QXD0013 - Sistemas Operacionais

Problemas Clássicos de IPC + Introdução a Escalonamento

Thiago Werlley Bandeira da Silva[1]

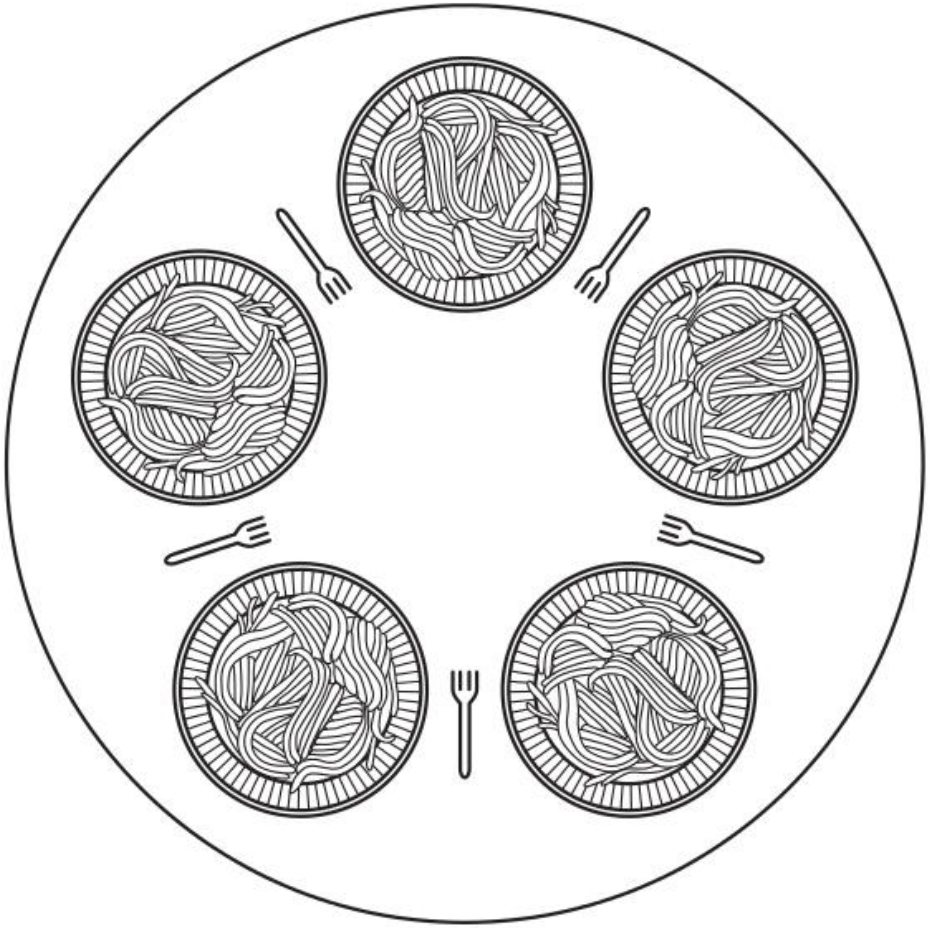
[1]Universidade Federal do Ceará, Brazil

22/11/2021

Problemas Clássicos de IPC - Jantar dos Filósofos

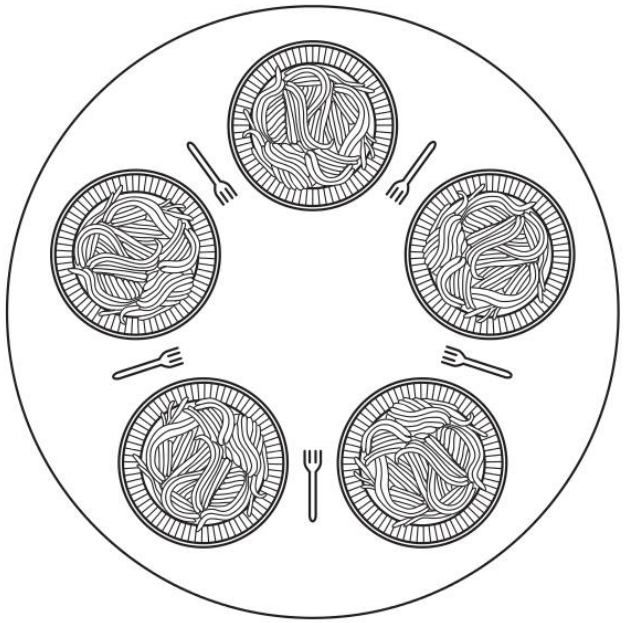
- Formulado por Dijkstra em 1965
- Problema clássico de sincronização
- Benchmark para avaliação de novas primitivas
- Descrição:
  - Cinco filósofos sentados em uma mesa circular
  - Cada um tem um prato de espaguete
  - Para comer, precisa-se de dois garfos
  - Um garfo entre cada par de pratos
  - Alternância de períodos: comer e pensar

Problemas Clássicos de IPC - Jantar dos Filósofos



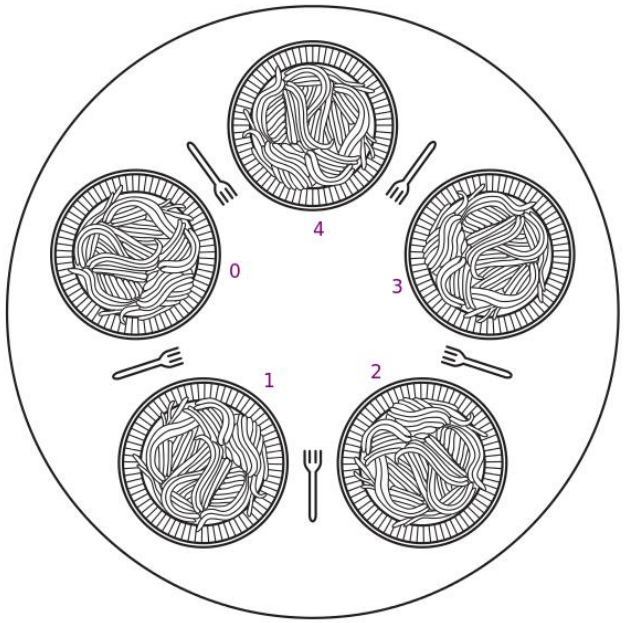
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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 */
        }
}
```



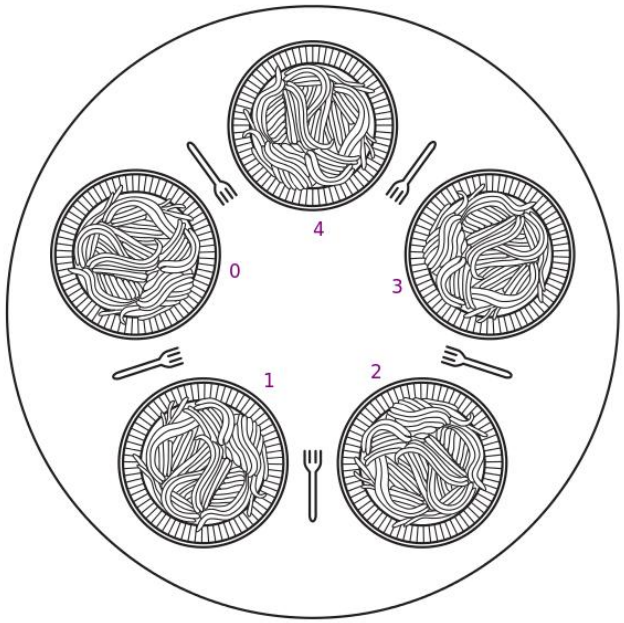
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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 */
        }
}
```



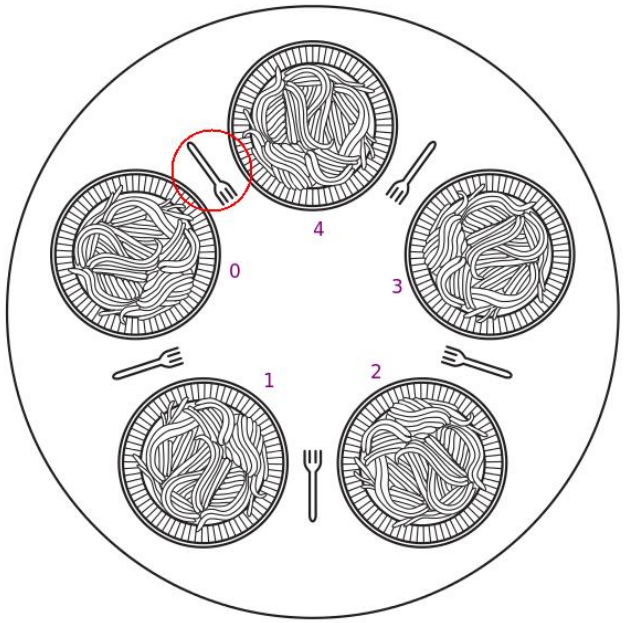
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=0

