



# ***SISTEMAS DISTRIBUÍDOS***



***Ismael H F Santos***

<b>1</b>	<b>BIBLIOGRAFIA</b>	<b>5</b>
1.1	<b>BIBLIOGRAFIA BÁSICA</b>	<b>5</b>
•	Sistemas Operacionais Modernos - Andrew S. Tanembaum, Prentice Hall 1992.	5
•	Introdução à Arquitetura de Sistemas Operacionais- Francis B Machado e Luis Paulo Maia, LTC 1997 2ª Edição.	5
•	Operating System Concepts: Internals and Design Principles – William Stallings, Prentice Hall 1998 3ª Edição	5
•	Operating System Concepts – James L. Peterson & Abraham Silberschatz, Addison-Wesley 1985, 2ª Edição	5
•	An Introduction to Operating Systems - Harrey M. Deitel, Addison-Wesley 1990, 2ª Edição	5
1.2	<b>BIBLIOGRAFIA ADICIONAL</b>	<b>5</b>
•	Computer Organization and Architecture: Principles Structure and Function – William Stallings, Addison-Wesley 1990, 2ª Edição	
<b>2</b>	<b>PROGRAMAÇÃO CONCORRENTE</b>	<b>7</b>
2.1	<b>COMUNICAÇÃO ENTRE PROCESSOS E CONCORRÊNCIA</b>	<b>7</b>
2.1.1	Grafo de Precedência	7
2.1.2	Condições para Concorrência	9
2.2	<b>ESPECIFICAÇÃO DE CONCORRÊNCIA EM PROGRAMAS</b>	<b>10</b>
2.2.1	Construção Fork e Join	10
2.2.2	Execução Paralela – PARBEGIN e PAREND	12
2.2.3	Comparação FORK-JOIN x PARBEGIN-PAREND	13
2.3	<b>PROBLEMA COMPARTILHAMENTO DE RECURSOS – REGIÃO CRÍTICA</b>	<b>15</b>
2.3.1	Produtor x Consumidor com Buffer Limitado - The Bounded-Buffer Problem	17
2.4	<b>DEFINIÇÃO DO PROBLEMA DE EXCLUSÃO MÚTUA</b>	<b>19</b>
2.4.1	Soluções Algorítmicas para o caso de 2 Processos	20
2.4.2	Solução Algorítmica para o caso N Processos	23
2.5	<b>SOLUÇÕES DE HARDWARE</b>	<b>25</b>
2.5.1	Desabilitar Interrupções	25
2.5.2	Instrução Test and Set e Swap	26
2.6	<b>SOLUÇÕES DE SOFTWARE (SEM BUSY-WAITING)</b>	<b>27</b>
2.6.1	Semáforos	28
2.6.2	Monitores	32
2.6.3	Troca de Mensagens	35
2.7	<b>EQUIVALÊNCIA ENTRE PRIMITIVAS DE SINCRONIZAÇÃO</b>	<b>38</b>
2.7.1	Usando Semáforos para Simulação de Monitores e Troca de Mensagens	39
2.7.2	Usando Monitores para Simulação de Semáforos e Troca de Mensagens	40
2.7.3	Usando Troca de Mensagens para Simulação de Semáforos e Monitores	41
2.8	<b>PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO</b>	<b>42</b>
2.8.1	The Dining Philosophers Problem	42
2.8.2	The Readers and Writers Problem	44
2.8.2.1	Leitores têm Prioridade - Readers Have Priority	45
2.8.2.2	Escritores têm Prioridade - Writers Have Priority	45
2.8.3	The Sleeping Barber Problem	47
<b>3</b>	<b>COMUNICAÇÃO ENTRE PROCESSOS</b>	<b>48</b>
<b>4</b>	<b>DEADLOCK</b>	<b>49</b>
4.1	<b>O PROBLEMA DE DEADLOCK</b>	<b>49</b>
4.1.1	Caracterização de Deadlock	50
4.1.2	Grafo de Alocação de Recursos	51
4.2	<b>PREVENÇÃO DE DEADLOCK</b>	<b>51</b>
4.2.1	Exclusão Mútua	55
4.2.2	Condição Hold and Wait	55
4.2.3	Ausência de Preempção	55
4.2.4	Espera Circular	55
4.3	<b>ALGORITMOS DE BANKER</b>	<b>56</b>
4.4	<b>DETECÇÃO DE DEADLOCK</b>	<b>57</b>

4.5	REMOÇÃO DO DEADLOCK	58
4.6		58
5	<b>SISTEMAS DISTRIBUÍDOS</b>	<b>60</b>
5.1	<b>SISTEMA DISTRIBUÍDO (SD) X SISTEMA CENTRALIZADO (SC)</b>	<b>60</b>
5.1.1	Vantagens dos SD sobre SC	60
5.1.2	Vantagens dos SD sobre conjunto independente de PCs	61
5.1.3	Desvantagens dos SDs	61
5.2	<b>ARQUITETURAS UTILIZADAS EM SISTEMAS DISTRIBUÍDOS</b>	<b>61</b>
5.2.1	Multiprocessadores com Barramento - Bus-Based Multiprocessors	64
5.2.2	Multiprocessadores Chaveados - Switched Multiprocessors	65
5.2.3	Multicomputadores com Barramento - Bus-Based MultiComputers	66
5.2.4	Multicomputadores Chaveados - Switched Multicomputers	67
5.3	<b>CLASSIFICAÇÃO DOS SISTEMAS OPERACIONAIS</b>	<b>67</b>
5.3.1	Sistemas Operacionais de Rede - SOR	68
5.3.2	Sistemas Operacionais Distribuídos - SOD	69
5.3.3	Multiprocessadores de Tempo Compartilhado - MTC	70
5.4	<b>REQUISITOS PRINCIPAIS DOS SISTEMAS DISTRIBUÍDOS</b>	<b>71</b>
5.4.1	Transparência	71
5.4.2	Flexibilidade	72
5.4.3	Tolerância a Falhas	74
5.4.4	Escalabilidade	74
6	<b>COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS</b>	<b>76</b>
6.1	<b>INTRODUÇÃO</b>	<b>76</b>
6.2	<b>PADRÕES E SISTEMAS ABERTOS</b>	<b>76</b>
6.3	<b>MODELO DE REFERÊNCIA OSI/ISO</b>	<b>76</b>
6.3.1	Cadada OSI	77
6.3.1.1	Camada Física	77
6.3.1.2	Camada de Enlace de Dados	78
6.3.1.3	Camada de Rede	78
6.3.1.4	Camada de Transporte	78
6.3.1.5	Camada de Sessão	79
6.3.1.6	Camada de Apresentação	79
6.3.1.7	Camada de Aplicação	79
6.3.2	Camadas: Visão Funcional	79
6.4	<b>O PROTOCOLO TCP/IP</b>	<b>80</b>
6.4.1	Pilha de Protocolos do TCP/IP	80
6.4.1.1	Camada de Aplicação TCP/IP	81
6.4.1.2	Camada de Transporte TCP/IP	81
6.4.1.3	Camada de Rede TCP/IP	82
6.4.1.4	Camada de Interface da Rede TCP/IP	84
7	<b>ARQUITETURA CLIENTE-SERVIDOR</b>	<b>86</b>
7.1	<b>COMPONENTES DE UMA APLICAÇÃO DISTRIBUÍDA</b>	<b>87</b>
7.2	<b>DISTRIBUIÇÃO DOS COMPONENTES DA APLICAÇÃO</b>	<b>88</b>
7.2.1	Apresentação Distribuída	88
7.2.2	Apresentação Remota	89
7.2.3	Função Distribuída	89
8	<b>COORDENAÇÃO EM SISTEMAS DISTRIBUÍDOS</b>	<b>90</b>
8.1	xxx	90
8.2		90
9	<b>SINCRONIZAÇÃO EM SISTEMAS DISTRIBUÍDOS</b>	<b>91</b>

9.1	xxx	91
9.2		91
10	<b>GERÊNCIA DE PROCESSOS EM SISTEMAS DISTRIBUÍDOS</b>	<b>92</b>
10.1		92
10.2		92
11	<b>GERÊNCIA DE ARQUIVOS EM SISTEMAS DISTRIBUÍDOS</b>	<b>93</b>
11.1		93
11.2		93

## **1 Bibliografia**

### **1.1 Bibliografia Básica**

- Sistemas Operacionais Modernos - Andrew S. Tanembaun, Prentice Hall 1992.
- Introdução à Arquitetura de Sistemas Operacionais- Francis B Machado e Luis Paulo Maia, LTC 1997 2ª Edição.
- Operating System Concepts: Internals and Design Principles – William Stallings, Prentice Hall 1998 3ª Edição
- Operating System Concepts – James L. Peterson & Abraham Silberschatz, Addison-Wesley 1985, 2ª Edição
- An Introduction to Operating Systems - Harrey M. Deitel, Addison-Wesley 1990, 2ª Edição

### **1.2 Bibliografia Adicional**

- Computer Organization and Architecture: Principles Structure and Function – William Stallings, Addison-Wesley 1990, 2ª Edição
- Organização Estruturada de Computadores - Andrew S. Tanembaun, Prentice Hall 1988.

# *Parte I*

# *Programação*

# *Concorrente*

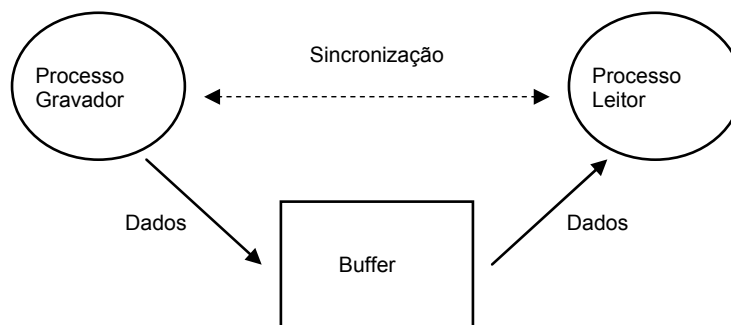
## 2 Programação Concorrente

### 2.1 Comunicação entre Processos e Concorrência

**Problema** – Como deve ser gerenciado o compartilhamento de recursos do sistema, quando diversos processos trabalham juntos, isto é, de forma concorrente ?

Vejamos um exemplo onde dois processos trocam informações através de operações de gravação e leitura em um “buffer”. Um processo só poderá gravar dados no buffer quando ele não estiver cheio.

Da mesma forma, o outro processo só poderá ler dados armazenados no “buffer” se existir algum dado para ser lido. Em ambos os casos, os processos deverão aguardar até que o “buffer” esteja pronto para operações de gravação ou leitura.



