Sistemas Operacionais

Problemas Clássicos de Comunicação entre Processos

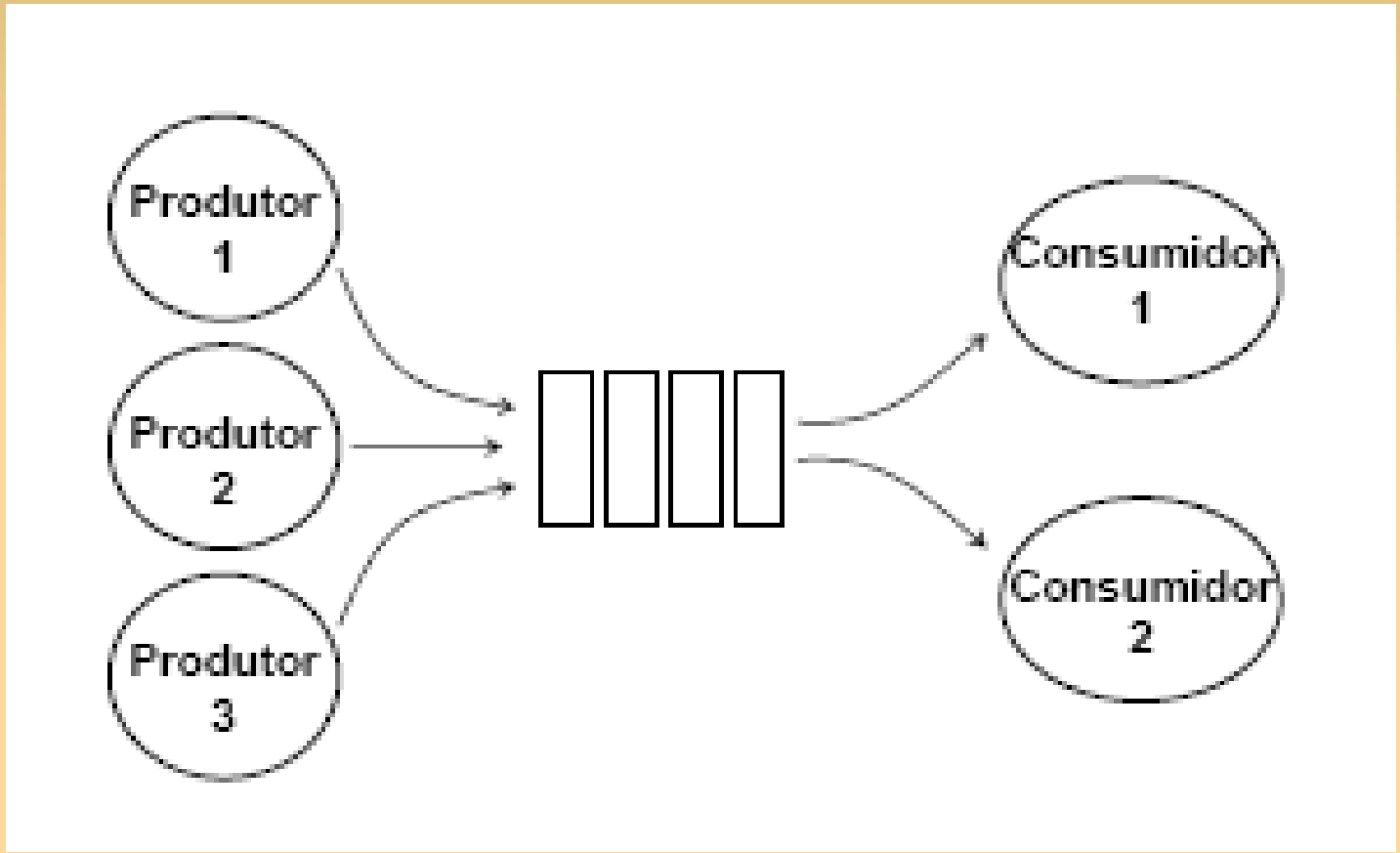
Norton Trevisan Roman
Marcelo Morandini
Jó Ueyama

Apostila baseada nos trabalhos de Kalinka Castelo Branco, Antônio Carlos Sementille, Paula Prata e nas transparências fornecidas no site de compra do livro "Sistemas Operacionais Modernos"

Produtor – Consumidor

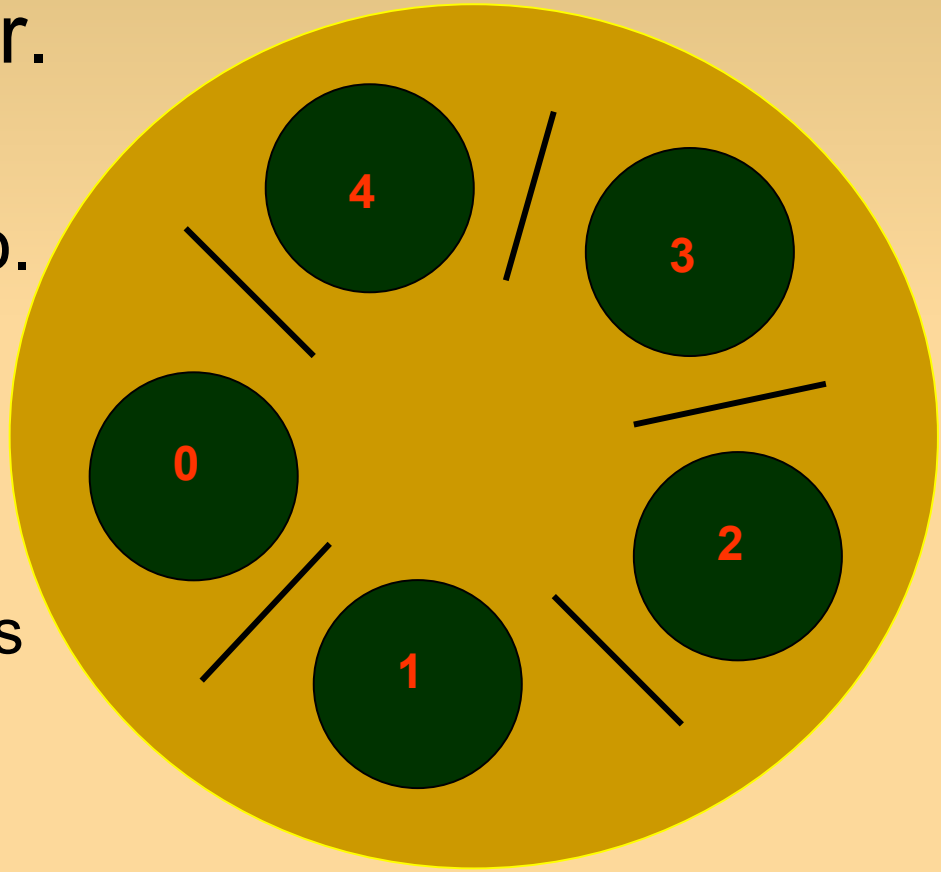
- Um sistema é composto por entidades produtoras e entidades consumidoras.
- Entidades produtoras
  - Responsáveis pela produção de itens que são armazenados em um buffer (ou em uma fila)
  - Itens produzidos podem ser consumidos por qualquer consumidor
- Entidades consumidoras
  - Consomem os itens armazenados no buffer (ou na fila)
  - Itens consumidos podem ser de qualquer produtor

