SISTEMAS OPERACIONAIS

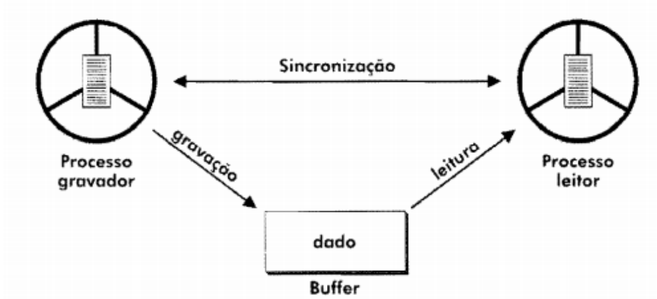
SINCRONIZAÇÃO E COMUNICAÇÃO DE PROCESSOS (PARTE 2)

Capítulo 7

Professor Fábio Angelo

Conceitos Iniciais

RELEMBRANDO AS SITUAÇÕES



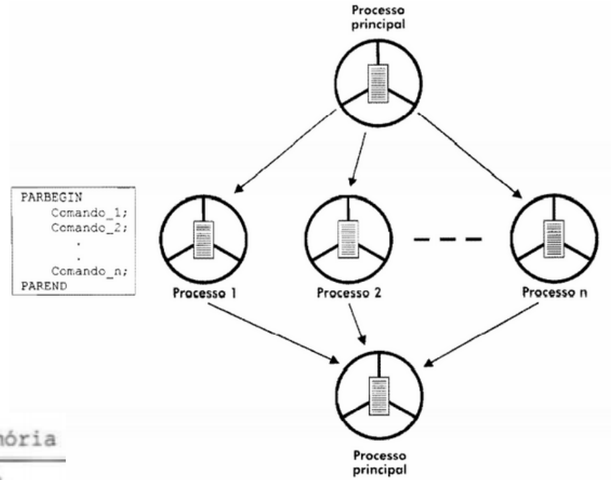


Fig. 7.2 Concorrência em programas.

Caixa	Comando	Saldo arquivo	Valor dep/ret	Saldo memória
1	READ	1.000	*	1.000
1	READLN	1.000	-200	1.000
1	:=	1.000	-200	800
2	READ	1.000	*	1.000
2	READLN	1.000	+300	1.000
2	:=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

Situações:

1) Concorrência;
2) O problema do compartilhamento de recursos;
3) Race Conditions;
4) Região Crítica;
5) Starvation (espera indefinida)
6) Exclusão Mútua por Hardware;
7) Exclusão Mútua por Software;
8) Sincronização Condicional

O que ficou na mente?

1) O que é exclusão mútua e como é implementada?

2) O que é "starvation"?

3) Qual o problema gerado se usarmos o recurso de desabilitar as interrupções para implementar a exclusão mútua?

4) O que é espera ocupada e qual seu impacto no sistema computacional?

5) Explique o que é sincronização condicional. Cite algum exemplo prático.

Respostas Recebidas

Gabriela, Larissa e Vinicius Reinert

1- Exclusao mutua ocorre quando um recurso é utilizado separadamente por processo, ou seja, um começa apenas quando o outro termina.

2 - Starvation é quando um processo não consegue ser executado, de forma alguma, pois sempre existem processos de prioridade maior para serem executados, de forma que o processo "faminto" nunca consiga tempo de processamento.

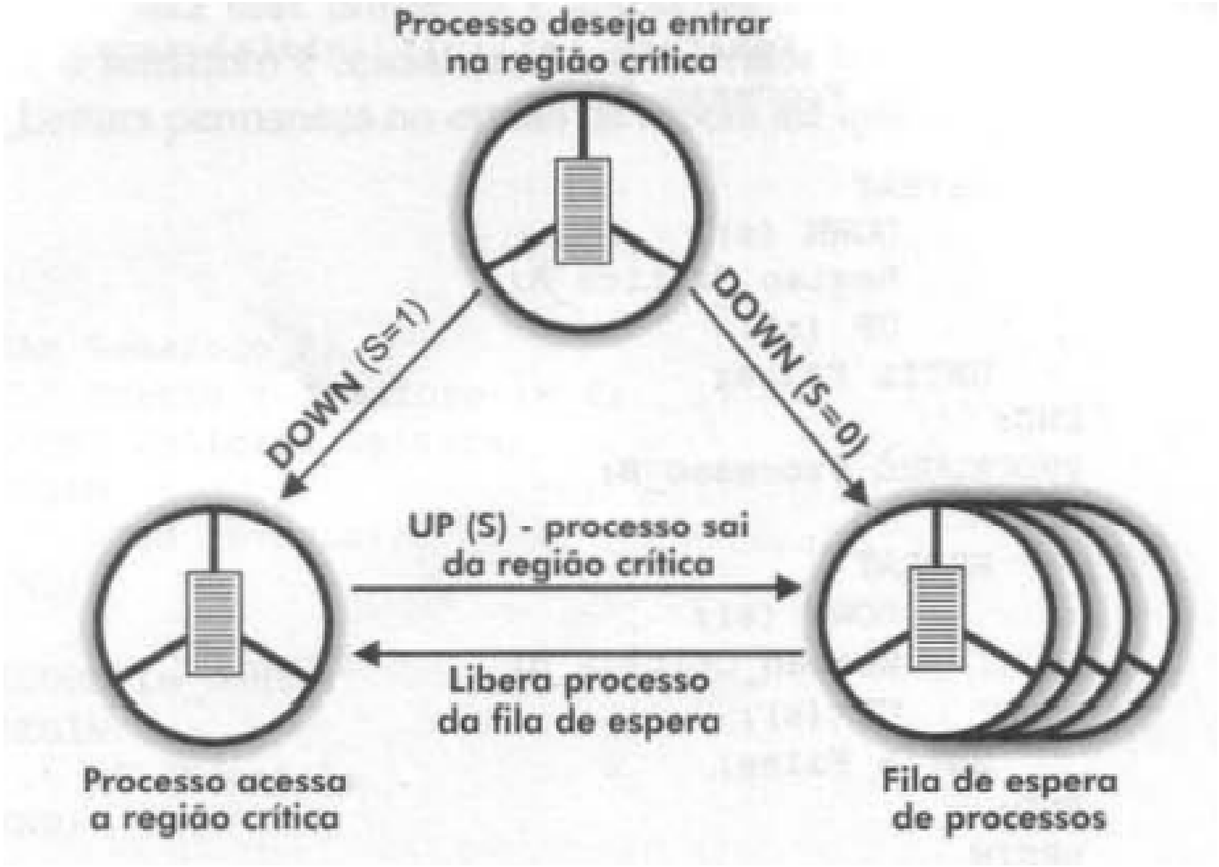
3 - Tal artifício impede a mudança de contexto, deixando o acesso exclusivo, por outro lado, compromete a concorrência de recursos.

