COM120 – Sistemas Operacionais

Sincronização e Comunicação entre Processos

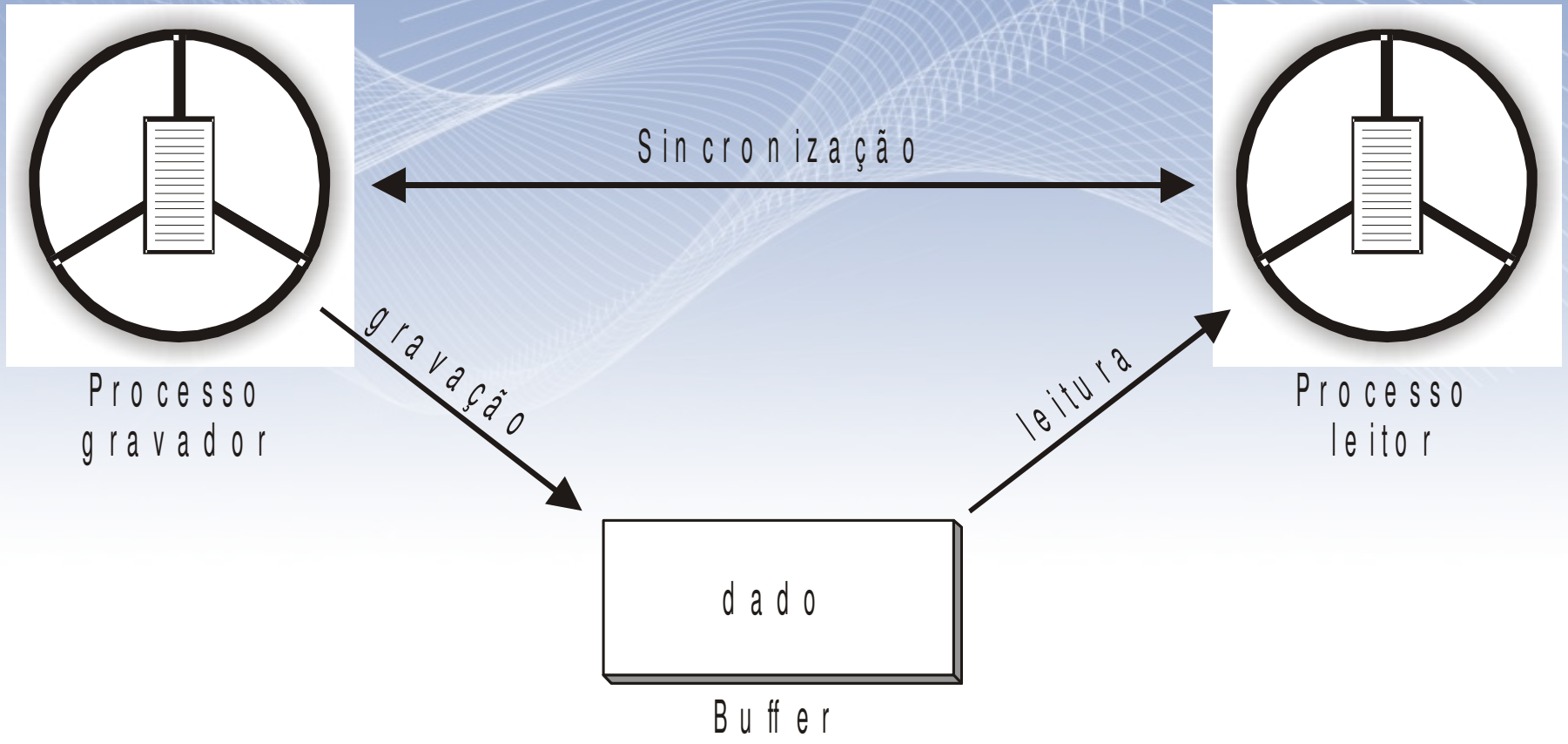
UNIFEI
Universidade Federal de Itajubá
IMC – Instituto de Matemática e Computação

