

# Sincronização entre processos

---

- Problema do Produtor-consumidor
- Soluções para exclusão mútua com bloqueio de processos
  - Sleep/wakeup
  - Semáforos
- Semáforos
- Problemas clássicos de sincronização

# Problema do Produtor/Consumidor (1)

---

- Exemplo de utilização de mecanismos de exclusão mútua
- O problema do produtor/consumidor consiste em um processo (**produtor**) que produz dados para serem consumidos por outro processo (**consumidor**)
- Os dados, ou mensagens, são armazenados temporariamente em um **buffer** enquanto esperam para serem utilizados

# Problema do Produtor/Consumidor (2)

---

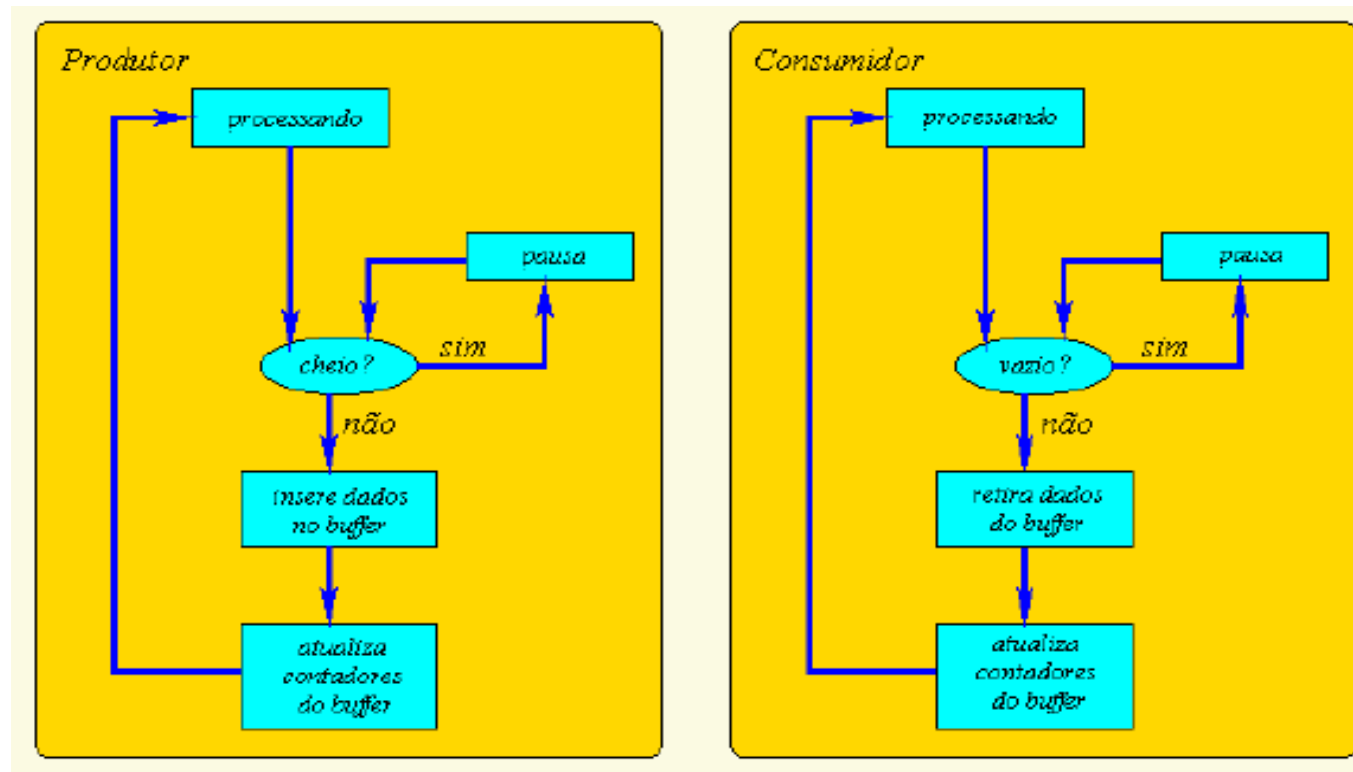
- 2 processos compartilham um *buffer* de tamanho fixo
  - processo *produtor* coloca dados no *buffer*
  - processo *consumidor* retira dados do *buffer*