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 */
        }
}
```



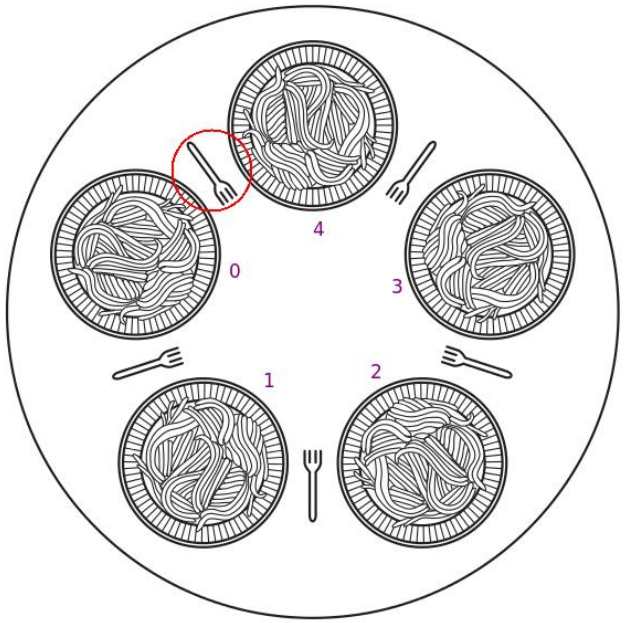
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=0
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 */
        }
}
```



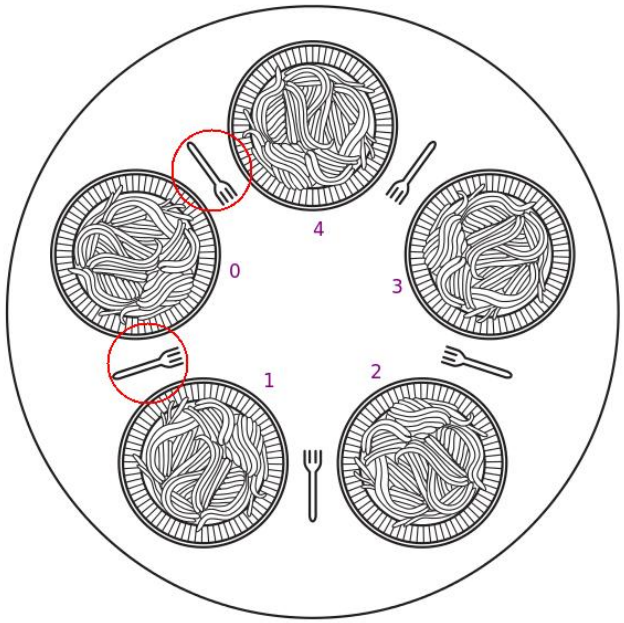
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=1
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 */
        }
}
```



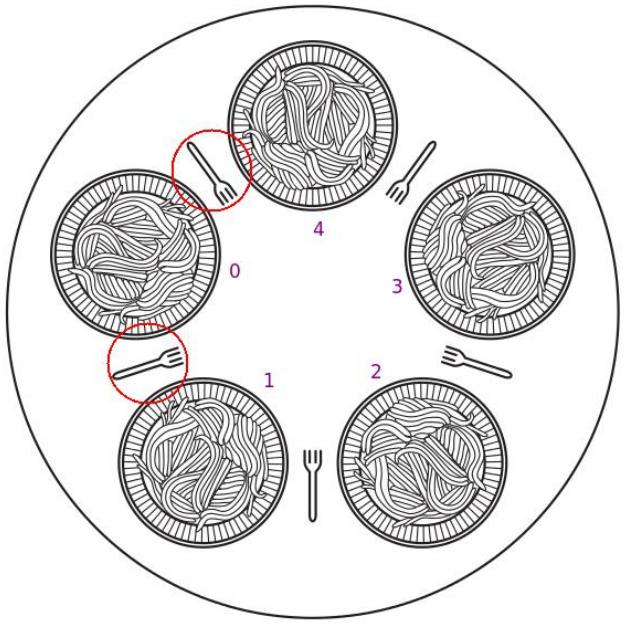
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=1
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 */
        }
}
```



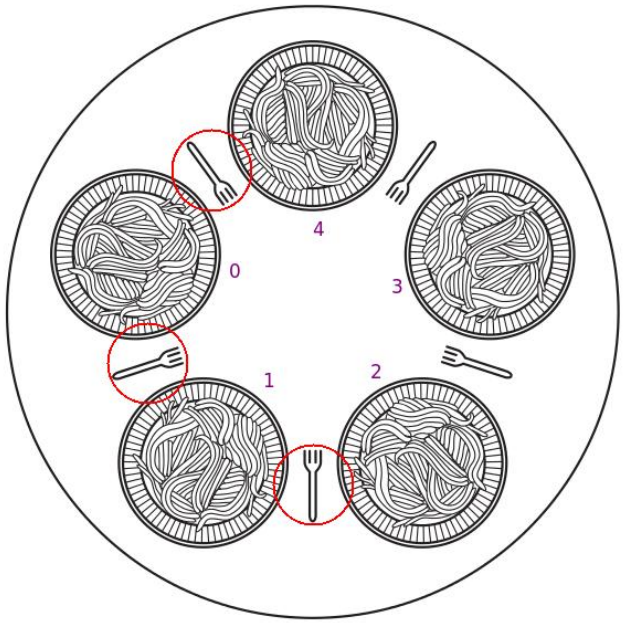
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=2
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 */
        }
}
```



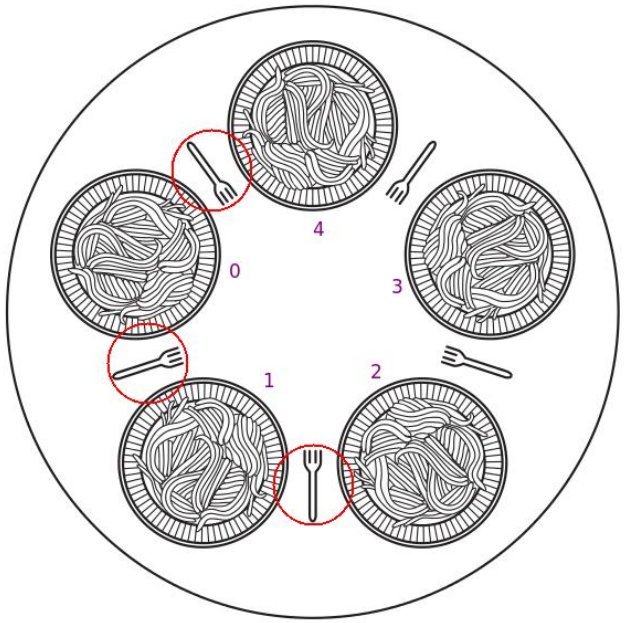
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=2
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 */
        }
}
```



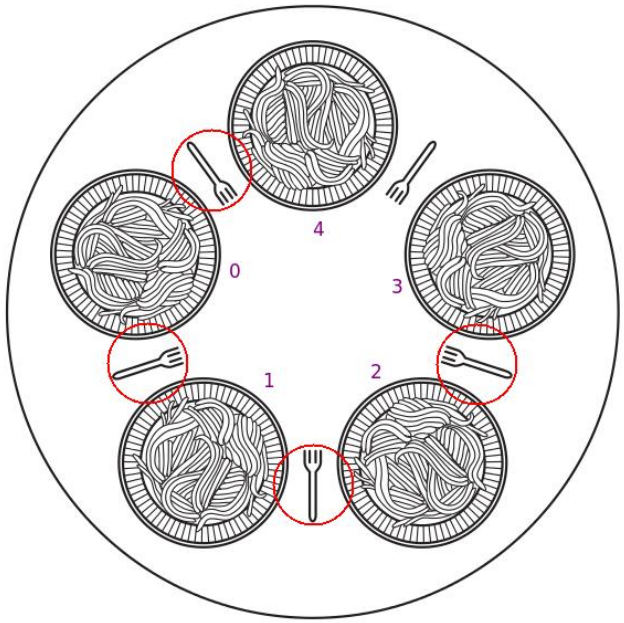
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=3
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 */
        }
}
```



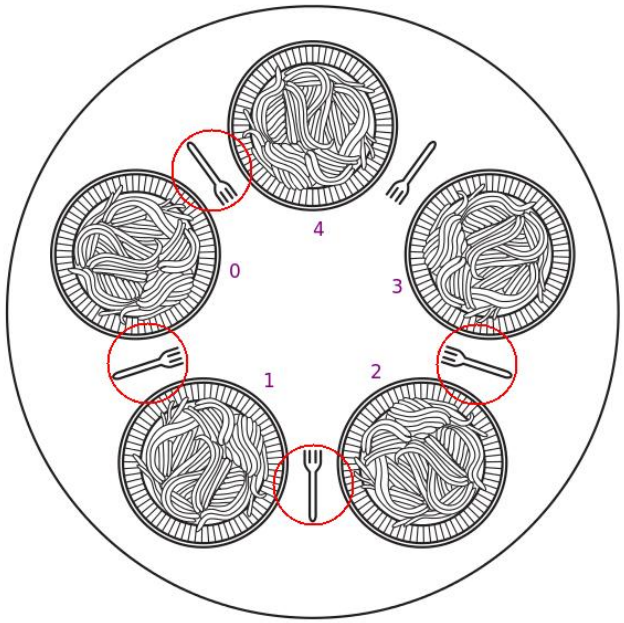
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=3
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 */
        }
}
```