Prof. Carlos Minoru Tamaki

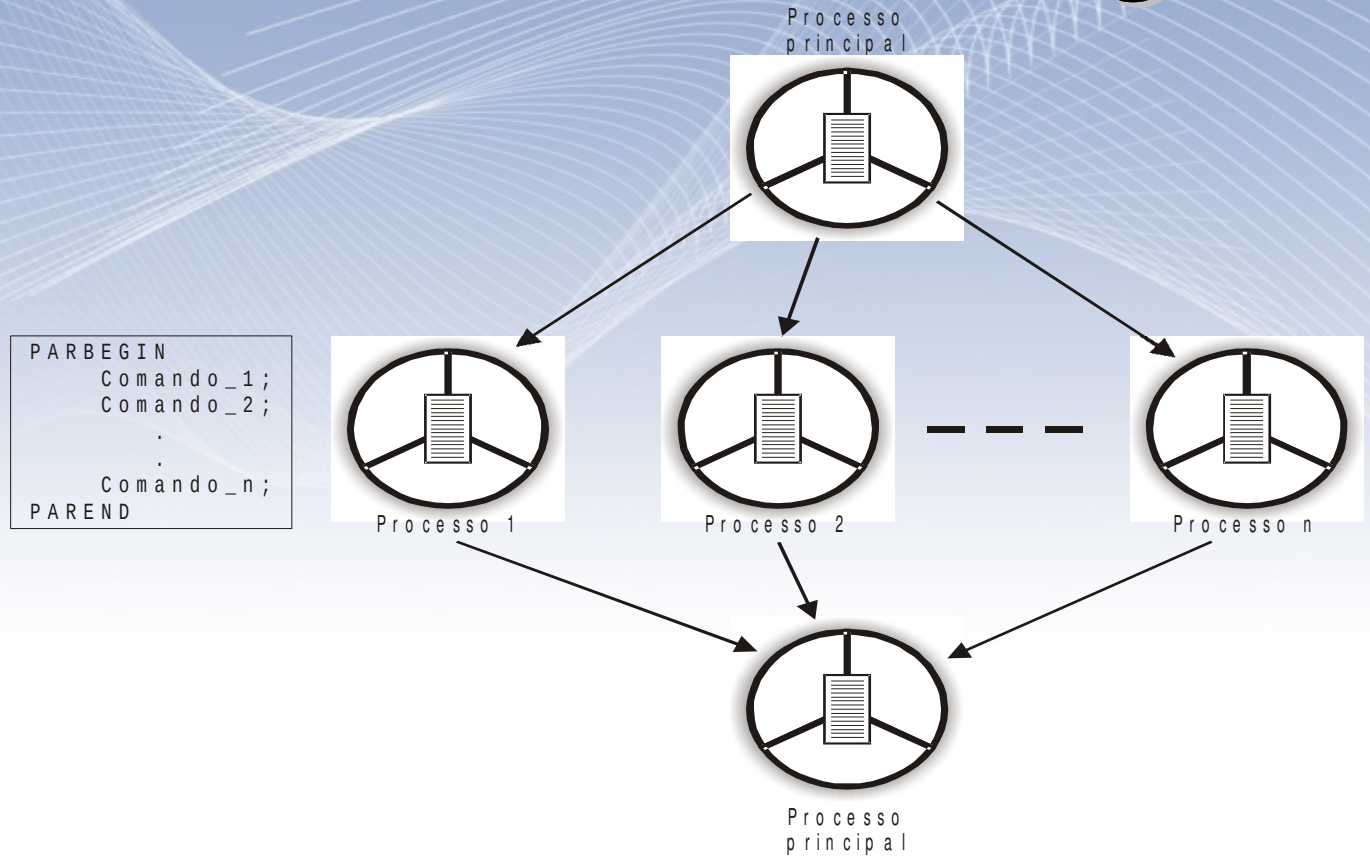
Introdução

- Com o surgimento dos sistemas multiprogramáveis na década de 1960, os código passaram a poder executar concorrentemente
- Esses códigos tem como base a execução cooperativa de múltiplos processos e threads a trabalharem na busca de um resultado comum
- Em sistemas multiprogramáveis com único processador, os processos alternam sua execução, segundo critérios estabelecidos pelo SO

Sincronização e Comunicação



Concorrências em Programas



Processamento paralelo

- **X = SQRT(1024)+(35.4 * 0.23)-(302 / 7)**

```
PROGRAM Expressao;
    VAR X, Temp1, Temp2, Temp3 : REAL;
BEGIN
    PARBEGIN
        Temp1 := SQRT( 1024 );
        Temp2 := 35.4 * 0.23;
        Temp3 := 302 / 7;
    PAREND;
    X := Temp1 + Temp2 – Temp3;
    WRITELN( 'X = ', X );
END.
```

Problemas de compartilhamento de recursos

- O primeiro problema é analisado a partir do programa de Conta_Corrente, que atualiza o saldo após um lançamento de débito ou de crédito:

```
Program conta_corrente;
        :
        READ( Arq_Contas, Reg_Cliente );
        READLN( Valor_Dep_Ret );
        Reg_Cliente.Saldo := Reg_Cliente.Saldo + Valor_Dep_Ret;
        WRITE( Arq_Contas, Reg_Cliente );
        :
END.
```

Condição de Corrida

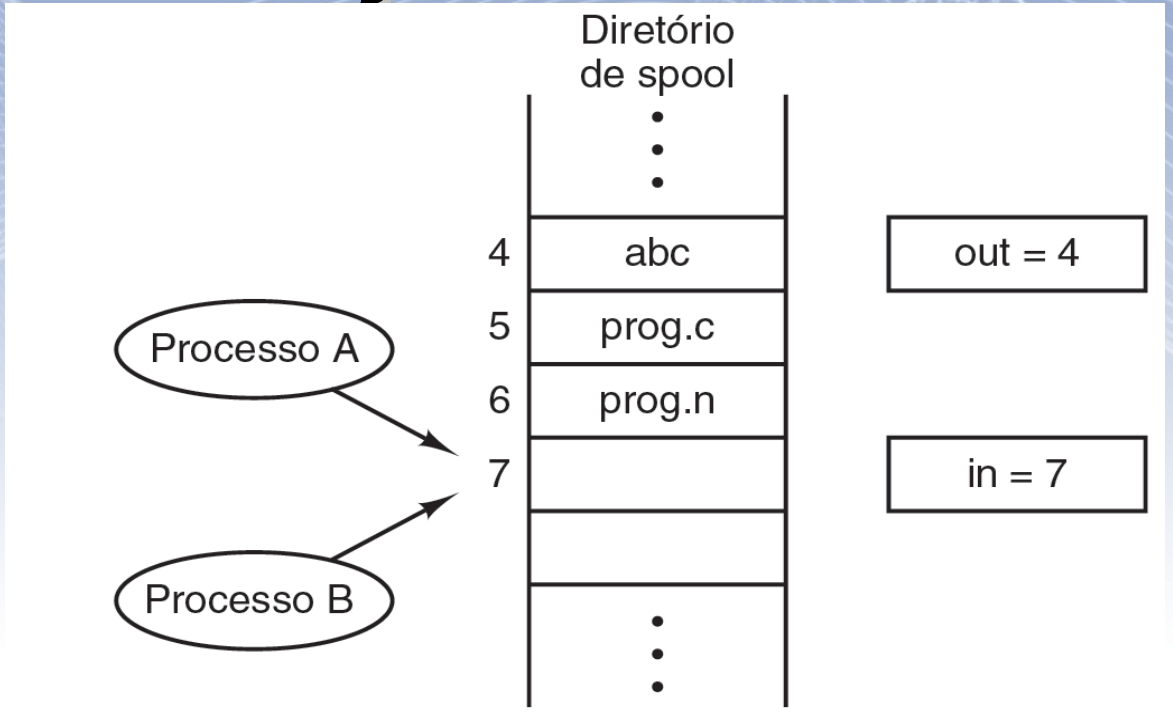


Figura 2.16 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

Regiões Críticas

- Condições necessárias para evitar condições de corrida:
  - Dois processos não podem estar simultaneamente dentro de suas regiões críticas
  - Nada pode ser afirmado sobre a velocidade ou sobre o número de CPU's
  - Nenhum processo sendo executado fora de sua região crítica pode bloquear outros processos
  - Nenhum processo deve esperar eternamente para entrar em sua região crítica