## *Problemas:*

- Buffer está cheio e produtor quer colocar mais dados
- Buffer está vazio e consumidor quer retirar dados

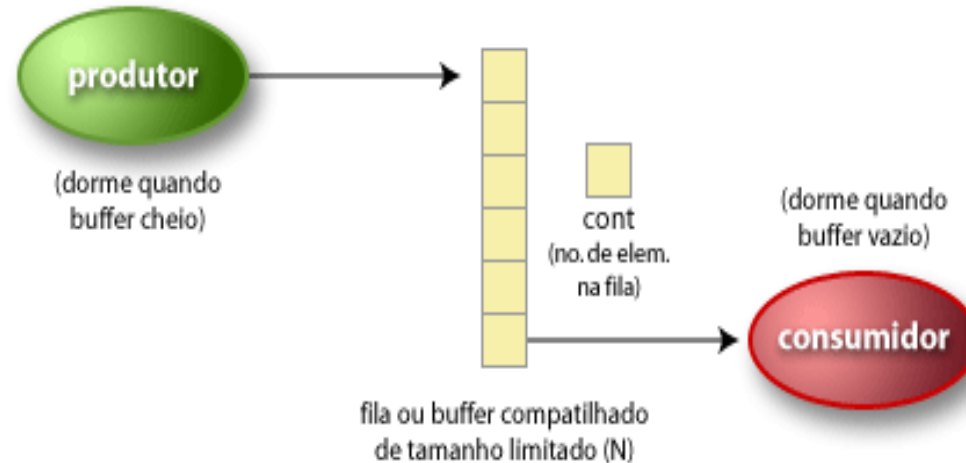
# Problema do Produtor/Consumidor (3)

---



# Produtor/Consumidor: comportamento básico – **sem controle de concorrência**

---



Ver código: [prod-cons-basico.c](#)

# **Produtor/Consumidor sem controle de concorrência, mas com controle de bugs**

---

Código: prod-cons-basico.c

– Problemas:

- Produtor insere em posição que ainda não foi consumida
- Consumidor remove de posição que já foi consumida

**Ver código: prod-cons-basico-bug.c**

# **Produtor/Consumidor com tentativa de controle de concorrência baseada em *espera ocupada***

---

**Ver código: *prod-cons-basico-busy-wait.c***

Código: *prod-cons-basico-busy-wait.c*

- Problema:
  - Espera ocupada !!!

# Soluções para exclusão mútua

---

Espera Ocupada

Primitivas *Sleep/Wakeup*

Semáforos



# Produtor/Consumidor com tentativa de controle de concorrência baseada em **sleep/wakeup** (1)

```
# define N 5
int count = 0; //qtd itens

void producer(void){

    while (true) {
        produce_item();
        if (count == N) // cheio
            sleep();
        insert_item();
        count++;
        if (count == 1)
            wakeup(consumer);
    }
}
```

```
void consumer(void){

    while (true) {
        if (count == 0) //vazio
            sleep();
        remove_item();
        count--;
        if (count == N - 1)
            wakeup(producer);
        consume_item();
    }
}
```

Problema: (a seguir)

- Lost wakeup

# Produtor/Consumidor com tentativa de controle de concorrência baseada em **sleep/wakeup** (2)

---

- Exercício: testar condição de corrida
  - *Buffer* está vazio ( $cont = 0$ );
  - Consumidor testa o valor de *count*, que é zero, mas não tem tempo de executar *sleep*, pois o escalonador selecionou o produtor;
  - O Produtor produz um item, insere-o no *buffer* e incrementa *count*. Como  $count = 1$ , produtor chama *wakeup* para acordar consumidor;
  - Sinal *wakeup* não tem efeito (é perdido), pois o consumidor ainda não está dormindo;
  - Consumidor ganha a CPU, executa *sleep* e vai dormir;
  - Produtor ganha a CPU e, cedo ou tarde, encherá o *buffer*, indo também dormir;
  - Ambos dormirão eternamente.