Produtor – Consumidor



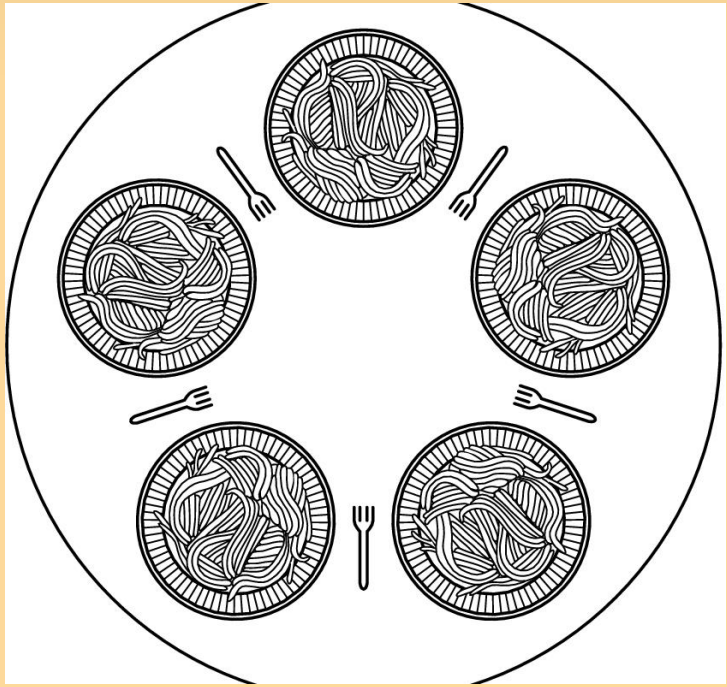
Jantar dos Filósofos

- Cinco filósofos estão sentados ao redor de uma mesa circular para o jantar.
  - Cada filósofo possui um prato para comer macarrão.
  - Além disso, eles dispõem de hashis, em vez de garfos
    - Cada um precisa de 2 hashis
  - Entre cada par de pratos existe apenas um hashi.
    - Hashis precisam ser compartilhados de forma sincronizada

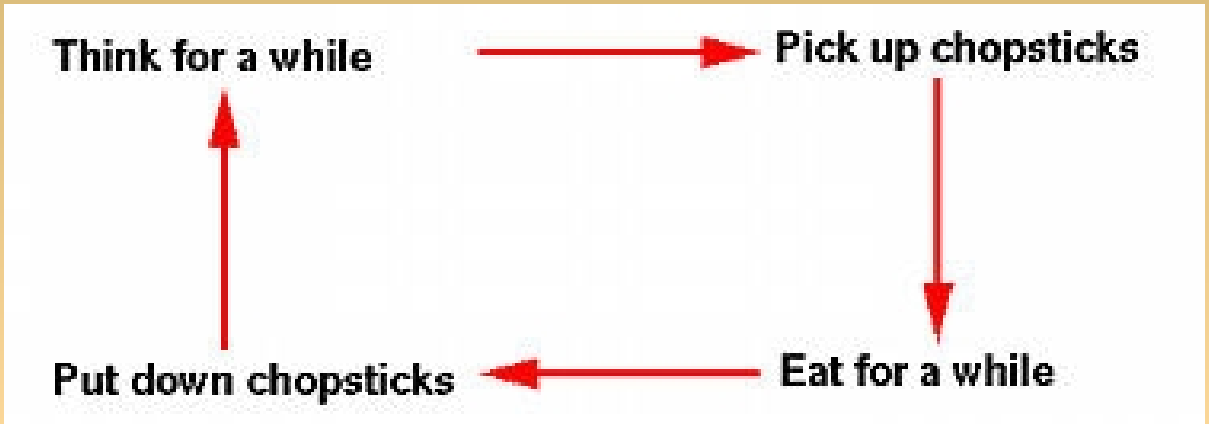


Jantar dos Filósofos

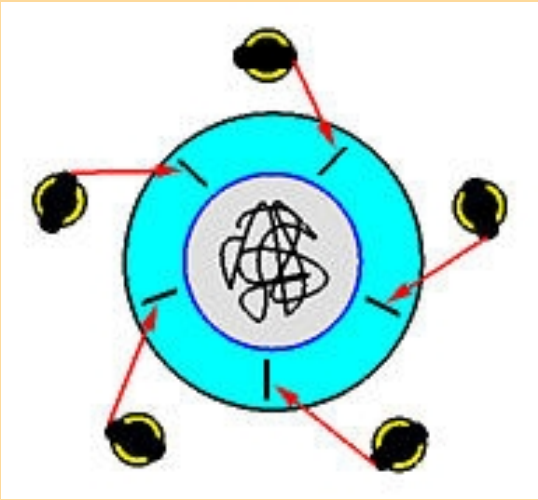
- Os filósofos comem e pensam, alternadamente
  - Não se atém a apenas uma das tarefas
- Além disso, quando comem, pegam apenas um hashi por vez
  - Se conseguir pegar os dois, come por alguns instantes e depois larga os hashis
- Como evitar que fiquem bloqueados?