Regiões Críticas

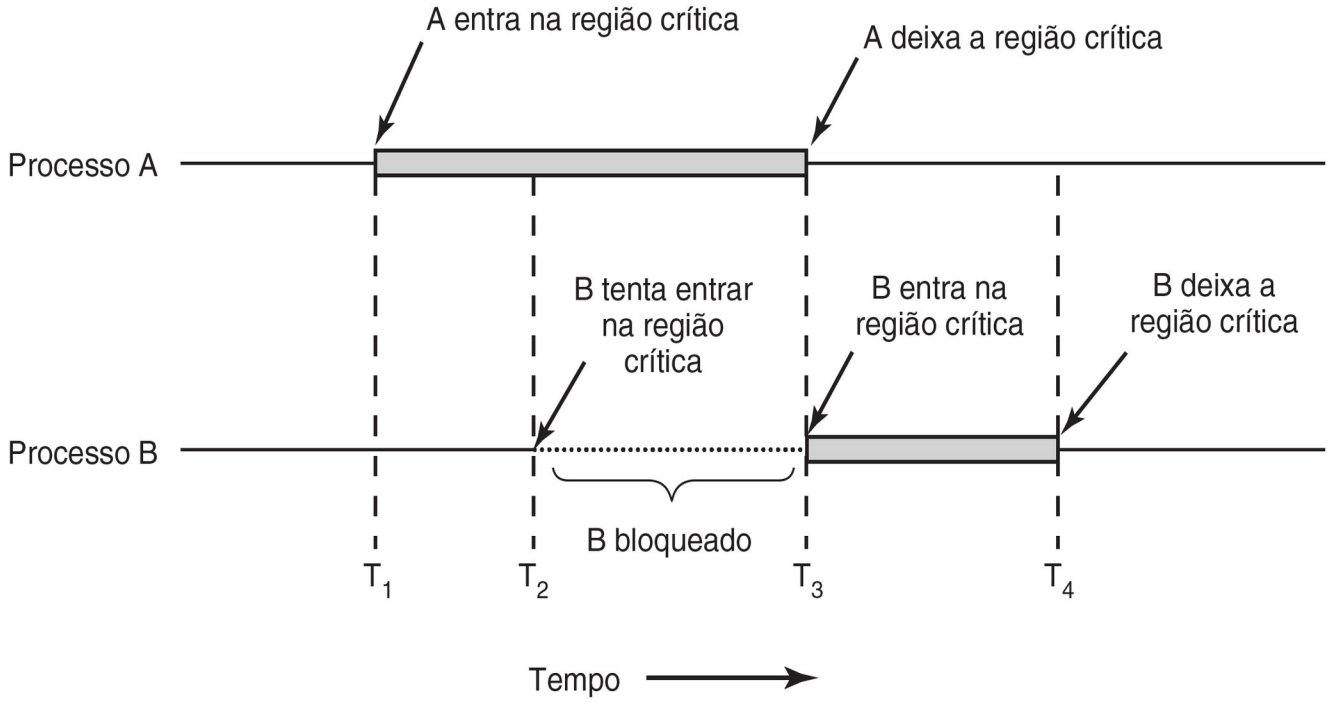


Figura 2.17 Exclusão mútua usando regiões críticas.

Exclusão Mútua

- A solução mais simples para evitar os problemas de compartilhamento é impedir que dois ou mais processos acessem um mesmo recurso simultaneamente.
- A ideia de exclusividade de acesso é chamada exclusão mútua (mutual exclusion).
- Os mecanismos que implementam a exclusão mútua utilizam protocolos de acesso à região crítica

```
BEGIN
        :
        Entra_Região_Crítica; (* Protocolo de entrada *);
        Região_Crítica;
        Sai_Região_Crítica; (* Protocolo de saída *);
        :
END.
```

Soluções de Hardware

- Desabilita interrupções: a solução mais simples é fazer com que o processador desabilite todas as interrupções

```
BEGIN
        :
        Desabilita_Interrupcoes;
        Regiao_Critica;
        Habilita_Interrupcoes;
        :
END.
```

Soluções de Hardware

- Instruções test-and-set, muitos processadores possuem uma instrução de máquina especial que permite ler uma variável, armazenar seu conteúdo em uma outra área e atribuir um novo valor à mesma variável e trata-se de uma instrução indivisível.
- Esta instrução possui o seguinte formato:

  Test-and-Set (X, Y);

- Quando executada o valor lógico da variável Y é copiado para X, sendo atribuído à variável Y o valor lógico verdadeiro.

Exemplo Test_and_Set

```
Program Test_and_Set;
  VAR Bloqueio : BOOLEAN;
Procedure Processo_A;
  VAR Pode_A : BOOLEAN;
BEGIN
  REPEAT
    Pode_A := true;
    WHILE (Pode_A) DO
      Test_and_Set (Pode_A, Bloqueio);
    Regiao_Critica_A;
    Bloqueio := false;
  UNTIL false;
END;
Procedure Processo_B;
  VAR Pode_B : BOOLEAN;
BEGIN
  REPEAT
    Pode_B := true;
    WHILE (Pode_B) DO
      Test_and_Set (Pode_B, Bloqueio);
    Regiao_Critica_B;
    Bloqueio := false;
  UNTIL false;
END;

BEGIN
  Bloqueio := false;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Execução do Programa Test_and_Set

Processo	Instrucão	Pode_A	Pode_B	Bloqueio
A	REPEAT	*	*	false
B	REPEAT	*	*	false
A	Pode_A := true	true	*	false
B	Pode_B := true	*	true	false
B	WHILE	*	true	false
B	Test_and_Set	*	false	true
A	WHILE	true	*	true
A	Test_and_Set	true	*	true
B	Regiao_Critica_B	*	false	true
A	WHILE	true	*	true
A	Test_and_Set	true	*	true
B	Bloqueio := false	*	false	false
B	UNTIL false	*	false	false
B	REPEAT	*	false	false
A	WHILE	true	*	false
A	Test_and_Set	false	*	true
A	Regiao_Critica_A	false	*	true

Soluções de Software – Primeiro Algoritmo

