Aula 09
Sistemas Operacionais I

Comunicação e Sincronismo de Processos

– Parte 03

Prof. Julio Cezar Estrella
jcezar@icmc.usp.br

Material adaptado de

Sarita Mazzini Bruschi

baseados no livro Sistemas Operacionais Modernos de A. Tanenbaum

Processos

Introdução

Escalonamento de Processos

- **Comunicação entre Processos**
  - Condição de Disputa
  - Região Crítica
  - Formas de Exclusão Mútua
  - **Problemas Clássicos**

Threads

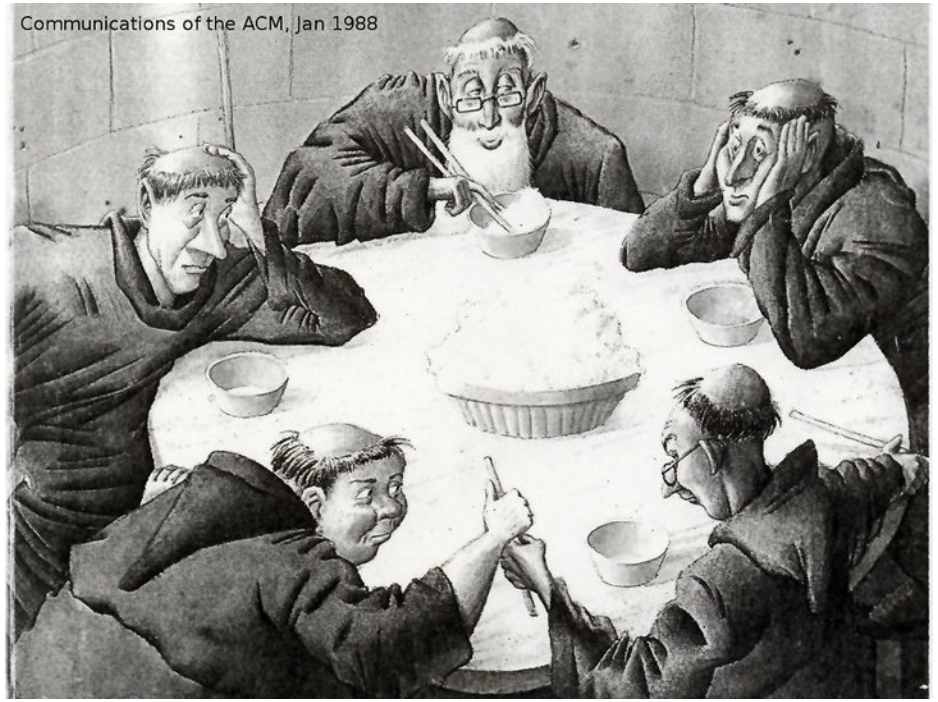
Deadlock

Problemas clássicos de comunicação entre processos

Problema do Jantar dos Filósofos

Cinco filósofos desejam comer espaguete; No entanto, para poder comer, cada filósofo precisa utilizar dois garfos e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.

Os filósofos comem e pensam;