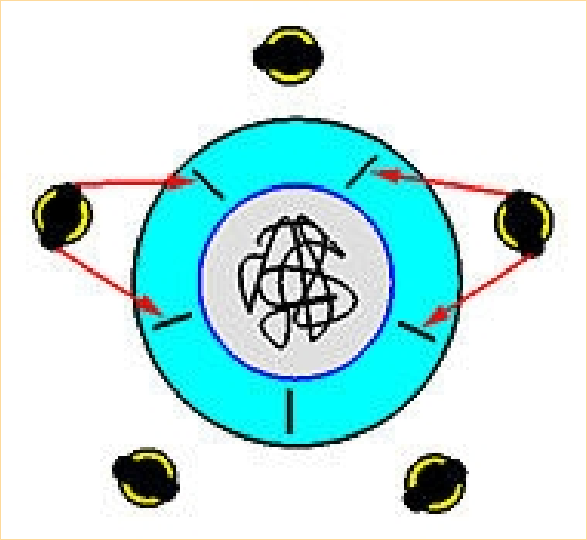


Jantar dos Filósofos



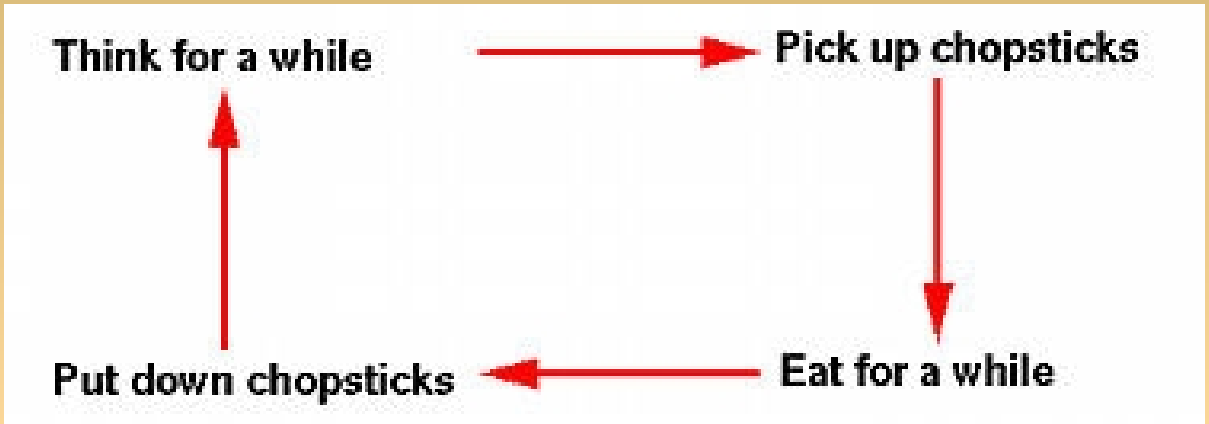
tópicos



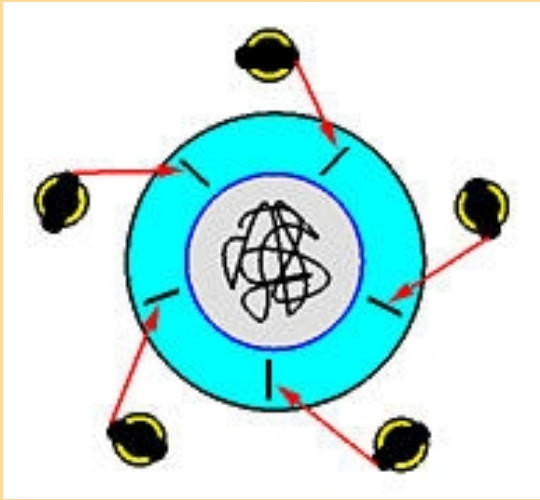


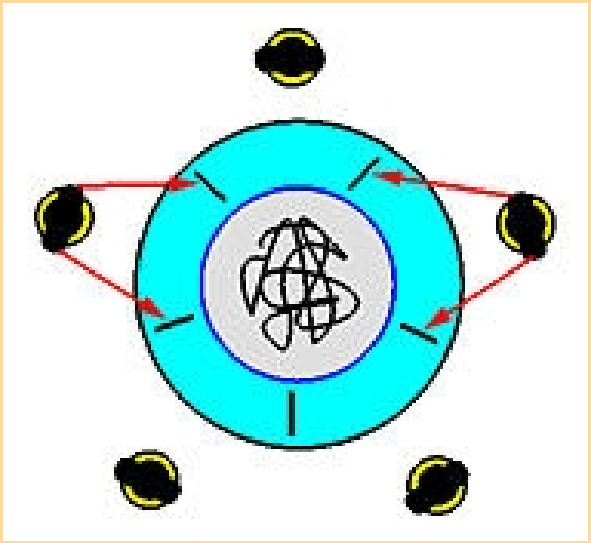
Que fazer???

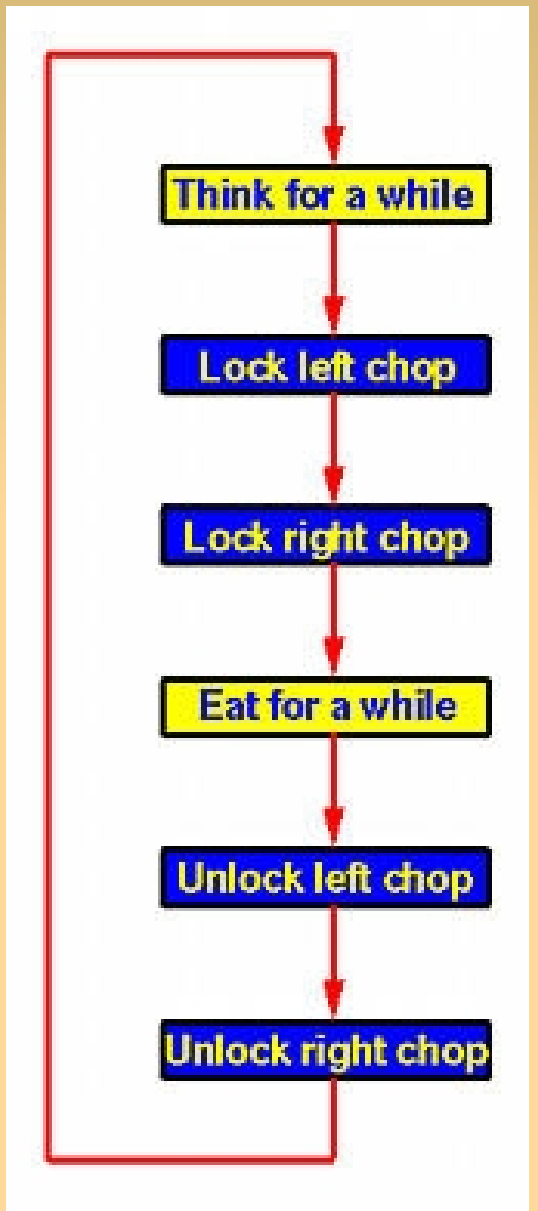
Jantar dos Filósofos



tópicos

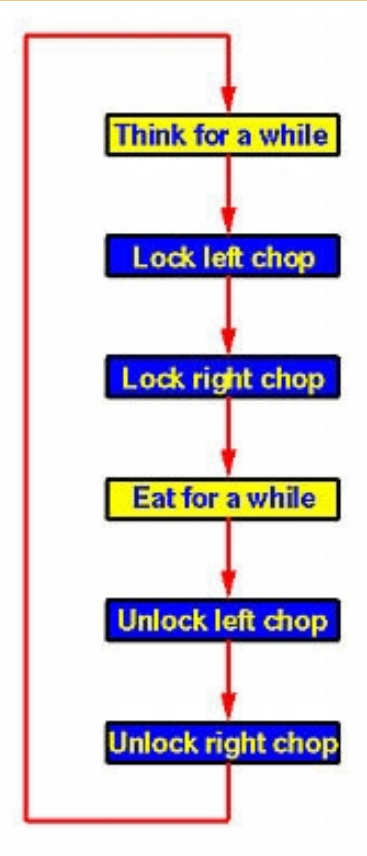






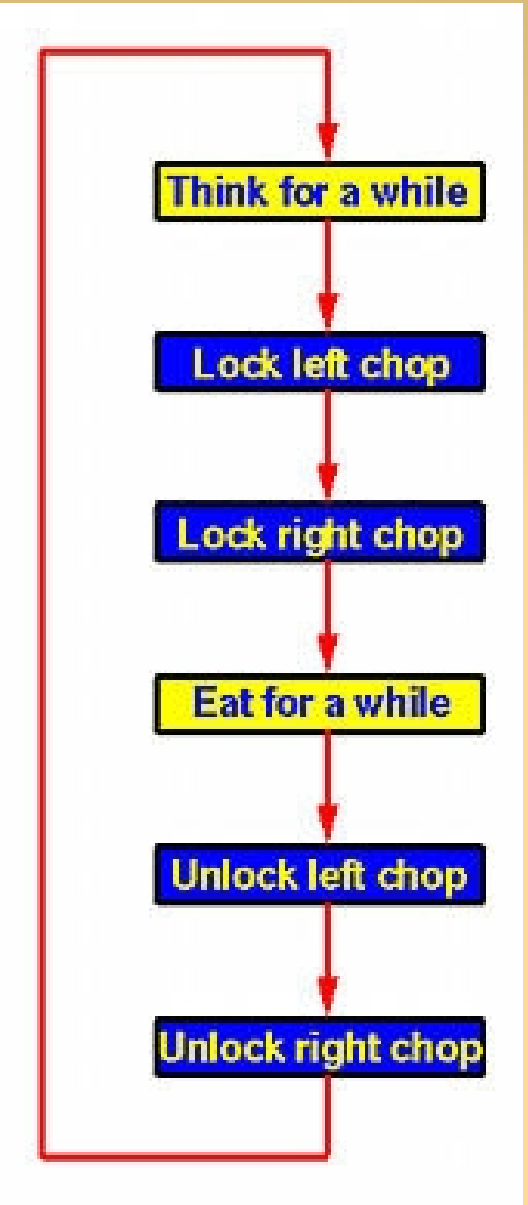
Jantar dos Filósofos

```
#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 */
    }
}
```