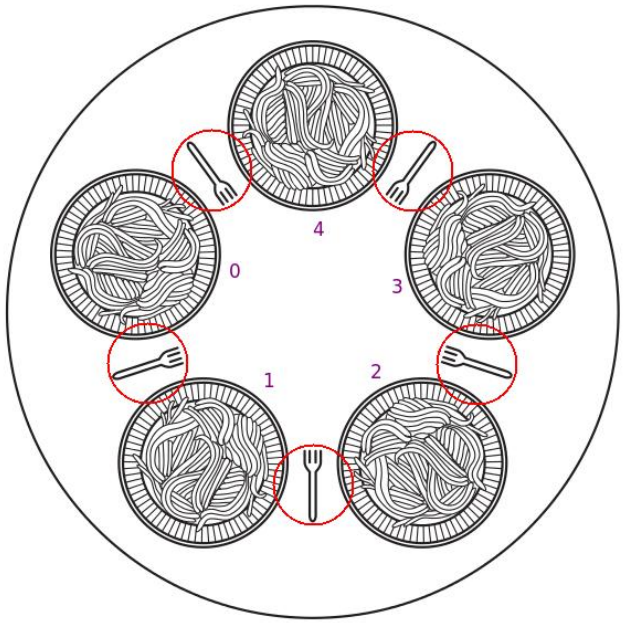
Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=4
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 */
        }
}
```



Problemas Clássicos de IPC - Jantar dos Filósofos

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

                                        i=4
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 */
        }
}
```

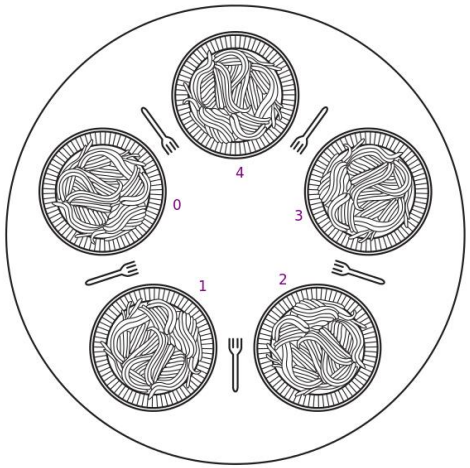


Problemas Clássicos de IPC - Jantar dos Filósofos

- Primeira alternativa
  - Pegar o garfo da esquerda
  - Verificar se o da direita está livre
  - Caso ocupado, liberar garfo da esquerda e repetir após um tempo
  - Todos ocupando o garfo esquerdo simultaneamente
    - Inanição (*starvation*)
- Segunda alternativa
  - Similar à anterior mas utilizando um tempo aleatório entre tentativas
    - Ethernet $\rightarrow$ Colisão de pacotes
  - Falta previsibilidade
- Terceira alternativa
  - Proteger o acesso ao primeiro garfo com um mutex
  - Permite apenas um filósofo comer por vez
  - Desempenho pior

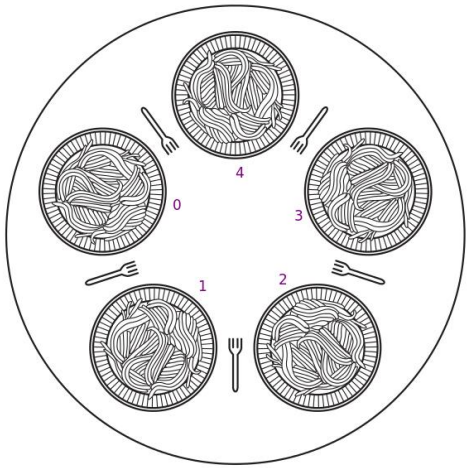
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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 */
```



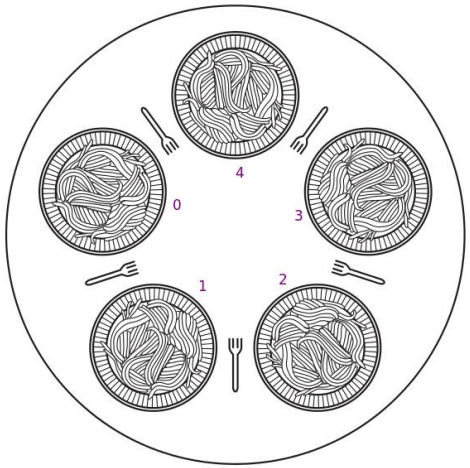
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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 */
    }
}
```



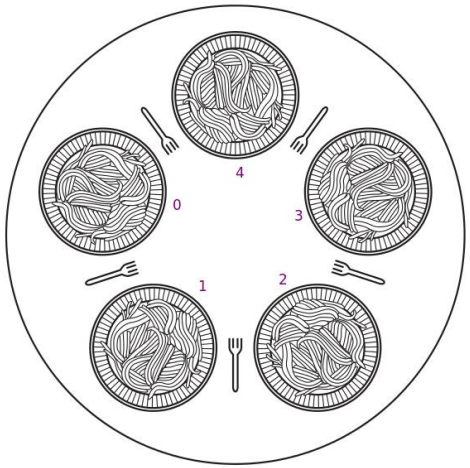
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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 */
}
```



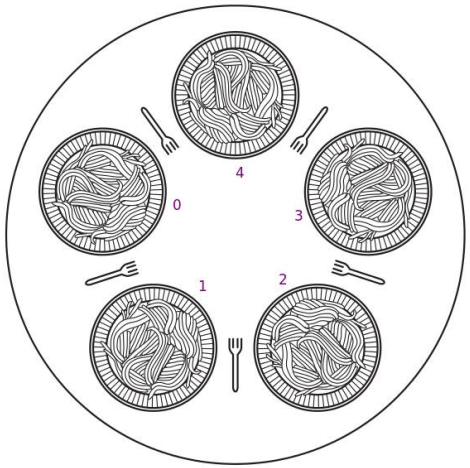
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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 */
}
```



