Sistemas Operacionais

Problemas clássicos de IPC

Profª Drª Thaína Aparecida Azevedo Tosta

tosta.thaina@unifesp.br

UNIFESP
Universidade Federal de São Paulo

São José dos Campos
2024/01

Aula passada

- Condições de corrida
- Regiões críticas
- Exclusão mútua com espera ocupada
- Dormir e acordar
- Mutexes e semáforos
- Monitores
- Troca de mensagens
- Barreiras

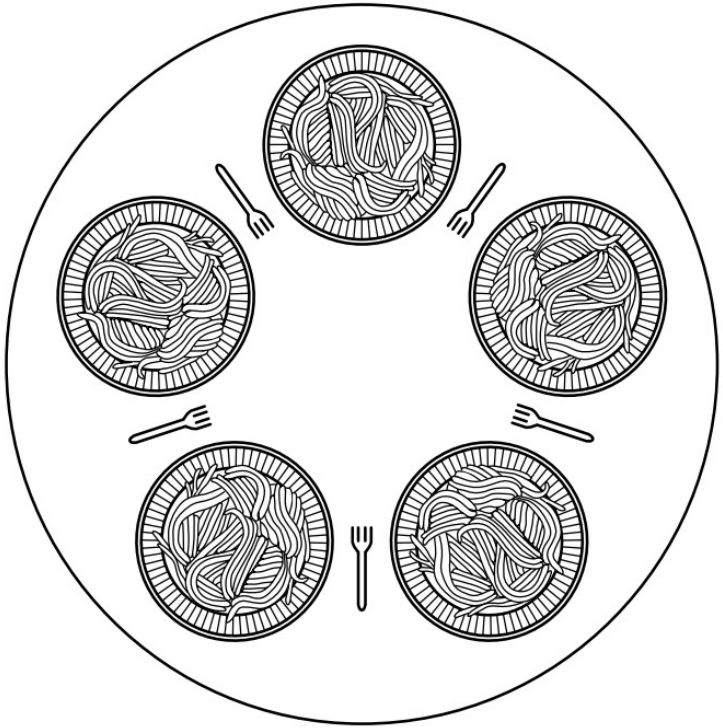
Sumário

- O problema do jantar dos filósofos
- O problema dos leitores e escritores

Objetivo: qual a aplicação desses problemas?

O problema do jantar dos filósofos

- Esse problema permite testar primitivas de sincronização;
- A vida de um filósofo consiste em alternar períodos de alimentação e pensamento;
- Se o filósofo for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar;
- **Você consegue escrever um programa para cada filósofo que faça o que deve fazer e jamais fique travado?**



O problema do jantar dos filósofos

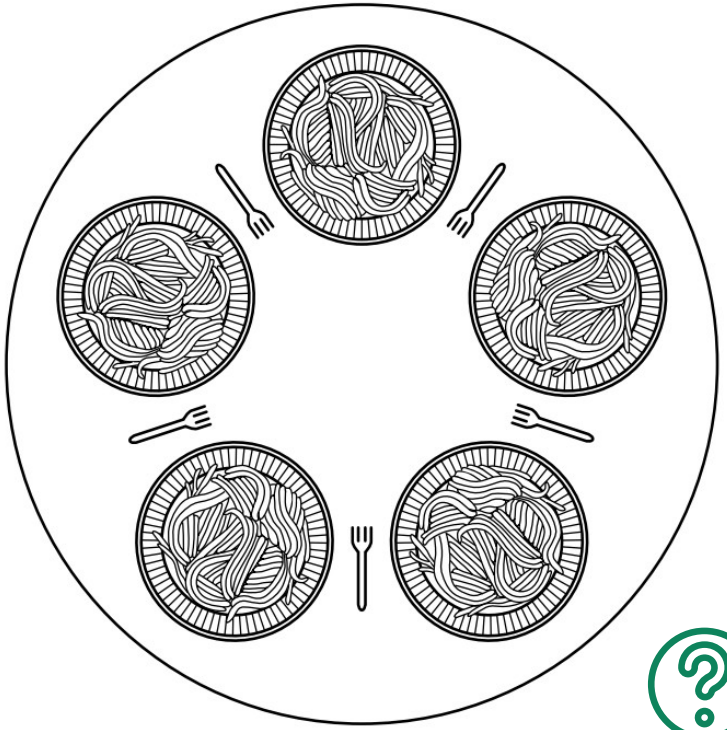
```
#define N 5                                     /* numero de filosofos */

void philosopher(int i)                         /* i: numero do filosofo, de 0 a 4 */
{
        while (TRUE) {
                think( );                       /* o filosofo esta pensando */
                take_fork(i);                   /* pega o garfo esquerdo  */
                take_fork((i+1) % N);           /* pega o garfo direito; % e o operador modulo */
                eat( );                         /* hummm, espaguete */
                put_fork(i);                    /* devolve o garfo esquerdo a mesa */
                put_fork((i+1) % N);            /* devolve o garfo direito a mesa */
        }
}
```

