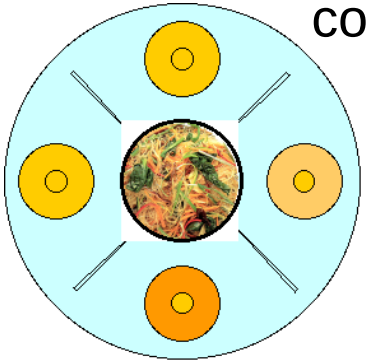
Sistemas Operacionais

Problemas Clássicos de Coordenação/Sincronização

Problemas Clássicos de Coordenação

Barbeiro dorminhoco

Filósofos comensais

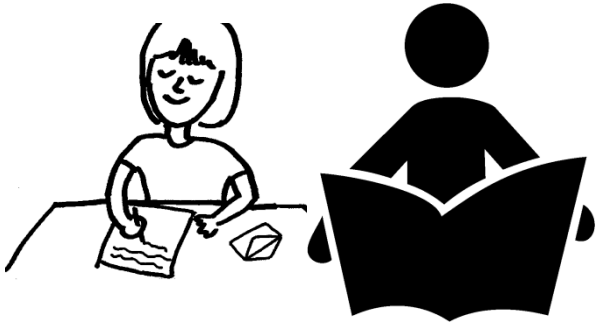


Produtor

Consumidor

Escritor

Leitor



Índice

- Exemplos de utilização de semáforos
  - Produtor – Consumidor
  - Leitor – Escritor
  - Filósofos comensais
    - Impasse
  - Outras aplicações de semáforos
    - Barreiras e Ordenação (*Grafos de Precedência*)

Problemas Clássicos de Coordenação/Sincronização

- Situações de coordenação e sincronização ocorrem com muita frequência na programação de sistemas multitarefa;

Problemas Clássicos de Coordenação/Sincronização

•Situações de coordenação e sincronização ocorrem com muita frequência na programação de sistemas multitarefa;

•Os *problemas clássicos de coordenação* **retratam** essas situações e permitem compreender como podem ser implementadas as soluções;

Ex. de Problemas Clássicos:

-*produtores/consumidores;*
-o problema dos *leitores/escritores;*
-e o *jantar dos filósofos.*

Problemas Clássicos de Coordenação/Sincronização

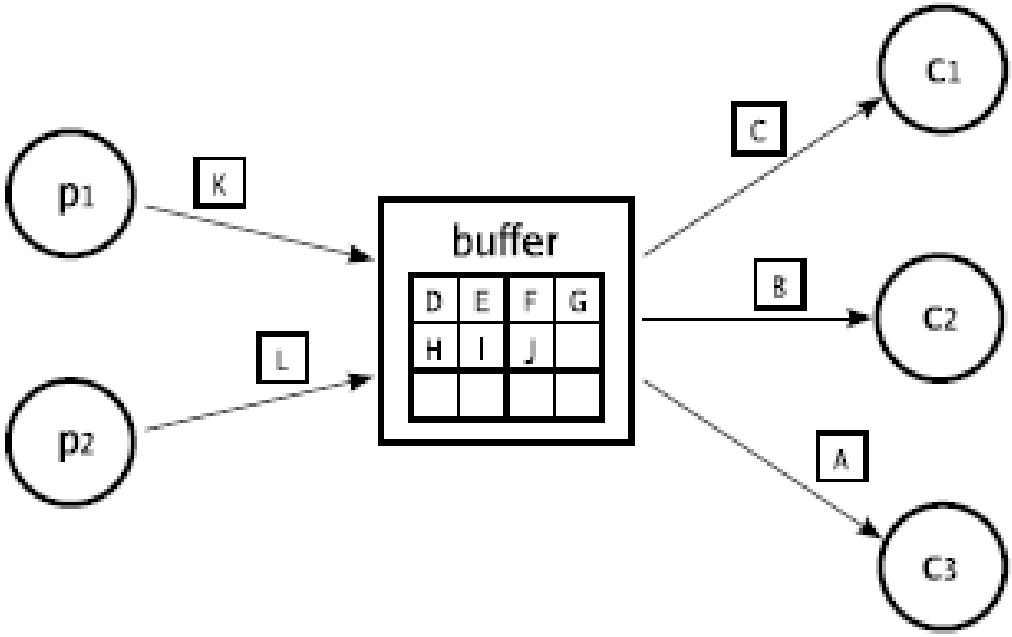
Vamos utilizar semáforos para implementar as soluções dos problemas clássicos!