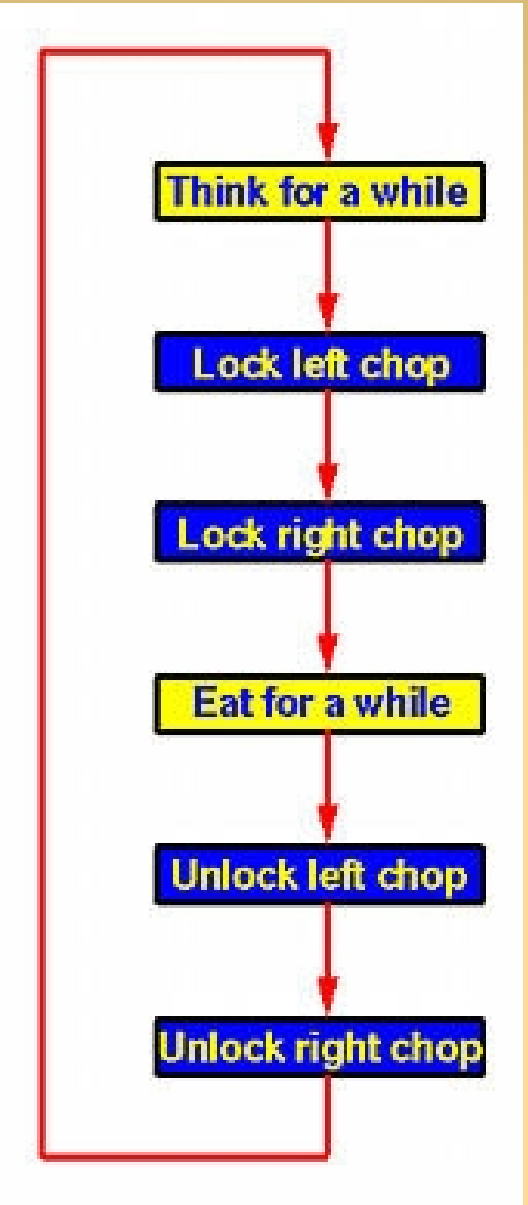
Jantar dos Filósofos

- Isso funciona?



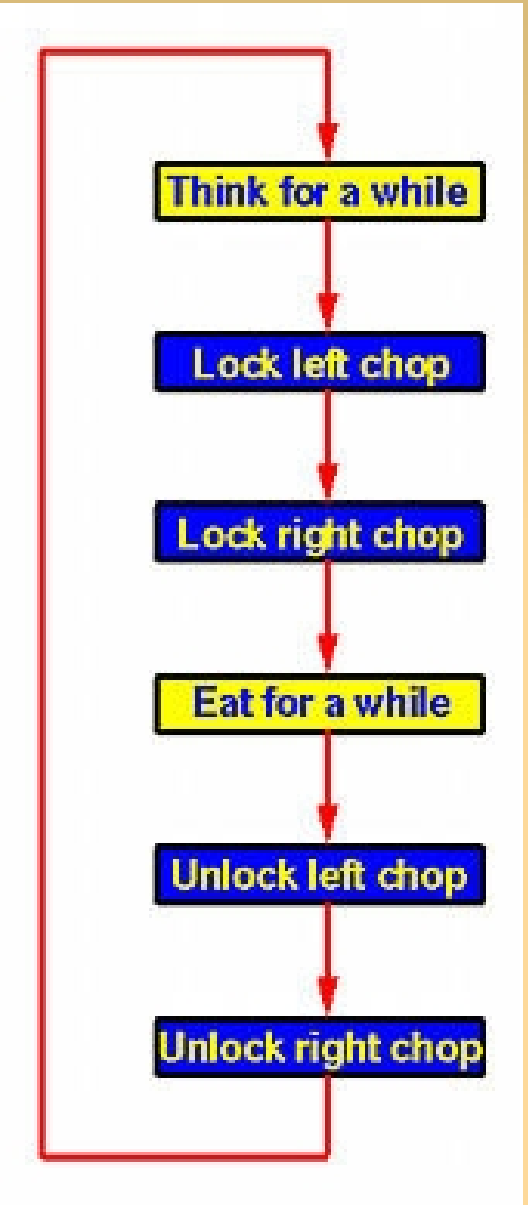
Jantar dos Filósofos

- Isso funciona?
  - Em take_fork():
    - Se todos os filósofos pegarem o hashi da esquerda, nenhum pegará o da direita – deadlock
- E como solucionar?



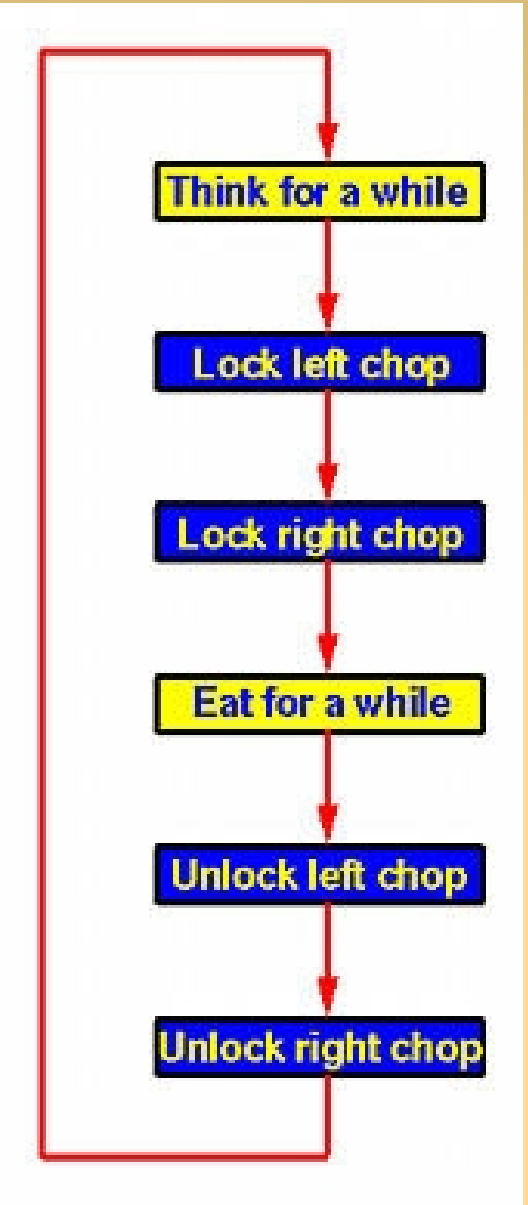
Jantar dos Filósofos

- Isso funciona?
  - Em take_fork():
    - Se todos os filósofos pegarem o hashi da esquerda, nenhum pegará o da direita – deadlock
- E como solucionar?
  - Após pegar o hashi da esquerda, o filósofo verifica se o da direita está livre. Se não estiver, devolve o hashi que pegou, espera um pouco e tenta novamente