O problema do jantar dos filósofos

- Problema: Se todos os cinco filósofos pegarem seus garfos esquerdos simultaneamente, nenhum será capaz de pegar seus garfos direitos → impasse;
- Solução:

  Após pegar o garfo esquerdo, o programa confere para ver se o garfo direito está disponível;

  Se não estiver, o filósofo coloca de volta o esquerdo sobre a mesa, espera por um tempo, e repete todo o processo.



O problema do jantar dos filósofos

- Problema: todos os filósofos começam o algoritmo simultaneamente → pega garfo esquerdo, verifica indisponibilidade do direito, devolve esquerdo, espera, ... → inanição (*starvation*);
- Solução: espera de tempo aleatório;
- Rede *vs* controle de segurança de usina de energia nuclear.

O problema do jantar dos filósofos

- Solução: proteger os cinco comandos (`take_fork`, `eat`, `put_fork`) com um semáforo binário;
- O **semáforo** mutex recebe 0 quando o processo entra na região crítica, e o semáforo bloqueia quem o chama nessa condição.

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE e a constante 1 */
          item = produce_item( );                /* gera algo para por no buffer */
          down(&empty);                          /* decresce o contador empty */
          down(&mutex);                          /* entra na regiao critica */
          insert_item(item);                     /* poe novo item no buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&full);                             /* incrementa o contador de lugares preenchidos */
     }
}
```

```
void consumer(void)
{
     int item;

     while (TRUE) {                             /* laco infinito */
          down(&full);                           /* decresce o contador full */
          down(&mutex);                          /* entra na regiao critica */
          item = remove_item( );                 /* pega item do buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&empty);                            /* incrementa o contador de lugares vazios */
          consume_item(item);                    /* faz algo com o item */
     }
}
```

O problema do jantar dos filósofos

- Proteger os cinco comandos (`take_fork`, `eat`, `put_fork`) com um semáforo binário:

  Um filósofo realiza um down em mutex;

  Após substituir os garfos, ele realizaria um up em mutex.

- Problema: só um filósofo pode estar comendo a qualquer instante.

```
#define N 5                                     /* numero de filosofos */

void philosopher(int i)                         /* i: numero do filosofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                               /* o filosofo esta pensando */
        take_fork(i);                           /* pega o garfo esquerdo  */
        take_fork((i+1) % N);                   /* pega o garfo direito; % e o operador modulo */
        eat( );                                 /* hummm, espaguete */
        put_fork(i);                            /* devolve o garfo esquerdo a mesa */
        put_fork((i+1) % N);                    /* devolve o garfo direito a mesa */
    }
}
```

O problema do jantar dos filósofos

Solução sem impasse e com máximo paralelismo (1/2):

```
#define N           5                   /* numero de filosofos */
#define LEFT        (i+N–1)%N           /* numero do vizinho a esquerda de i */
#define RIGHT       (i+1)%N             /* numero do vizinho a direita de i */
#define THINKING    0                   /* o filosofo esta pensando */
#define HUNGRY      1                   /* o filosofo esta tentando pegar garfos */
#define EATING      2                   /* o filosofo esta comendo */

typedef int semaphore;                  /* semaforos sao um tipo especial de int */
int state[N];                           /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;                    /* exclusao mutua para as regioes criticas */
semaphore s[N];                         /* um semaforo por filosofo */

void philosopher(int i)                 /* i: o numero do filosofo, de 0 a N–1 */
{
    while (TRUE) {                      /* repete para sempre */
        think( );                       /* o filosofo esta pensando */
        take_forks(i);                  /* pega dois garfos ou bloqueia */
        eat( );                         /* hummm, espaguete! */
        put_forks(i);                   /* devolve os dois garfos a mesa */
    }
}
```

O problema do jantar dos filósofos

Solução sem impasse e com máximo paralelismo (2/2):