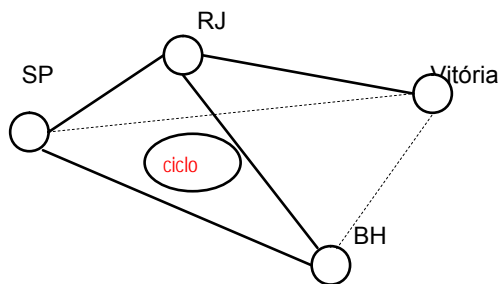
#### 2.1.1 Grafo de Precedência

**Definição** – Um grafo  $G(V,E)$  é uma estrutura algébrica formada por um conjunto de vértices  $V$  e um conjunto de arestas  $E$  ( $E \subset V \times V$ ) que representam as ligações entre os vértices. Um grafo pode ser direcionado ou não-direcionado.

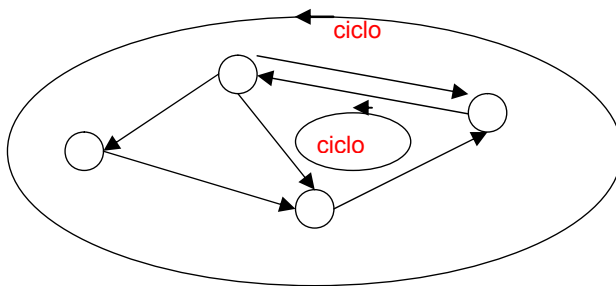
**Definição** – Um grafo é não-direcionado se, e somente se, dados quaisquer  $v, w \in V$  e se tivermos  $(v, w) \in E \Rightarrow (w, v) \in E$  necessariamente.

Vejamos um exemplo:

$V = \{\text{Rio, São Paulo, Belo Horizonte, Vitória}\}$

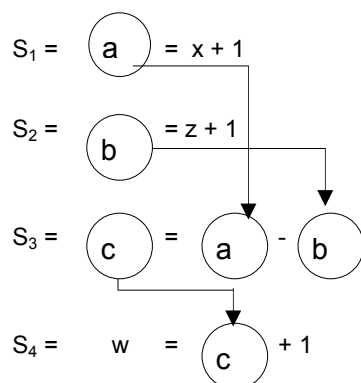


grafo não-direcionado



grafo direcionado

Seja o seguinte segmento de programa, o qual desejamos que execute de forma concorrente (estamos supondo a existência de múltiplas unidades funcionais ou múltiplos processadores):

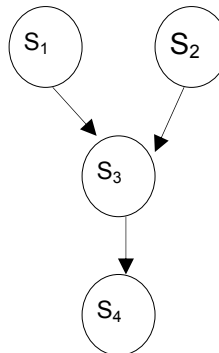


É fácil perceber que existem restrições de precedência entre as diversas instruções. Estas restrições podem ser representadas por um grafo.

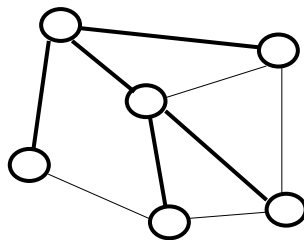
**Definição** – Grafo de precedências é o grafo  $G(V,E)$  ( $V \subset E \times E - V$  conjunto de vértices,  $E$  conjunto de arestas) direcionado acíclico (não tem ciclos) cujos vértices correspondem as instruções, exceto nos casos de loop. Uma aresta do vértice  $S_i$  para o vértice  $S_j$  significa que a instrução  $S_j$  só pode ser executada após a execução da instrução  $S_i$ .



Para o caso anterior (\*) teríamos o seguinte grafo de precedências. Observe que o grafo não tem ciclos (ciclo – caminho fechado):



**Definição:** Uma árvore é um grafo acíclico não direcionado.



Árvore Geradora Mínima  
do Grafo

### 2.1.2 Condições para Concorrência

Quando duas instruções em um programa podem ser executadas concorrentemente e ainda assim produzir o mesmo resultado? A resposta a este problema depende de algumas definições:

**Definição :** Seja  $R(S_i) = \{a_1, \dots, a_n\}$ , é o conjunto de todas as variáveis cujos valores são referenciados (lidos) na instrução  $S_i$  durante a sua execução.

**Definição :** Seja  $W(S_i) = \{b_1, \dots, b_n\}$ , é o conjunto de todas as variáveis cujos valores são alterados (escritos) durante a execução de  $S_i$ .

Exemplos:

$$\begin{aligned}
 &\left\{ \begin{array}{l} R(c = a - b) = \{a, b\} \\ W(c = a - b) = \{c\} \end{array} \right. \\
 &\left\{ \begin{array}{l} R(w = c + 1) = \{c\} \\ W(w = c + 1) = \{w\} \end{array} \right. \\
 &\left\{ \begin{array}{l} R(x = x + z) = \{x, z\} \\ W(x = x + z) = \{x\} \end{array} \right. \Rightarrow R \cap W \neq \{\} \\
 &\left\{ \begin{array}{l} R(\text{read}(a)) = \{\} \\ W(\text{read}(a)) = \{a\} \end{array} \right. \\
 &\left\{ \begin{array}{l} R(\text{write}(a)) = \{\} \\ W(\text{write}(a)) = \{a\} \end{array} \right.
 \end{aligned}$$

**Bernstein's Conditions** - Condições para que e as instruções consecutivas  $S_1$  e  $S_2$  possam ser executadas concorrentemente de forma coerente (isto é, correta).

$$R(S_1) \cap W(S_2) = \{ \} \quad (1)$$

$$W(S_1) \cap R(S_2) = \{ \} \quad (2)$$

$$W(S_1) \cap W(S_2) = \{ \} \quad (3)$$

Exemplo:  $S_1 = a = x + y$   
 $S_2 = b = z + 1$

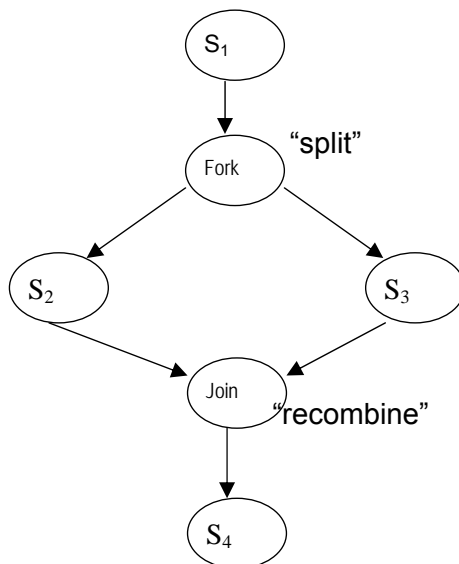
$$\left. \begin{array}{l} R(S_1) = \{x, y\} \\ R(S_2) = \{z\} \\ W(S_1) = \{a\} \\ W(S_2) = \{b\} \end{array} \right\}$$

Neste caso verificamos que as condições estão OK !

## 2.2 Especificação de Concorrência em Programas

O grafo de precedência é uma ferramenta difícil de ser usada computacionalmente em uma linguagem de programação. Existem outros meios de especificar as partes de um programa que devem executar de forma concorrente. As técnicas de compilação mais modernas permitem expressar de forma estruturada e clara a concorrência.

### 2.2.1 Construção Fork e Join



Exemplo – Programa (\*)

```

count = 2;
Fork L1;
a = x + y;
goto L2;
L1: b = z + 1;
L2: Join count;
c = a - b;
w = c + 1;
  
```

count = 2; } Processo A original  
 $S_1$ ; }  
**Fork** B; }

$S_2$ ; }  
 Goto L; } Processo B  
 B: } concorrente com  
 $S_3$ ; } processo  
 , } original  
 , }  
 , }

L: **Join** count; } Sincronização dos  
 Processo A e B

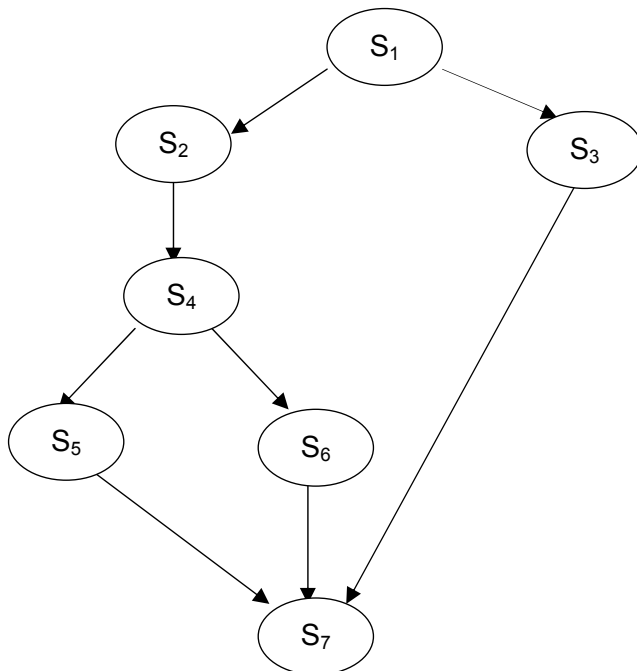
Nós precisamos saber o número de computações que serão "joined", de tal forma que nós terminemos todas as computações exceto a última que deve prosseguir. Assim:

```

Join count => --count;
               if ( count ) exit();
  
```

O comando FORK cria uma cópia (clone) do processo corrente e executa o trecho de código correspondente de forma concorrente. O comando JOIN precisa saber o número de computações concorrentes (dado pelo variável count) de forma que possa terminar todas as computações exceto a última, que deve prosseguir.

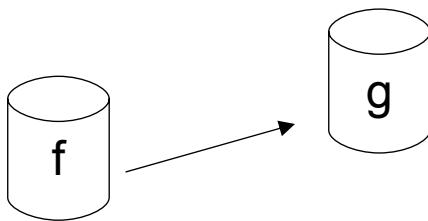
Exercício – Faça a especificação



```

S1;
Count = 3;
Fork L1;
  S2;
  S4;
Fork L2;
  S5;
  Go to L3;
  L2: S6;
  Go to L3;
  L1: S3;
L3: Join count;
S7;
  
```

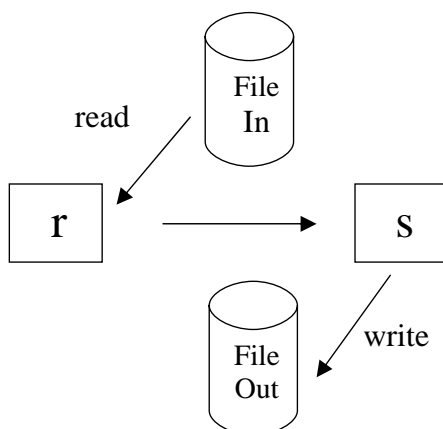
Como exemplo final, consideremos um programa que copie um arquivo *f* para um arquivo *g*.



```

Program CopiaArq;
var f,g:File of Text; r:string[132];
begin
  reset(f);
  rewrite(g);
  read(f, r);
  While not eof(f) do
    begin
      write(g, r);
      read(f, r);
    end;
  write(g, r);
end.
  
