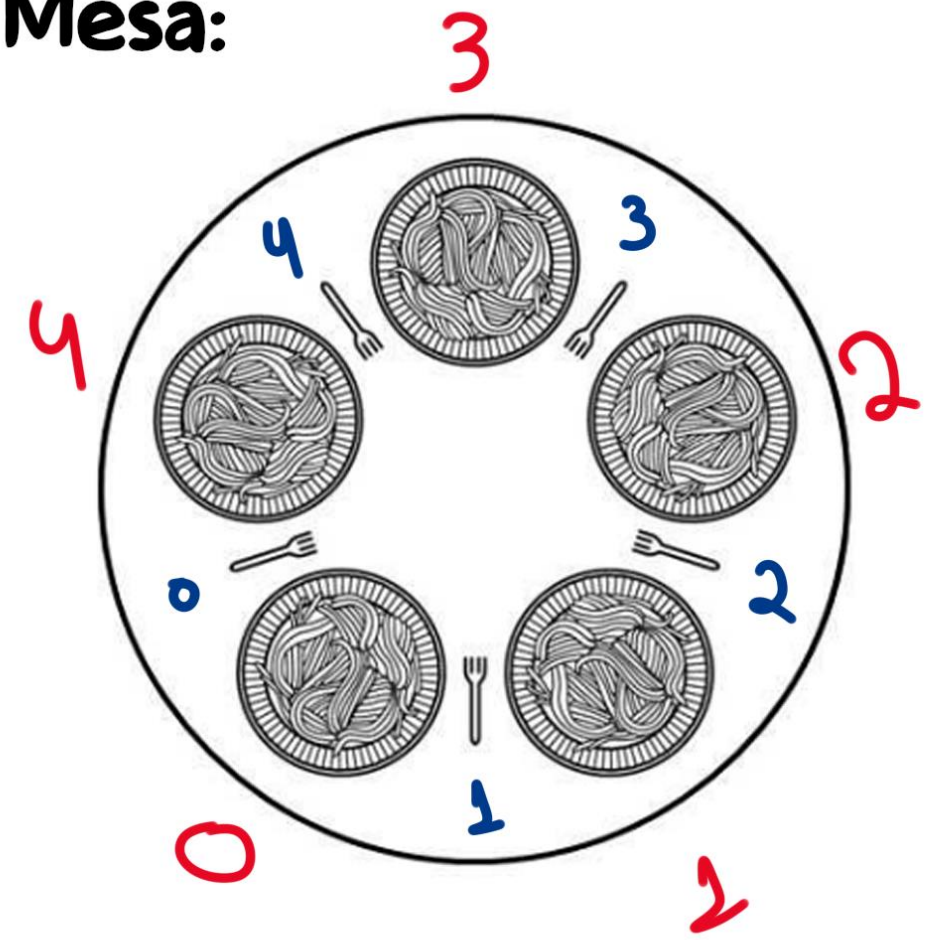


Se todos os filosofos pegarem o garfo da esquerda, nenhum pega o da direita.

4.2. O algoritmo (Solução 2 – uso de mutex):

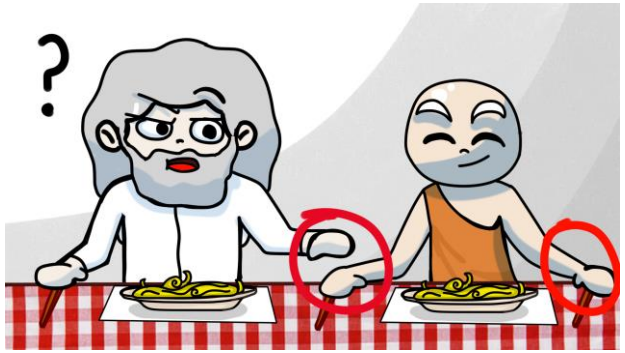
```
#define N 5                                     /* number of philosophers */

semaphore mutex = 1;
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        down(&mutex);
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
        up(&mutex);
```