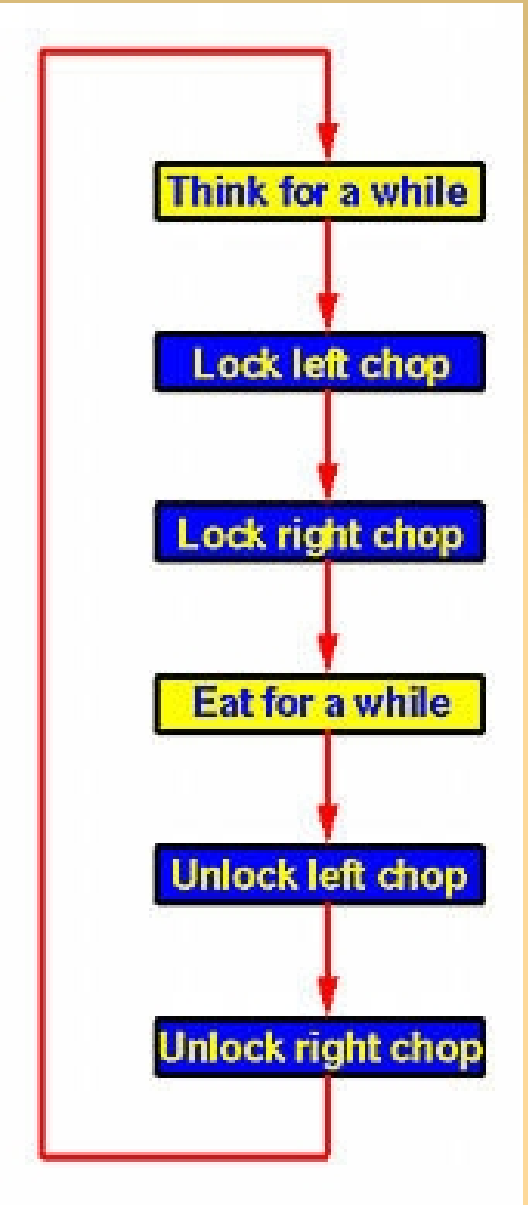
Jantar dos Filósofos

- Isso funciona?



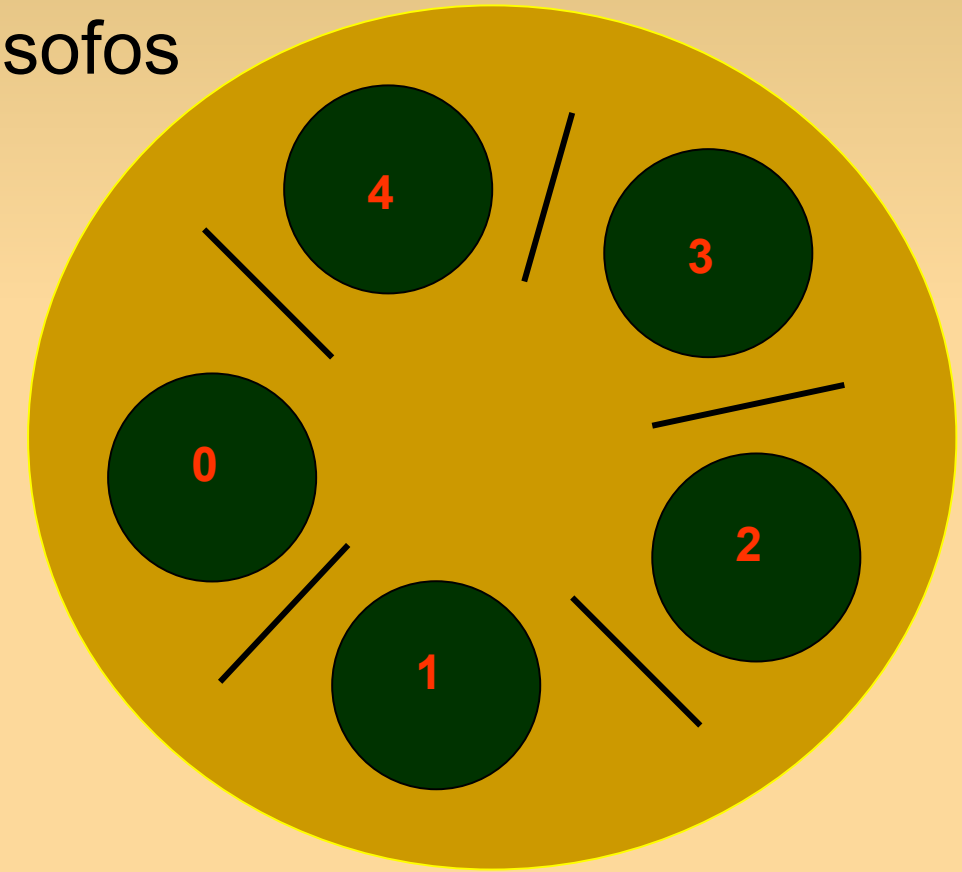
Jantar dos Filósofos

- Isso funciona?
  - Se todos os filósofos pegarem o hashi da esquerda ao mesmo tempo:
    - Verão que o da direita não está livre
    - Largarão seu garfo e e esperarão
    - Pegarão novamente o garfo da esquerda
    - Verão que o da direita não está livre
    - ...
  - Starvation (inanição)



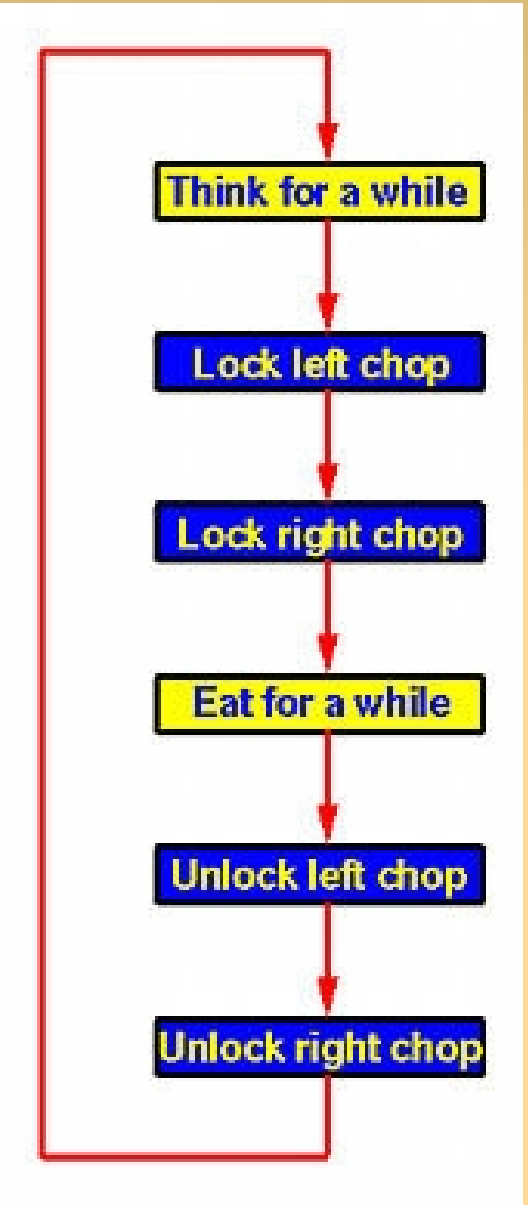
Jantar dos Filósofos

- Problemas que devem ser evitados:
  - Deadlock – todos os filósofos pegam um único hashi ao mesmo tempo;
  - Starvation – os filósofos ficam indefinidamente pegando hashis simultaneamente;