1 - Produtores/Consumidores

1 - Produtores/Consumidores

•Também conhecido como o problema do *buffer limitado.*

•Consiste em coordenar o acesso de tarefas a um buffer compartilhado com capacidade de armazenamento limitada a ***N*** itens.



Lembram das Mailboxes?

1 - Produtores/Consumidores

•Também conhecido como o problema do *buffer limitado.*

•Consiste em coordenar o acesso de tarefas a um buffer compartilhado com capacidade de armazenamento limitada a ***N*** itens.

•São considerados dois tipos de processos com comportamentos simétricos:

-**Produtor**
-**Consumidor**

1 - Produtores/Consumidores

Produtor : periodicamente produz e deposita um item no buffer, caso este tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga.

1 - Produtores/Consumidores

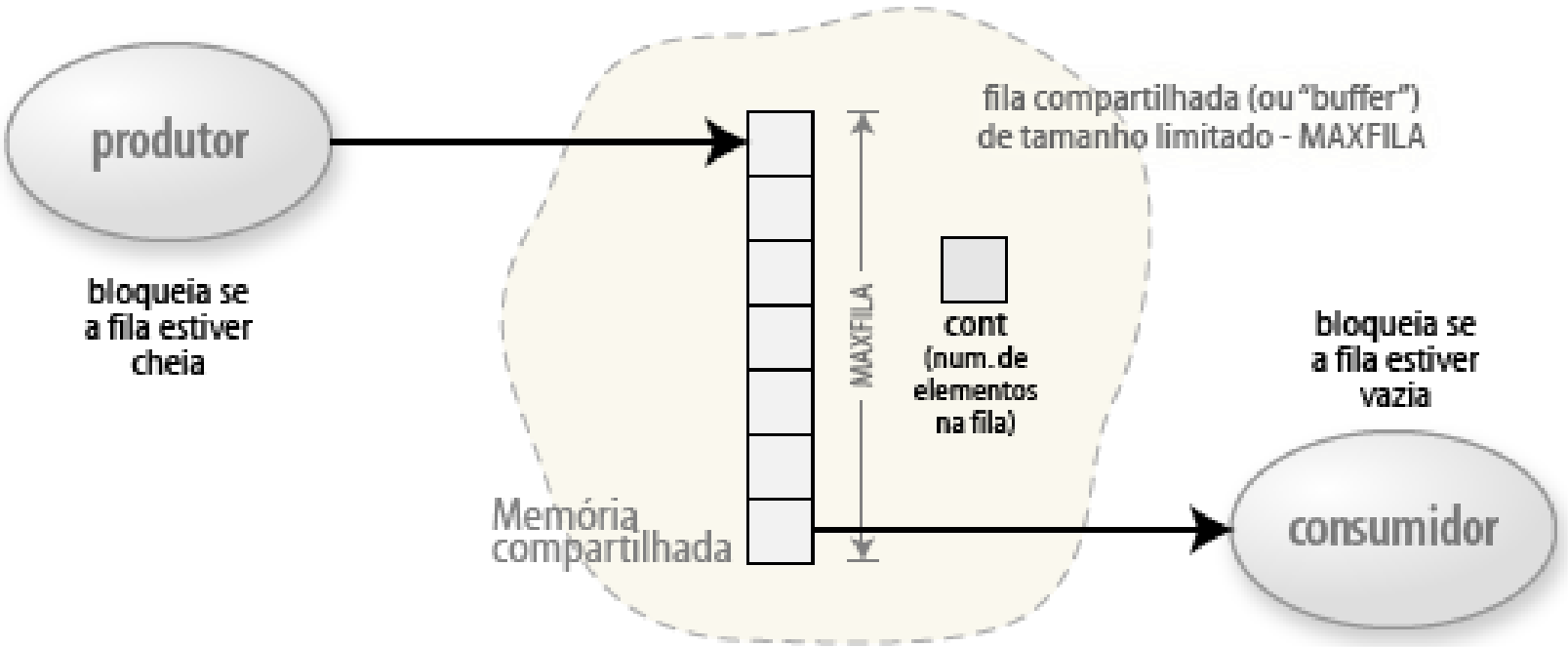
Produtor : periodicamente produz e deposita um item no buffer, caso este tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga.