Mesa:



PROBLEMA:

```
Down(&mutex);
Up(&mutex);
```



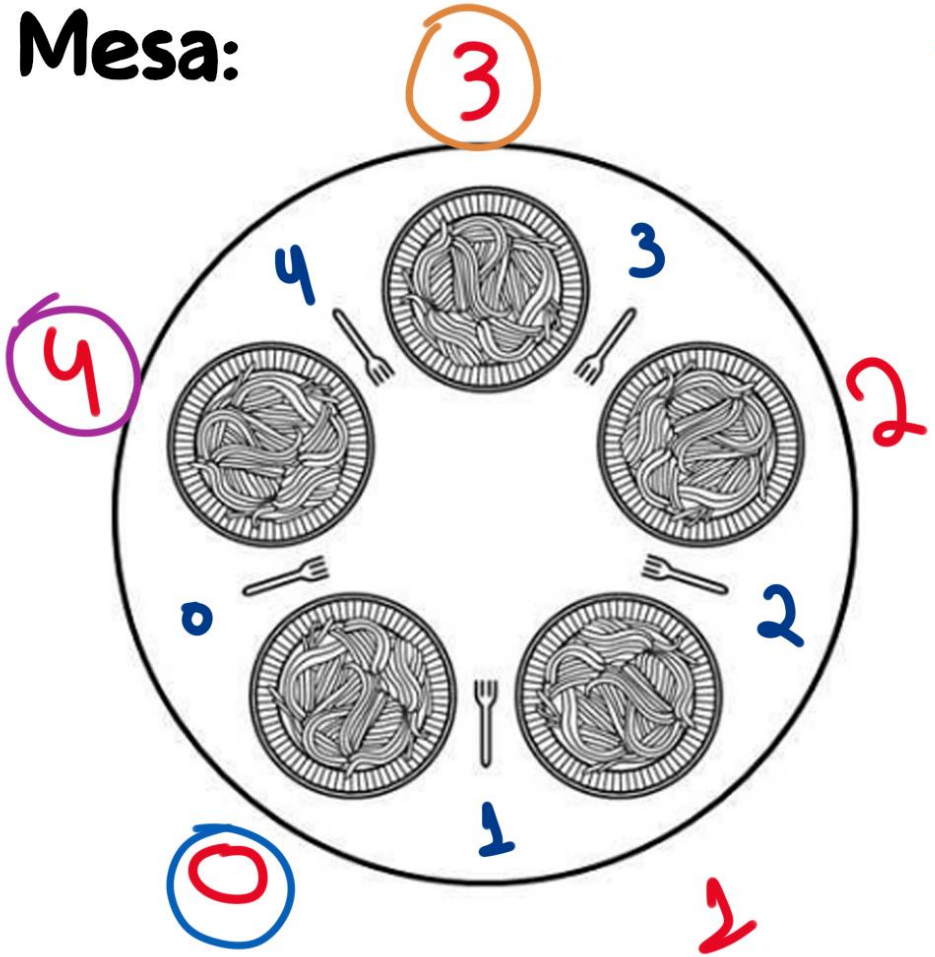
somente um filosofo pode comer de cada vez.

4.3. O algoritmo (Solução 3 – uso de semáforos):

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N–1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
```

```
void take_forks(int i)          /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)               /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                    /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Mesa:




```
#define LEFT    (i+N-1)%N
#define RIGHT   (i+1)%N
```

Ex(3):