```
PROGRAM Algoritmo_1;
   VAR Vez: CHAR;
PROCEDURE Processo_A;
BEGIN
   REPEAT
      WHILE ( Vez = 'B' ) DO (* não faz
nada *);
      Regiao_Critica_A;
      Vez := 'B';
      Processamento_A;
   UNTIL false;
END;
```

```
PROCEDURE Processo_B;
BEGIN
   REPEAT
      WHILE ( Vez = 'A' ) DO (* não faz nada *);
      Regiao_Critica_B;
      Vez := 'A';
      Processamento_B;
   UNTIL false;
END;

BEGIN
   Vez := 'A';
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

Soluções de Software – Segundo Algoritmo

```
PROGRAM Algoritmo_2;
  VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
  REPEAT
    WHILE (CB) DO (*do nothing *);
    CA := true;
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
  UNTIL false;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    WHILE (CA) DO (*do nothing *);
    CB := true;
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
  UNTIL false;
END;

BEGIN
  CA := false;
  CB := false;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Soluções de Software - Terceiro Algoritmo

```
PROGRAM Algoritmo_3;
  VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
  REPEAT
    CA := true;
    WHILE (CB) DO (*do nothing
*);
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
  UNTIL false;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    CB := true;
    WHILE (CA) DO (*do nothing *);
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
  UNTIL false;
END;
```

```
BEGIN
  CA := false;
  CB := false;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Soluções de Software - Quarto Algoritmo

```
PROGRAM Algoritmo_4;
  VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
  REPEAT
    CA := true;
    WHILE (CB) DO
    BEGIN
      CA := false;
      {pequeno intervalo de tempo aleatório}
      CA := true;
    END;
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
  UNTIL false;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    CB := true;
    WHILE (CA) DO
    BEGIN
      CB := false;
      {pequeno intervalo de tempo
aleatório}
      CB := true;
    END;
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
  UNTIL false;
END;
```

Soluções de Software

- ***Algoritmo de Dekker***

A primeira solução de software que garantiu exclusão mútua entre dois processos sem a incorrência de outros problemas foi proposta pelo matemático holandês T. Dekker, com base no primeiro e no quarto algoritmos. O algoritmo de Dekker possui uma lógica bastante complexa e pode ser encontrado em Ben-Ari (1990) e Stallings (1997). Posteriormente, G. L. Peterson propôs outra solução mais simples para o mesmo problema.

Soluções de Software – Algoritmo de Peterson

```
PROGRAM Algoritmo_Peterson;
  VAR CA, CB : BOOLEAN;
  Vez : CHAR;

PROCEDURE Processo_A;
BEGIN
  REPEAT
    CA := true;
    Vez := 'B';
    WHILE (CB and Vez = 'B') DO
(*do nothing *);
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
  UNTIL false;
END;
```

```
PROCEDURE Processo_B;
BEGIN
  REPEAT
    CB := true;
    Vez := 'A';
    WHILE (CA and Vez = 'A') DO
(*do nothing *);
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
  UNTIL false;
END;
```

```
BEGIN
  CA := false;
  CB := false;
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Soluções de Software

- Algoritmo para exclusão mútua entre N processos
  - Dekker e Peterson garantem exclusão mútua para 2 processos.
  - Hofri (1990) generalizou o algoritmo de Peterson para N processos
  - Lamport (1974) propôs o algoritmo do padeiro (bakery algorithm), solução clássica encontrado em Ben-Ari (1990) e Silbershatz, Galvin e Gagne(2001)
  - Outros algoritmos foram apresentados em Peterson (1983) e Lamport (1987)

Sincronização condicional

- Sincronização condicional é uma situação em que o acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso.
- Exemplo clássico: processo produtor e processo consumidor

Sincronização condicional

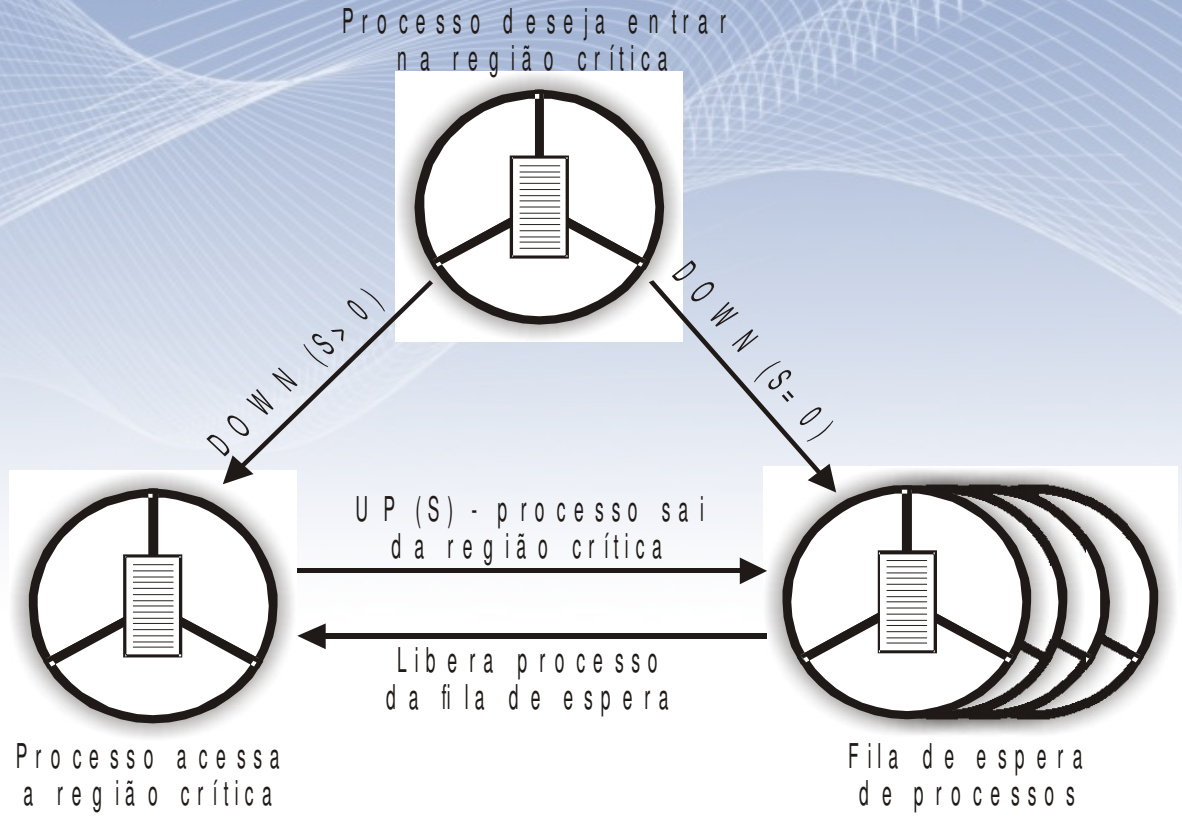
```
PROGRAM Produtor_Consumidor_1;
  CONST TamBuf = (*Tamanho Qualquer*);
  TYPE Tipo_Dado = (*Tipo Qualquer*);
  VAR Buffer : ARRAY [ 1..TamBuf ] OF
Tipo_Dado;
    Dado_1 : Tipo_Dado;
    Dado_2 : Tipo_Dado;
    Cont : 0..TamBuf;