Consumidor : continuamente retira um item do buffer e o consome. Caso o buffer esteja vazio aguarda que novos itens sejam depositados pelos produtores.

1 - Produtores/Consumidores

Obs.: O acesso ao buffer é bloqueante, ou seja, cada processo fica bloqueado (***suspenso***) até conseguir fazer seu acesso, seja para depositar ou para consumir um item.

1 - Produtores/Consumidores



Atenção: Execução Concorrente

```
...
while (1)
    {
        PARBEGIN;
            consumidor() ;
            produtor() ;
        PAREND;
    }
}
```

Solução

3 aspectos de coordenação/sincronização:

•A exclusão mútua no acesso ao buffer → evita condições de disputa entre produtores e/ou consumidores.

Solução

3 aspectos de coordenação/sincronização:

•A exclusão mútua no acesso ao buffer → evita condições de disputa entre produtores e/ou consumidores.

• O bloqueio dos *produtores,* caso o buffer esteja cheio: os produtores devem aguardar vagas *livres* no buffer.

Solução

3 aspectos de coordenação/sincronização:

•A exclusão mútua no acesso ao buffer → evita condições de disputa entre produtores e/ou consumidores.

• O bloqueio dos *produtores,* caso o buffer esteja cheio: os produtores devem aguardar vagas *livres* no buffer.

• O bloqueio dos *consumidores,* no caso de o buffer estar *vazio*: os consumidores devem aguardar novos itens a consumir no buffer.

Solução

A solução para esse problema exige **três** semáforos: um para atender cada aspecto de coordenação acima descrito.

Solução

```
sem_t mutex ; // controla o acesso ao buffer (inicia em 1)
sem_t item ;  // número de itens no buffer (inicia em 0)
sem_t vaga ;  // número de vagas livres no buffer (inicia em N)
```

Solução

```
sem_t mutex ; // controla o acesso ao buffer (inicia em 1)
sem_t item ;  // número de itens no buffer (inicia em 0)
sem_t vaga ;  // número de vagas livres no buffer (inicia em N)

produtor ()
{
while (1)
   {
        ... // produz um item
        sem_down(&vaga) ;  // aguarda uma vaga no buffer
        sem_down(&mutex) ; // aguarda acesso exclusivo ao buffer
        ... // deposita o item no buffer
        sem_up(&mutex) ;   // libera o acesso ao buffer
        sem_up(&item) ;    // indica presença de novo item no buffer
   }
}
```

Solução

```
consumidor () {
while (1)
    {
        sem_down(&item) ;    // aguarda um novo item no buffer
        sem_down(&mutex) ; // aguarda acesso exclusivo ao buffer
        ... // retira o item do buffer
        sem_up(&mutex) ;     // libera o acesso ao buffer
        sem_up(&vaga) ;      // indica liberação de vaga no buffer
        ... // consome o item retirado do buffer
    }
}
```

Tarefas **NÃO** podem ser suspensas ao executar **up(s)**

2 - Leitores/escritores

2 - Leitores/escritores

•Um conjunto de tarefas acessam de forma **concorrente** uma área de memória comum (**compartilhada**) na qual podem fazer leituras ou escritas de valores (*variáveis*);

2 - Leitores/escritores

•Um conjunto de tarefas acessam de forma **concorrente** uma área de memória comum (**compartilhada**) na qual podem fazer leituras ou escritas de valores (*variáveis*);

•As leituras podem ser feitas simultaneamente, pois não interferem umas com as outras;