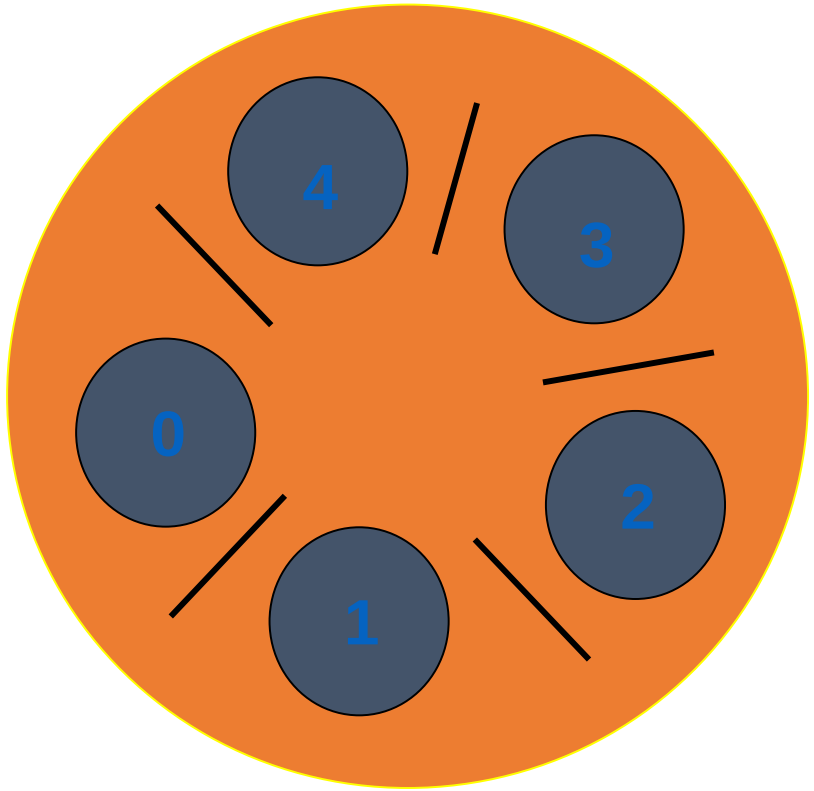


Problemas clássicos de comunicação entre processos

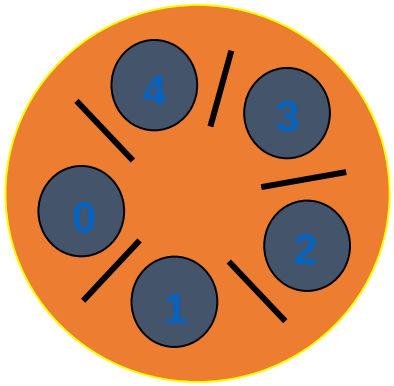
Problemas que devem ser evitados:

Deadlock – um ou mais processos impedidos de continuar;

Starvation – processos executam mas não progridem;



Solução 1 para Filósofos

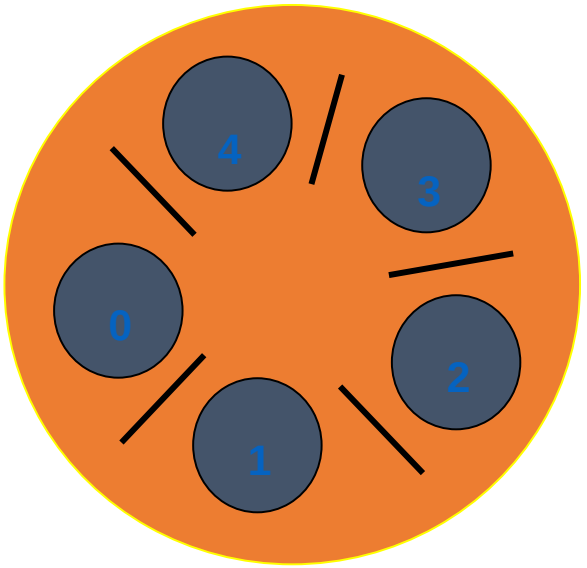



```
#define N 5                                     /* number of philosophers */

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 */
    }
}
```

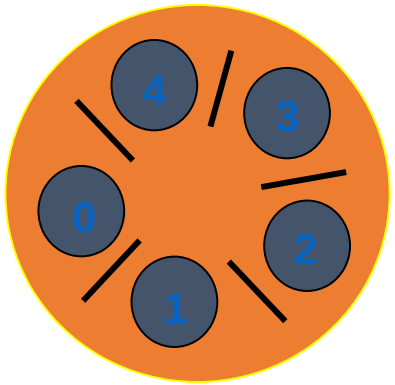
Solução 1 para Filósofos

Problema da solução 1:

Execução do `take_fork(i)` → Se todos os filósofos pegarem o garfo da esquerda, nenhum pega o da direita → *Deadlock*;



Solução 1 para Filósofos



Se modificar a solução:

Pegar o garfo da esquerda

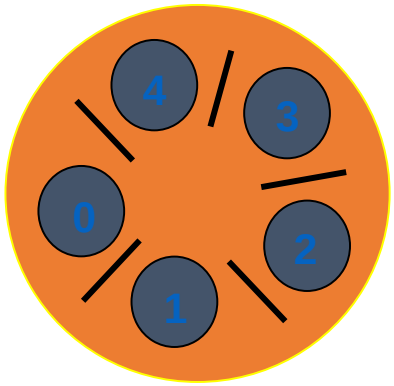
Verificar se o garfo da direita está disponível. Se não está, devolve o da esquerda e começa novamente