Problema:

- Lost wakeup

# Soluções para exclusão mútua

---

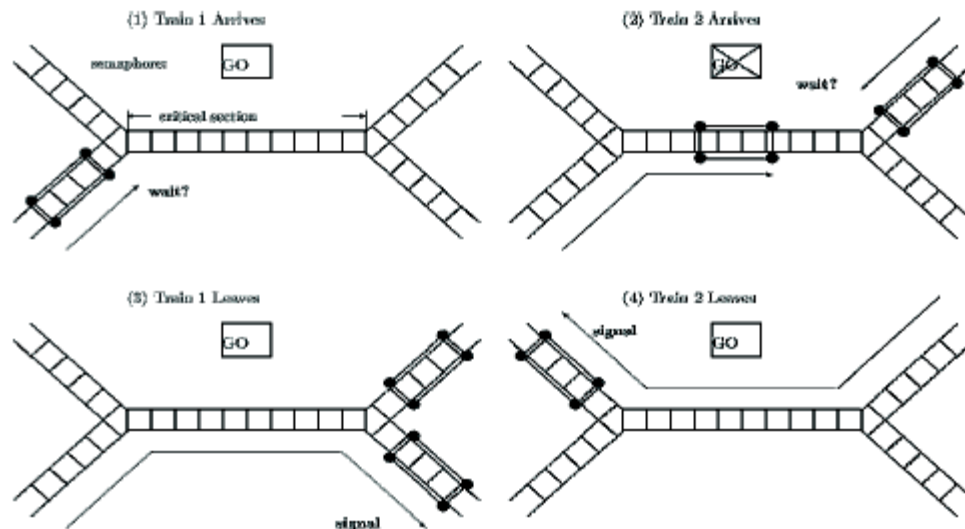
Espera Ocupada

Primitivas *Sleep/Wakeup*

Semáforos

# Semáforo

- Mecanismo para sincronização e sinalização entre processos
- Voltado ao compartilhamento de recursos e dependência entre processos
- Proposto por Dijkstra (1965)
- Inspiração: sinais de trens



# Semáforos: definição

---

- variável inteira não negativa que pode ser manipulada por duas operações P (down/wait) e V (up/signal)
  - As modificações feitas no valor do semáforo usando Down e Up são **atômicas**
  - Para exclusão mútua, as operações Down e Up funcionam como protocolos de entrada e saída de SCs
    - Down é executada quando o processo deseja entrar na SC - Decrementa o semáforo de 1
    - Up é executada quando o processo sai de sua SC – Incrementa o semáforo em 1
-

# Semáforos: primitivas

---

## Primitiva **P** (*down/wait*)

- solicita acesso à SC associada ao semáforo **s**
  - livre → processo continua sua execução
  - não livre → processo é suspenso e adicionado à fila do semáforo; contador do semáforo é decrementado

```
P(s): s.valor = s.valor - 1;  
      se s.valor < 0 {  
        Bloqueia processo (sleep);  
        Insere processo em s.fila;  
      }
```

## Primitiva **V** (*up/signal*):

- libera a SC associada ao semáforo **s**
- contador associado ao semáforo é incrementado
  - se a fila do semáforo não está vazia → 1º processo da fila é acordado, sai da fila e volta à fila de prontos

```
V(s): s.valor = s.valor + 1  
      se s.valor <= 0 {  
        Retira processo de s.fila;  
        Acorda processo (wakeup);  
      }
```

# Semáforos: implementação

---

## Semaphore Structure:

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

## Wait Operation:

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

## Signal Operation:

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

# Semáforos: considerações

---

- suspender processos que aguardam acesso à SC elimina espera ocupada
  - uso mais eficiente do processador
- fila associada ao semáforo
  - contém todos os processos suspensos
  - obedece política FIFO → justiça no acesso à SC
- valor associado ao semáforo → contador de recursos
  - positivo → nº de instâncias do recurso disponíveis
  - negativo → nº de processos aguardando pelo recurso
  - valor inicial → permite expressar diferentes situações de sincronização