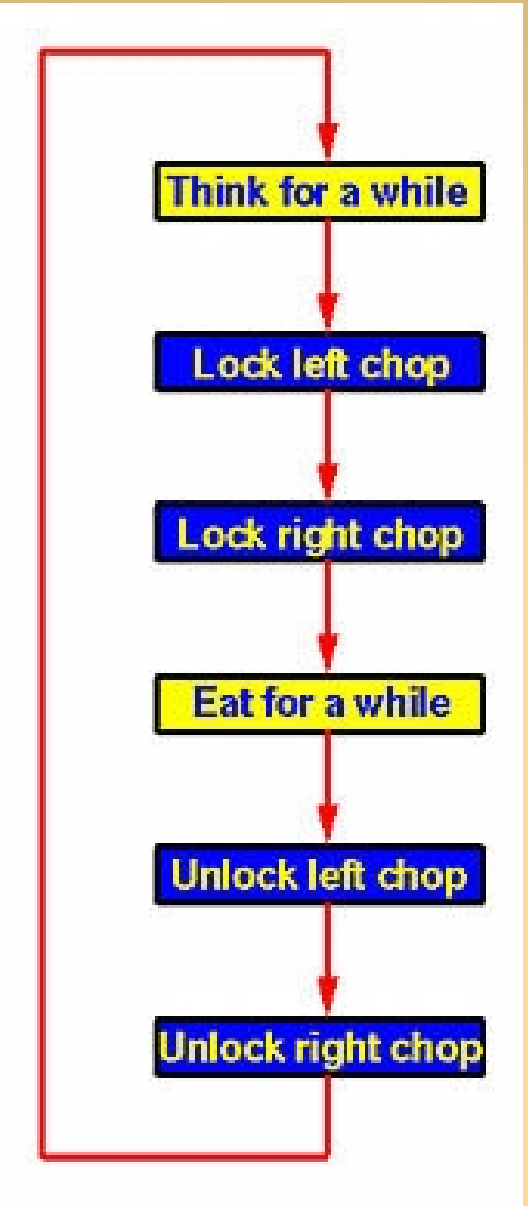
Jantar dos Filósofos

- E agora?
  - Poderíamos fazer com que eles esperassem um tempo aleatório
    - Reduz a chance de starvation
  - Na maioria das aplicações, tentar novamente não é problema
    - Via ethernet, é exatamente isso que é feito com envio de pacotes
    - Sistemas não críticos
    - E controle de segurança em usina nuclear? Será uma boa idéia?



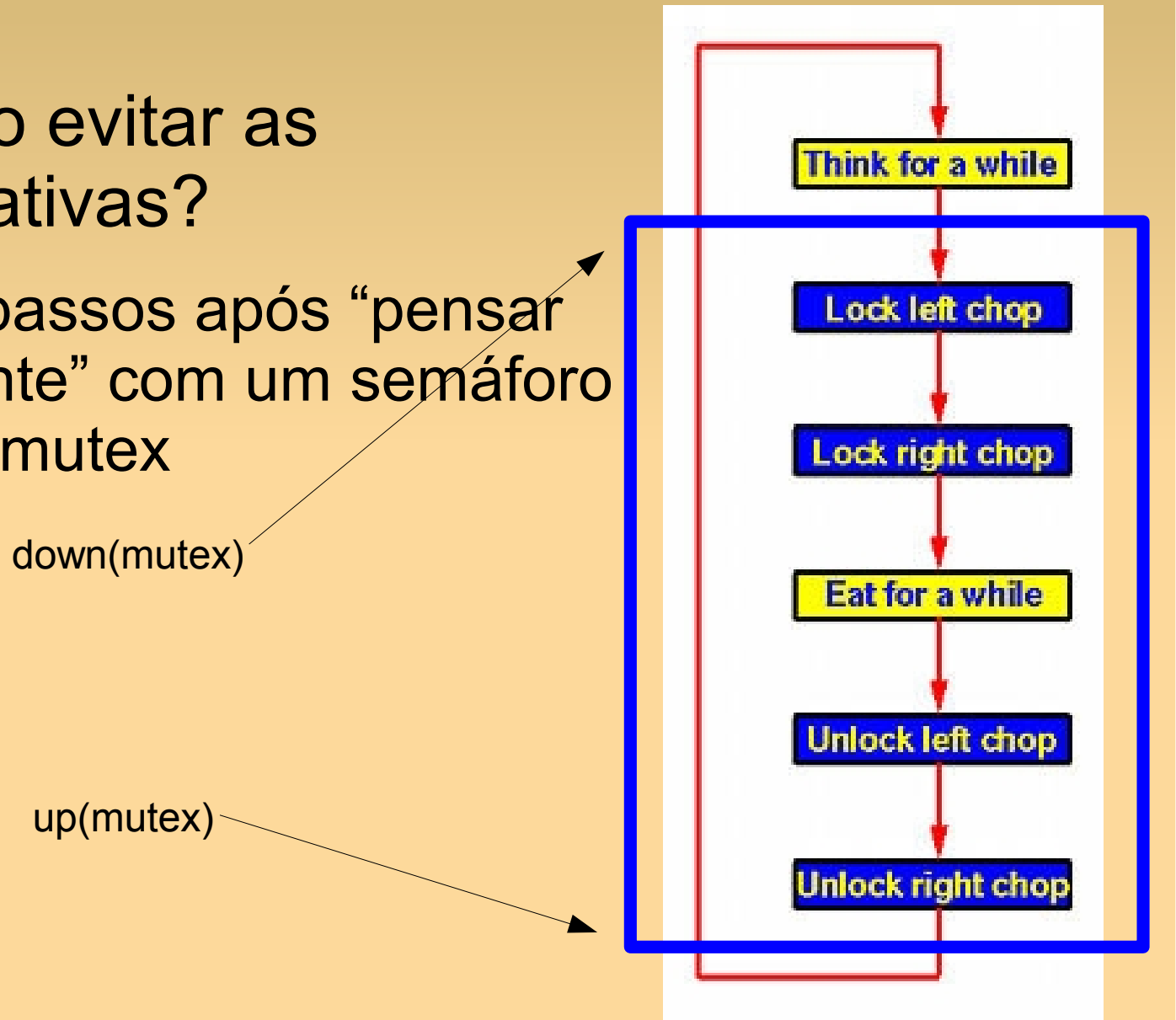
Jantar dos Filósofos

- E agora, como evitar as múltiplas tentativas?