4 - Espera ocupada é um modelo de programação paralela caracterizado por testes repetidos de um condição que impedem o progresso de um processo e que só pode ser alterada por outro processo. Possui a grande desvantagem de levar a desperdícios de tempo em um monoprocessador, já que este passa parte do tempo testando condições (cujo resultado é falso), ao invés de realizar trabalho útil. Porém, pode ser uma solução aceitável para multiprocessadores.

5 - Sincronização condicional é uma situação onde o acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso. Um recurso pode não se encontrar pronto para uso devido a uma condição específica. Nesse caso, o processo que deseja acessá-lo deverá permanecer bloqueado até que o recurso fique disponível. Um exemplo clássico desse tipo de sincronização é a comunicação entre dois processos através de operações de gravação e leitura em um buffer

Semáforos

TRATANDO A EXCLUSÃO MÚTUA



Exclusão Mútua:

1) Pode ser implementada através de um semáforo binário;

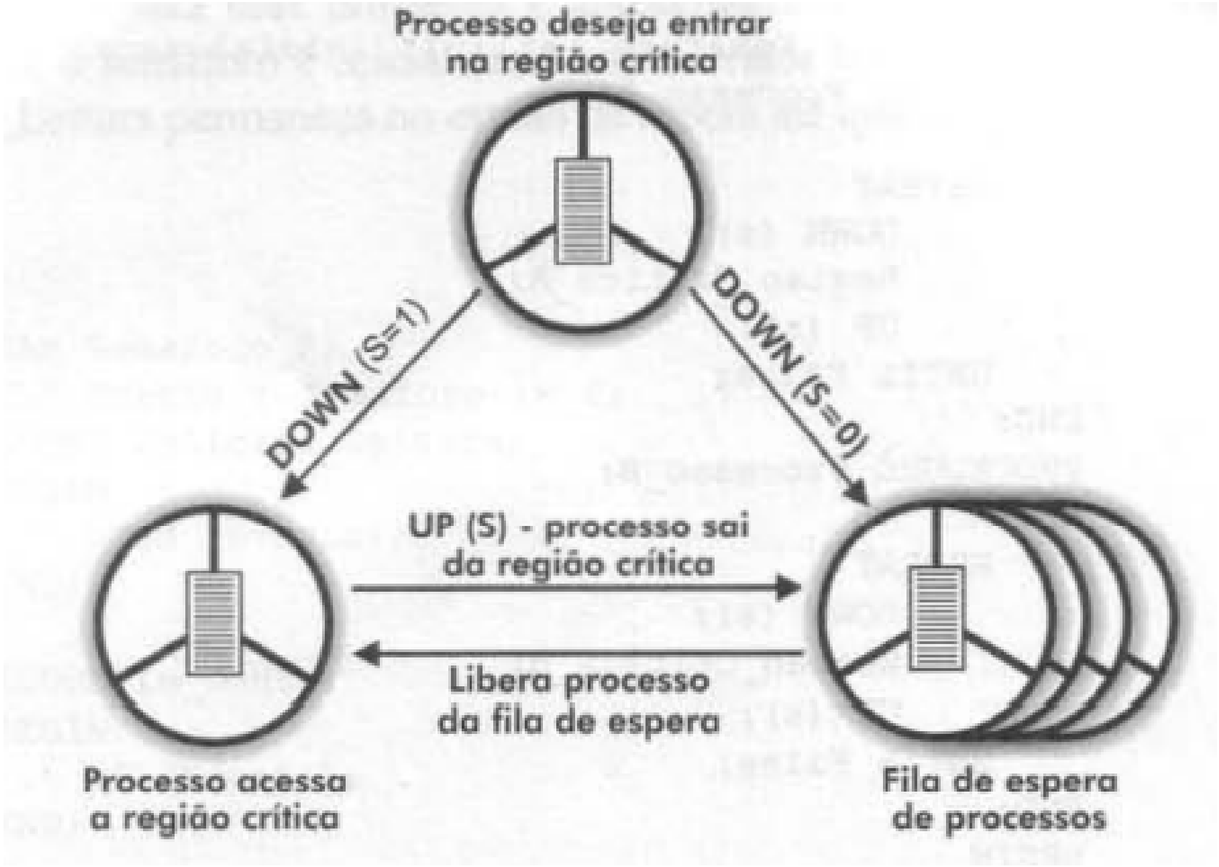
2) O semáforo é associado a um recurso compartilhado;

3) Trata a situação de Busy Wait (espera ocupada);

4) Processos que não conseguem entrar na região crítica são adicionados a uma fila de espera.

Semáforos

TRATANDO A EXCLUSÃO MÚTUA



Implementação:

1) As instruções UP e DOWN funcionam como protocolos de entrada/saída das regiões críticas;

2) O semáforo associado ao recurso compartilhado faz a devida sinalização quando está ocupado/livre;

3) Semáforo com valor 1, indica estar livre para uso;

4) Semáforo com valor 0, indica estar que o recurso está ocupado.

Olhando internamente...

```
TYPE Semaforo = RECORD
        Valor : INTEGER;
        Fila_Espera : (* Lista de processos pendentes *);
    END;
PROCEDURE DOWN (VAR S : Semaforo);
BEGIN
    IF (S = 0) THEN Coloca_Processo_na_Fila_de_Espera
    ELSE
        S := S - 1;
END;

PROCEDURE UP (VAR S : Semaforo);
BEGIN
    S := S + 1;
    IF (Tem_Processo_Esperando) THEN Retira_da_Fila_de_Espera;
END;
```

```
PROGRAM Semaforo_1;
    VAR s : Semaforo := 1;
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.
```

Olhando internamente...

Processo_A	Processo_B	S	Pendente
REPEAT	REPEAT	1	*
DOWN (s)	REPEAT	0	*
Regiao_Critica_A	DOWN (s)	0	Processo_B
UP (s)	DOWN (s)	1	Processo_B
REPEAT	Regiao_Critica_B	0	*

EXECUTANDO...