Se tempo para tentativa for fixo → *Starvation (Inanição);*

Se tempo for aleatório (abordagem utilizada pela rede Ethernet) – resolve o problema

- Serve para sistemas não-críticos;

Solução 1 para 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 */
        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 */
    }
}
```

down(&mutex);

up(&mutex);

Somente um filósofo come!

Solução 2 para Filósofos usando Semáforos

Permite o máximo de “paralelismo”;

Não apresenta:

Deadlocks;

Starvation;

Solução 2 para Filósofos usando Semáforos (2/3)

```
#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 */
    }
}
```

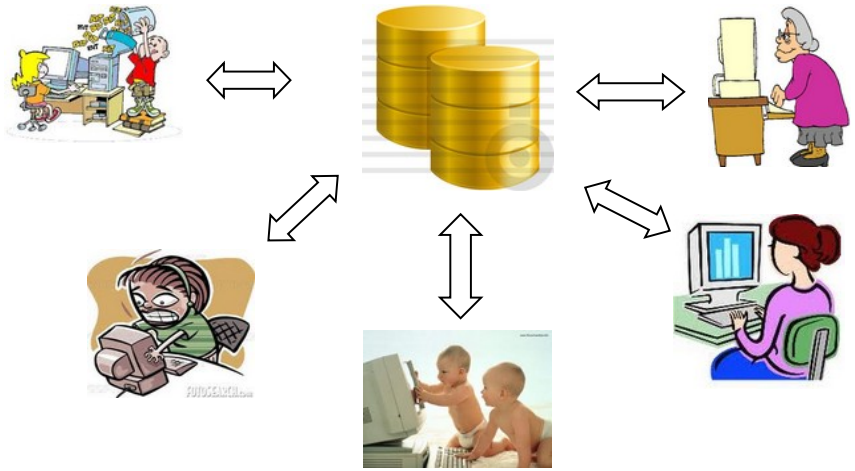
Solução 2 para Filósofos usando Semáforos (3/3)

```
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]);
        }
}
```

Exclusão Mútua
Problema dos Leitores / Escritores

Modela o acesso compartilhado a uma base de dados

Processos leitores e processos escritores competem por um acesso a essa base



Exclusão Mútua
Problema dos Leitores / Escritores

Vários processos leitores podem acessar a base ao mesmo tempo

- Variável compartilhada por todos processos leitores controla o número de leitores na base - **rc**
- Vários processos acessam rc – necessidade de exclusão mútua – **semáforo mutex**

Exclusão Mútua
Problema dos Leitores / Escritores

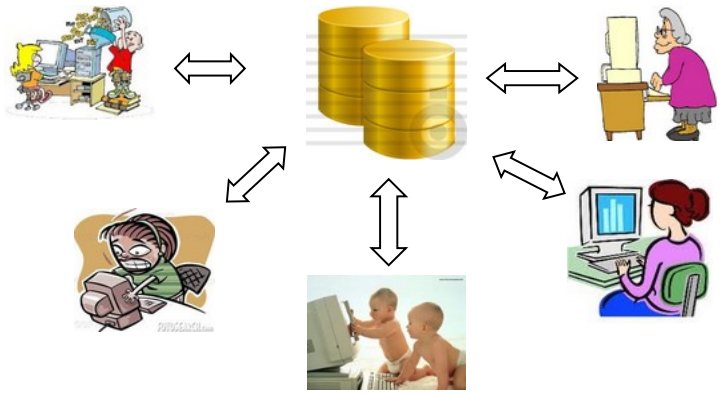
Um único processo escritor pode escrever (modificar) a base de dados em um determinado instante

Quando a base está sendo modificada não pode haver processos leitores acessando a base.

Controlar acesso a leitor **ou** a escritor

Semáforo - **db**

Exclusão Mútua Problema dos Leitores / Escritores



```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
        while (TRUE) {
                down(&mutex);
                rc = rc + 1;
                if (rc == 1) down(&db);
                up(&mutex);
                read_data_base( );
                down(&mutex);
                rc = rc – 1;
                if (rc == 0) up(&db);
                up(&mutex);
                use_data_read( );
        }
}


void writer(void)
{
        while (TRUE) {
                think_up_data( );
                down(&db);
                write_data_base( );
                up(&db);
        }
```