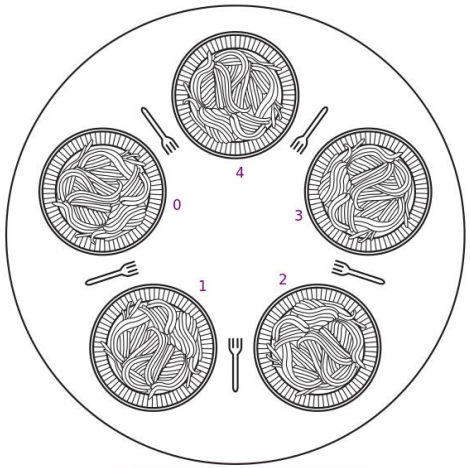
Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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]);
    }
}
```



Problemas Clássicos de IPC - Jantar dos Filósofos

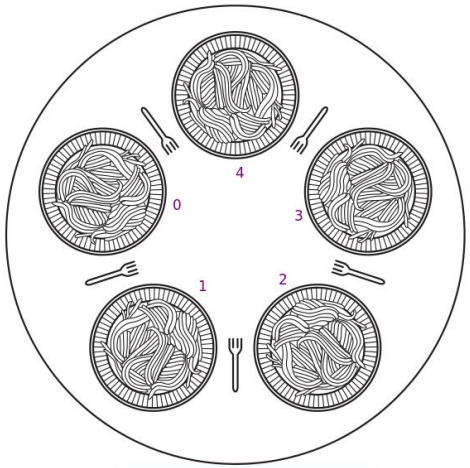
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=0
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=THINKING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

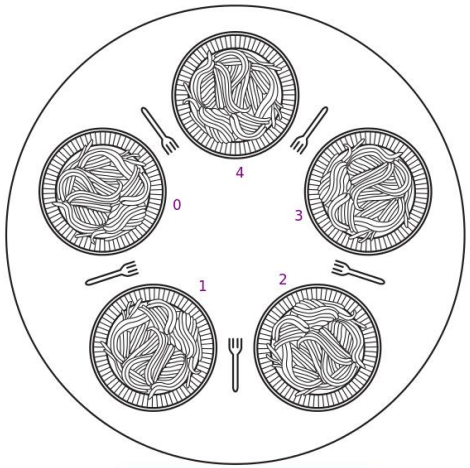
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=0
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=THINKING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

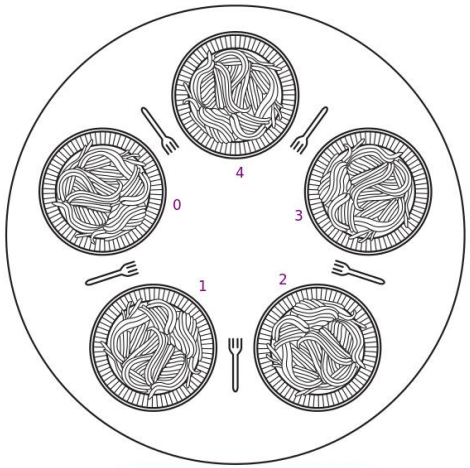
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=0
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=HUNGRY
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

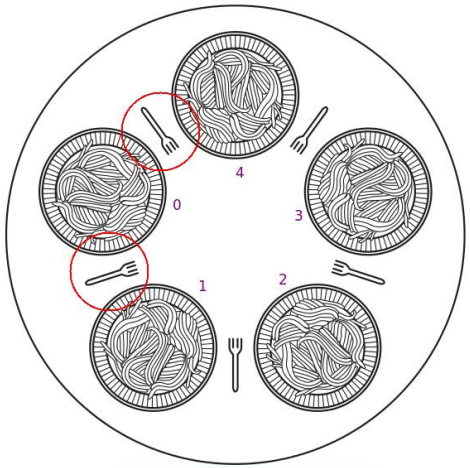
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=0
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=HUNGRY
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

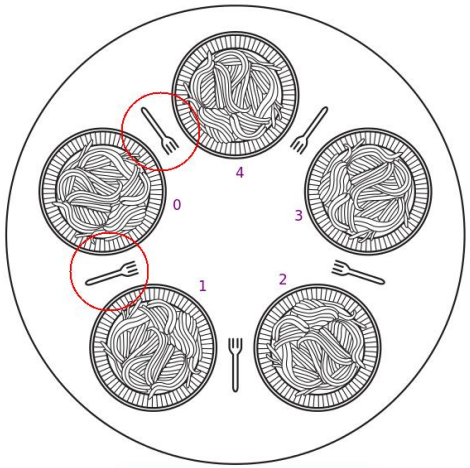
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=0
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

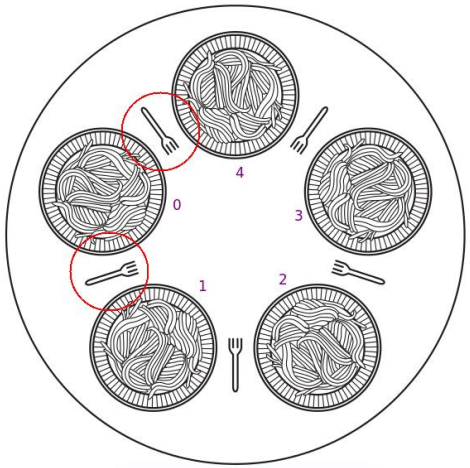
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=0
mutex=0
s[0]=1
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

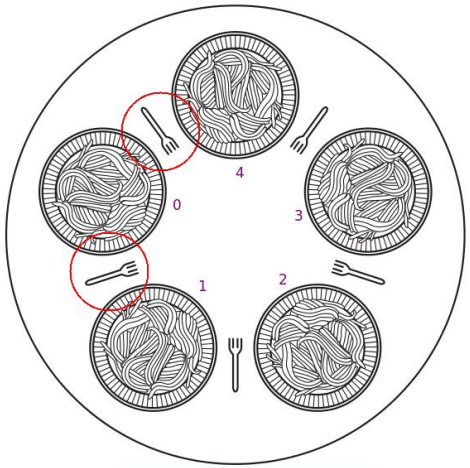
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=0
mutex=1
s[0]=1
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

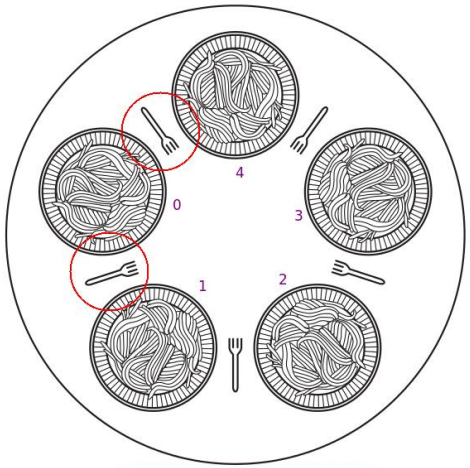
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=0
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

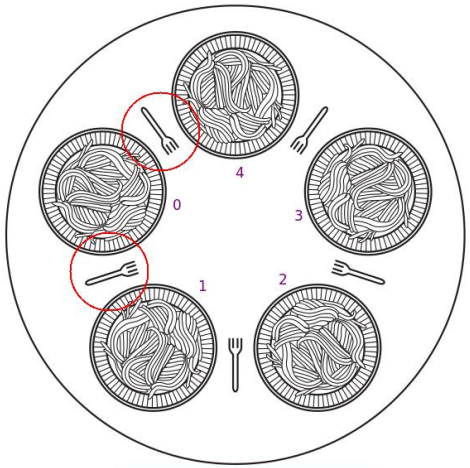
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=3
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

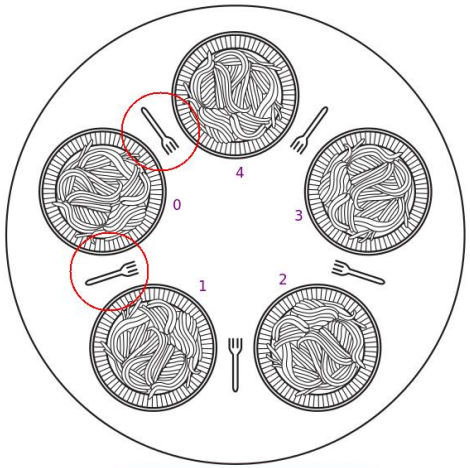
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

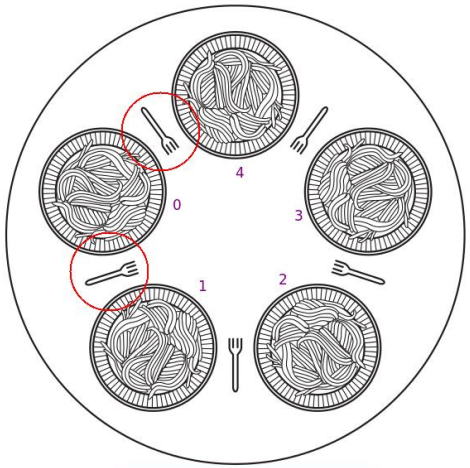
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=HUNGRY
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

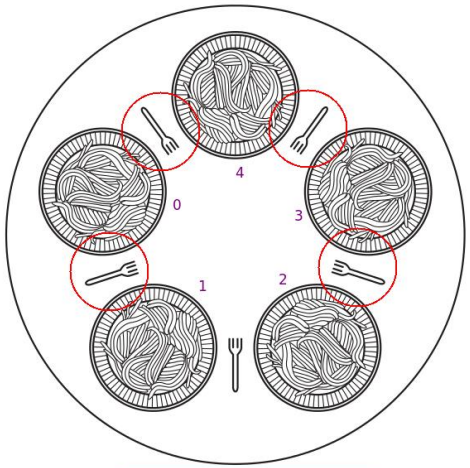
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=HUNGRY
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

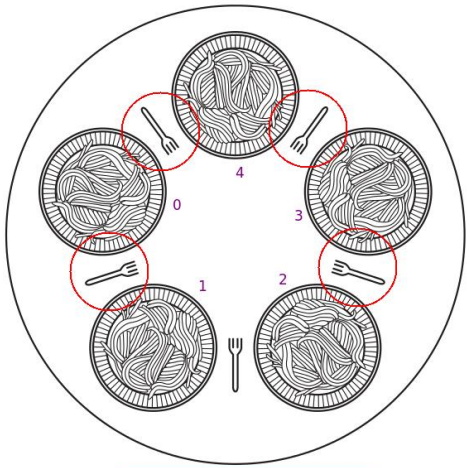
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

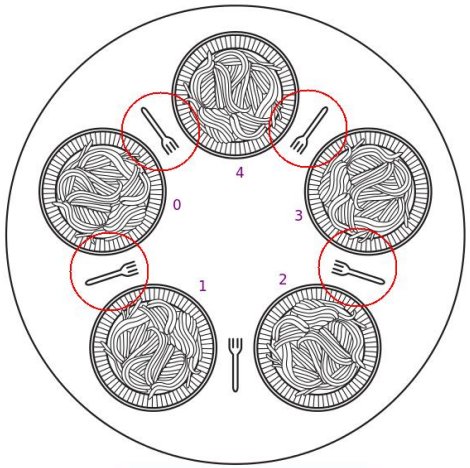
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=1
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

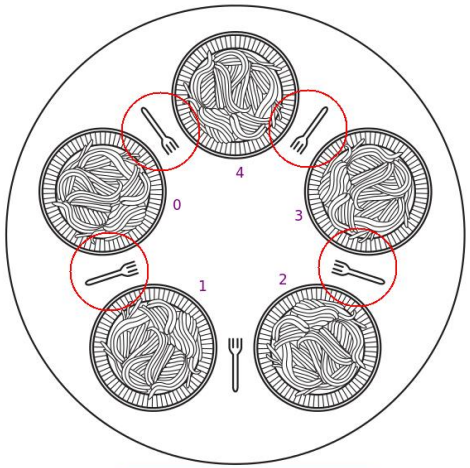