```

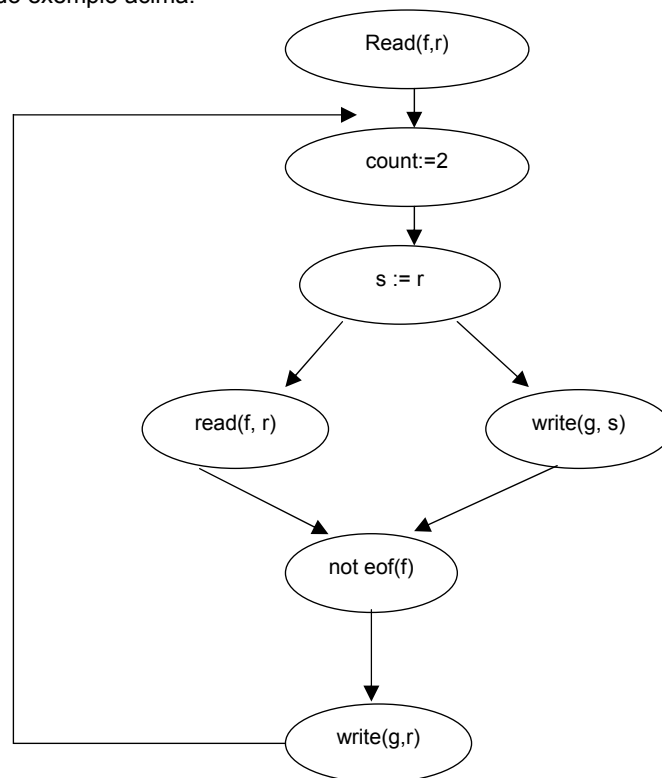
Será que existe alguma forma de explorarmos a concorrência neste exemplo? A solução é a utilização das técnicas de double-buffering para permitir a concorrência entre read e write. Vejamos.



```

Program CopiaArq2
var f,g: File of Text; r,s: string[132];
    count: integer;
begin;
  reset(f);
  rewrite(g);
  read(f, r);
  While not eof(f) do
    begin
      count:= 2;
      s:=r;
      Fork L1;
        write(g, s);
        goto L2;
      L1: read(f, r);
      L2: Join count;
    end;
    Write(g, r);
  end.
  
```

Grafo de precedência do exemplo acima:

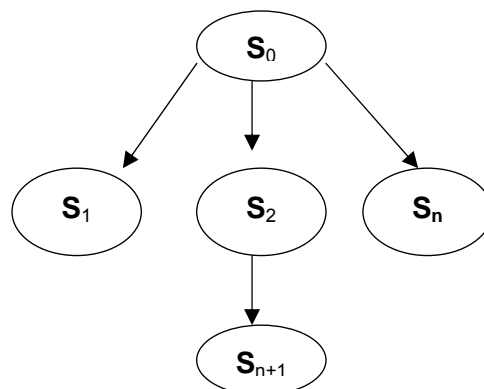


### 2.2.2 Execução Paralela – PARBEGIN e PAREND

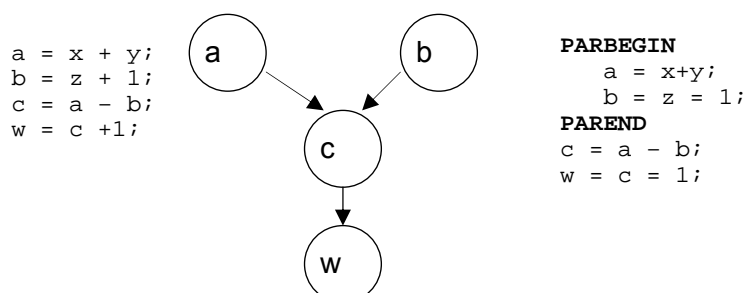
Uma das implementações mais simples de expressar concorrência é a utilização dos comandos PARBEGIN/PAREND, propostos por **Dijkstra** em 1965. A sua forma é a seguinte:

```

S0 ;
PARBEGIN ;
    S1 ;
    S2 ;
    \
    \
    \
    Sn ;
PAREND ;
Sn+1 ;
  
```



Cada comando é uma instrução que pode ser executada concorrentemente e sem ordem previsível. Portanto, o grafo de precedência para este comando pode ser visto acima. O comando PAREND define um ponto de sincronização, onde o processamento só continuará quando todos os processos iniciados já tiverem terminado sua execução. Considerando novamente o programa 1, poderíamos rescrevê-lo da seguinte forma:

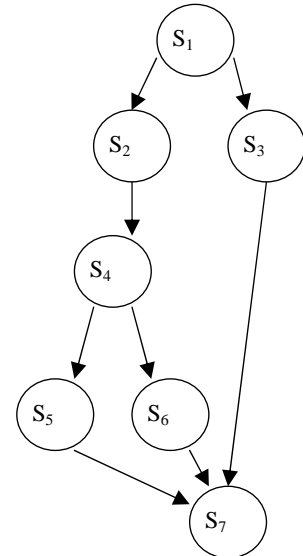


E para o grafo de precedência abaixo teríamos:

```

S1;
PARBEGIN;
  S3;
  begin;
    S2;
    S4;
    PARBEGIN
      S5;
      S6;
    PAREND;
  end;
PAREND;
S7;

```



Rescrevendo o problema da cópia de arquivo, teríamos:

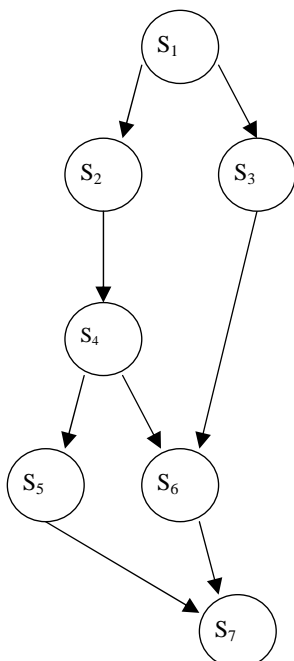
```

Program CopiaArq3
var f, g: File of Text;
    r, s: string[132];
begin
  reset(f);
  read(f, r);
  rewrite(g);
  While not eof(f) do
    begin
      s := r;
      PARBEGIN
        write(g, s);
        read(f, r);
      PAREND;
    end;
    write(g, s);
  end.

```

### 2.2.3 Comparação FORK-JOIN x PARBEGIN-PAREND

Observamos que a construção **PARBEGIN-PAREND** não é poderosa o suficiente para moldar todos os possíveis grafos de precedência. Para ilustrar, vejamos o seguinte exemplo:



```

S1;
count1=2;
Fork L1;
S2;
S4;
count2=2;
Fork L2
S5;
Goto L3
L1: S3;
L2: Join count1;
S6;
L3: Join count2;
S7

```

Para implementarmos este grafo com **PARBEGIN-PAREND** será necessário o uso de mecanismos de sincronização do Sistema Operacional que serão estudados mais adiante.

Para completar, mostramos como implementar **PARBEGIN** e **PAREND** com **FORK** e **JOIN**:

**PARBEGIN/PAREND**

```
S0 ;
PARBEGIN ;
S1 ;
S2 ;
    \
    \
    \
    \
Sn ;
PAREND ;
Sn+1 ;
```

**FORK/JOIN**

```
S0 ;
count := N ;
fork L2 ;
fork L3 ;
fork L4 ;
    \
    \
    \
    \
fork Ln ;
S1 ;
Goto Ln+1 ;
L2 : S2 ;
Goto Ln+1 ;
    \
    \
    \
    \
Ln : Sn ;
Ln+1 : join count ;
```

Para terminar, colocamos mais um exemplo do uso do comando de concorrência **PARBEGIN-PAREND**. Suponhamos o caso em que desejamos calcular o valor da expressão aritmética:-

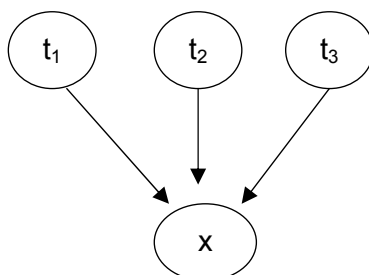
$x := \text{sqrt}(1024) + 35.4 * y - 302 / 2;$

Rescrevendo a expressão teríamos:

```
t1 = sqrt(1024);
t2 = 35.4 * y
t3 = 302 / z;

x := t1 + t2 + t3
```

Grafo de precedência:



```
Program Expressao;
var
    x,t1,t2,t3:real;
begin
    PARBEGIN;
        t1 = sqrt(1024);
        t2 = 35.4 * y;
        t3 = 302 / z;
    PAREND;
    x := t1 + t2 + t3
    writeln('x= ', x);
end.
```

```
Program Expressao2;
var
    x,t1,t2,t3:real;
    Count:integer;
begin
    count := 3;
    fork L1;
    fork L2;
    t1;
    goto L3;
    L1: t2;
    Goto L3;
    L2:t3;
    L3: join count;
    x := t1 + t2 + t3;
end.
```

## 2.3 Problema Compartilhamento de Recursos – Região Crítica

Vamos agora estudar os problemas relacionados ao compartilhamento de recursos entre processos que estejam cooperando para realizar uma determinada tarefa. Vejamos alguns exemplos:

### (i) Atualização de saldo de cliente em um arquivo de contas bancárias:

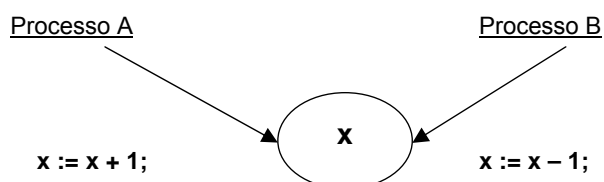
```
read(arqcontas, regcliente);
readln(valordepret);  $\leftarrow$  valor depositado ou retirado
regcliente.saldo := regcliente.saldo + valordepret;
write(arqcontas, regcliente);
```

Imaginemos duas atualizações simultâneas do saldo do cliente, geradas através de dois acessos distintos:

Atualização	Comando	Saldo Arquivo	Valor Dep Ret	Saldo Memória
1	Read	1.000,00	-	1.000,00
1	Readln	1.000,00	-200,00	1.000,00
1	:=	1.000,00	-200,00	800,00
2	Read	1.000,00	-	1.000,00
2	Readln	1.000,00	+300,00	1.000,00
2	:=	1.000,00	+300,00	1.300,00
1	Write	800,00	-200,00	800,00
2	Write	1.300,00	+300,00	1.300,00

Observe que independentemente de que atualização seja feita primeiro, o resultado no final será inconsistente.