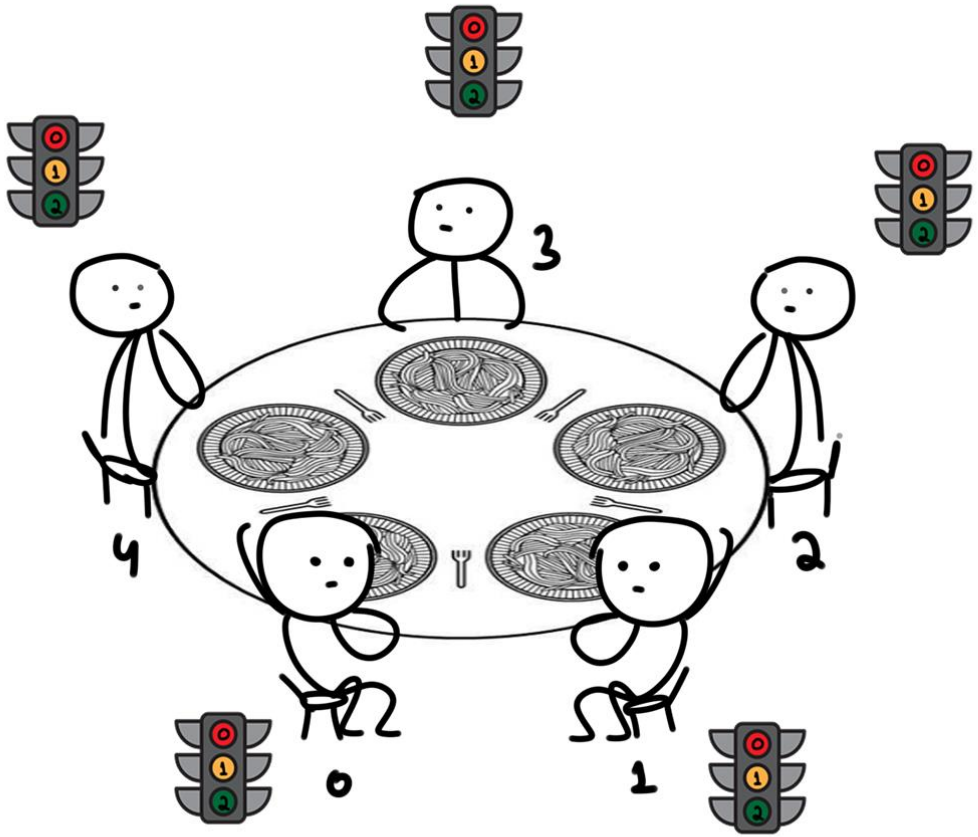
(i+N-1)%N = (3+5-1)%5 = 2,,

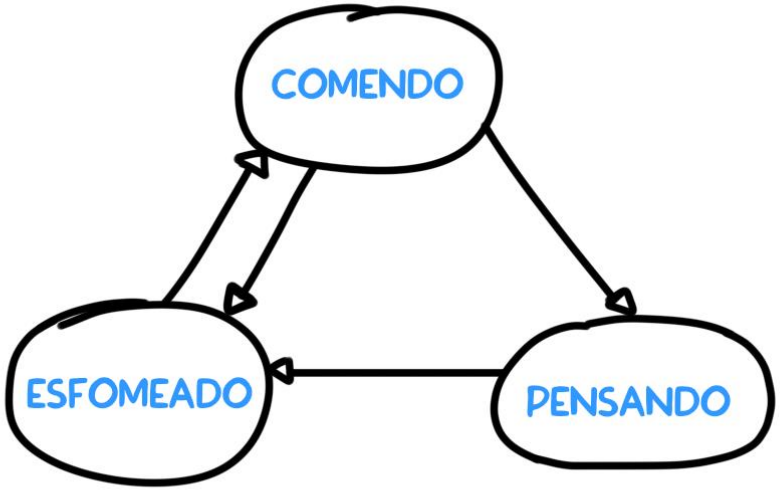
Ex.(0):

(0+5-1)%5 = 4%5 = 4,,

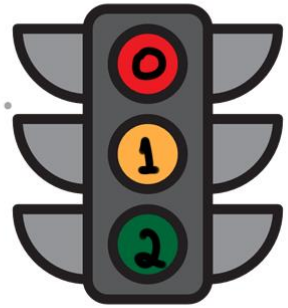
Ex.(4):

