

# Sistemas Operacionais

Interação entre tarefas - problemas de coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Agosto de 2020

# Conteúdo

**1** Produtores/consumidores

**2** Leitores/escritores

**3** O jantar dos filósofos

# Problemas clássicos de coordenação

Retratam situações de coordenação típicas em muitos sistemas.

Descrevem soluções genéricas e eficientes de problemas usuais.

Exemplos:

- Produtores/consumidores
- Leitores/escritores
- O barbeiro dorminhoco
- Jantar dos filósofos

**Leitura:** *The Little Book of Semaphores*, Allen B. Downey

# Produtores/consumidores

Buffer compartilhado com capacidade para  $N$  itens.

Acesso concorrente por dois tipos de tarefas:

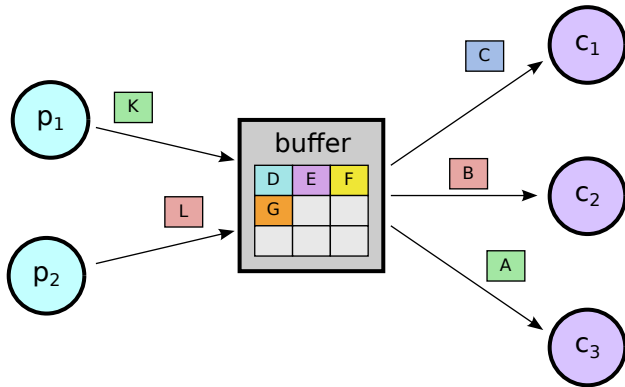
## Produtor:

- produz e deposita itens no buffer
- se o buffer estiver cheio, espera uma vaga

## Consumidor:

- retira e consome itens do buffer
- se o buffer estiver vazio, espera um item

# Produtores/consumidores



Parece a comunicação por *mailbox*?

# Produtores/consumidores

Há três aspectos de coordenação a resolver:

- Exclusão mútua no acesso ao buffer
- Bloqueio dos produtores na falta de vagas
- Bloqueio dos consumidores na falta de itens

A solução usa **três semáforos**:

- **mutex** controla exclusão mútua do buffer (inicia em 1)
- **item** controla itens disponíveis (inicia em 0)
- **vaga** controla vagas disponíveis (inicia em  $N$ )

# Produtores/consumidores

Comportamento básico das tarefas:

## **Produtor**

```
{  
    while (true)  
    {  
        produz_item ;  
  
        coloca_item ;  
  
    }  
}
```

## **Consumidor**

```
{  
    while (true)  
    {  
  
        retira_item ;  
  
        consome_item ;  
    }  
}
```

# Produtores/consumidores

Coordenação do acesso ao buffer: **mutex**

## Produtor

```
{  
    while (true)  
    {  
        produz_item ;  
  
        sem_down (mutex) ;  
        coloca_item ;  
        sem_up (mutex) ;  
    }  
}
```

## Consumidor

```
{  
    while (true)  
    {  
  
        sem_down (mutex) ;  
        retira_item ;  
        sem_up (mutex) ;  
  
        consome_item ;  
    }  
}
```



# Produtores/consumidores

Coordenação de itens: **item**

## Produtor

```
{  
    while (true)  
    {  
        produz_item ;  
  
        sem_down (mutex) ;  
        coloca_item ;  
        sem_up (mutex) ;  
        sem_up (item) ;  
    }  
}
```

## Consumidor

```
{  
    while (true)  
    {  
        sem_down (item) ;  
        sem_down (mutex) ;  
        retira_item ;  
        sem_up (mutex) ;  
  
        consome_item ;  
    }  
}
```

# Produtores/consumidores

Coordenação de vagas: **vaga**

## Produtor

```
{  
    while (true)  
    {  
        produz_item ;  
        sem_down (vaga) ;  
        sem_down (mutex) ;  
        coloca_item ;  
        sem_up (mutex) ;  
        sem_up (item) ;  
    }  
}
```

## Consumidor

```
{  
    while (true)  
    {  
        sem_down (item) ;  
        sem_down (mutex) ;  
        retira_item ;  
        sem_up (mutex) ;  
        sem_up (vaga) ;  
        consome_item ;  
    }  
}
```

# Solução com variáveis condicionais

```
1  mutex mbuf ;           // controla o acesso ao buffer
2  condition item ;       // condição: existe item no buffer
3  condition vaga ;       // condição: existe vaga no buffer
```

```
1  task produtor ()
2  {
3      while (1)
4      {
5          ...             // produz um item
6          lock (mbuf) ;    // obtém o mutex do buffer
7          while (num_itens == N) // enquanto o buffer estiver cheio
8              wait (vaga, mbuf) ; // espera uma vaga, liberando o buffer
9          ...             // deposita o item no buffer
10         signal (item) ;  // sinaliza um novo item
11         unlock (mbuf) ;  // libera o buffer
12     }
13 }
```

# Solução com variáveis condicionais

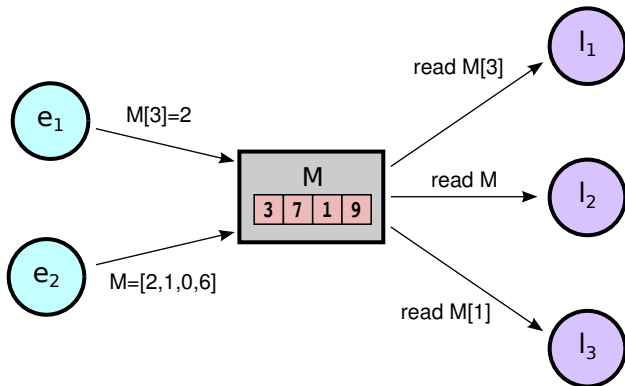
```
1 task consumidor ()
2 {
3     while (1)
4     {
5         lock (mbuf) ;           // obtem o mutex do buffer
6         while (num_items == 0) // enquanto o buffer estiver vazio
7             wait (item, mbuf) ; // espera um item, liberando o buffer
8         ...                     // retira o item no buffer
9         signal (vaga) ;         // sinaliza uma vaga livre
10        unlock (mbuf) ;         // libera o buffer
11        ...                     // consome o item retirado do buffer
12    }
13 }
```