### (ii) Compartilhamento de Variáveis em Memória



Inicialmente, o valor de x é 2, os comandos de atribuição em linguagem de alto nível são traduzidos para linguagem de máquina da seguinte forma:

#### Processo A

```
Load R0,x
Add R0,1
Store R0,x
```

#### Processo B

```
Load R0,x
Sub R0,1
Store R0,x
```

Processo	Instrução	X	R <sub>0</sub> A	R <sub>0</sub> B
A	Load	2	2	-
A	Add	2	3	-
B	Load	2	3	2
B	Sub	2	3	1
A	Store	3	3	1
B	Store	1	3	1

(iii)    **“Too Much Milk Problem”**Pessoa A

9.00    Vai a geladeira;  
          Procura Leite;  
       9.05    Se não tem Leite então  
       9.10    Vai para supermercado;  
       9.15    Compra Leite;  
       9.20    Chega em casa;  
          Toma Leite;

Pessoa B

-  
 -  
 -  
 Vai a geladeira;  
 Procura Leite;  
 Se não tem Leite então  
   Vai para supermercado;  
   Compra Leite;  
   Chega em casa;  
**“Agora tem muito Leite na geladeira !”;**

Por estes exemplos, fica fácil observar que, para evitarmos o problema de compartilhamento é preciso impedir que dois ou mais processos acessem um mesmo recurso no mesmo “instante de tempo”. Para isso, enquanto o processo estiver acessando determinado recurso, todos os outros que queiram acessar este mesmo recurso, deverão esperar até que o primeiro processo termine o acesso.

Essa idéia de exclusividade de acesso é chamada de **Exclusão Mútua (Mutual Exclusion)**. A exclusão mútua deve apenas afetar os processos concorrentes quando um deles estiver fazendo acesso ao recurso compartilhado. A parte do código onde é feito o acesso ao recurso compartilhado é denominada **Região Crítica**. A idéia é evitar que dois processos entrem em suas regiões críticas ao mesmo tempo, i.e., garantindo-se a execução mutuamente exclusiva das regiões críticas, evitaremos os problemas decorrentes do **Compartilhamento de Recursos**.

Os mecanismos que implementam a exclusão mútua utilizam um protocolo de acesso à região. Toda vez que um processo for executar sua região crítica ele obrigatoriamente executará antes, um protocolo de entrada para verificar a possibilidade de execução da região crítica, da mesma forma que, ao sair, deverá executar um protocolo de saída.

BEGIN

**Entrada\_Região\_Crítica;**    /\* protocolo de entrada \*/

**Região\_Crítica;**

**Saída\_Região\_Crítica;**    /\* protocolo de saída \*/

END;

Portanto, para os nossos exemplos deveríamos fazer o seguinte:

- (i) Acesso ao arquivo de contas deve ser exclusivo; este teste de exclusividade pode ser feito na abertura do arquivo (ou na leitura do registro) que será atualizado simultaneamente, desde que seja garantido pelo S.O.

```

Unix           }  open ( arq, proteção, O_EXCL );
Windows        }
```

- (ii) O acesso ao valor da variável **x** também deve ser exclusivo, bloqueando a outros processos, enquanto o processo estiver atualizando a variável **x**.

- (iii) Deixar uma nota indicando a compra do leite.

Processos A & B

```

if (No leite) {
  if (No Nota) {
    Deixa_Nota;
    Vai até supermercado;
    Compra leite;
    Retorna para casa;
    Remove_Nota;
  }
}
```

Antes de colocarmos formalmente o problema de Compartilhamento de Recursos, que conforme já vimos, pode ser resolvido criando-se o conceito de Região Crítica dentro da qual impomos a condição de exclusão mútua, vamos apresentar um problema bastante comum em Sistemas Operacionais que é o problema de processos Produtor/Consumidor.



Este problema acontece em diversas situações como por exemplo, quando um driver de impressora produz caracteres que devem ser consumidos, isto é, impressos pela impressora; ou quando o compilador produz código em linguagem assembly para o montador assembler que por sua vez produz o módulo objeto que será consumido pelo link-editor para ligá-lo com as bibliotecas do sistema para gerar o módulo executável.

Para permitirmos que os processos Produtor e Consumidor executem de forma concorrente devemos criar um buffer que possa ser gravado pelo Produtor e lido pelo Consumidor de forma sincronizada, entretanto devemos evitar que o Consumidor não tente consumir um item que ainda não foi produzido pelo Produtor.

Existem 2 versões deste problema com relação ao tamanho do Buffer. No caso de Buffer ilimitado (**unbounded-buffer**) apenas o Consumidor deve esperar pela produção de um novo item indefinidamente. No caso do Buffer de tamanho limitado (**bounded-buffer**) o Produtor deve esperar se todos os buffers estiverem cheios, isto é, se ao menos um buffer ainda não foi consumido pelo Consumidor.

### **2.3.1 Produtor x Consumidor com Buffer Limitado - The Bounded-Buffer Problem**

A seguir apresentamos uma solução para o problema Produtor-Consumidor com Buffer limitado, que é implementado como um ARRAY de TAMBUF elementos compartilhados. Utilizamos um contador de elementos disponíveis no Buffer ( variável *cont* ), dois índices *in* e *out* para indicar a próxima posição livre do Buffer ( variável *in* ) e a próxima posição ocupada do Buffer ( variável *out* ). O Buffer está vazio quando tivermos *in* = *out* ( ou *cont* = 0 ) e cheio quando *in* + 1 mod TAMBUF = *out* ( ou *cont* = TAMBUF ).

```

Program Produtor_Consumidor_1;
  CONST TAMBUF = ... (* tamanho qualquer *);
  TYPE TIPO_DADO = ... (* tipo qualquer *);
  VAR Buffer : ARRAY[0..TAMBUF-1] of TIPO_DADO; cont: 0..TAMBUF;
Procedure Produtor;
  VAR dado: TIPO_DADO;
      in : 0..TAMBUF-1;
  BEGIN
    in:= 0;
    Repeat
      ProduzDado(dado);
      if (cont = TAMBUF) then sleep();      (* Buffer cheio *)
      GravaBuffer(dado, in);
      cont := cont + 1;
      if (cont = 1) wakeup(consumidor);    (* Buffer estava vazio *)
    until False;                          (* Acorda um provável consumidor *)
  END;
Procedure Consumidor;
  VAR dado: TIPO_DADO;
      out: 0..TAMBUF-1;
  BEGIN
    out:=0;
    Repeat
      if (cont = 0) sleep();                (* Buffer vazio *)
      LeBuffer(dado, out);
      cont := cont - 1;
      if (cont == TAMBUF-1) wakeup(Produtor); (* Buffer estava cheio *)
      ConsomeDado(dado);                    (* Acorda um provável produtor *)
    until False;
  END;

```

```

Procedure GravaBuffer(VAR dado: TIPO_DADO; VAR in:0..TAMBUF-1)
  BEGIN
    Buffer[in] := dado;
    in := in + 1 mod TAMBUF;
  END;

```

```

Procedure LeBuffer(VAR dado: TIPO_DADO; VAR out:0..TAMBUF-1)
  BEGIN
    Dado := Buffer[out];
    Out := out + 1 mod TAMBUF;
  END;

```

O código do programa principal deve disparar de forma corrente os processos **Produtor** e **Consumidor**.

```

BEGIN                                (* Programa Principal - Bounded Buffer Problem *)
  cont := 0;
  PARBEGIN
    Produtor;
    Consumidor;
  PAREND;
END;

```

Observamos que apesar das implementações das rotinas Produtor e Consumidor estarem corretas separadamente, esta implementação pode não funcionar corretamente quando executada concorrentemente. Vejamos o porquê de tal fato.

Suponha que **cont** seja igual a 5 e que os processos Produtor e Consumidor executem as instruções abaixo:

```

cont := cont + 1;    - Produtor

cont := cont - 1;    - Consumidor

```

Estas instruções serão implementadas em linguagem de máquina da seguinte forma:

Produtor	Consumidor
LR Ro, cont	LR Ro, cont
ADD Ro, 1	SUB Ro, 1
STO cont, Ro	STO cont, Ro

Aqui novamente podemos obter o seguinte resultado:

```

t0: Produtor executa LR Rop, cont    (* Rop = 5 *)
t1: Produtor executa ADD Rop, 1      (* Rop = 6 *)
t2: Consumidor executa LR Roc, cont  (* Roc = 5 *)
t3: Consumidor executa SUB Roc, 1    (* Roc = 4 *)
t4: Produtor executa STO cont, Rop   (* cont = 6 *)
t5: Consumidor executa STO cont, Roc (* cont = 4 *)

```

Onde obteremos um valor incorreto para a variável cont, indicando que existem 4 buffers completos quando no realidade existem 5!

Um segundo problema pode ocorrer. Suponha que o Buffer esteja vazio e o processo Consumidor executa uma instrução de LR para consultar o valor da variável cont (LR Roc, cont). Neste instante suponha que haja uma troca de contexto para o processo Produtor, que produz um dado e incrementa o valor de cont para 1.

Ao perceber que o valor de `cont` era zero ele acredita que existe um processo Consumidor em estado `sleep` e executa a `system call wakeup` para acordar o Consumidor. Como o Consumidor não tinha entrado no estado de `sleeping` (pois tinha sido interrompido antes de testar o valor de `cont`) o sinal de `wakeup` será perdido (a menos que o SO gerencie este problema, como por exemplo bloquear o processo que enviou o sinal até que outro processo possa receber este sinal). Quando o Consumidor retornar a sua execução ele entrará em `sleep` (sem ser acordado, pois o sinal de `wakeup` já foi perdido). Mais cedo ou mais tarde quando o processo Produtor encher o Buffer, ele também entrará no estado de `sleep` e **ambos os processos Produtor e Consumidor dormirão para sempre!**

A essência do problema nos dois casos é que nós permitimos o acesso concorrente a variável `cont` de forma irrestrita. Observamos que a execução concorrente dos comandos viola as condições de Bernstein:

Processo Produtor	Processo Consumidor
S1: <code>cont := cont + 1;</code>	S2: <code>cont := cont - 1;</code>

E neste caso nós temos os seguintes valores para os conjuntos R e S das instruções de S1 e S2:

$$\begin{aligned}
 R(S1) &= \{\text{cont}\} = W(S1) & R(S2) &= \{\text{cont}\} = W(S2) \\
 \text{E portanto} & & R(S1) \cap W(S2) &= \{\text{cont}\} \neq \emptyset \\
 & & W(S1) \cap R(S2) &= \{\text{cont}\} \neq \emptyset \\
 & & W(S1) \cap W(S2) &= \{\text{cont}\} \neq \emptyset
 \end{aligned}$$

Para resolvermos este problema devemos garantir que o acesso/alteração da variável `cont` caracteriza a presença de uma Região Crítica. Passemos agora a definição do Problema de Exclusão Mútua.