PROCEDURE Produtor;
BEGIN
  REPEAT
    Produz_Dado (Dado_1);
    WHILE (Cont = TamBuf) DO (*Do Nothing*);
    Grava_Buffer (Dado_1, Buffer);
    Cont := Cont + 1;
  UNTIL false;
END;
```

```
PROCEDURE Consumidor;
BEGIN
  REPEAT
    WHILE (Cont = 0) DO (*Do Nothing*);
    Le_Buffer (Dado_2, Buffer);
    Consome_Dado (Dado_2);
    Cont := Cont - 1;
  UNTIL false;
END;

BEGIN
  Cont := 0;
  PARBEGIN
    Produtor;
    Consumidor;
  PAREND;
END.
```

Semáforos

- O conceito de semáforos foi proposto por E. W. Dijkstra em 1965, sendo apresentado como um mecanismo de sincronização que permitia implementar, de forma simples, a exclusão mútua e a sincronização condicional entre processos.
- Semáforo é uma variável inteira, não negativa que só poder ser manipulada pelas instruções: Down e Up.

Utilização do Semáforo Binário na Exclusão Mútua



Utilização do Semáforo Binário na Exclusão Mútua

```
PROGRAM Semaforo_1;
  VAR s : Semaforo := 1;
(*inicialização do semáforo*)

PROCEDURE Processo_A;
BEGIN
  REPEAT
    DOWN( s );
    Regiao_Critica_A;
    UP( s );
  UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
  REPEAT
    DOWN( s );
    Regiao_Critica_B;
    UP( s );
  UNTIL false;
END;

BEGIN
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

Produtor/Consumidor

```
PROGRAM Produtor_Consumidor_2;
  CONST TamBuf = 2;
  TYPE Tipo_Dado = (*Tipo Qualquer*);
  VAR Vazio : Semaforo := TamBuf;
        Cheio : Semaforo := 0;
        Mutex : Semaforo := 1;
        Buffer : ARRAY [ 1..TamBuf ] OF Tipo_Dado;
        Dado_1 : Tipo_Dado;
        Dado_2 : Tipo_Dado;
```

```
PROCEDURE Produtor;
BEGIN
  REPEAT
    Produz_Dado( Dado_1 );
    DOWN( Vazio );
    DOWN( Mutex );
    Grava_Buffer( Dado_1, Buffer );
    UP( Mutex );
    UP( Cheio );
  UNTIL false;
END;
```

```
PROCEDURE Consumidor;
BEGIN
  REPEAT
    DOWN( Cheio );
    DOWN( Mutex );
    Le_Buffer( Dado_2, Buffer );
    UP( Mutex );
    UP( Vazio );
    Consome_Dado( Dado_2 );
  UNTIL false;
END;
```

```
BEGIN
  PARBEGIN
    Produtor;
    Consumidor;
  PAREND;
END.
```

Problema dos Filósofos

Problema clássico de sincronização proposto por Dijkstra, proposição:

“Há uma mesa com cinco pratos e cinco garfos onde os filósofos podem sentar, comer e pensar. Toda vez que um filósofo para de pensar e deseja comer é necessário que ele utilize dois garfos, posicionados à sua direita e à sua esquerda.

```
PROGRAM Filosofo_1;
  VAR Garfos : ARRAY [ 0..4 ] OF Semaforo := 1;
        I : INTEGER;

PROCEDURE Filosofo ( I : INTEGER );
BEGIN
  REPEAT
    Pensando;
    DOWN( Garfo[ I ] );
    DOWN( Garfo[ ( I + 1 ) MOD 5 ] );
    Comendo;
    UP( Garfo[ I ] );
    UP( Garfo[ ( I + 1 ) MOD 5 ] );
  UNTIL false;
END;

BEGIN
  PARBEGIN
    FOR I := 0 TO 4 DO
      Filosofo( I );
  PAREND;
END.
```

Problema dos Filósofos

Analisando o algoritmo anterior vimos que pode ocorrer o travamento denominado deadlock, portanto apresentaremos algumas sugestões que podem evitar esse problema:

(a) Permitir que apenas 4 filósofos sentem à mesa simultaneamente;

(b) Permitir que um filósofo pegue um garfo apenas se o outro estiver disponível;