1) Processo A executa a instrução DOWN;
2) Semáforo é decrementado de 1;
3) Processo A tem acesso a área crítica;
4) Processo B executa a instrução DOWN;
5) Semáforo em 0, coloca processo em fila;
6) Processo A executa a instrução UP;
7) Semáforo é incrementado de 1;
8) Fila de pendentes é acionada.

```
PROGRAM Semaforo_1;
    VAR s : Semaforo := 1;
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.
```

Semáforos

TRATANDO A SINCRONIZAÇÃO CONDICIONAL

```
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.
```

Semáforos

TRATANDO A SINCRONIZAÇÃO CONDICIONAL

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

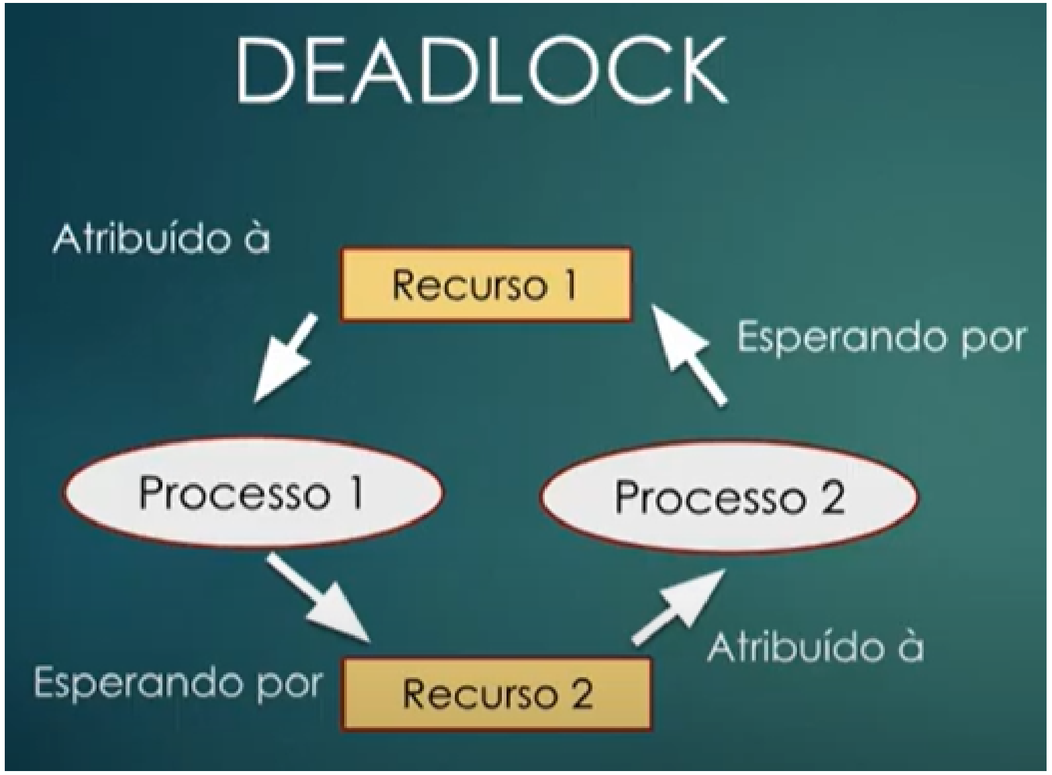
```
PROCEDURE Consumidor;
BEGIN
    REPEAT
        DOWN (Cheio);
        DOWN (Mutex);
        Le_Buffer (Dado_2, Buffer);
        UP (Mutex);
        UP (Vazio);
        Consome_Dado (Dado_2);
    UNTIL False;
END;
```

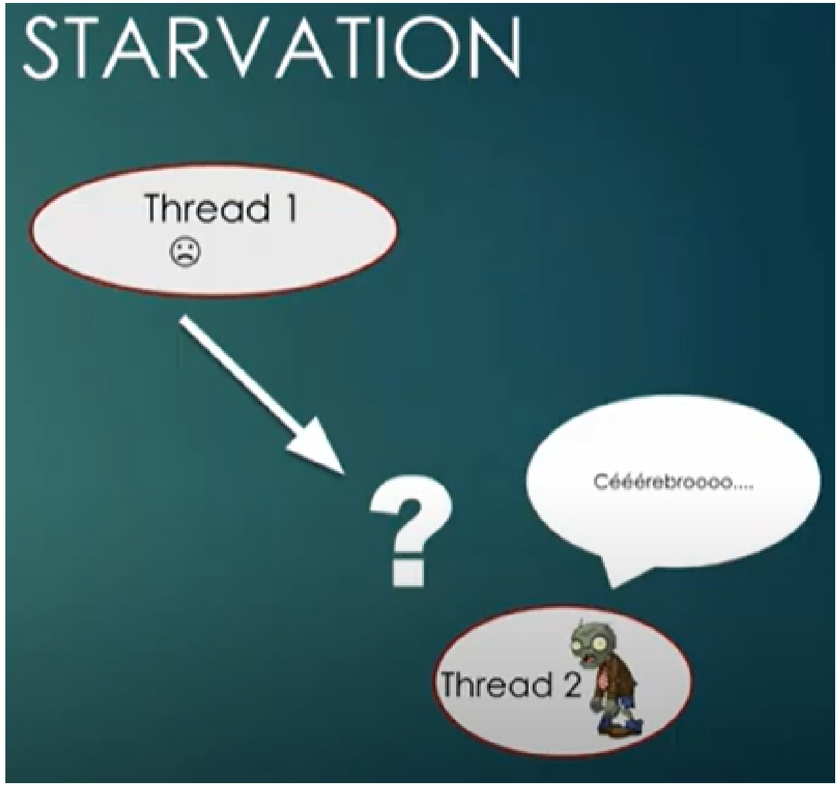
Produtor	Consumidor	Vazio	Cheio	Mutex	Pendente
*	*	2	0	1	*
*	DOWN (Cheio)	2	0	1	Consumidor
DOWN (Vazio)	DOWN (Cheio)	1	0	1	Consumidor
DOWN (Mutex)	DOWN (Cheio)	1	0	0	Consumidor
Grava_Buffer	DOWN (Cheio)	1	0	0	Consumidor
UP (Mutex)	DOWN (Cheio)	1	0	1	Consumidor

Produtor	Consumidor	Vazio	Cheio	Mutex	Pendente
UP (Cheio)	DOWN (Cheio)	1	1	1	*
UP (Cheio)	DOWN (Cheio)	1	0	1	*
UP (Cheio)	DOWN (Mutex)	1	0	0	*
UP (Cheio)	Lê_Dado	1	0	0	*
UP (Cheio)	UP (Mutex)	1	0	1	*
UP (Cheio)	UP (Vazio)	2	0	1	*