## 2.4 Definição do Problema de Exclusão Mútua

Considere um sistema de  $n$  processos  $\{P_1, P_2, \dots, P_n\}$  operando de forma concorrente e cooperativa. Cada processo tem um segmento de código, chamado **Região Crítica**, no qual cada processo pode estar lendo variáveis, atualizando tabelas ou arquivos compartilhados. Assumimos que quando um processo está executando na sua **Região Crítica** nenhum outro processo pode estar executando a sua respectiva **Região Crítica**, isto é, **a execução das Regiões Críticas de cada processo deve ser mutuamente exclusiva no tempo**. O problema de **Região Crítica** é então o de projetar um protocolo (de acesso e liberação da **Região Crítica**) que possa ser usado pelos processos para permitir a sua cooperação.

Uma solução para o problema de exclusão mútua deve satisfazer então os seguintes requisitos:

### (i) "Exclusão Mútua"

Se o processo  $P_i$  está executando na sua Região Crítica, então nenhum outro processo  $P_j$ ,  $j \neq i$  pode estar executando na sua respectiva Região Crítica.

### (ii) "Progressividade"

Nenhum processo rodando fora de sua Região Crítica pode bloquear a execução de outros processos, isto é, somente os processos que estejam querendo entrar em suas respectivas Regiões Críticas ou esteja executando na sua Região Crítica podem participar da decisão de qual será o próximo processo a executar a sua Região Crítica. Resumindo:

Se  $P_j$  não está na Região Crítica  $j$ , então  $P_j$  não bloqueia nenhum  $P_i$ ,  $i \neq j$

(iii) “Espera Limitada”

Nenhum processo deverá esperar indefinidamente para entrar em sua Região Crítica.

(iv) “Independência de HW”

Nenhuma hipótese pode ser assumida com relação a velocidade de execução de cada processo nem quanto ao número de CPUs estarão disponíveis.

A partir de agora nós estudaremos as soluções algorítmicas (por software) do problema de **Região Crítica** satisfazendo os requisitos levantados anteriormente. Começaremos com o caso de 2 processos  $P_0$  e  $P_1$  onde a estrutura geral do problema é:

```

Program Região_Crítica;
BEGIN
  (* Variáveis e tipo globais *)
  .
  .
  .
  PARBEGIN
    P0;
    P1;
  PAREND;
END.

Procedure P0; (idem P1)
BEGIN
  Entrada_Região_Crítica;    ← 1 - Protocolo de Entrada
  Região_Crítica;
  Saída_Região_Crítica;      ← 2 - Protocolo de Saída
END;
  
```

### 2.4.1 Soluções Algorítmicas para o caso de 2 Processos

#### Algoritmo 1 – Strict Alternation

Este algoritmo se baseia no uso de uma variável inteira compartilhada “**Vez**”

Processo $P_0$	Processo $P_1$
Repeat	Repeat
While Vez $\neq$ 0 do sleep();	While Vez $\neq$ 1 do sleep();
Região_Crítica_ $P_0$ ;	Região_Crítica_ $P_1$ ;
Vez := 1;	Vez := 0;
Processamento Longo;	Until False;
Until False;	

Esta solução garante que apenas um processo executará a sua **Região Crítica** (i). Inicialmente a variável **Vez** é igual a 0 e o processo  $P_0$  pode executar a sua região crítica. O processo  $P_1$ , pôr sua Vez, fica esperando enquanto **Vez** for igual a 0, ou seja, enquanto  $P_0$  estiver em sua região crítica. O processo  $P_1$  só executará sua região crítica quando  $P_0$  atribuir 1 a variável **Vez**.

O problema desta solução surge no próprio mecanismo de controle utilizado. É possível que um processo queira executar sua região crítica ( $P_1$ ) e não possa, por estar bloqueado por outro processo  $P_0$ , que está fora de sua região crítica. Como o processo  $P_1$  executa seu loop mais rapidamente que o processo  $P_0$ , a possibilidade de executar a sua região crítica fica limitada pela velocidade do processo  $P_0$ .

Processo $P_0$	Processo $P_1$	Vez
While Vez <> 0	While Vez <> 1	0
Regiao_Critica_Po;	While Vez <> 1	0
Vez := 1;	While Vez <> 1	1
Processamento_Longo;	Regiao_Critica_P1	1
Processamento_Longo;	Vez := 0;	0
Processamento_Longo;	While Vez <> 1	0

**=> Desta forma violamos a condição (ii), pois  $P_1$  está sendo bloqueado pelo processo  $P_0$  quando está fora de sua região crítica !**

### Algoritmo 2

Para solucionar o problema da falta de memória do estado de cada processo individualmente, criamos uma variável array **Flag** contendo a informação se cada processo está ou não executando a sua **Região Crítica**, isto é:

```
VAR Flag: array[0..1] of boolean;
```

```
Flag[0] := false; Flag[1] := false;
```

```
e Flag[i] = True => Processo  $P_i$  (i=1,n) está executando sua Região Crítica.
```

A estrutura geral de cada processo é:

Processo $P_0$	Processo $P_1$
Repeat	Repeat
While Flag[1] do sleep(); Flag[0] := True;	While Flag[0] do sleep(); Flag[1] := True;
Região_Critica_Po;	Região_Critica_P1;
Flag[0] := False;	Flag[1] := False;
Until False;	Until False;

Este algoritmo não garante que somente um processo de cada Vez esteja executando sua região crítica. Por exemplo, considere a seguinte sequência de execução:

```
t0:  $P_0$  executa While Flag[1] e encontra Flag[1] = false
t1:  $P_1$  executa While Flag[0] e encontra Flag[0] = false
t2:  $P_1$  faz Flag[1] := True e entra na sua Região Crítica;
t3:  $P_0$  faz Flag[0] := True e entra na sua Região Crítica;
```

**=> E dessa forma nós violamos a condição (i) de exclusão mútua !**

**Algoritmo 3**

O problema do Algoritmo 2 é que  $P_1$  tomou uma decisão a respeito do estado de  $P_0$  antes de  $P_0$  ter a oportunidade de alterar seu estado. Podemos tentar resolver este problema fazendo uma pequena modificação no código do Algoritmo 2:

Processo $P_0$	Processo $P_1$
Repeat  Flag[0] := True; While Flag[1] do sleep();	Repeat  Flag[1] := True; While Flag[0] do sleep();
Região_Crítica_Po;	Região_Critica_P1;
Flag[0] := False;  Until False;	Flag[1] := False;  Until False;

Aqui a semântica da variável Flag muda e passa a indicar que quando **Flag[i] = True indica que  $P_i$ , ( $i=0..1$ ) "deseja entrar" em sua Região Crítica**. Nesta situação, diferentemente do Algoritmo 2, a **exclusão mútua é garantida**. Infelizmente a condição (ii) ainda não é satisfeita pois no caso em que tivermos a seguinte seqüência de execução teremos:

to:  $P_0$  faz Flag[0] := True;  
t1:  $P_1$  faz Flag[1] := True;

**=> Agora  $P_0$  e  $P_1$  entrarão em loop infinito em seus respectivos While, o que viola a condição de Progressividade !**

**Algoritmo 4 – Peterson's Solution**

Em 1981, **G. L. Peterson** descobriu uma solução para este problema que se mostrou não ser trivial, conforme podemos observar. A solução é basicamente uma combinação do Algoritmo 3 com uma pequena modificação do Algoritmo 1 de **Strictly Alternation**.

Os processos compartilham duas variáveis: VAR **Flag**: array[0..1] of boolean;  
**Vez**: 0..1;

Inicialmente fazemos Flag[0] := false; Flag[1] := false; e Vez := 0. A estrutura dos Processos é dada abaixo:

Processo $P_0$	Processo $P_1$
Repeat  Flag[0] := True; Vez := 1; While Flag[1] and Vez = 1 do sleep();	Repeat  Flag[1] := True; Vez := 0; While Flag[0] and Vez = 0 do sleep();
Região_Crítica_Po;	Região_Critica_P1;
Flag[0] := False;  Until False;	Flag[1] := False;  Until False;

Para provarmos a correção do Algoritmo devemos mostrar que as **condições (i) – “Exclusão Mútua”, (ii) – “Progressividade”, (iii) – “Espera Limitada”** sejam satisfeitas. A condição **(iv)** é claramente satisfeita pela própria implementação.

Provemos (i) – Cada processo  $P_i$  entra em sua região crítica somente se  $\text{Flag}[j, j \neq i] = \text{false}$  ou  $\text{Vez} = i$ . Suponhamos que ambos processos  $P_0$  e  $P_1$  estejam executando suas **Regiões Críticas** ao mesmo tempo.. Dessa forma teremos  $\text{Flag}[0] = \text{Flag}[1] = \text{True}$ ; esta condição mostra que  $P_0$  e  $P_1$  não puderam executar os seus respectivos **While** ao mesmo tempo, já que o valor de  $\text{Vez}$  só pode ser 0 ou 1. Dessa forma um dos processos, digamos  $P_1$  deve ter executado com sucesso o **While** enquanto  $P_0$  teve que executar uma instrução a mais  $\text{Vez} := 1$ ; mas como tínhamos  $\text{Flag}[0] = \text{Flag}[1] = \text{True}$  e esta condição permanecerá enquanto  $P_1$  estiver na sua **Região Crítica** concluímos que a exclusão mútua é preservada.

Para provarmos (ii) e (iii) observamos que um processo  $P_i$  deixa de entrar na **Região Crítica** apenas se está preso na execução do loop com a condição  $\text{Flag}[j] = \text{True}$  e  $\text{Vez} = j$ . Se  $P_j$  não quer entrar na **Região Crítica** então  $P_i$  pode entrar. Se  $P_j$  faz  $\text{Flag}[j] := \text{True}$  e está executando o loop **While** então ou  $\text{Vez} = i$  ou  $\text{Vez} = j$ . Se  $\text{Vez} = i$  então  $P_i$  entrará na **Região Crítica**. Se  $\text{Vez} = j$ , então  $P_j$  entrará na **Região Crítica**. Entretanto, quando  $P_j$  sair da **Região Crítica** ele colocará  $\text{Flag}[j] := \text{false}$ , permitindo  $P_i$  entrar na **Região Crítica**. Se  $P_j$  colocar  $\text{Flag}[j] := \text{True}$  ele colocará também  $\text{Vez} = i$ . Assim, já que  $P_i$  não altera o valor da variável  $\text{Vez}$  enquanto executando o loop **While**,  $P_i$  entrará na **Região Crítica** ( portanto isto mostra que a condição de **Progressividade** é satisfeita ) após no máximo uma entrada de  $P_j$  ( o que nos garante que a condição de **Espera Limitada** também é satisfeita ).

## 2.4.2 Solução Algorítmica para o caso N Processos

No item anterior nós vimos que o **Algoritmo de Peterson** resolve o problema de **Região Crítica** para dois processos. Mostraremos agora a solução para o problema de N processos querendo entrar na **Região Crítica**. O primeiro algoritmo é de autoria de **Eisemberg e McGuire (1972)** e o segundo é de **Leslie Lamport (1974)**.