2 - Leitores/escritores

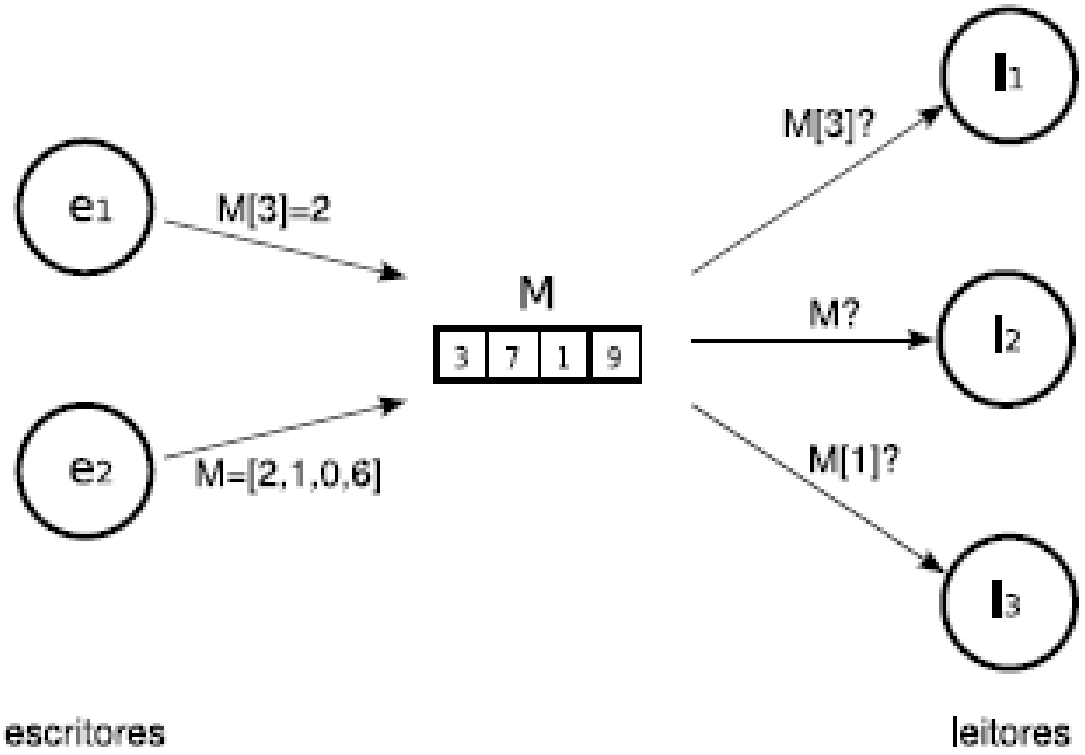
•Um conjunto de tarefas acessam de forma **concorrente** uma área de memória comum (**compartilhada**) na qual podem fazer leituras ou escritas de valores (*variáveis*);

•As leituras podem ser feitas simultaneamente, pois não interferem umas com as outras;

•As escritas têm de ser feitas com acesso exclusivo à área compartilhada, para evitar condições de disputa.

2 - Leitores/escritores: Exemplo

Leitores e escritores acessam de forma concorrente uma matriz de inteiros *M.*



Atenção: Execução Concorrente

```
...
while (1)
    {
        PARBEGIN;
            escritor () ;
            leitor () ;
        PAREND;
    }
}
```

Solução

Proteção da área compartilhada com um semáforo inicializado em 1; Somente um processo por vez poderia acessar a área.

```
sem_t mutex_area ;                    // controla o acesso à área (inicia em 1)
```

Solução

Proteção da área compartilhada com um semáforo inicializado em 1; Somente um processo por vez poderia acessar a área.

```
sem_t mutex_area ;                  // controla o acesso à área (inicia em 1)

leitor ()
{
   while (1)
      {
         sem_down (&mutex_area) ;   // requer acesso exclusivo à área
         ...                        // lê dados da área compartilhada
         sem_up (&mutex_area) ;     // libera o acesso à área
         ...
      }
}
```

Solução

```
escritor ()
{
   while (1)
      {
         sem_down (&mutex_area) ; // requer acesso exclusivo à
                                   // área
         ...                       // escreve dados na área
                                   // compartilhada
         sem_up (&mutex_area) ;    // libera o acesso à área
         ...
      }
}
```

Problema

Porque restringe desnecessariamente o acesso dos leitores à área compartilhada.