Semáforos

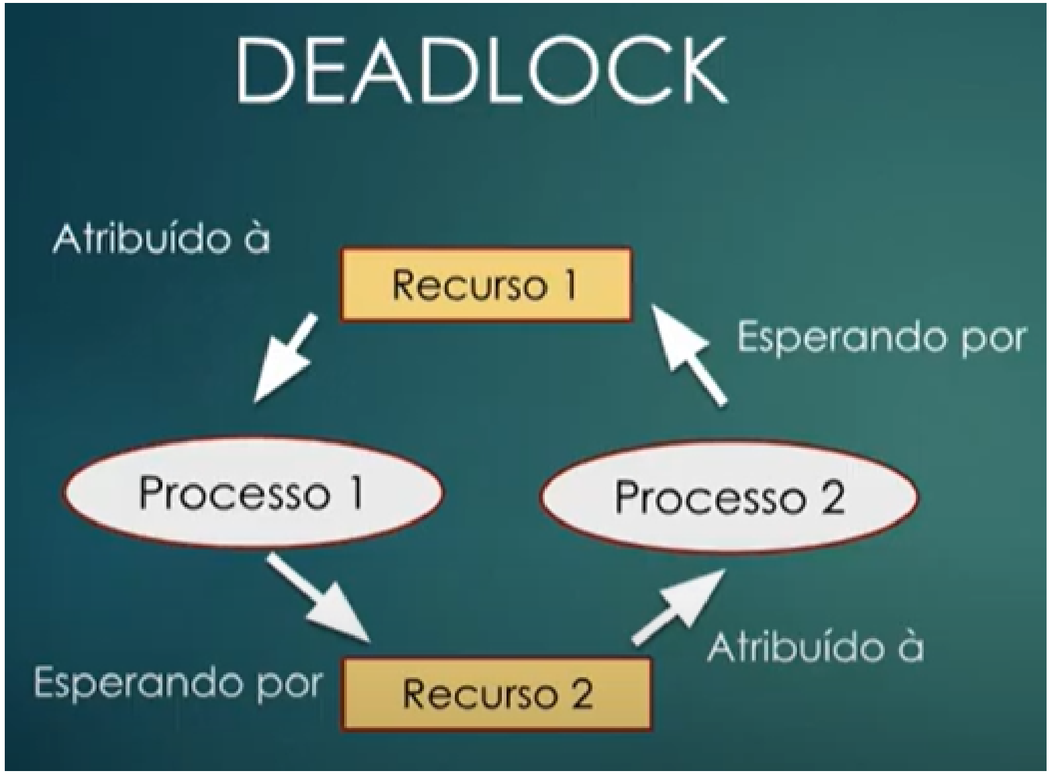
E OS PROBLEMAS DA CONCORRÊNCIA...