### Algoritmo 5 – Eisemberg & McGuire

```

Program RegiaoCriticaNProcessos;
CONST N=...;
TYPE TSTATUS = (IDLE, WANT_IN, IN_CS);

VAR Flag: array[0..N-1] of TSTATUS;
    Vez: 0..N-1;

Procedure Processo_i; /*Código do "iésimo" processo comum a todos os processos */
VAR                                     /* Inicialmente temos Flag[j]:= IDLE; j=0,N-1 */
    j: 0..N;                             /* e Vez é qualquer valor entre 0 e N-1 */
BEGIN
    Repeat
        Repeat                                     /* Inicio do Protocolo de Entrada */
            Flag[i] := WANT_IN;
            j := Vez;
            While j <> i do
                if Flag[j] <> IDLE then
                    j := Vez;
                else
                    j := j + 1 mod N;
            Flag[i] := IN_CS;
            j := 0;
            While (j < N) and (j = i or Flag[j] <> IN_CS) do
                j := j + 1;
        Until (j >= N) and (Vez = i or Flag[Vez] = IDLE);
        Vez := i;                                     /* Fim do Protocolo de Entrada */

        Regiao_Critica;

        j := Vez + 1 mod N;                             /* Inicio do Protocolo de Saída */
        While Flag[j] = IDLE do
            j := j + 1 mod N;
        Vez := j;
        Flag[i] := IDLE;                                 /* Fim do Protocolo de Saída */
        ...
    Until false;
END.

```

Para provarmos a correção deste algoritmo é necessário mostrar:

- (i) **Exclusão Mútua é preservada;**
- (ii) **Nenhum processo executando fora de sua Região Crítica pode ser bloqueado por outros processos;**
- (iii) **Nenhum processo deve esperar para sempre para entrar na sua Região Crítica.**

- Provemos (i)

Note que cada processo  $P_i$  entra na sua **Região Crítica** somente se  $Flag[j] \neq IN\_CS \forall j \neq i$ . A exclusão mútua vem de fato de que somente  $P_i$  seta  $Flag[i] = IN\_CS$  e só o faz quando  $Flag[j] \neq IN\_CS \forall j = 0, N-1$ .

- Provemos (ii)

O valor da variável *Veiz* só pode ser mudado quando um processo entra na sua **Região Crítica**. Portanto se nenhum processo entra ou sai de sua **Região Crítica**, o valor de *Veiz* permanece inalterado. Além disso, **o primeiro processo, segundo a ordem cíclica (i, i+1, ..., n-1, 0, ..., i-1), que estiver querendo entrar na sua Região Crítica acabará entrando.**

- Provemos (iii)

Observe que quando um processo deixa a **RC (Região Crítica)**, ele deve designar o seu sucessor (alterando o valor da variável *Veiz*) seguindo a ordem cíclica assegurando dessa forma que qualquer processo que queira entrar na **RC** o fará em, no máximo,  $n-1$  rodadas (alterações cíclicas da variável *Veiz*).

O próximo algoritmo, devido a **Lamport (1974)** apresenta uma estratégia diferente. O algoritmo foi projetado para um ambiente distribuído e é baseado em algoritmo de escalonamento comumente utilizado em padarias, confeitarias, açougue, etc. Uma *Veiz* o cliente entrando na padaria, ele recebe um número e o cliente com o menor número é atendido primeiro, o que caracteriza um esquema **FCFS ou FIFO**. Pôr ser projetado para um ambiente distribuído torna-se impossível garantir que dois processos distintos recebam números diferentes. Neste caso, quando tivermos uma situação de empate (isto é,  $Numero[i] = Numero[j]$ ) o desempate se dará através da escolha do processo de menor ordem, isto é, se  $Numero[i] = Numero[j]$  então  $P_i$  será atendido se  $i < j$ . Para efeito didático criamos a notação: **(a, b) < (c, d) significando que:  $a \leq c$  e  $b < d$ .**

#### Algoritmo 6 – Bakery Algorithm – Leslie Lamport

```

Program RegiaoCriticaNProcessos;
CONST
    N = ...;
VAR
    Choosing: array[0..N-1] of boolean;    /* Inicializada com false */
    Numero:   array[0..N-1] of integer;    /* Inicializada com 0 */

Procedure Processo_i;
VAR
    j: integer;
BEGIN
    Repeat
        Choosing[i] := True;                /* Início do Protocolo de Entrada */
        Numero[i] := max(Numero[0], Numero[i], ..., Numero[N-1]) + 1;
        Choosing[i] := false;
        for j:=0 to N-1 do
            BEGIN
                While Choosing[j] do
                    ;                        /* espera j obter o ticket */

                While Numero[j] <> 0 and (Numero[j],j) < (Numero[i],i) do
                    ;                        /* espera j sair da RC */

            END;
        /* Fim do Protocolo de Entrada */

        Regiao_Critica;

        Numero[i] := 0;                    /* Início/Fim do Protocolo de Saída */
    Until false;
END.

```



### Prova do Bakery Algorithm

A prova do algoritmo consiste em mostrar primeiro que se  $P_i$  está na **RC** e  $P_k$  ( $k \neq i$ ) já fez  $\text{Numero}[k] < 0$ , então a condição  $(\text{Numero}[i], i) < (\text{Numero}[k], k)$  deve ser satisfeita pois é ela que controla a entrada de  $P_i$  na sua **RC**. É fácil ver que se  $P_i$  está na **RC** então  $\text{Choosing}[i] = \text{false}$  e  $\text{Numero}[k] \geq \text{Numero}[i]$ . Se  $\text{Numero}[k] > \text{Numero}[i]$  então teremos  $(\text{Numero}[i], i) < (\text{Numero}[k], k)$  mas se tivermos  $\text{Numero}[k] = \text{Numero}[i]$  concluímos que ambos começaram a execução do protocolo de entrada na **RC** em paralelo, entretanto já que  $P_i$  está na sua **RC** então devemos ter  $i < k$  o que mostra que a condição  $(\text{Numero}[i], i) < (\text{Numero}[k], k)$  é satisfeita.

A **Exclusão Mútua** é decorrência da condição do item anterior, pois se  $P_i$  está na **RC** e  $P_k$  está tentando entrar, ele encontrará  $\text{Numero}[k] < 0$  e  $(\text{Numero}[i], i) < (\text{Numero}[k], k)$  e dessa forma  $P_k$  continuará preso ao **While** até que  $P_i$  deixe a **RC**.

Observe que a condição de **Espera Limitada** também é satisfeita já que neste caso os processos entram na **RC** segundo uma política **First Come First Served (FCFS)**.

## 2.5 Soluções de Hardware

Até agora foram apresentadas algumas soluções para o problema de **Exclusão Mútua (PEM)**. Todas estas soluções têm em comum o fato de provocarem que processos que desejem entrar na Região Crítica e que estejam impedidos de fazê-lo, entram em um estado de “**Espera Ocupada**” (também chamado de **busy-waiting**), aguardando a liberação da área compartilhada. Antes de passarmos a mecanismos mais sofisticados, que evitem a ocorrência de **busy-waiting**, vamos estudar alguns mecanismos de hardware que permitem implementar soluções para o problema de **Exclusão Mútua** com o auxílio do HW.

### 2.5.1 Desabilitar Interrupções

A solução mais simples para o problema de **Exclusão Mútua (PEM)** é fazer com que cada processo que deseje entrar na sua **Região Crítica (RC)** primeiro possa desabilitar todas as interrupções externas e as reabilite após deixar a **Região Crítica (RC)**. Como a mudança de contexto só pode ser realizada através de interrupções (de relógio, término de uma SVC de E/S, etc.), o processo que as desabilitou terá acesso exclusivo garantido, para ler e atualizar as variáveis compartilhadas da **Região Crítica**.

```
Program Região_Crítica_DisableInterrupts;
BEGIN
  (* Variáveis e tipos globais *)
  .
  .
  PARBEGIN
    P0;
    P1;
  PAREND;
END.
Procedure P0; (idem P1)
BEGIN
  Desabilita_Interrupções;      ← 1 - Protocolo de Entrada

  Região_Crítica;

  Habilita_Interrupções;        ← 2 - Protocolo de Saída
END;
```

Esse mecanismo é inconveniente por vários motivos. O maior deles acontece quando o processo que desabilita interrupções não retorna a habilitá-las, o que nesse caso fará com que o sistema operacional não funcione corretamente.

A inibição de interrupções, no entanto, pode ser útil ao SO quando este necessita manipular estruturas de dados compartilhadas do Sistema, como lista de processos, etc.

## 2.5.2 Instrução Test and Set e Swap

Muitas máquinas possuem uma instrução especial, que permite ler o conteúdo de uma variável e ao mesmo tempo atribuir um novo valor a essa variável. Este tipo de instrução é chamado **TEST-AND-SET** e tem como característica principal o fato de ser sempre executada sem interrupção, ou seja, trata-se de uma instrução indivisível. Esta característica possibilita a implementação de uma solução para o problema de Exclusão Mútua, já que não existe a possibilidade de dois processos entrarem ao mesmo tempo em RCs que estejam protegidas pela execução de instruções **TEST-AND-SET**, conforme veremos adiante.

```
Function TestAndSet(Var target: boolean):boolean;
BEGIN
    TestAndSet:=target;
    target:=true;
END;
```

Existe ainda a instrução **Swap** que implementa a troca do conteúdo de duas variáveis em um único ciclo de memória.

```
Procedure Swap(var a,b: boolean);
BEGIN
    temp := a; a:= b; b:= temp;
END;
```

Uma solução para o **PEM** usando as instruções **TEST-AND-SET** e **SWAP** poderia ser como abaixo:

<pre>REPEAT     while ret:=TestAndSet(lock) do         ;      Região-Crítica;      lock:= false;     ... UNTIL false;</pre>	<pre>REPEAT     key := true;     repeat         Swap(lock, key);     until key = false;      Região-Crítica;      lock := false;     ... UNTIL false;</pre>
---	---

Uma possível sequência de execução seria a mostrada abaixo onde o Processo 1 entra primeiro na **RC**.