# Semáforos em C

---

- Biblioteca semaphore.h
- Declarações e uso:
  - sem\_t semaphore
  - sem\_init (&semaphore, 0, some\_value);
  - sem\_wait(&semaphore);
  - sem\_post(&semaphore);
- Obs.: em Pthreads usa-se:
  - wait no lugar de down; post no lugar de up

# Produtor/consumidor usando semáforos (1)

---

- Solução usa 3 semáforos:
  - Full: conta nº de posições ocupadas no buffer; inicializado c/ 0
  - Empty: conta nº de posições vazias no buffer; inicializado c/ nº total de posições do buffer (N)
  - Mutex: garante que produtor e consumidor não acessem o buffer ao mesmo tempo; inicializado c/ 1 (semáforo binário)

# Produtor/consumidor usando semáforos (2)

```
# define N 5
```

```
typedef int semaphore;  
semaphore mutex = 1;  
semaphore empty = N;  
semaphore full = 0;
```

```
void producer (){  
    while (true){  
        produce_item();  
        down(empty);  
        down(mutex);  
        insert_item();  
        up(mutex);  
        up(full);  
    }  
}
```

```
void consumer (){  
    while (true){  
        down(full);  
        down(mutex);  
        remove_item();  
        up(mutex);  
        up(empty);  
        consume_item();  
    }  
}
```

# Produtor/consumidor usando semáforos (3)

---

Ver código: [produtor-consumidor-semaforo.c](#)

# Semáforos: problemas

---

- programador deve definir os pontos de sincronização no programa
  - eficaz p/ programas pequenos e problemas de sincronização simples
  - inviável p/ sistemas mais complexos
  - Ex.:
    - a) programador esquece de liberar um semáforo previamente alocado → programa pode entrar em *deadlock*
    - b) programador esquece de requisitar um semáforo → exclusão mútua sobre um recurso pode ser violada

# Problemas clássicos de sincronização

---

- Problema do Produtor/Consumidor
- Problema dos Filósofos Jantadores
- Problema dos Leitores/Escritores

# Problema do Produtor/Consumidor

---

- Problema
  - não deixar o produtor inserir dados em um *buffer cheio*
  - não deixar o consumidor retirar dados de um *buffer vazio*
- Apresentadas 2 soluções:
  - Solução 1: `sleep()` e `wakeup()`
    - Condição de corrida
  - Solução 2: semáforos

# Problema dos Filósofos Jantadores

---

- 5 filósofos desejam comer espaguete
- para comer, cada filósofo precisa de dois garfo
- logo, os filósofos precisam compartilhar o uso do garfo de forma sincronizada
- os filósofos alternam entre comer e pensar





# Problema dos Filósofos Jantadores: solução com semáforos (1)

---

```
#define N          5          /* número de filósofos */
#define PENSANDO   0          /* filósofo pensando */
#define FAMINTO    1          /* filósofo tentando pegar garfos */
#define COMENDO    2          /* filósofo comendo */

#define ESQ        (i+1)%N
#define DIR        (i-1+N)%N

int estado[N];               /* vetor (compartilhado) p/ controlar estados */
Semaforo mutex = 1;          /* exclusão mútua para região crítica */
Semaforo s[N];               /* 1 semáforo por filósofo. Inicialmente, todos == 0 */

void filosofo(int i)
{
    while (true) {
        pensa();
        pega_garfos(i);
        come();
        devolve_garfos(i);
    }
}
```