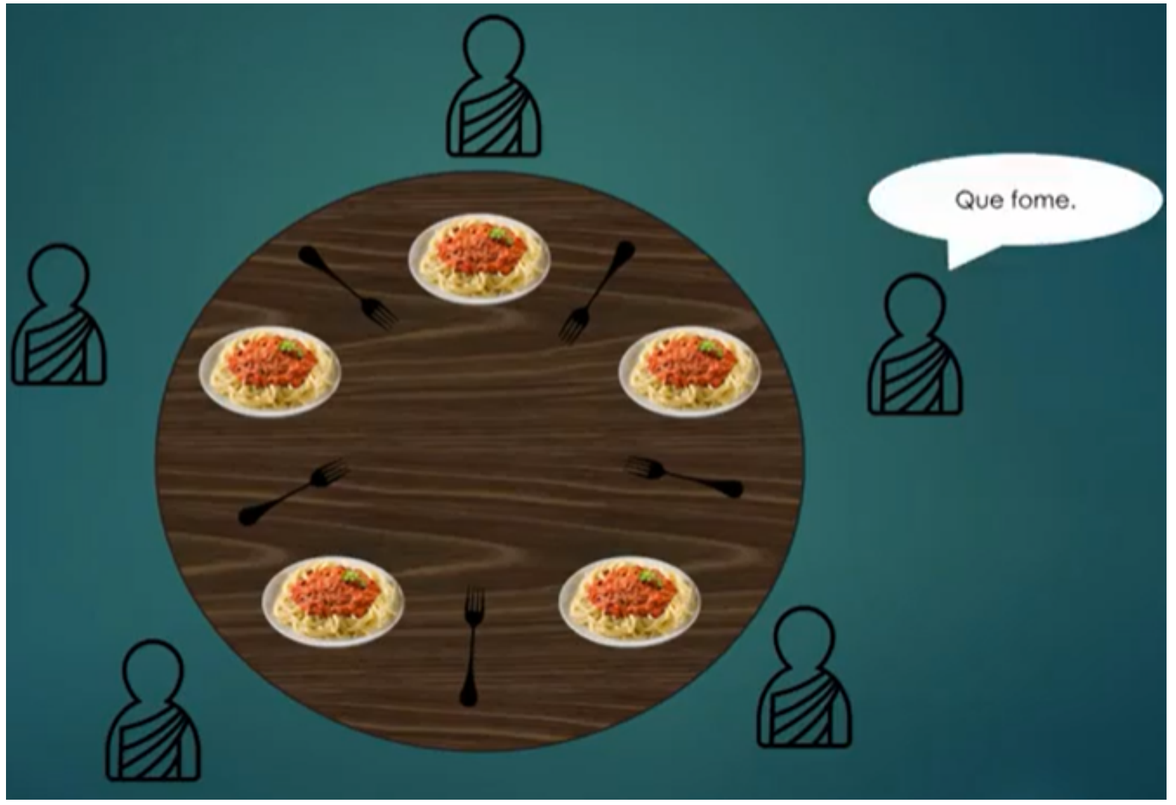




Semáforos

O JANTAR DOS FILÓSOFOS



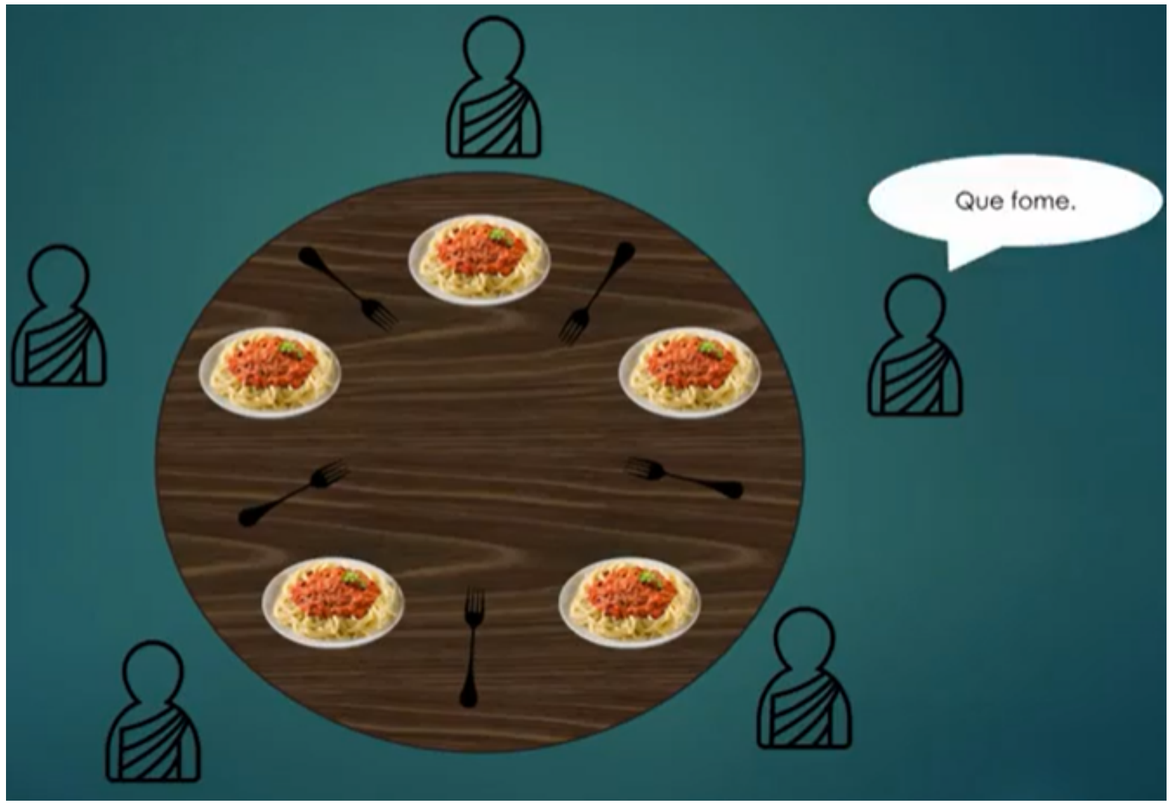


Semáforos

O JANTAR DOS FILÓSOFOS

PREMISSAS

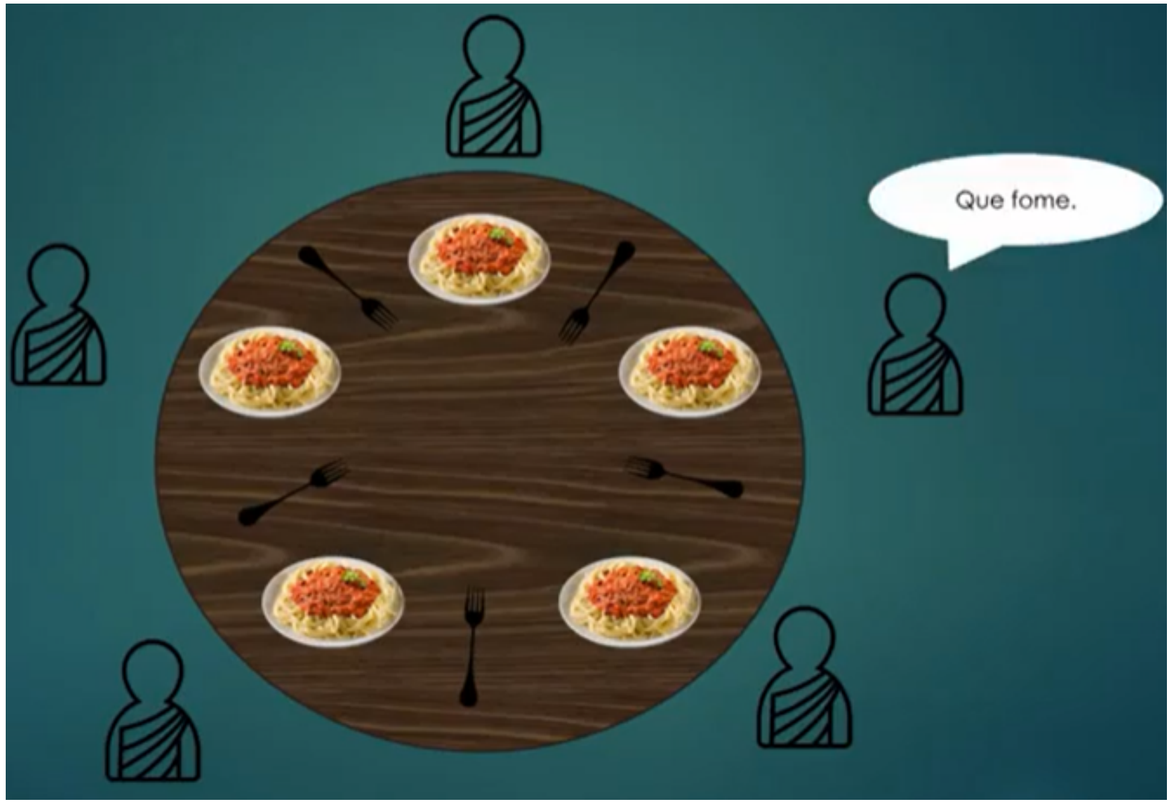
1) 5 filósofos;
2) 5 garfos;
3) o filósofo pode comer e pensar;
4) para comer, deverá pegar 2 garfos;
5) os garfos devem ser os da sua direta e esquerda;



Semáforos

O JANTAR DOS FILÓSOFOS

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

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

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



Praticando...

Em uma aplicação concorrente que controla saldo bancário em contas-correntes, dois processos compartilham uma região de memória onde estão armazenados os saldos dos clientes A e B. Os processos executam concorrentemente os seguintes passos:

Processo 1 (Cliente A)	Processo 2 (Cliente B)
/* saque em A */	/*saque em A */
1a. x := saldo_do_cliente_A;	2a. y := saldo_do_cliente_A;
1b. x := x - 200;	2b. y := y - 100;
1c. saldo_do_cliente_A := x;	2c. saldo_do_cliente_A := y;
/* deposito em B */	/* deposito em B */
1d. x := saldo_do_cliente_B;	2d. y := saldo_do_cliente_B;
1e. x := x + 100;	2e. y := y + 200;
1f. saldo_do_cliente_B := x;	2f. saldo_do_cliente_B := y;

Supondo que os valores dos saldos de A e B sejam, respectivamente, 500 e 900, antes de os processos executarem, pede-se:

a) Quais os valores esperados para os saldos dos clientes A e B após o término da execução dos processos?

b) Quais os valores finais dos saldos dos clientes se a seqüéncia temporal de execução das operações for: 1a, 2a, 1b, 2b, 1c, 2c, 1d 2d, 1c, 2c, 1d, 2d, 1e, 2e, 1f, 2f?