(c) Permitir que um filósofo ímpar pegue primeiro o seu garfo da esquerda e depois o da direita, enquanto o filósofo par pegue o garfo da direita e, em seguida o da esquerda.

```
PROGRAM Filosofo_2;
  VAR Garfos : ARRAY [ 0..4 ] OF Semaforo := 1;
        Lugares : Semaforo := 4;
        I : INTEGER;

PROCEDURE Filosofo ( I : INTEGER );
BEGIN
  REPEAT
    Pensando;
    DOWN( Lugares );
    DOWN( Garfo[ I ] );
    DOWN( Garfo[ ( I + 1 ) MOD 5 ] );
    Comendo;
    UP( Garfo[ I ] );
    UP( Garfo[ ( I + 1 ) MOD 5 ] );
    UP( Lugares );
  UNTIL false;
END;

BEGIN
  PARBEGIN
    FOR I := 0 TO 4 DO
      Filosofo( I );
  PAREND;
END.
```

Problema do Barbeiro

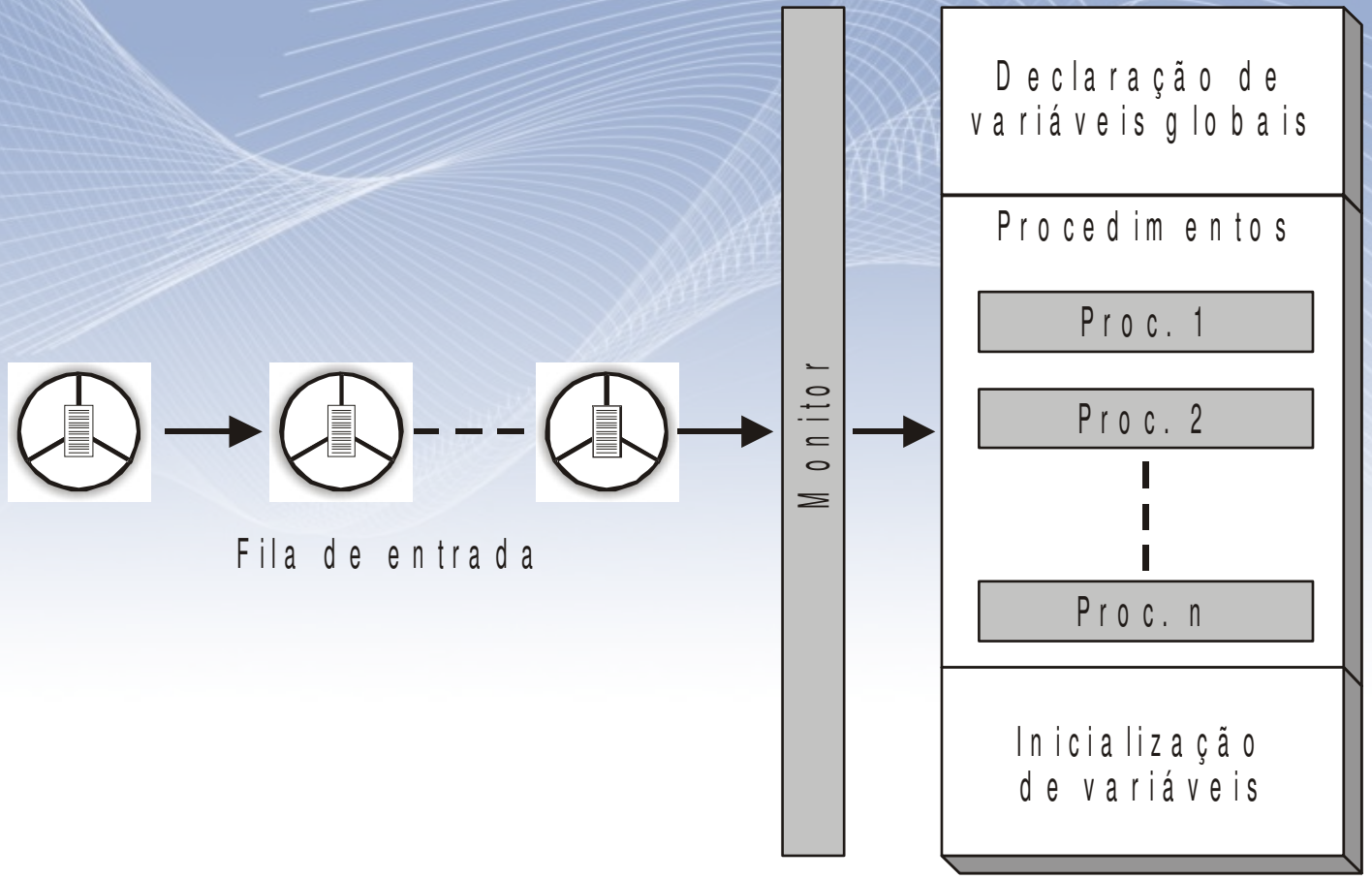
Outro exemplo clássico, neste problema um barbeiro recebe clientes para cortar o cabelo. Na barbearia há uma cadeira de barbeiro e apenas 5 cadeiras para clientes esperarem. Quando um cliente chega, caso o barbeiro esteja trabalhando ele se senta, se houver cadeira vazia, ou vai embora, se todas as cadeiras estiverem ocupadas. No caso do barbeiro não ter nenhum cliente para atender, ele senta na cadeira e dorme até chegar um cliente.

```
PROGRAM Barbeiro;
  CONST Cadeiras = 5;
  VAR Clientes : Semaforo := 0;
      Barbeiro : Semaforo := 0;
      Mutex : Semaforo := 1;
      Espera : INTEGER := 1;
```

```
PROCEDURE Barbeiro;
BEGIN
  REPEAT
    DOWN ( Clientes );
    DOWN ( Mutex );
    Espera := Espera – 1;
    UP ( Barbeiro );
    UP ( Mutex );
    Corta_Cabelo;
  UNTIL false;
END;
```

```
PROCEDURE Clientes;
BEGIN
  DOWN ( Mutex );
  IF ( Espera < Cadeiras )
  BEGIN
    Espera := Espera + 1;
    UP ( Clientes );
    UP ( Mutex );
    DOWN ( Barbeiro );
    Ter_Cabelo_Cortado;
  END
  ELSE UP ( Mutex );
END;
```