Jantar dos Filósofos

- E agora, como evitar as múltiplas tentativas?
  - Proteger os passos após “pensar por um instante” com um semáforo binário – um mutex



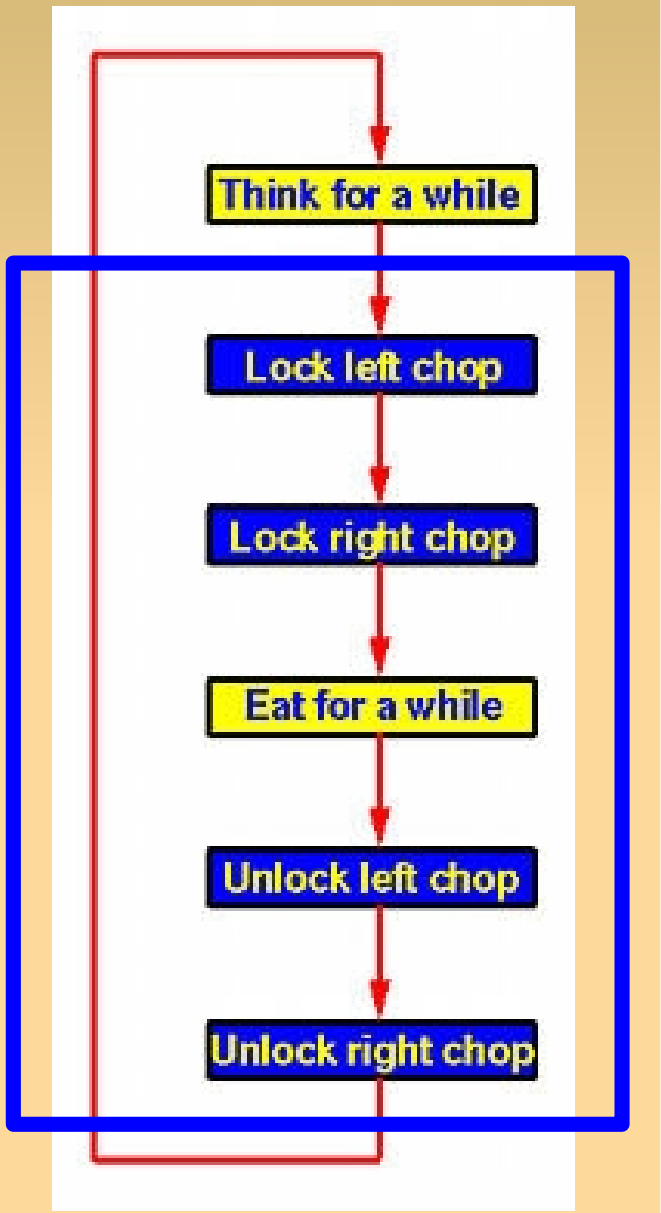
Jantar dos Filósofos

```
#define N 5                                     /* number of philosophers */
semaphore mutex = 1;
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        down(&mutex);
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
        up(&mutex);
    }
}
```

Somente um filósofo come!

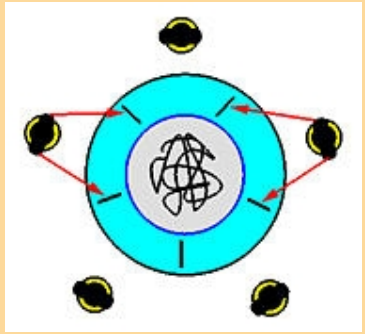
Jantar dos Filósofos

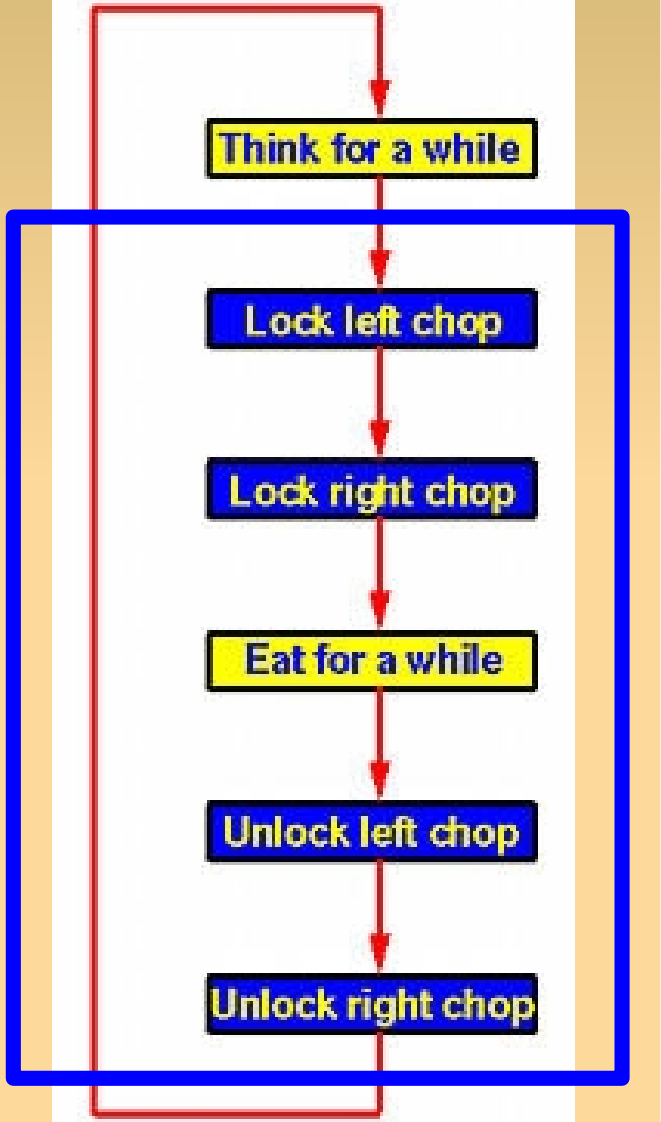
- Há problemas?



Jantar dos Filósofos

- Há problemas?
  - Teoricamente, é uma solução adequada
  - Na prática, contudo, tem um problema de performance:
    - Somente um filósofo pode comer em um dado momento
    - Com 5 hashis, deveríamos permitir que 2 filósofos comessem ao mesmo tempo





Jantar dos Filósofos

- Como solucionar?
  - Sem deadlocks ou starvation
  - Com o máximo de paralelismo para um número arbitrário de filósofos

Jantar dos Filósofos

- Como solucionar?
  - Sem deadlocks ou starvation
  - Com o máximo de paralelismo para um número arbitrário de filósofos
  - Usar um arranjo – state – para identificar se um filósofo está comendo, pensando ou faminto (pensando em pegar os hashis)
    - Um filósofo só pode comer (estado) se nenhum dos vizinhos estiver comendo
  - Usar um arranjo de semáforos, um por filósofo
    - Filósofos famintos podem ser bloqueados se os hashis estiverem ocupados