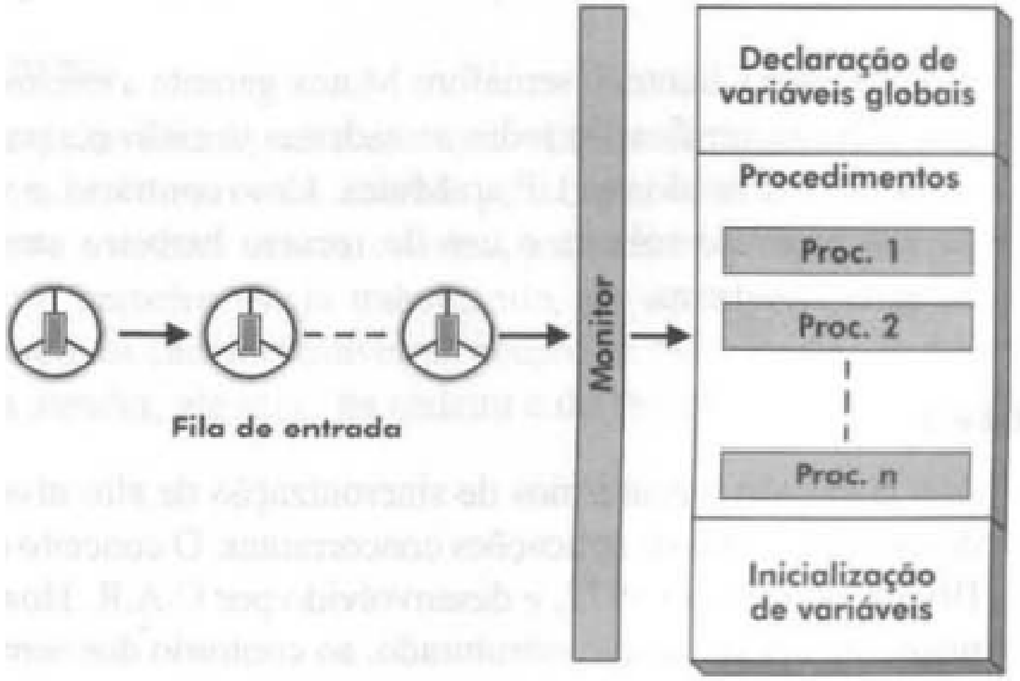
c) Utilizando semáforos, proponha uma solução que garanta a integridade dos saldos.

Monitores

MECANISMO DE SINCRONIZAÇÃO

CONTEXTUALIZANDO...

- Proposto por Brinch Hansen em 1972 para simplificar o desenvolvimento de aplicações concorrentes;
- É mecanismo de sincronização estrutura, em oposições aos semáforos que são não-estruturados;
- Além de simplificar também reduz as chances dos erros comuns em aplicações concorrentes;
- O monitor é formado por procedimentos e variáveis encapsulados dentro de um módulo, ou seja, apenas um processo pode estar executando um procedimento do monitor em determinado momento.



Monitores

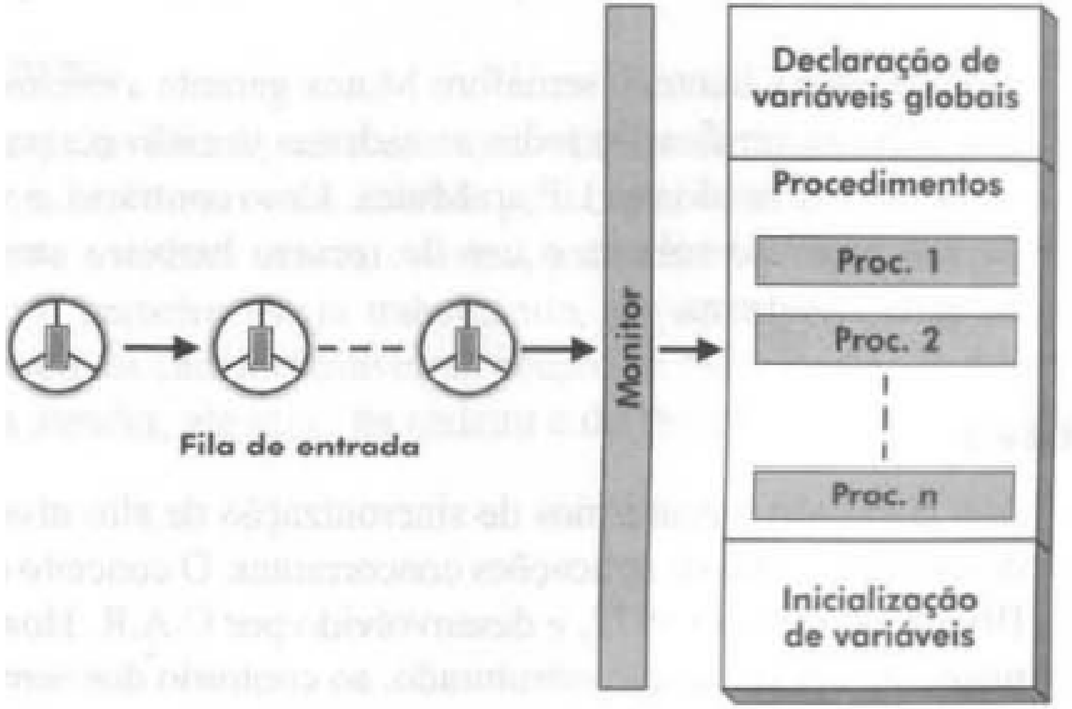
MECANISMO DE SINCRONIZAÇÃO

MONITOR Exclusao_Mutua:
(*Declaracao das variaveis do monitor*)

PROCEDURE Regiao_Critica_1;
BEGIN
END;

PROCEDURE Regiao_Critica_2;
BEGIN
END;

BEGIN
(* Codigo de inicializacao *)
END;