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=3
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=1
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

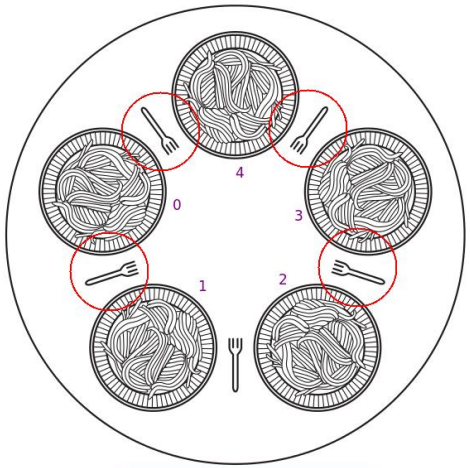
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=3
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

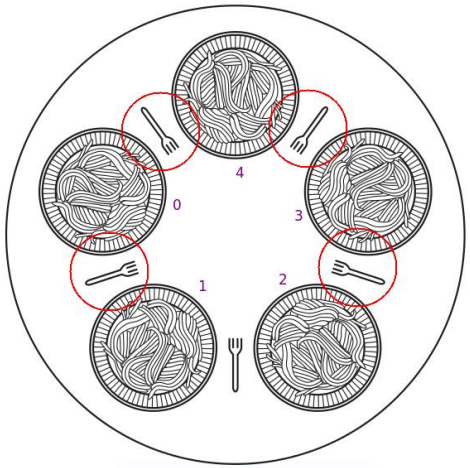
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=2
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

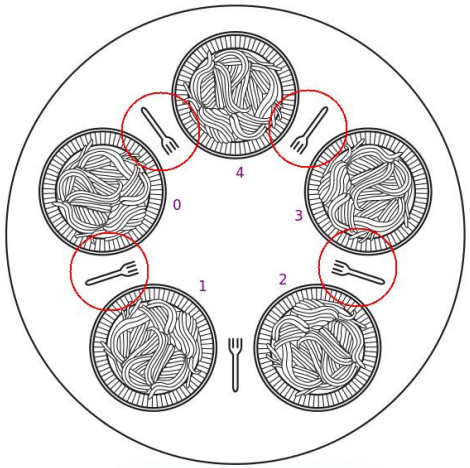
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=2
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=THINKING
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

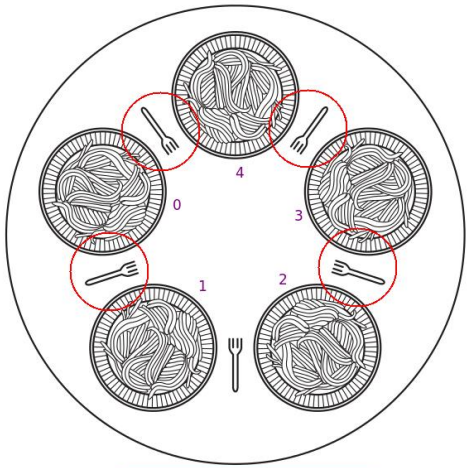
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=2
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

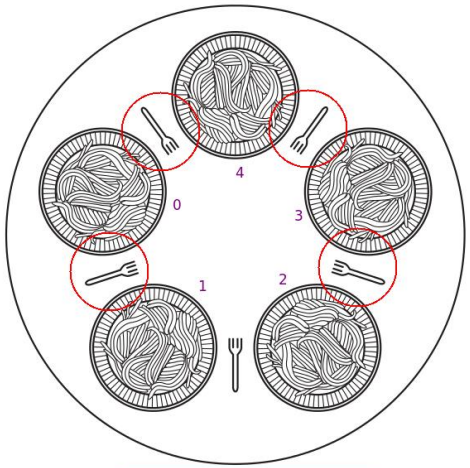
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=2
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

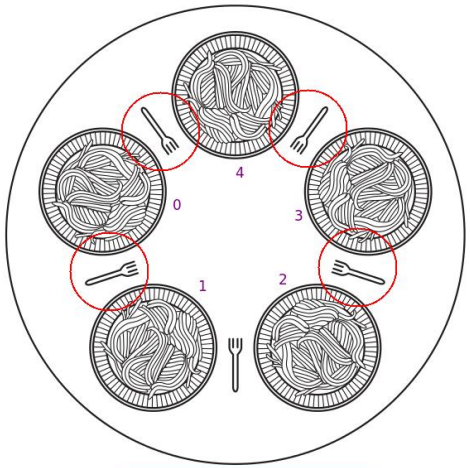
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=2
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

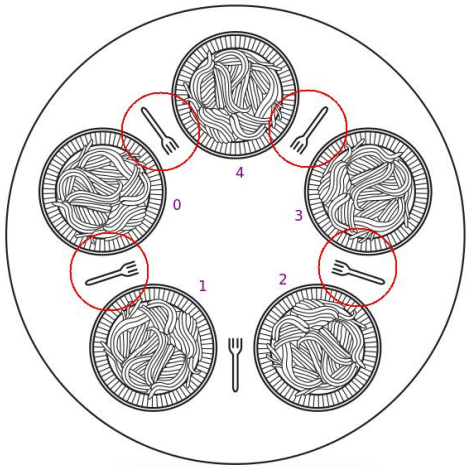
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      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]);
    }
}
```




```
i=2
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

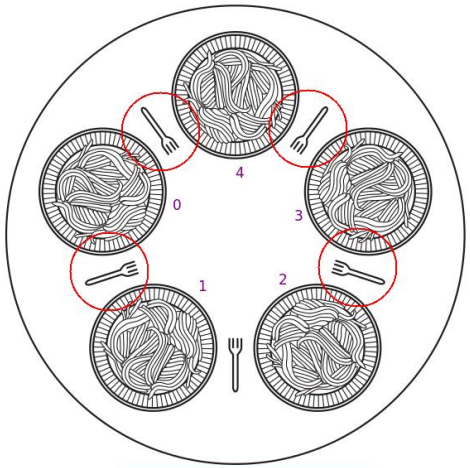
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```



i=3
mutex=1
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=EATING
state[4]=THINKING

Problemas Clássicos de IPC - Jantar dos Filósofos

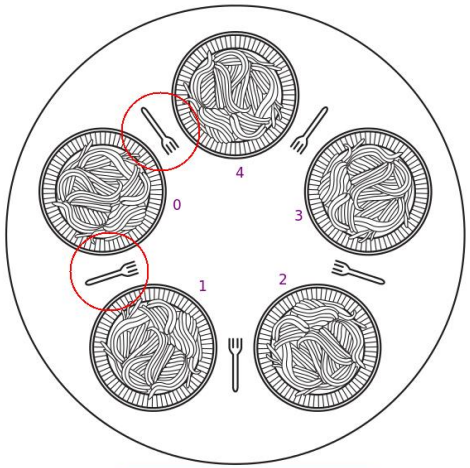
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=EATING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

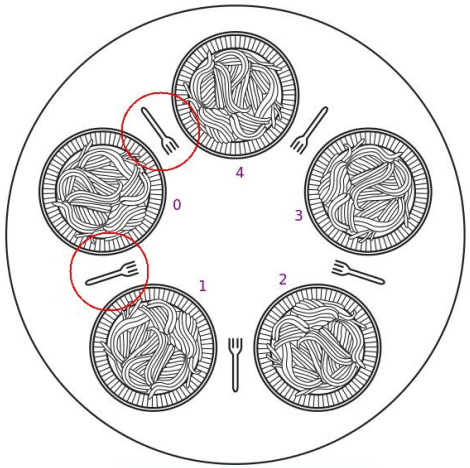
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

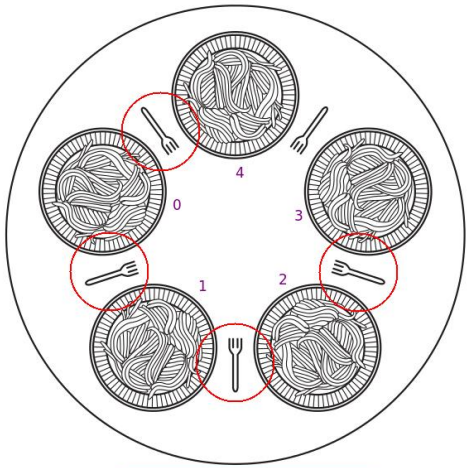
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=HUNGRY
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

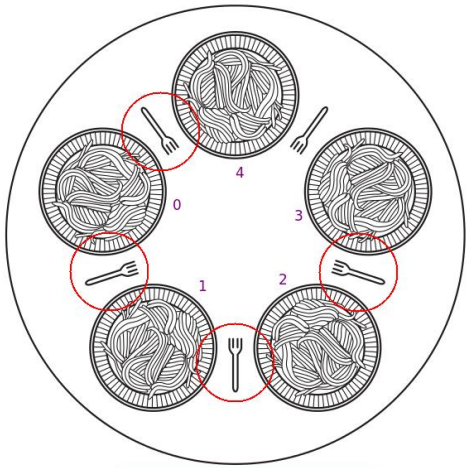
```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher 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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=EATING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Jantar dos Filósofos

```
#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]);
    }
}
```




```
i=3
mutex=0
s[0]=0
s[1]=0
s[2]=0
s[3]=0
s[4]=0
state[0]=EATING
state[1]=THINKING
state[2]=EATING
state[3]=THINKING
state[4]=THINKING
```