Processo 0	Processo 1	ret 0	ret 1	Lock
	Repeat	-	-	False
Repeat	While ret := ...	-	False	True
While ret := ...	Região-Crítica	True	False	True
While ret := ...	Região-Crítica	True	False	True
While ret = ...	Região-Crítica	True	False	True
While ret = ...	Região-Crítica	True	False	True
While ret = ...	Lock := false	True	False	False
While ret = ...	...	False	False	True
Região-Crítica	While ret = ...	False	True	True
Lock := false	While ret = ...	False	True	False
...	Região-Crítica	False	False	True
While ret = ...	Região-Crítica	True	True	True

Os dois algoritmos apresentados anteriormente não satisfazem a condição de **Espera Limitada** para o **PEM**. A seguir, apresentamos um algoritmo de autoria de **Burns** [1978] que usa a instrução **TestAndSet** e que satisfaz todas as condições requeridas pelo **PEM**.

```

Program Região_Crítica_Test_and_Set;
  VAR waiting: array[0..N-1] of boolean; /* Inicializada com false */
      Lock   : boolean;
      \
      \
Processo_Pi;
  VAR j:0..N-1;
      key: boolean;
  BEGIN
    Repeat

      Waiting[i] := True;                /* Início do Protocolo de Entrada */
      key := True;
      While waiting[i] and key do
        key := TestAndSet(lock);
      Waiting[i] := false;                /* Fim do Protocolo de Entrada */

      Região-Crítica;

      j := i+1 mod N;                    /* Início do Protocolo de Saida */
      While (j<>i) and (not waiting[j]) do
        j:= j+1 mod N;
      If j=i then Lock := false
        else waiting[j] := false; /* Fim do Protocolo de Saida */

    Until false;
  END;

```

**Prova de Exclusão Mútua** → O processo Pi pode entrar na sua RC somente se waiting[i]=false ou key=false. key se torna false apenas quando executada a instrução **TestAndSet**. O primeiro processo a executar **TestAndSet** encontrará key=false e bloqueará todos os restantes. Como apenas um waiting[i] é setado False garante-se a exclusão mútua.

-

**Prova da Progressividade** → É garantida já que uma vez que o processo Pi sai de sua RC coloca a variável Lock=false, ou waiting [i]=false. Isto permite que um próximo processo possa entrar em sua RC.

**Prova da Espera Limitada** → Observe que quando o processo Pi libera a sua RC, ele varre o array waiting de forma cíclica (i+1, i+2, .....n-1,0,....,i-1) e escolhe o próximo processo nesta ordem que deseje entrar (waiting[j]=True).

## 2.6 Soluções de Software (sem Busy-waiting)

Apesar de as soluções até então apresentadas resolverem o problema **PEM**, todas apresentam a deficiência da espera ocupada (**busy-waiting**). Na espera ocupada, toda vez que um processo tenta entrar em sua RC e é impedido de fazê-lo, por já existir outro processo acessando a **RC**, ele fica em loop, testando uma condição, até que seja permitido o acesso. Dessa forma o processo bloqueado consome tempo de CPU desnecessário.

A solução para o problema foi a introdução de primitivas do SO que permitissem que um processo, quando não pudesse entrar em sua RC, fosse colocado no estado de espera até que outro processo o liberasse. As soluções inicialmente inventadas não eram de fácil aplicação em problemas mais complexos e fatores como legibilidade, simplicidade e correção dos programas concorrentes ganharam grande importância com o aumento da complexidade no desenvolvimento de programas concorrentes. Para atender a essas necessidades, **Dijkstra** [1965] introduziu uma nova ferramenta de sincronização, denominada **Semáforo** e, posteriormente **Hanssem e Hogre** [1974] desenvolveram uma estrutura de sincronização, chamada **Monitor**. A seguir discutiremos estas duas ferramentas.

## 2.6.1 Semáforos

Voltando ao problema **Produtor x Consumidor** com Buffer limitado (programa **Produto\_Consumidor\_1** do item 2.3.1), lembramos que a solução apresentada se utilizava de duas primitivas *sleep* / *wakeup* para comunicação entre processo que bloqueiam/desbloqueiam um processo ao invés de mantê-lo em busy-waiting, evitando o consumo de tempo de CPU.

O problema da solução apresentada no algoritmo **Produtor\_Consumidor\_1** é, conforme já comentamos, a possibilidade dos dois processos entrarem em estado de espera. A essência do problema era a possibilidade de um envio de um sinal de *wakeup* para o processo **Consumidor** que ainda não estivesse dormindo. A perda do sinal de *wakeup* levaria o processo a entrar no estado de *sleep* indevidamente já que o aviso para permanecer acordado foi perdido (para maiores detalhes veja a discussão no item 2.3.1). Uma solução para tal problema é a criação de um bit chamado "**wakeup-waiting bit**" no contexto de hardware do processo, para avisar ao processo que um sinal de *wakeup* lhe foi enviado. Logo, quando um *wakeup* é enviado para um processo que já está acordado, este bit é ligado. Quando, mais tarde, o processo tentar dormir, se o "**wakeup-waiting bit**" estiver ligado, ele será desligado e o processo permanecerá acordado. É fácil ver que quando tivermos mais de dois processos executando de forma concorrente teremos a necessidade de criar um número também maior de "**wakeup-waiting bits**".

Esta era a situação em 1965, quando **E. W. Dijkstra** sugeriu o uso de uma variável inteira para contar o número de **wakeups** recebidos, ao invés do uso de diversos bits. Esta variável inteira foi chamada de **Semáforo**. Portanto, temos a seguinte definição:

**Definição** – Um semáforo *S* é uma variável inteira, não negativa, que só pode ser manipulada por duas operações atômicas padrão *Down* e *Up* (ou *P* e *V*, no artigo original de Dijkstra) que são generalizações das instruções *sleep* e *wakeup*. Logicamente, as operações *Down* e *Up* são:

```
Down(S):  While ( S ≤ 0 ) do
            ;
            S := S - 1;
Up(S):    S := S + 1;
```

Modificações no valor da variável *S* pelas instruções **Down** e **Up** são executados de forma atômica, isto é, são indivisíveis. No caso do **PEM** as instruções **Down** e **Up** funcionam como protocolos de entrada e saída para acesso a **Região Crítica**. A idéia é colocar o semáforo associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes. Se seu valor for maior do que zero, nenhum processo está utilizando o recurso; caso contrário, o processo tem seu acesso a **RC** impedido.

Para o caso de *N* processos tentando acessar sua **Região Crítica**, os *N* processos devem compartilhar uma variável **semáforo**, e cada processo *P<sub>i</sub>* pode ser codificado da seguinte forma:

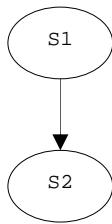
```

Processo Pi
Var mutex: semáforo;          /* variável global compartilhada iniciada com 1 */
BEGIN
  Repeat                      /* semáforos aplicados ao PEM são chamados
    Down(mutex);              Mutexes (mutual exclusion semaphores) ou
    Região-Crítica;         Binários, por assumirem os valores 0 ou 1 */
    Up(mutex);
    .....
  Until false;

END;

```

**Semáforos** podem ser também usados para resolver diversos problemas de sincronização, problema que chamaremos de **Sincronização Condicional**. Suponha que tivéssemos o seguinte grafo de precedências:



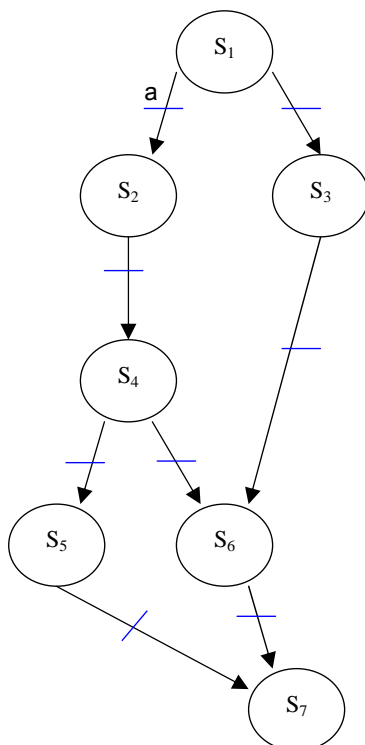
Com o uso de **semáforos** podemos executar cada instrução em um processo diferente desde que possamos garantir que a instrução S2 seja executada após a instrução S1, o que poderia ser feito da seguinte forma:

P1 → S1;  
Up(sync);      onde sync é um semáforo inicializado com 0.

Por que ?

P2 → Down(sync);  
S2;

Como exemplo consideremos agora o grafo de precedências abaixo e o respectivo código concorrente:



```

Program
Var a,b,c,d,e,f,g: semáforo;  /* todas inicializadas com 0 */
BEGIN
  PARBEGIN
    begin
      S1; UP(a); UP(b);
    begin
      Down(a); S2; S4; UP(c); UP(d);
    end;
    begin
      Down(b); S3; UP(e);
    end;
    begin
      Down(c); S5; UP(f);
    end;
    begin
      Down (d); Down(e); S6; UP(g);
    end;
    begin
      Down(f); Down(g); S7;
    end;
  PAREND;
END.

```

Observe que no item 2.2.3 observamos que não podíamos resolver este problema com as instruções **ParBegin** e **ParEnd**, somente com as instruções **Fork** e **Join**. **Você é capaz de dizer o porquê !!!**

Passemos agora a implementação física das instruções **Down** e **Up** de forma a evitar o busy-waiting. Para isto, será necessário a criação de uma lista de processos pendentes, isto é, bloqueados, associados a cada semáforo. Portanto, uma implementação para o semáforo seria:

```
Type Semáforo: record
    valor: integer;
    L      : lista_procs_blocked; /* usualmente implementada como uma
                                   lista de PCBs (Process Control Blocks) */
end;
```

**Procedure Down(var S: semáforo);**

```
BEGIN
    If S.valor <= 0 then
        begin
            adiciona processo a lista de processos bloqueados S.L;
            sleep();
        end
    else
        S.valor := S.valor - 1;
    END;
```

**Procedure Up(var S: semáforo);**

```
BEGIN
    S.valor := S.valor + 1;
    if not vazia(S.L) then
        begin
            remove processo P da lista de processos bloqueados S.L;
            wakeup(P);
        end;
    END;
```

O gerenciamento da lista de processos pode seguir qualquer política (FIFO, LIFO, prioridade, etc.) e não tem influência na correção ou na performance de uma solução que utilize **semáforos**. O importante é garantir que as instruções **Down** e **Up** não possam ser executadas simultaneamente por processos distintos. **Em máquinas com um único processador**, uma solução simples seria a de **inibir interrupções durante as execuções das instruções**, em **máquinas com múltiplos processadores** a solução, no caso de não existir alguma solução disponibilizada pelo Hardware é a *utilização de alguns dos algoritmos apresentados* nas seções 2.3.2 (**Peterson**), 2.3.3 (**Eisemberg e McGuire**) e 2.3.3 (**Bakery Algorithm – Leslie Lamport**).

É importante observar que não eliminamos totalmente a ocorrência do **busy-waiting**. Na realidade, nós reduzimos a execução das instruções **Down** e **Up**, que são muito pequenas (se propriamente codificadas, esta primitivas não devem conter mais do que 10 instruções). Dessa forma, podemos dizer com que o uso de **semáforos** acabamos “virtualmente” com o **busy-waiting**.