Monitores

MECANISMO DE SINCRONIZAÇÃO

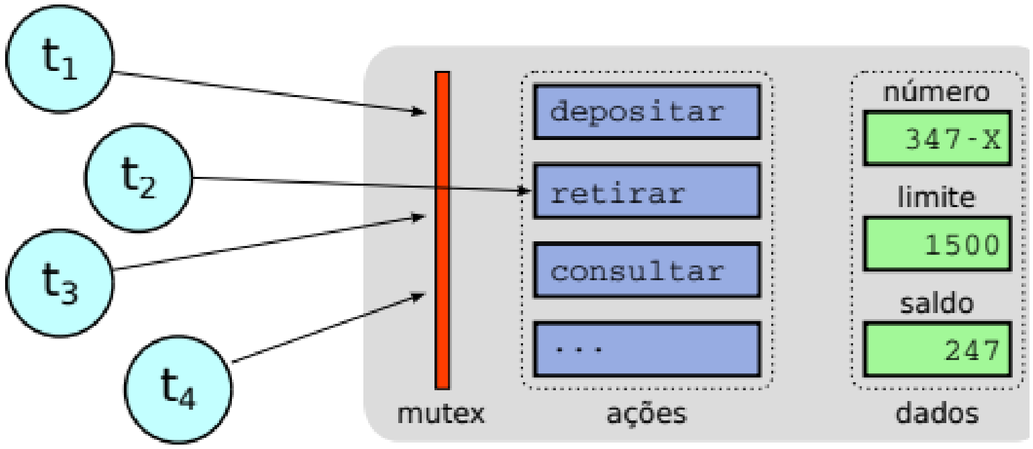
```
PROGRAM Monitor_1;
    MONITOR Regiao_Critica;
        VAR X : INTEGER;

        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.
```

IMPLEMENTANDO...

- O programador não implementa diretamente a exclusão mútua;
- As regiões críticas devem ser definidas como procedimentos no monitor;
- Compilador se encarrega de garantir a exclusão mútua entre os procedimentos;
- A comunicação com o monitor é feita unicamente através de chamadas a seus procedimentos e dos parâmetros passados.

Monitores

MECANISMO DE SINCRONIZAÇÃO



Um monitor em Java

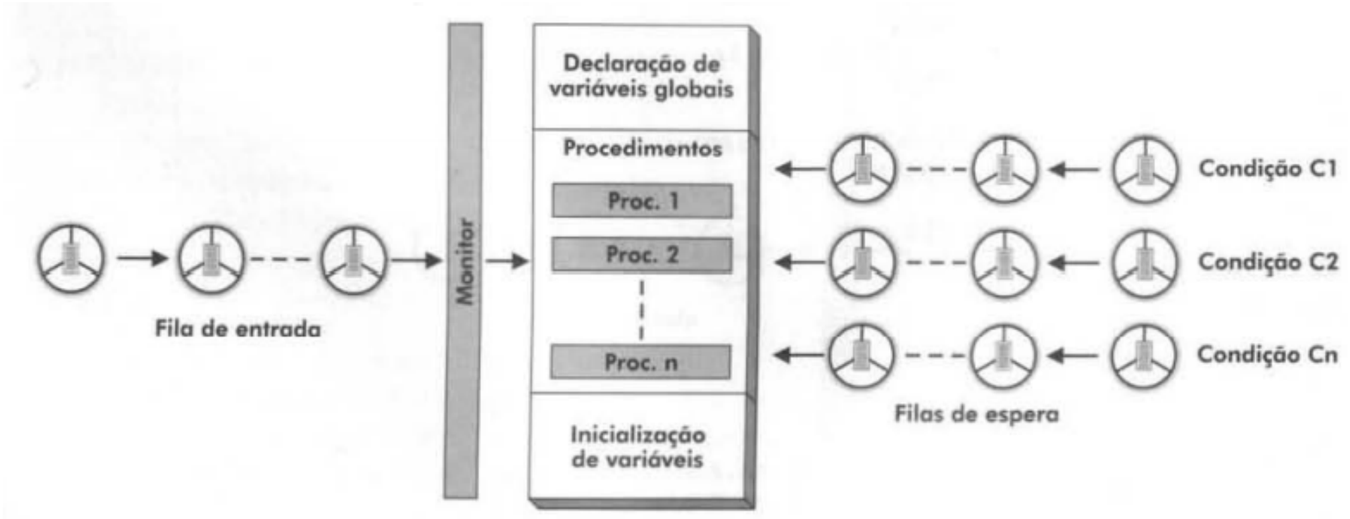
```
1  class Conta
2  {
3     private float saldo = 0;
4
5     public synchronized void depositar (float valor)
6     {
7        if (valor >= 0)
8           saldo += valor ;
9        else
10          System.err.println("valor negativo");
11    }
12
13    public synchronized void retirar (float valor)
14    {
15       if (valor >= 0)
16          saldo -= valor ;
17       else
18          System.err.println("valor negativo");
19    }
20 }
```

Monitores

SINCRONIZAÇÃO CONDICIONAL

CONTEXTUALIZANDO...

- Os monitores podem executar um procedimento condicionados ao valor de variáveis especiais;
- Estas variáveis são manipuladas por intemédio de duas intruções, conhecidas como WAIT e SIGNAL;



- A Instrução WAIT faz o processo ficar em estado de espera, até que outro processo sinalize com a intrução SIGNAL;
- O monitor organiza os processos em filas.

Monitores

SINCRONIZAÇÃO CONDICIONAL

MONITOR Condicional ;

VAR Cheio, Vazio (* Variáveis especiais *)

PROCEDURE **Produz**;

BEGIN
IF (Cont = TamBuf) THEN WAIT (Cheio);
....
IF (Cont =1) THEN SIGNAL (Vazio) ;
END;

PROCEDURE **Consome**;

BEGIN
IF (Cont = O) THEN WAIT (Vazio);
....
IF (Cont = TamBuf - 1) THEN SIGNAL (Cheio);
END;

BEGIN
END ;

Troca de Mensagens

COMUNICAÇÃO E SINCRONIZAÇÃO DE PROCESSOS

CONTEXTUALIZANDO...