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 */
      }
}
```

Jantar dos Filósofos

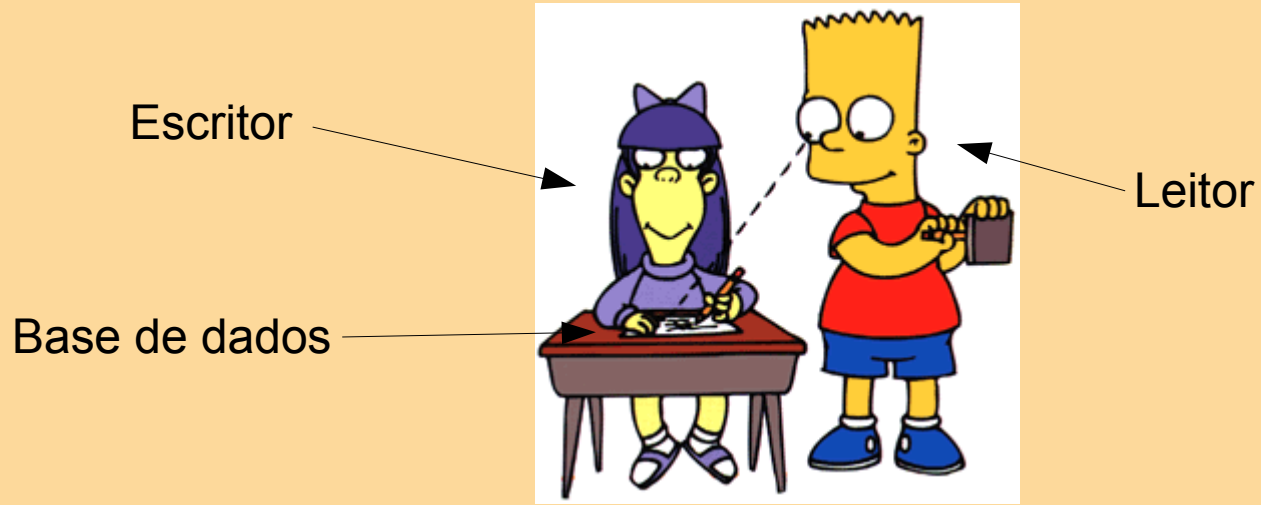
```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 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]);
        }
}
```

Jantar dos Filósofos

- Útil para modelar processos que competem por acesso exclusivo a um número limitado de recursos:
  - Periféricos em geral
  - etc

Leitores e Escritores

- Modela acessos a uma base de dados
- Um sistema com uma base de dados é acessado simultaneamente por diversas entidades. Estas entidades realizam dois tipos de operações:
  - Leitura
  - Escrita



Leitores e Escritores

- Neste sistema é aceitável a existência de diversas entidades lendo a base de dados ao mesmo tempo.

- Porém, se um processo necessita escrever na base, nenhuma outra entidade pode estar realizando acesso a ela
  - Nem mesmo leitura

Leitores e Escritores

- O que fazer?

Leitores e Escritores

- O que fazer?
  - Escritores devem bloquear a base de dados
  - Leitores:
    - Se a base estiver desbloqueada:
      - Se for o primeiro leitor a usar a base, deve bloqueá-la, para que nenhum escritor entre
      - Se, contudo, já houver outro leitor lá, basta usar a base
    - Ao sair, os leitores verificam se há ainda outro usando a base.
      - Se não houver, desbloqueia a base
      - Se houver, deixa bloqueada

Leitores e Escritores

- Possível solução:

```
typedef int semaphore;                  /* use sua imaginação */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* número de processos lendo ou querendo ler */

void reader(void)
{
    while (TRUE) {                      /* repete para sempre */
        down(&mutex);                   /* obtém acesso exclusivo a 'rc' */
        rc = rc + 1;                    /* um leitor a mais agora */
        if (rc == 1) down(&db);         /* se este for o primeiro leitor ... */
        up(&mutex);                     /* libera o acesso exclusivo a 'rc' */
        read_data_base( );              /* acesso aos dados */
        down(&mutex);                   /* obtém acesso exclusivo a 'rc' */
        rc = rc – 1;                    /* um leitor a menos agora */
        if (rc == 0) up(&db);           /* se este for o último leitor ... */
        up(&mutex);                     /* libera o acesso exclusivo a 'rc' */
        use_data_read( );               /* região não crítica */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repete para sempre */
        think_up_data( );               /* região não crítica */
        down(&db);                      /* obtém acesso exclusivo */
        write_data_base( );             /* atualiza os dados */
        up(&db);                        /* libera o acesso exclusivo */
    }
}
```

Jantar dos Filósofos

- Há algum problema com esse procedimento?

Jantar dos Filósofos

- Há algum problema com esse procedimento?
  - Ele esconde uma decisão tomada por nós
  - Suponha que um leitor acesse a base
  - Enquanto isso, outro leitor aparece, e entra sem problemas.
  - Mais leitores aparecem, entrando na base
  - Agora suponha que um escritor aparece
    - Não poderá entrar, devido aos leitores. Então ele é suspenso
  - Mas não param de chegar leitores

Jantar dos Filósofos

- E como resolvemos?

Jantar dos Filósofos

- E como resolvemos?
  - Podemos fazer com que, quando um leitor chegar e um escritor estiver esperando, o leitor é suspenso também, em vez de ser admitido imediatamente
    - Escritores precisam apenas esperar que leitores ativos completem
    - Não precisam esperar por leitores que chegam depois dele
  - Desvantagem:
    - Há menos concorrência → menor performance

Barbeiro Sonolento

- Uma barbearia possui:
  - 1 barbeiro
  - 1 cadeira de barbeiro
  - N cadeira para clientes esperarem
- Quando não há clientes, o barbeiro senta em sua cadeira e dorme

Barbeiro Sonolento

- Quando um cliente chega, ele acorda o barbeiro.
- Quando um cliente chega e o barbeiro estiver atendendo um cliente, ele aguarda sua vez sentado na cadeira de espera.
- Quando um cliente chega e não existem cadeiras de espera disponíveis, o cliente vai embora.
- O problema é programar o barbeiro e os clientes sem que haja condição de disputa

Barbeiro Sonolento – Solução

Para controle de região crítica

Conta os clientes à espera de atendimento

Ao chegar para trabalhar, se não houver clientes, o barbeiro dorme

Um cliente que entra na barbearia deve contar o número de clientes à espera de atendimento. Se este for menor que o número de cadeiras, ele ficará; do contrário, sairá

Qualquer outro cliente (e até o barbeiro) deve esperar a liberação de mutex

```
#define CHAIRS 5                    /* número de cadeiras para os clientes à espera */

typedef int semaphore;              /* use sua imaginação */

semaphore customers = 0;            /* número de clientes à espera de atendimento*/
semaphore barbers = 0;              /* número de barbeiros à espera de clientes */
semaphore mutex = 1;                /* para exclusão mútua */
int waiting = 0;                    /* clientes estão esperando (não estão cortando) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* vai dormir se o número de clientes for 0 */
        down(&mutex);               /* obtém acesso a 'waiting' */
        waiting = waiting – 1;      /* decresce de um o contador de clientes à espera */
        up(&barbers);               /* um barbeiro está agora pronto para cortar cabelo */
        up(&mutex);                 /* libera 'waiting' */
        cut_hair( );                /* corta o cabelo (fora da região crítica) */
    }
}

void customer(void)
{
    down(&mutex);                   /* entra na região crítica */
    if (waiting < CHAIRS) {         /* se não houver cadeiras livres, saia */
        waiting = waiting + 1;      /* incrementa o contador de clientes à espera */
        up(&customers);             /* acorda o barbeiro se necessário */
        up(&mutex);                 /* libera o acesso a 'waiting' */
        down(&barbers);             /* vai dormir se o número de barbeiros livres for 0 */
        get_haircut( );             /* sentado e sendo servido */
    } else {
        up(&mutex);                 /* a barbearia está cheia; não espere */
    }
}
```

Problemas clássicos de comunicação entre processos

- Sugestão de Exercícios:
  - Entender a solução para o problema dos Filósofos utilizando semáforos:
    - Identificando a(s) região(ões) crítica(s);
    - Descrevendo exatamente como a solução funciona;
  - Entender a solução para o problema dos Produtores/Consumidores utilizando monitor:
    - Identificando a(s) região(ões) crítica(s);
    - Descrevendo exatamente como a solução funciona;

Links Interessantes

- http://www.anylogic.pl/fileadmin/Modele/Traffic/filozof/Dining%20Philosophers%20-%20Hybrid%20Applet.html
- http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html
- http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/deadlock.html
- http://users.erols.com/ziring/diningAppletDemo.html