"As leituras podem ser feitas simultaneamente, pois não interferem umas com as outras."

Nova solução

Nova solução

```
sem_t mutex_area ;          // controla o acesso à área (inicia em 1)
int conta_leitores = 0 ;    // número de leitores acessando a área
sem_t mutex_conta ;         // controla o acesso ao contador (inicia
                            // em 1)
```

Nova solução

```
sem_t mutex_area ;              // controla o acesso à área (inicia em 1)
int conta_leitores = 0 ;        // número de leitores acessando a área
sem_t mutex_conta ;             // controla o acesso ao contador (inicia
                                // em 1)


leitor ()
{
   while (1)
      {
         sem_down (&mutex_conta) ; // requer acesso exclusivo ao
                                   // contador
         conta_leitores++ ;        // incrementa contador de
                                   // leitores
         if (conta_leitores = = 1) // sou o primeiro leitor a entrar?
           sem_down (&mutex_area) ;// requer acesso à área
         sem_up (&mutex_conta) ;   // libera o contador
```

Nova solução

```
        ...                                 // lê dados da área
                                            // compartilhada

        sem_down (&mutex_conta) ; // requer acesso exclusivo ao
                                            // contador
        conta_leitores-- ;                  // decrementa contador de
                                            // leitores
        if (conta_leitores = = 0)           // sou o último leitor a sair?
            sem_up (&mutex_area) ;          // libera o acesso à área
        sem_up (&mutex_conta) ;             // libera o contador
        ...
    }
}
```

Nova solução

```
escritor ()
{
    while (1)
        {
            sem_down(&mutex_area) ;     // requer acesso exclusivo à área
            ...                         // escreve dados na área
                                        // compartilhada
            sem_up(&mutex_area) ;       // libera o acesso à área
            ...
        }
}
```

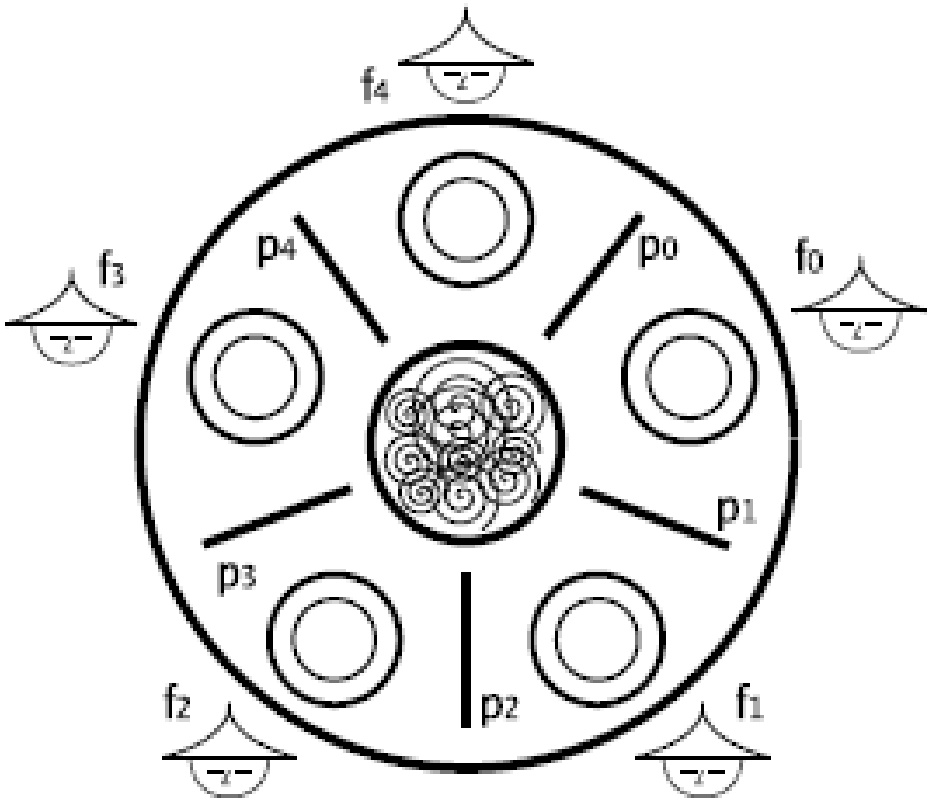
Nova solução

```
escritor ()
{
   while (1)
      {
         sem_down(&mutex_area) ;    // requer acesso exclusivo à área
         ...                        // escreve dados na área
                                    // compartilhada
         sem_up(&mutex_area) ;      // libera o acesso à área
         ...
      }
}
```