(4+5-1)%5 = 8%5 = 3,,





Semáforo



Pensando

Esfomeado

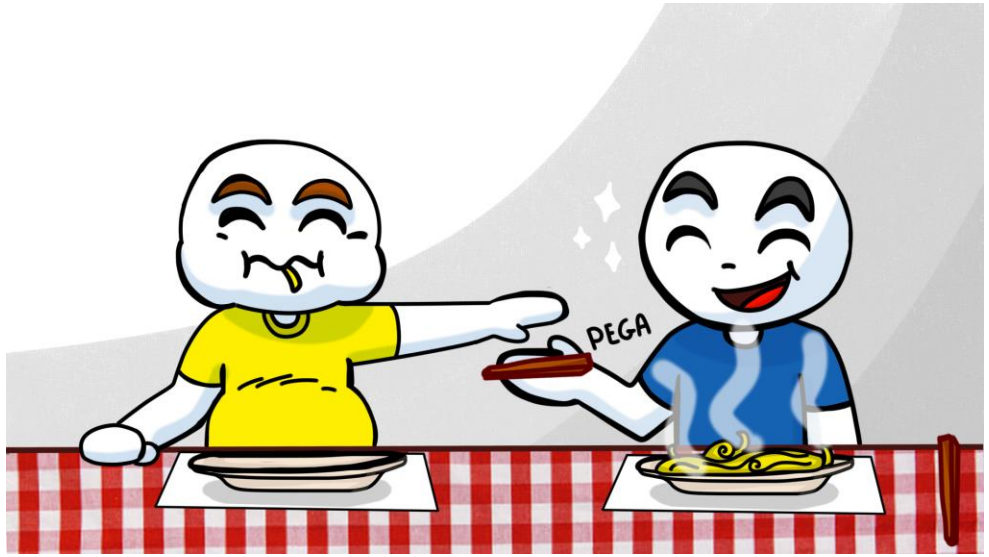
Comendo

main.c

```c
//-- INCLUDE
//--------------------------------------------------------------------
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
//-- CONSTANTES
//--------------------------------------------------------------------
#define QUANT      (5)                    //Quantidade de filósofos
#define ESQUERDA   (id_filosofo + QUANT - 1) % QUANT   //Id do filósofo a esquerda do id
#define DIREITA    (id_filosofo + 1) % QUANT
//Id do filósofo a direita do id
#define PENSANDO   (0)                    //Id para estado pensado
#define FAMINTO    (1)                    //Id para estado de fome
#define COMENDO    (2)                    //Id para estado comendo
//-- GLOBAL
//--------------------------------------------------------------------
int estado [QUANT];                       //Estado dos filósofos
pthread_mutex_t mutex;                    //Região crítica
```

```
clang-7 -pthread -lm -o main main.c
./main
Filosofo 1 foi criado com sucesso
Filosofo 4 pensa por 4 segundos
Filosofo 3 foi criado com sucesso
Filosofo 5 pensa por 7 segundos
Filosofo 2 foi criado com sucesso
Filosofo 5 pensa por 8 segundos
Filosofo 4 foi criado com sucesso
Filosofo 5 pensa por 6 segundos
Filosofo 5 foi criado com sucesso
Filosofo 5 pensa por 4 segundos
Filosofo 5 esta faminto
Filosofo 5 esta com intencao de comer
Filosofo 5 ganhou a vez de comer
Filosofo 5 esta faminto
Filosofo 5 esta com intencao de comer
Filosofo 5 ganhou a vez de comer
exited, segmentation fault
```

Exemplo de solução em C



Obrigado pela atenção!

Contato:

Jorge Siqueira: jorgefilhoz@hotmail.com

William Thiago: Williamthiagoalmeida@gmail.com