Estrutura do Monitor

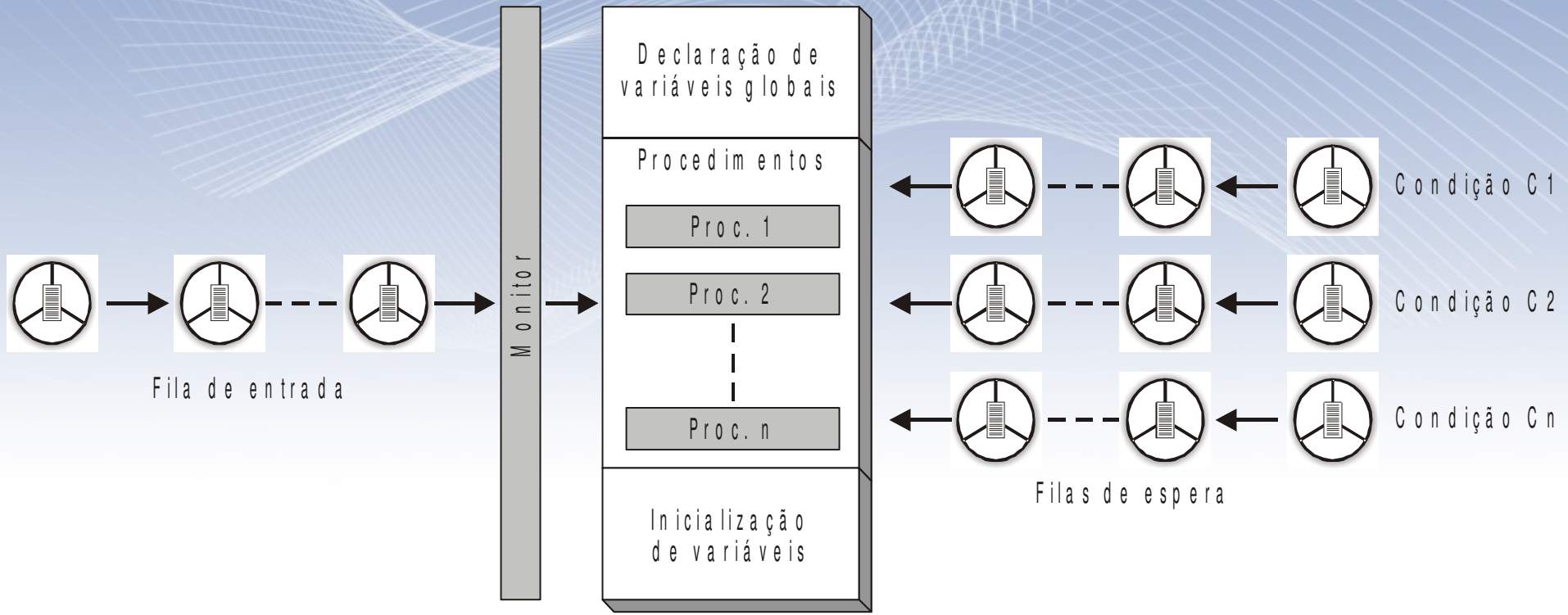


Algoritmo para Monitor

```
MONITOR Exclusao_Mutua;
    (* Declaração das variáveis do monitor *)
    PROCEDURE Regiao_Critica_1;
    BEGIN
        :
    END;

    PROCEDURE Regiao_Critica_2;
    BEGIN
        :
    END;

    PROCEDURE Regiao_Critica_3;
    BEGIN
        :
    END;

    BEGIN
        (* Código de inicialização *)
    END;
```

```
PROGRAM Monitor_1;
  MONITOR Exclusao_Mutua;
    VAR x : INTEGER;

  PROCEDURE Soma;
  BEGIN
    x := x + 1;
  END;

  PROCEDURE Diminui;
  BEGIN
    x := x – 1;
  END;

  BEGIN
    x := 0;
  END;

BEGIN
  PARBEGIN
    Regiao_Critica.Soma;
    Regiao_Critica.Diminui;
  PAREND;
END.
```