⇒ Problema: priorização dos Leitores

3 - O jantar dos filósofos

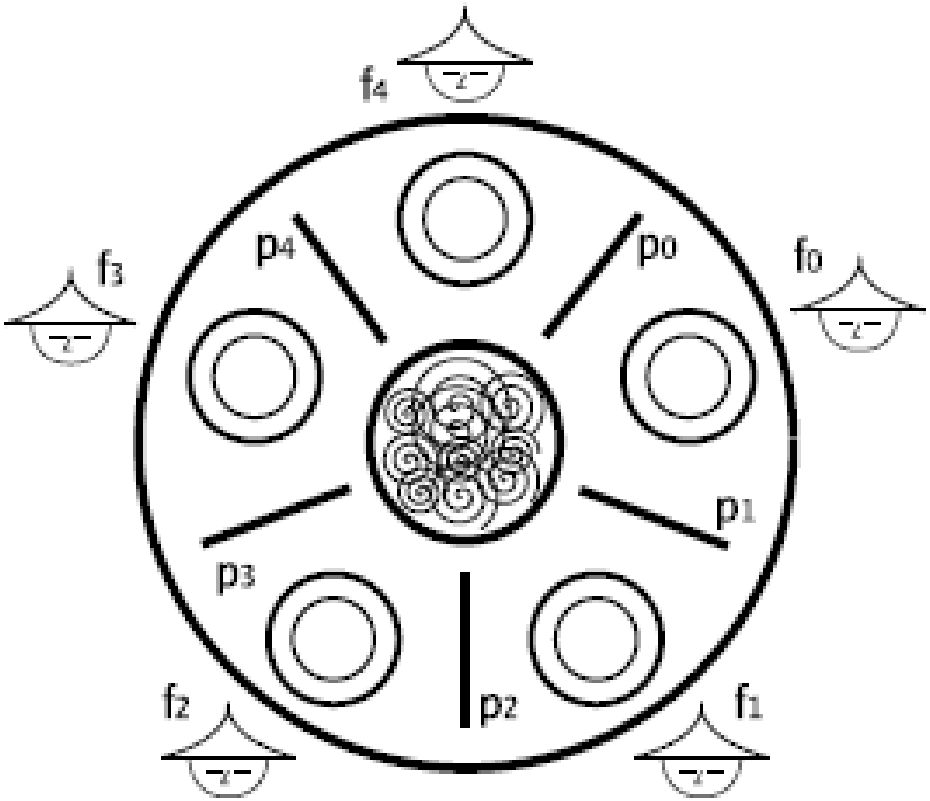
Dijkstra (novamente)