Problemas Clássicos de IPC - Escritores e Leitores

- Elaborado por Curtois em 1971
- Modelagem de acesso a banco de dados
- Exemplo: Sistema de reserva de passagens
  - Aceitável múltiplos processos lendo
  - Apenas um processo escreve $\rightarrow$ Demais têm acesso bloqueado

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

```
rc=0
mutex=1
db=1
```

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=0
mutex=0
db=1

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=0
db=1

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=2
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=1
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=1
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=0
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

rc=0
mutex=0
db=1

Problemas Clássicos de IPC - Escritores e Leitores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

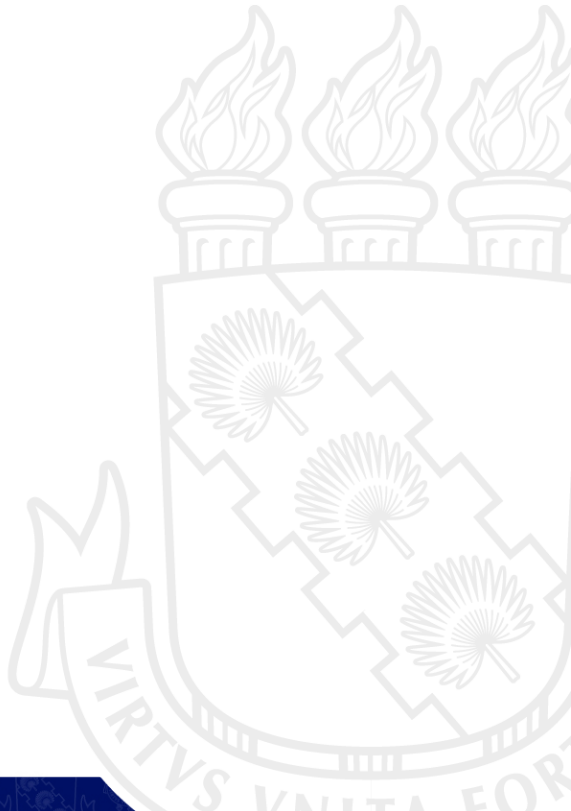
rc=0
mutex=0
db=0

Problemas Clássicos de IPC - Escritores e Leitores

- Admissão de leitores quase sempre permitida
- Admissão de escritores
  - Depende do fluxo de leitores
- Possível solução:
  - Fila de chegada

Problemas Clássicos de IPC - Montanha Russa

- Existem n passageiros que repetidamente:
  - Aguardam para entrar no carrinho
  - Fazem o passeio
  - Voltam a aguardar
- Montanha-russa tem somente um carrinho:
  - Com capacidade para C pessoas ($C < n$)
  - Somente inicia o percurso se estimver lotado



Problemas Clássicos de IPC - Montanha Russa

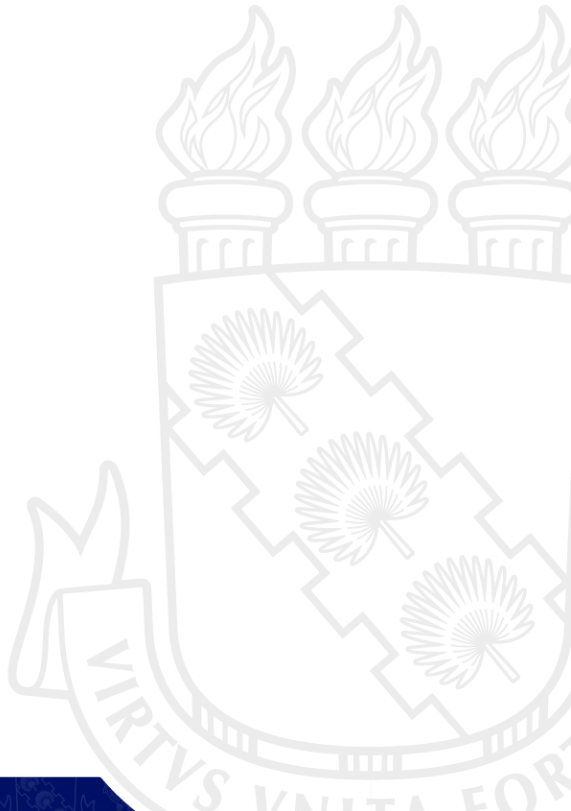
```
semaphore passageir = C
semaphore carrinho  = 0
semaphore andando   = 0
semaphore mutex     = 1

int Npass = 0;
Passageiro()
{
  while (true)
  {
    DOWN(passageiro);
    entra_no_carrinho(); /* varios passageiros podem entrar ao mesmo
        tempo */
    DOWN(mutex);
    Npass++;
    if (Npass == C) /* carrinho lotou */
    {
      UP(carrinho) /* autoriza carrinho a andar */
      DOWN(andando) /* espera carrinho parar */
      UP(mutex)
    }
    else
    {
      UP(mutex)
      DOWN(andando) /* espera carrinho lotar, passear e voltar */
    }
  }
}
```

Problemas Clássicos de IPC - Montanha Russa

```
Carrinho()
{
  while (true)
  {
    DOWN(carrinho) /* espera autoriza  o para andar */
    passeia() /* faz o passeio e volta */
    Npass := 0 /* esvazia carrinho */
    for (int i=0; i<C; i++)
    {
      UP(andando); /* libera passageiro que andou de volta    fila */
      UP(passageiro); /* libera entrada no carrinho */
    }
}
```

Escalonamento

- Múltiplos processos
- Recurso compartilhado $\rightarrow$ CPU
- SO deve escolher sequência de execução
- Parte responsável no SO: escalonador
- Política de escolha: algoritmo de escalonamento
- Aplicação em processos ou threads