Estrutura do Monitor com Variáveis de Condição

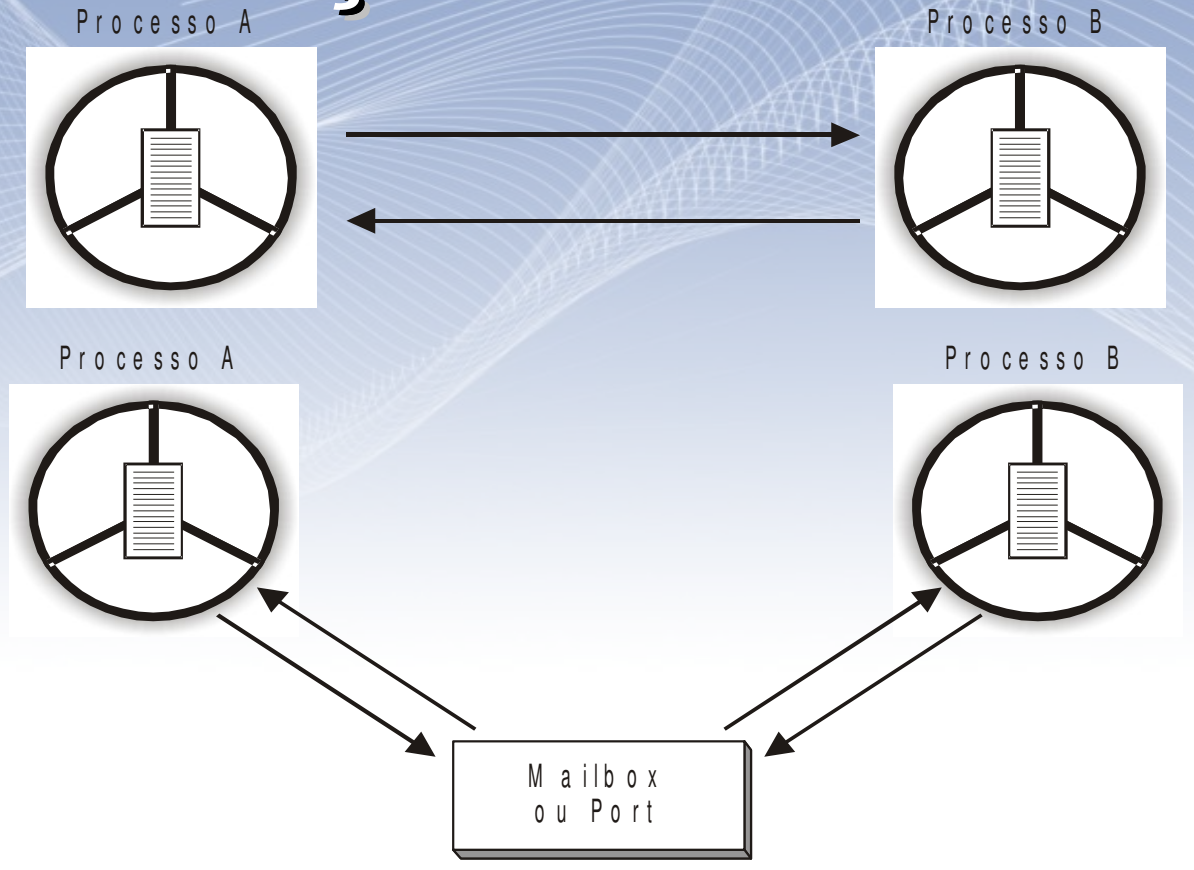


Sincronização Condicional Utilizando Monitores

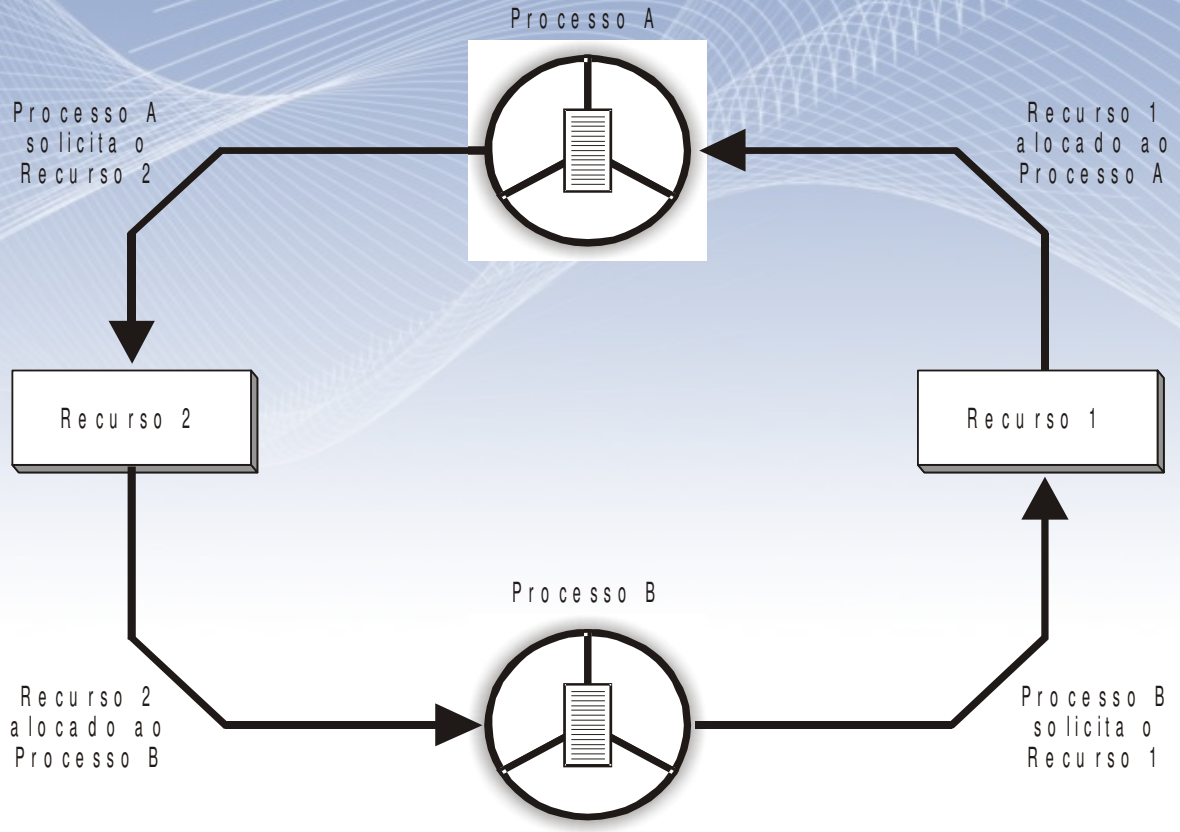
```
MONITOR Condicional;
    VAR Cheio, Vazio : (* Variáveis especiais de condição *);

    PROCEDURE Produz;
    BEGIN
        IF ( Cont = TamBuf ) THEN WAIT ( Cheio );
        :
        IF ( Cont = 1 ) THEN SIGNAL ( Vazio );
    END;

    PROCEDURE Consome;
    BEGIN
        IF ( Cont = 0 ) THEN WAIT ( Vazio );
        :
        IF ( Cont = TamBuf -1 ) THEN SIGNAL ( Cheio );
    END;

    BEGIN
        (* Código de inicialização *)
    END;
```

Transmissão de Mensagem

Comunicação Direta e Indireta



Deadlock – Espera Circular



Deadlock

- Para que ocorra um deadlock, 4 condições são necessárias simultaneamente:
  - Exclusão mútua: cada recurso só pode estar alocado a um único processo em um determinado instante;
  - Espera por recurso: um processo, além dos recursos já alocados, pode estar esperando por outros recursos;
  - Não-preempção: um recurso não pode ser liberado de um processo só porque outros porcessos desejam o mesmo recurso;
  - Espera circular: um processo pode ter de esperar por um recurso alocado a outro processo, e vice-versa.

DÚVIDAS