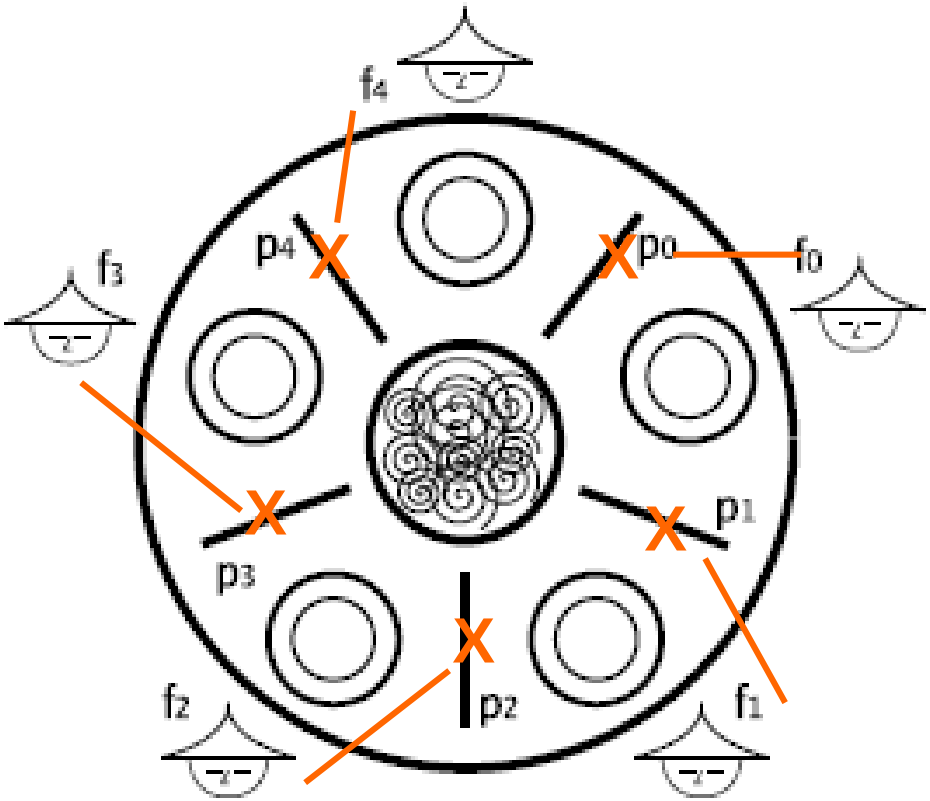
Solução Proposta

```
#define NUMFILO 5
semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)

task filosofo (int i)              // filósofo i (entre 0 e 4)
{
   int dir = i ;
   int esq = (i+1) % NUMFILO ;

   while (1)
   {
      meditar () ;
      down (hashi [dir]) ;         // pega palito direito
      down (hashi [esq]) ;         // pega palito esquerdo
      comer () ;
      up (hashi [dir]) ;           // devolve palito direito
      up (hashi [esq]) ;           // devolve palito esquerdo
   }
}
```