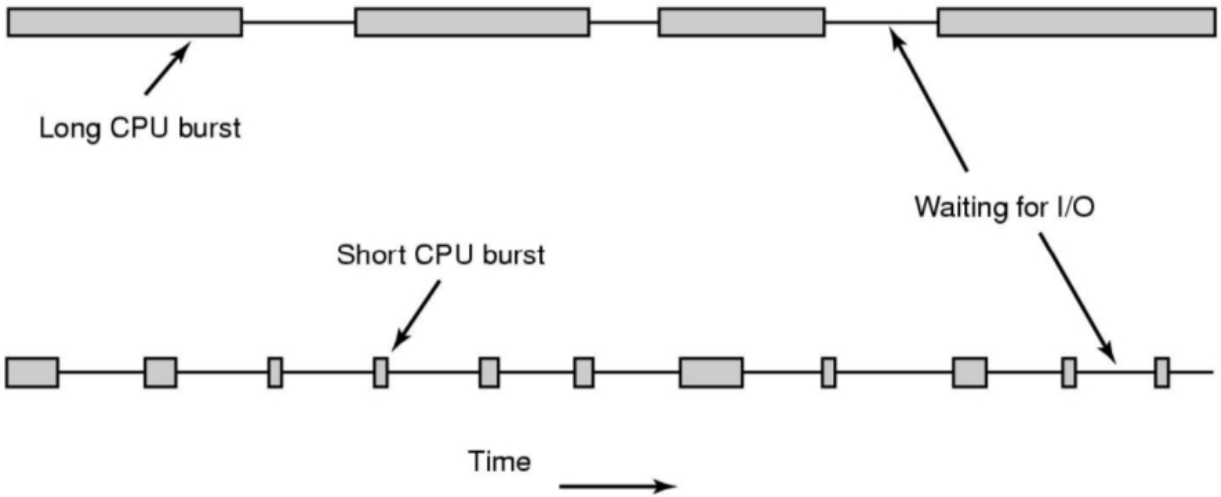


Escalonamento

- Sistemas em lotes
  - Próxima tarefa existente na fita
- Sistemas multiprogramadados
  - Usuários esperando serviço
  - Possível coexistência: lotes + usuários
  - Decisão sobre tipo de tarefa a executar
  - Desempenho percebido
- Computadores pessoais
  - Usuário monotarefa
  - CPU raramente um gargalo
  - Gargalo na entrada de dados
  - Escalonamento tem papel secundário

Escalonamento

- Servidores/Estações de trabalho
  - Múltiplos processos competindo
  - Decisão influencia no desempenho percebido
  - Exemplo: Solicitações de usuários x Acumulação de estatísticas diárias
  - Eficiência no uso da CPU: custo de chaveamento
- Comportamento dos processos
  - Alternância entre CPU e E/S
  - CPU x I/O bounded
  - CPUs mais rápidas → Tendência a I/O bounded

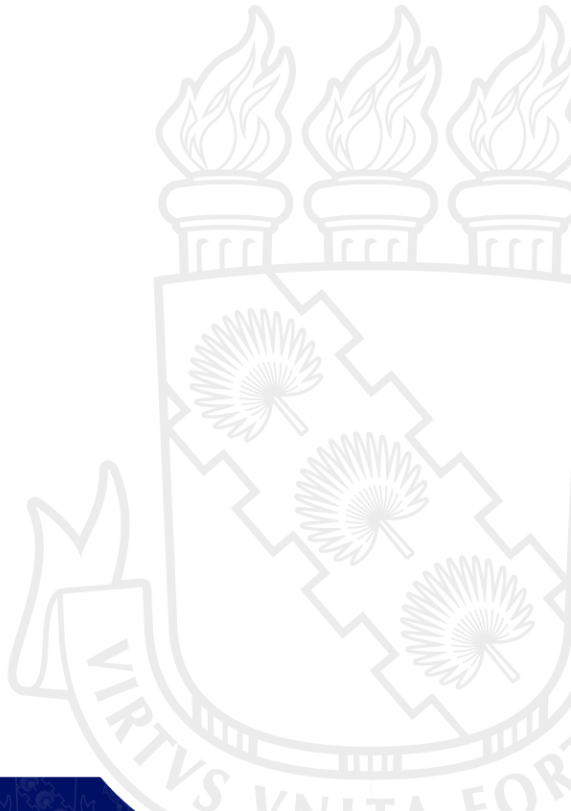


Pontos de Escalonamento

- Tomada de decisão
  - Criação de um novo processo
  - Término de um processo
  - Bloqueio
  - Interrupção
- Tipos de escalonadores
  - Tratamento de Interrupção: Preemptivo/Não preemptivo
  - Ambiente: Lote/Interativo/Tempo Real

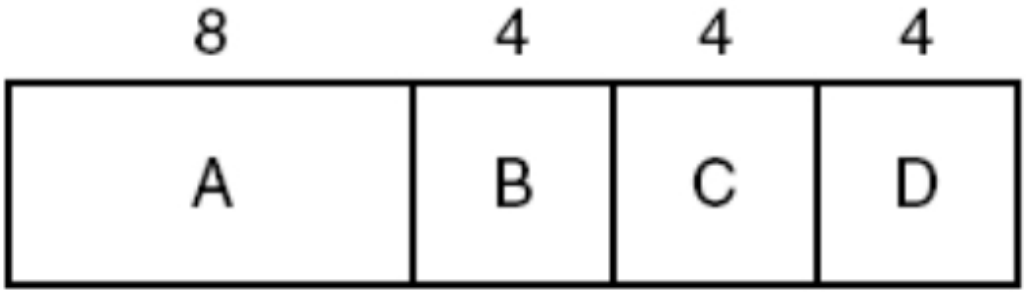
Objetivos de um Escalonador

- Gerais
  - Justiça
  - Aplicação da política
  - Equilíbrio: maximização no uso dos recursos
- Lote
  - + Vazão
  - - Tempo de retorno
  - + Utilização da CPU
- Interativo
  - - Tempo de resposta
  - Proporcionalidade
- Tempo Real
  - Cumprimento dos prazos
  - Previsibilidade



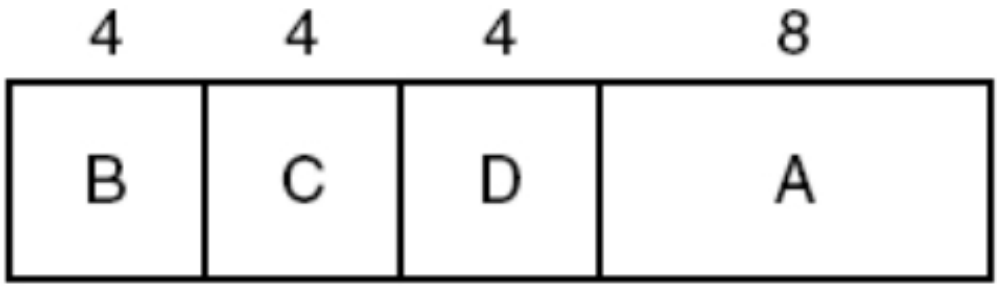
Escalonamento - Sistemas em Lote

- First Come, First Served (FCFS)
  - Não preemptivo
  - CPU alocada na ordem de requisição
  - Novos processos para o final da fila
  - Processo bloqueia $\rightarrow$ primeiro da fila executa
  - Processo bloqueado para o final da fila
- Shortest Job First (SJF)
  - Não preemptivo
  - Tempos de execução conhecidos
  - Adequado quando todos processos disponíveis



Escalonamento - Sistemas em Lote

- First Come, First Served (FCFS)
  - Não preemptivo
  - CPU alocada na ordem de requisição
  - Novos processos para o final da fila
  - Processo bloqueia → primeiro da fila executa
  - Processo bloqueado para o final da fila
- Shortest Job First (SJF)
  - Não preemptivo
  - Tempos de execução conhecidos
  - Adequado quando todos processos disponíveis

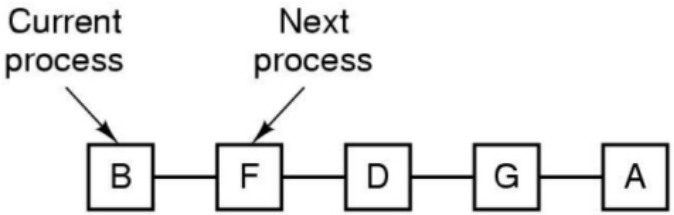


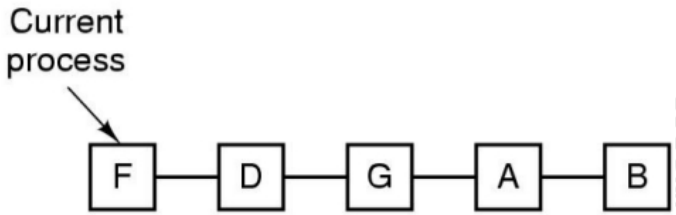
Escalonamento - Sistemas em Lote

- Shortest Remaining Time Next (SRTN)
  - Versão preemptiva do SJF
  - Tempos de execução conhecidos
  - Menor tempo de execução restante $\rightarrow$ Maior prioridade

Escalonamento - Sistemas Iterativos

- Round Robin
  - Chaveamento circular
  - Preemptivo
  - Cada processo tem um tempo para executar (*quantum*)
  - Ao final do *quantum*, outro processo é executado
  - Tamanho do *quantum* $\rightarrow$ Parâmetro de projeto
  - Overhead na troca de contexto x Tempo de resposta
  - Preferível minimizar preempção