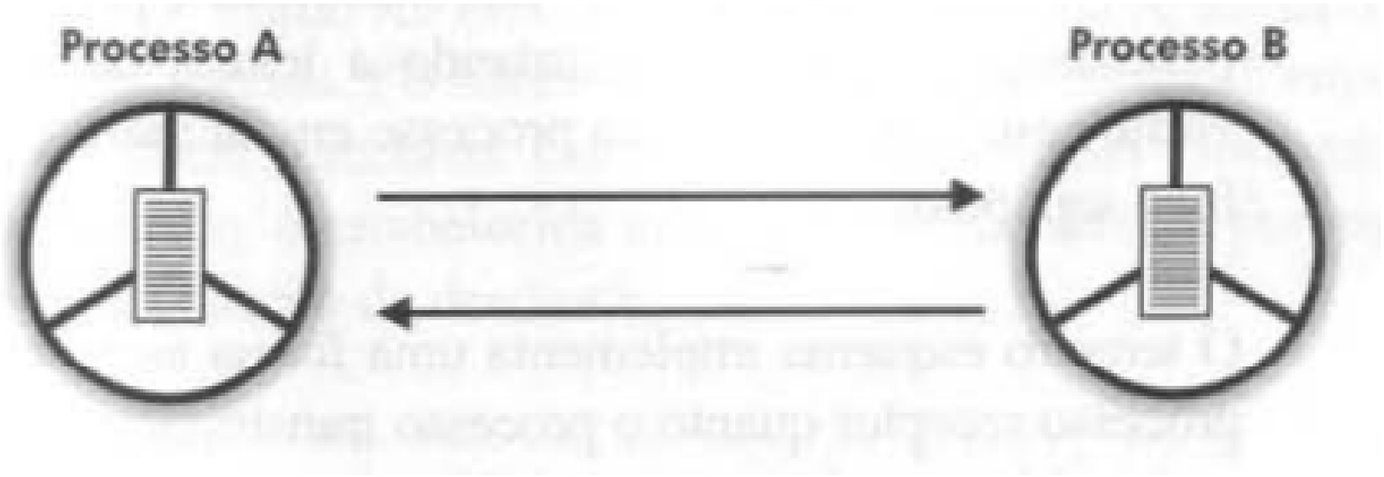
- O Sistema Operacional possui um subsistema de mensagem, devendo para isso existir um canal de comunicação;
- Esse canal pode ser um buffer ou link de rede de computadores;
- A troca de mensagens ocorre através de duas rotinas SEND (receptor, mensagem) e RECEIVE (transmissor, mensagem).

Troca de Mensagens

COMUNICAÇÃO DIRETA

CARACTERÍSTICAS

- Processos envolvidos devem estar com a execução sincronizada;
- É necessário ainda que a comunicação tenho o nome explícito do processo (receptor ou transmissor);
- Permite a troca de mensagem entre apenas dois processos;
- Havendo mudança na identificação dos processos, o código deve ser recompilado.

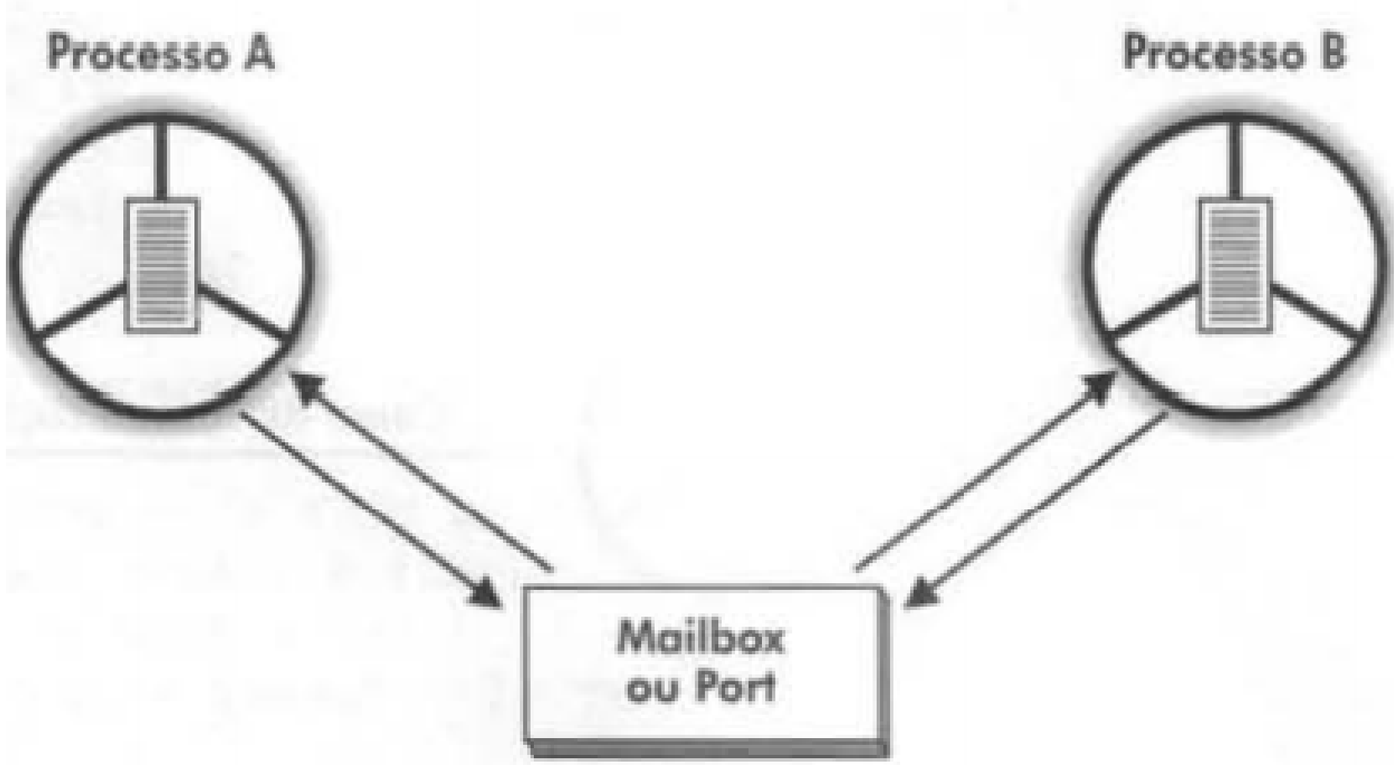


Troca de Mensagens

COMUNICAÇÃO INDIRETA

CARACTERÍSTICAS

- Neste modelo de comunicação, uma área é compartilhada para as mensagens serem colocadas ou retiradas;
- Esse tipo de buffer é conhecido como Mailbox ou Port;
- Permite a interoperação de vários processos usando o mesmo buffer;
- Os parâmetros do SEND e RECEIVE passam a ser nomes de mailboxes em vez do nome dos processos.



Deadlock

IMPASSE NA COMUNICAÇÃO

CONDIÇÕES PARA OCORRER (SEGUNDO COFFMANN, ELPHICK E SHOSHANI, 1971)

- **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 processos desejam o mesmo recurso;
- **Espera circular:** um processo pode ter de esperar por um reçurso alocado a outro processo e vice-versa.

Deadlock

PREVENÇÃO

ELIMINAR PELO MENOS 1 DAS 4 CONDIÇÕES NECESSÁRIAS

- **Exclusão mútua:** aferir a necessidade de garantir o acesso único em áreas críticas;
- **Espera por recurso**: garantir que um processo somente aloque um recurso se ele possuir todos os necessários para a execução; (risco de starvation)
- **Não-preempção**: permitir que um recurso seja retirado de um processo caso outro venha a necessitar (risco de starvation);
- **Espera circular:** implementar a utilização unitária de recursos.