# Leitores/escritores

Um conjunto de tarefas acessam uma área de memória compartilhada na qual fazem leituras ou escritas:

- As leituras podem ser feitas simultaneamente
  - A leitura não altera os dados
  - $N$  leitores podem acessar juntos
- As escritas têm de ser feitas com exclusão mútua
  - A escrita altera os dados
  - Um só escritor pode acessar por vez
  - Leitores não podem acessar durante a escrita

# Leitores/escritores



# Solução trivial

Comportamento básico das tarefas:

## **Leitor**

```
{  
  while (true)  
  {  
  
    lê_dados ;  
  
    trata_dados_lidos ;  
  }  
}
```

## **Escritor**

```
{  
  while (true)  
  {  
  
    obtém_dados ;  
  
    escreve_dados ;  
  }  
}
```

# Solução trivial

Coordenação do acesso ao buffer: **mutex**

**Leitor**

```
{  
  while (true)  
  {  
    sem_down (mutex) ;  
    lê_dados ;  
    sem_up (mutex) ;  
    trata_dados_lidos ;  
  }  
}
```

**Escritor**

```
{  
  while (true)  
  {  
    obtém_dados ;  
    sem_down (mutex) ;  
    escreve_dados ;  
    sem_up (mutex) ;  
  }  
}
```

**Problema:** os leitores não podem trabalhar juntos!



# Solução com leitores simultâneos

leitores : contador de leitores na área (0)

**Leitor ao entrar:**

```
leitores++ ;  
if (leitores == 1)  
    sem_down (mutex) ;
```

**Leitor ao sair:**

```
leitores-- ;  
if (leitores == 0)  
    sem_up (mutex) ;
```

O código do escritor não muda.

# Solução com leitores simultâneos

Precisa de um *mutex* para proteger o contador: **m\_cont**

**Leitor ao entrar:**

```
sem_down (m_cont) ;  
leitores++ ;  
if (leitores == 1)  
    sem_down (mutex) ;  
sem_up (m_cont) ;
```

**Leitor ao sair:**

```
sem_down (m_cont) ;  
leitores-- ;  
if (leitores == 0)  
    sem_up (mutex) ;  
sem_up (m_cont) ;
```

O código do escritor não muda.

# Solução com leitores simultâneos

Leitor

```
{
  while (true)
  {
    sem_down (m_cont) ;
    leitores++ ;
    if (leitores == 1)
      sem_down (mutex) ;
    sem_up (m_cont) ;

    lê_dados ;

    sem_down (m_cont) ;
    leitores-- ;
    if (leitores == 0)
      sem_up (mutex) ;
    sem_up (m_cont) ;

    trata_dados_lidos ;
  }
}
```

**mutex:** *mutex* do buffer

**leitores:** contador de  
leitores ativos

**m\_cont:** *mutex* do contador

Esta solução prioriza os  
leitores. **Por que?**

# O Jantar dos filósofos



# O Jantar dos filósofos

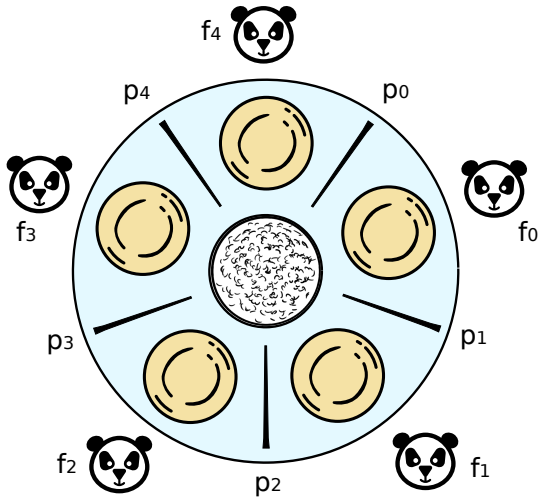
Cinco filósofos com uma vida bem simples...

- alternam entre meditar e comer
- usam uma mesa com 5 lugares
- cada um tem seu prato
- os 5 palitos são compartilhados

Envolve várias tarefas e vários recursos.

Não há um coordenador central.

# Modelagem



# Restrições

- Antes de comer, o filósofo  $f_i$  deve pegar os palitos da direita ( $p_i$ ) e da esquerda ( $p_{i+1}$ ), um de cada vez
- Após comer, o filósofo devolve os palitos a seus lugares
- Como os palitos são compartilhados, dois filósofos vizinhos não podem comer ao mesmo tempo
- Os filósofos não conversam entre si, nem conhecem os estados uns dos outros
- Não há um coordenador central

# Solução básica

```

1  #define NUMFILO 5
2  semaphore hashi [NUMFILO] ; // palitos são semáforos (iniciam em 1)
3
4  task filosofo (int i)          // filósofo i (entre 0 e 4)
5  {
6      int dir = i ;
7      int esq = (i+1) % NUMFILO ;
8
9      while (1)
10     {
11         meditar () ;
12         down (hashi [dir]) ;      // pega palito direito
13         down (hashi [esq]) ;      // pega palito esquerdo
14         comer () ;
15         up (hashi [dir]) ;        // devolve palito direito
16         up (hashi [esq]) ;        // devolve palito esquerdo
17     }
18 }
  
```



# Problema: risco de impasse!