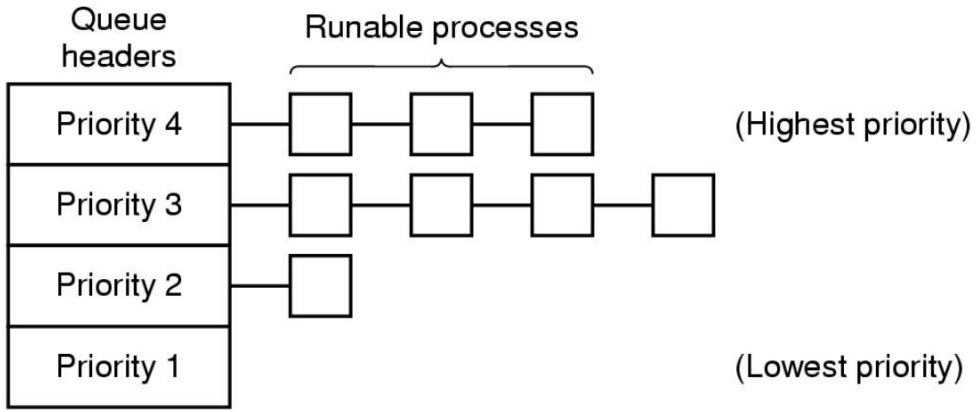


Escalonamento - Sistemas Iterativos

- Por prioridades
  - A cada processo é atribuida uma prioridade (estática ou dinâmica)
  - O processo com maior prioridade é executado
  - Possível problema: processos com alta prioridade não liberam CPU
  - Possíveis soluções:
    - Redução gradativa de prioridade do processo em execução
    - *Quantum* máximo
  - Prioridades dinâmicas para atingir objetivos
    - Processos orientados à E/S
    - Prioridade inverso da última fração de *quantum*

Escalonamento - Sistemas Iterativos

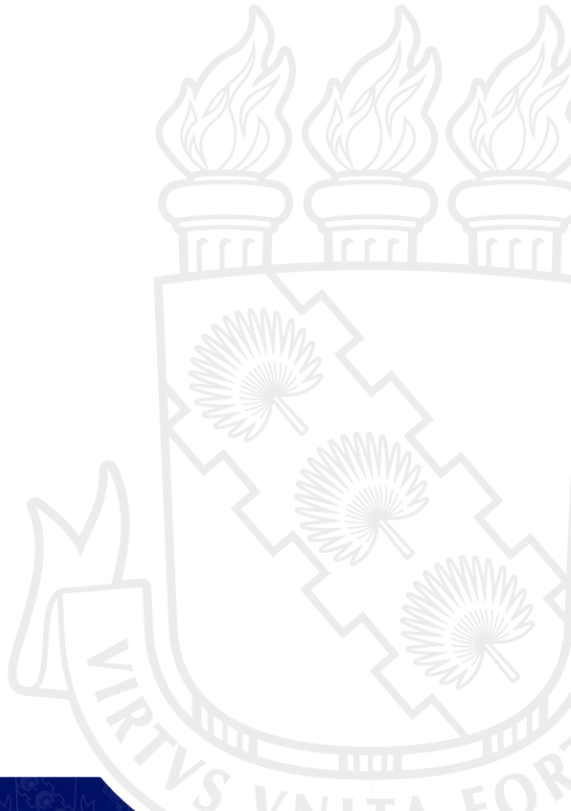
- Filas múltiplas



  - CTSS: MIT utilizando IBM 7094
  - Chaveamento de processo muito lento: um processo na memória
  - Processos CPU bounded $\rightarrow$ *quantum* grande
  - Classes de prioridade mais alta $\rightarrow$ um *quantum*
  - Cada classe de menor prioridade $\rightarrow$ 2x *quantum*
  - Processo preemtado $\rightarrow$ rebaixado de classe

Escalonamento - Sistemas Iterativos

- Filas múltiplas (Exemplo)
  - Processo com tempo de execução 100x *quantum*
  - Inicialmente com maior prioridade (1 *quantum*)
  - Próximas execuções: 2, 8, 16, 32, 64 *quanta*
  - 7 trocas de contexto (ao invés de 100)
  - Diminuição de frequência de execução
    - Mais tempo para processos iterativos rápidos

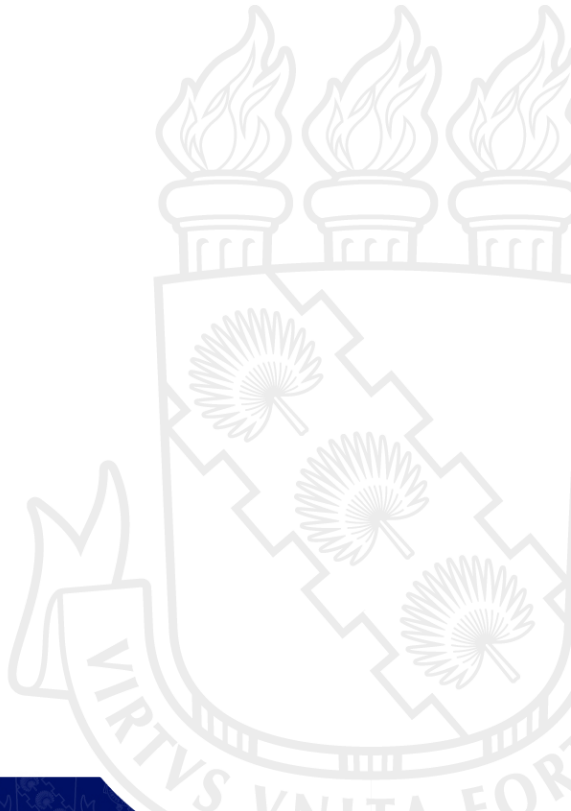


Escalonamento - Sistemas Iterativos

- Shortest Process First/Next (SPF/SPN)
  - Similar ao aplicado a sistemas em lote
  - Problema em conhecer a priori a duração dos processos
  - Estimativa baseada em execuções passadas
    $T_{n+1} = aT_{n-1} + (1-a)T_n$
    onde $a < 1$ é um fator de esquecimento
  - Também chamada de *aging* (envelhecimento)

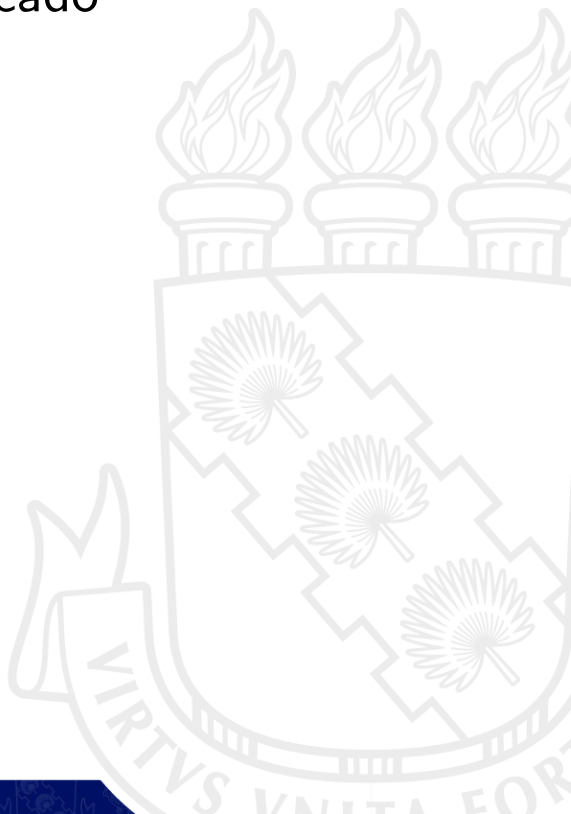
Escalonamento - Sistemas Iterativos

- Escalonamento garantido
  - Garantia de desempenho
  - Dados n processos, cada um deve ter $\frac{1}{n}$ da CPU
  - Processo criado no instante $t_{i,start}$
  - Processo tem direito de $t_{i,share} = \frac{t_{i,now} - t_{i,start}}{n}$
  - Tempo efetivamente utilizado: $t_{i,used}$
  - Fator de utilização $\frac{t_{i,used}}{t_{i,share}}$
  - Quanto menor o fator, maior a prioridade



Escalonamento - Sistemas Iterativos

- Escalonamento por loteria
  - Cada processo tem bilhetes de loteria
  - A cada ponto de escalonamento, um bilhete é sorteado
  - Prioridade implementada pelo número de bilhetes
  - Cooperação pela troca de bilhetes
- Escalonamento fração justa (*fair share*)
  - Preocupação com a propriedade dos processos
  - Alocação de CPU entre usuários
  - Diversas políticas de distribuição entre usuários



Escalonamento - Sistemas de Tempo Real

- Tempo tem uma função essencial
- Reação a estímulos externos dentro do prazo
- Duas principais categorias
  - Crítico
  - Não-crítico
- Escalonador deve tentar garantir atendimento a prazos
- Escalonamento de tempo real
  - Estático/dinâmico
  - Online/Offline
- Processos (tasks) periódicos, aperiódicos e esporádicos
- Teste de escalonabilidade