```
void take_forks(int i)                  /* i: o numero do filosofo, de 0 a N–1 */
{
        down(&mutex);                   /* entra na regiao critica */
        state[i] = HUNGRY;              /* registra que o filosofo esta faminto */
        test(i);                        /* tenta pegar dois garfos */
        up(&mutex);                     /* sai da regiao critica */
        down(&s[i]);                    /* bloqueia se os garfos nao foram pegos */
}

void put_forks(i)                       /* i: o numero do filosofo, de 0 a N–1 */
{
        down(&mutex);                   /* entra na regiao critica */
        state[i] = THINKING;            /* o filosofo acabou de comer */
        test(LEFT);                     /* ve se o vizinho da esquerda pode comer agora */
        test(RIGHT);                    /* ve se o vizinho da direita pode comer agora */
        up(&mutex);                     /* sai da regiao critica */
}

void test(i)/* i: o numero do filosofo, de 0 a N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

O problema dos leitores e escritores

- É aceitável ter múltiplos processos lendo ao mesmo tempo, mas se um processo está atualizando (escrevendo), nenhum outro pode ter acesso, nem mesmo os leitores;

- Como programar leitores e escritores?

O problema dos leitores e escritores

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

void reader(void)
{
    while (TRUE) {                      /* repete para sempre */
        down(&mutex);                   /* obtem 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);                   /* obtem acesso exclusivo a 'rc' */
        rc = rc – 1;                    /* um leitor a menos agora */
        if (rc == 0) up(&db);           /* se este for o ultimo leitor ... */
        up(&mutex);                     /* libera o acesso exclusivo a 'rc' */
        use_data_read( );               /* regiao nao critica */
    }
}
```

```
void writer(void)
{
    while (TRUE) {                      /* repete para sempre */
        think_up_data( );               /* regiao nao critica */
        down(&db);                      /* obtem acesso exclusivo */
        write_data_base( );             /* atualiza os dados */
        up(&db);                        /* libera o acesso exclusivo */
    }
}
```

O problema dos leitores e escritores

- Problema:

  Se um escritor aparece, ele é suspenso;

  Enquanto pelo menos um leitor ainda estiver ativo, leitores subsequentes serão admitidos;

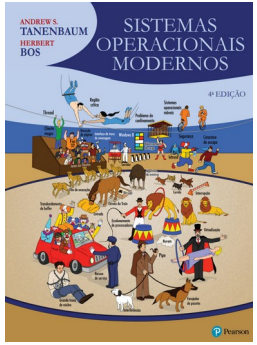
  O escritor será mantido suspenso até que nenhum leitor esteja presente.

- Solução:

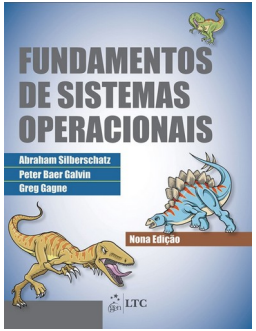
  Quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente.

Objetivo: qual a aplicação desses problemas?

Referências



TANENBAUM, Andrew S.; BOS, Herbert. Sistemas operacionais modernos. 4. edição. São Paulo: Pearson, 2016. xviii, 758 p. ISBN 9788543005676.



SILBERSCHATZ, Abraham.; GALVIN, Peter Baer.; GAGNE, Greg. Fundamentos de sistemas operacionais. 9. edição. Rio de Janeiro: LTC, 2015.

O modelo desta apresentação foi criado pelo Slidesgo.
Agradeço ao Prof. Bruno Kimura da Universidade Federal de São Paulo pelo material disponibilizado.