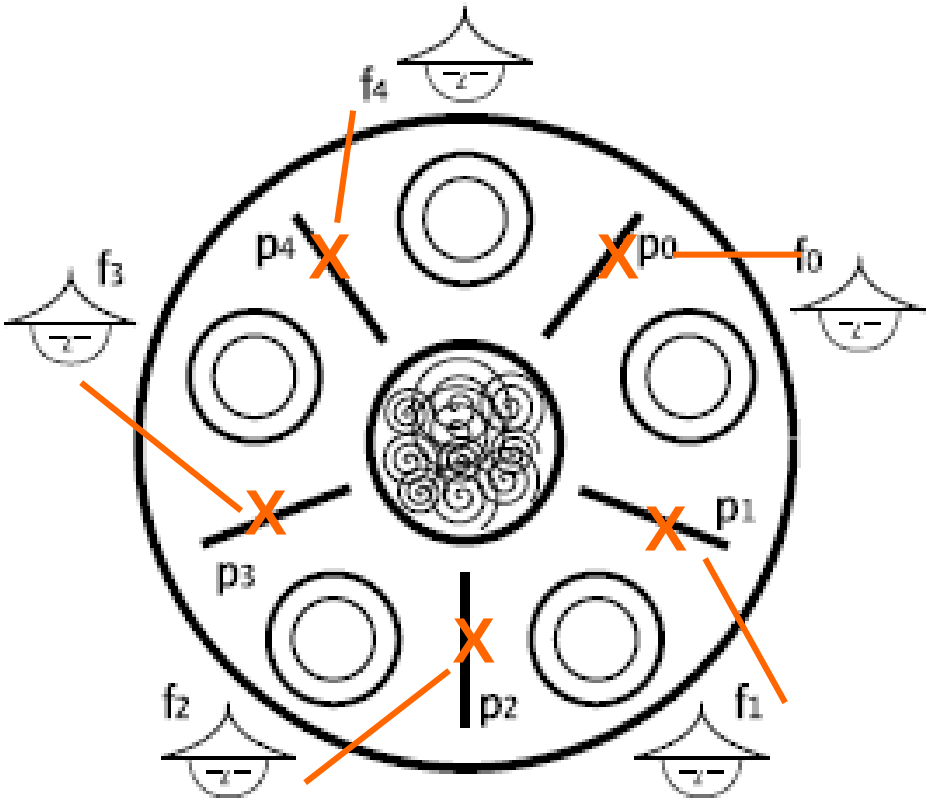
Problema



Problema



Problema

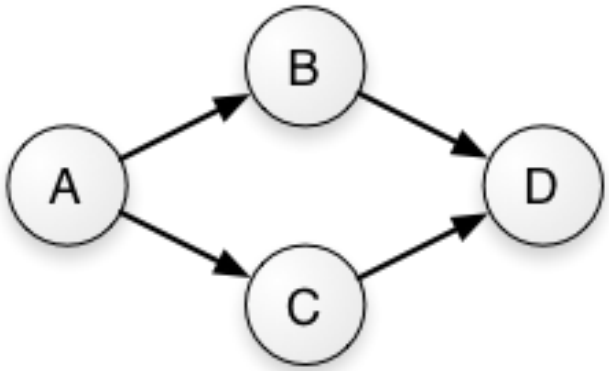


⇒ **IMPASSE (*Deadlock*)**

Outras aplicações de Semáforos

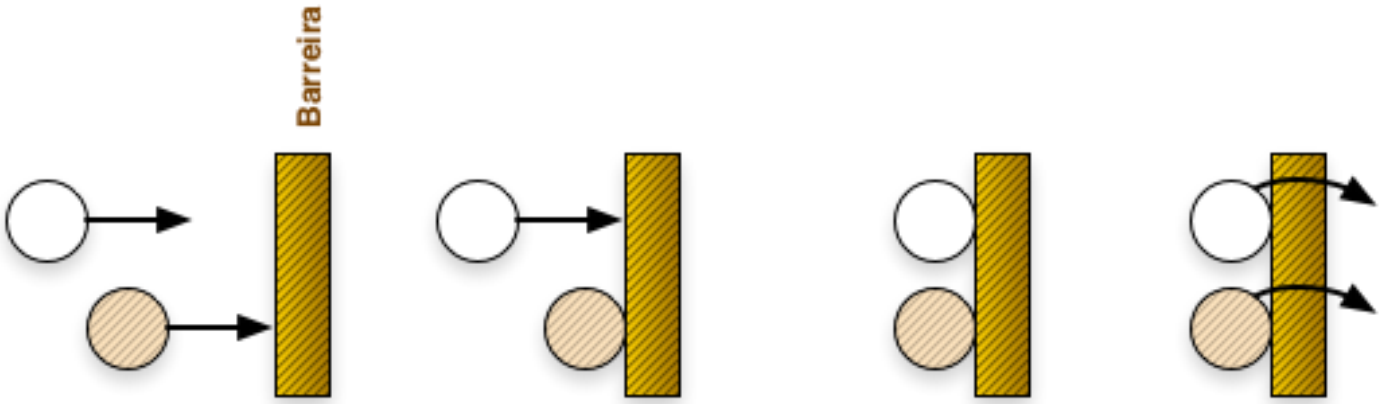
Grafos de Precedência

Grafo de precedência



Outras aplicações de Semáforos

Barreiras



Outras aplicações de Semáforos

Barreiras

Barreiras

```
Process {
        Bool        ÚLTIMO = false;
        Semaphore   BARREIRA; // semáforo contador
        Mutex       M;        // acesso exclusivo a COUNT
        Int         COUNT = N

        Init (& BARREIRA, 0)

        down(&M)
                COUNT--;
                if (COUNT == 0) ÚLTIMO = true;
        up(&M)

        if (NOT ÚLTIMO)  down (& BARREIRA); // espera pelos demais processos
        else for (i=0; i< N; i++)  up (& BARREIRA); // desbloqueia todos
        …
        }
```