---

# Problema dos Filósofos Jantadores: solução com semáforos (2)

---

```
void pega_garfos(int i)
{
    down(mutex);                /* entra na região crítica */
    estado[i] = FAMINTO;        /* filósofo i está com fome */
    teste(i);                   /* tenta mudar de estado para comendo */
    up(mutex);                  /* sai da região crítica */
    down(s[i]);                 /* fica bloqueado se não conseguiu mudar de estado */
}

void devolve_garfos(int i)
{
    down(mutex);                /* entra na região crítica */
    estado[i] = PENSANDO;       /* filósofo i terminou de comer */
    teste(DIR);                 /* verifica se vizinho da direita pode comer */
    teste(ESQ);                 /* verifica se vizinho da esquerda pode comer */
    up(mutex);                  /* sai da região crítica */
}

void teste(int i)
{
    if (estado[i] == FAMINTO && estado[ESQ] != COMENDO && estado[DIR] != COMENDO) {
        estado[i] = COMENDO;
        up(s[i]);
    }
}
```

---

## Problema dos Filósofos Jantadores: solução com semáforos (3)

---

Problemas que devem ser evitados:

*Deadlock* – todos os filósofos pegam um garfo ao mesmo tempo

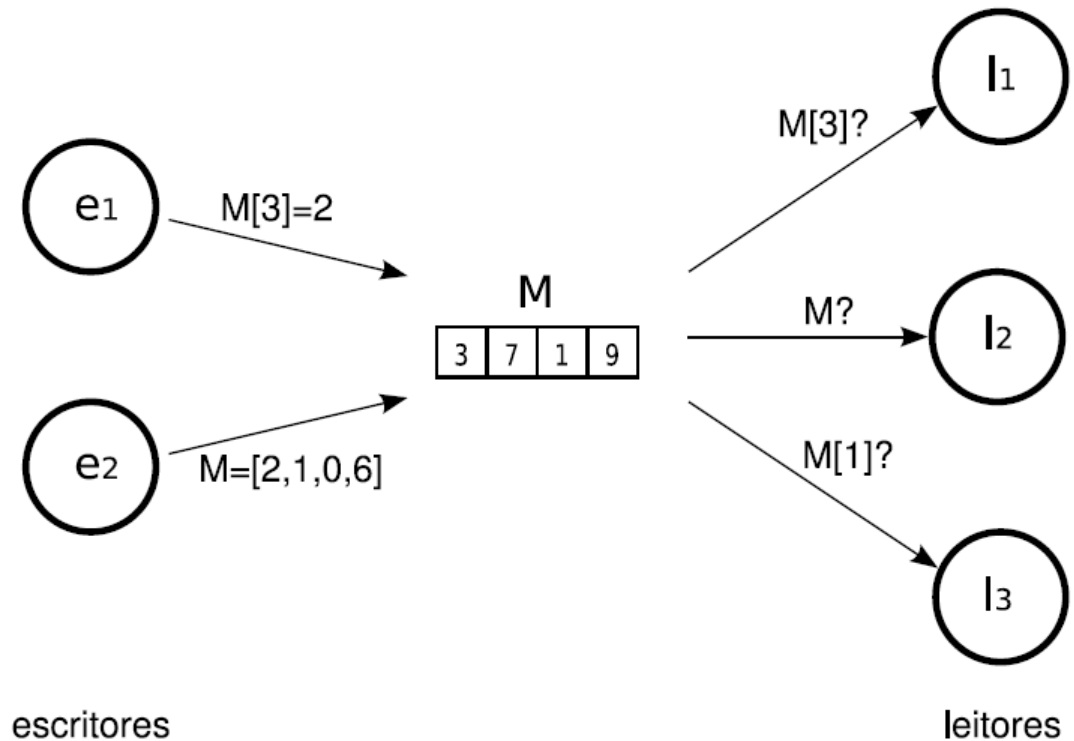
*Starvation* – os filósofos ficam indefinidamente pegando garfos simultaneamente

Ver código: [filosofos.c](#)

# Problema dos Leitores/Escritores (1)

---

- processos leitores e escritores competem pelo acesso a uma base de dados
- vários leitores podem acessar a base ao mesmo tempo
- quando um escritor está na base de dados, nenhum outro processo pode acessá-la (nem mesmo um leitor)



# Problema dos Leitores/Escretores (2)

---

```
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 */
    }
}
```

---

# Problema dos Leitores/Escritores (3)

---

```
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 */
    }
}
```

Ver código: leit-escri.c