A seguir, apresentamos a solução para o problema **Produtor x Consumidor** utilizando **semáforos**. A solução utiliza três semáforos, sendo um utilizado para implementar a **Exclusão Mútua** (mutex) e dois para a **Sincronização Condicional** (vazio e cheio). O semáforo mutex permite a execução das **Regiões Críticas** GravaDado e LeDado de forma mutuamente exclusiva. Os semáforos vazio e cheio representam, respectivamente, se há posições livres no buffer para serem gravadas e posições ocupadas a serem lidas.

```
Program Produtor_Consumidor_2;
  Const TAMBUF = N;
  Type TIPO_DADO = .....; /* tipo do dado */
  Var   Vazio : semáforo := TAMBUF;
        Cheio : semáforo := 0; /* inicialização dos semáforos */
        Mutex : semáforo := 1;
        Buffer: array[1..TAMBUF] of TIPO_DADO;

  Procedure Produtor;
  Var dado: TIPO_DADO;
  BEGIN
    Repeat
      ProduzDado(dado);
      Down(vazio);
      Down(mutex);
      GravaDado(dado, buffer); ← Região Crítica
      Up(mutex);
      Up(Cheio);
    Until false;
  END;

  Procedure Consumidor;
  Var dado: TIPO_DADO;
  BEGIN
    Repeat
      Down(Cheio);
      Down(mutex);
      LeDado(buffer, dado); ← Região Crítica
      Up(mutex);
      Up(vazio);
      ConsomeDado(dado);
    Until false;
  END;

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

Observe a simetria entre os processos Produtor e Consumidor. Podemos interpretar este código como o Produtor produzindo buffers cheios para Consumidor, ou como o Consumidor produzindo buffers vazios para o Produtor.

Semáforos do tipo vazio e cheio são chamados de **semáforos contadores**, sendo bastante úteis quando aplicados na alocação de recursos do mesmo tipo ( fazendo parte de um pool ) em um esquema que definimos como sendo de **Sincronização Condicional**. O semáforo é inicializado com o número total de recursos do pool, e sempre que um processo deseja um recurso, executa uma operação **Down**, subtraindo 1 do número de recursos disponíveis. Da mesma forma, sempre que o processo libera um recurso para o pool, executa um **Up**. Se o **semáforo contador** ficar igual a 0, isso significa que não existem mais recursos a serem utilizados, e o processo que solicitou o recurso permanece esperando, até que outro processo libere algum recurso para o pool.

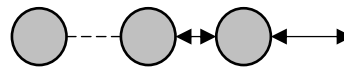
## 2.6.2 Monitores

Apesar de parecer simples, a comunicação entre processos com o uso de **Semáforos** exige do programador muito cuidado, pois qualquer engano pode levar a problemas de sincronização imprevisíveis. Como exemplo, suponha que na solução do **Produtor x Consumidor** usando **Semáforos**, invertamos a ordem dos dois comandos **Down** no código do Produtor de forma que o **Semáforo** mutex seja decrementado antes do **Semáforo** vazio.

```
Procedure Produtor;  
Var dado: TIPO_DADO;  
BEGIN  
  Repeat  
    ProduzDado(dado);  
    Down(mutex);           ← Inversão  
    Down(vazio);  
    GravaDado(dado, buffer); ← Região Crítica  
    Up(mutex);  
    Up(Cheio);  
  Until false;  
END;
```

Se o buffer estiver totalmente cheio, a execução do **Down** em vazio irá bloquear o Produtor e quando o Consumidor executar o **Down** em mutex ficará bloqueado (já que o Produtor executou o **Down** em mutex antes e ficou bloqueado). Dessa forma, os dois processos ficarão bloqueados para sempre, configurando uma situação de **Deadlock**, que estudaremos no próximo capítulo.

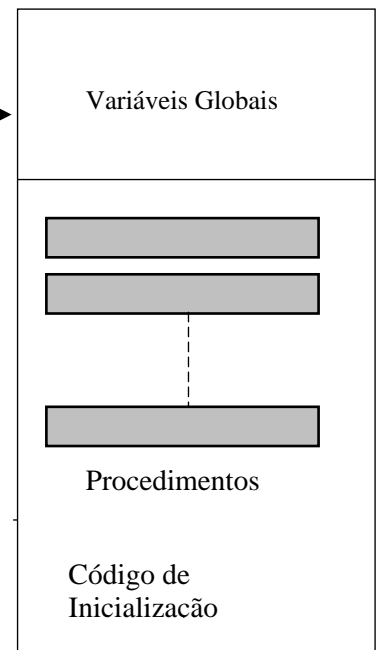
Para facilitar a escrita de programas concorrentes corretos, **Hoare** (1974), e **Brinch Hansem** (1975) propuseram um mecanismo de sincronização de alto nível chamado **Monitor**.



O **Monitor** é um conjunto de procedimentos, variáveis e estruturas de dados definidos dentro de um módulo (Pascal Concorrente) ou classe (em Java). A característica mais importante do Monitor é a implementação automática da **Exclusão Mútua** entre seus procedimentos, isto é, apenas um processo pode estar executando um dos procedimentos do **Monitor** em um determinado instante. Além disso, nenhum processo pode acessar diretamente as estruturas de dados internas do **Monitor**. Toda vez que algum processo chama um desses procedimentos, o **Monitor** verifica se já existe outro processo (ou thread) executando algum de seus procedimentos. Caso exista um procedimento sendo executado por outro processo, o processo (ou thread) requisitando a execução do procedimento ficará aguardando até que tenha permissão para executá-lo.

É importante observar que o **Monitor** é um mecanismo disponibilizado pela linguagem de programação, portanto toda a implementação da **Exclusão Mútua** é realizada pelo compilador, o que diminui a possibilidade de erros da parte do programador.

A única preocupação do programador é transformar as Regiões Críticas da aplicação em procedimentos do **Monitor** e dessa forma nunca teremos dois processos executando a Região Crítica ao mesmo tempo, o que será garantido pelo **Monitor**. Vejamos um exemplo, onde dois processos incrementam e decrementam de forma concorrente, o valor de uma variável compartilhada inteira **x**.





```

Program Exemplo_Monitor;
  Monitor Região_Crítica;
    Var x: integer;
    Procedure Incrementa;
    BEGIN
      x:=x+1;
    END;

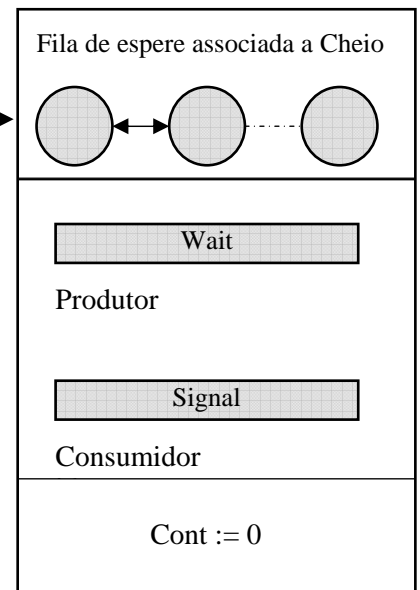
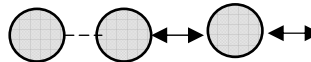
    Procedure Decrementa;
    BEGIN
      x:=x-1;
    END;
  BEGIN
    x:=0;
  END; /* Fim Monitor */
VAR
  RegiaoCritica : Monitor;
BEGIN
  RegiaoCritica.Create;
  PARBEGIN
    RegiãoCritica.Incrementa;
    RegiãoCritica.Decrementa;
  PAREND;
END.

```

A inicialização da variável compartilhada **x** será realizada uma única vez, no momento da criação do Monitor durante a execução.

Conforme vimos no problema **Produtor Consumidor**, torna-se necessária a criação de uma **Região Crítica** para o acesso ao Buffer de Dados (o que foi feito com o uso do semáforo binário mutex) e a criação de um mecanismo para que um processo fique bloqueado quando não houver mais recursos disponíveis (semáforos vazio e cheio para os processos Produtor e Consumidor), o que chamamos de **Sincronização Condicional**.

A solução, dentro do **Monitor**, para a obtenção da **Sincronização Condicional** é a criação das chamadas **Variáveis de Condição** juntamente com duas operações sobre elas, operações **wait** e **signal**. Quando um procedimento do **Monitor** descobre que não pode continuar (no caso do Produtor, quando o Buffer estiver cheio) ele executa uma operação de **wait** em uma variável de condição do **Monitor**. A execução da instrução **wait** acarretará o bloqueio do processo Produtor permitindo que o processo Consumidor que, por ventura, tenha tentado entrar na **Região Crítica** possa fazê-lo agora. O processo Consumidor pode, ao sair da **Região Crítica**, acordar o processo Produtor que estava dormindo através da chamada da primitiva **signal** associada a mesma **Variável de Condição** compartilhada entre os processos Consumidor e Produtor.



Para evitar que tenhamos dois processos ativos executando rotinas do **Monitor** ao mesmo tempo, é necessário uma política estabelecendo o que deve acontecer após a execução de um **signal**. **Hoare** propôs que o novo processo acordado suspenda automaticamente a execução do processo corrente. **Brinch Hansen** propôs uma solução mais amena impondo que o processo que executa o **signal** deve sair do **Monitor** imediatamente, isto é, o comando **signal** deve ser o último comando dentro de uma Procedure do **Monitor**. Cabe ao SO, através do algoritmo do Escalonador, a escolha do próximo processo a ser executado.

Vejamos a solução do Problema **Produtor x Consumidor** utilizando Monitor.

```
Program Produtor_Consumidor_3;  
  Const TAMBUF = ...           /* tamanho do Buffer */  
  Type TIPO_DADO = ...         /* tipo do dado */  
  Var   Buffer : ARRAY[1..TAMBUF] of TIPO_DADO;  
  
  Monitor ProdutorConsumidor;  
    VAR vazio, cheio: condition;    /* variáveis condição */  
    cont: integer;  
  
    Procedure Insere(dado: TIPO_DADO);  
    BEGIN  
      If cont = TAMBUF then wait(vazio);  
      GravaDado(dado, Buffer);  
      cont := cont + 1;  
      If cont = 1 then signal(cheio);  
    END;  
  
    Procedure Remove(VAR dado: TIPO_DADO);  
    BEGIN  
      If cont = 0 then wait(cheio);  
      LeDado(dado, Buffer);  
      cont := cont - 1;  
      If cont = TAMBUF-1 then signal(vazio);  
    END;  
  BEGIN  
    cont := 0;  
  END Monitor;           /* Fim do código do Monitor */  
  
  Procedure Produtor;  
  VAR dado: TIPO_DADO;  
  BEGIN  
    Repeat  
      ProduzDado(dado);  
      ProdutorConsumidor.Insere(dado);  
    Until False;  
  END;  
  
  Procedure Consumidor;  
  VAR dado: TIPO_DADO;  
  BEGIN  
    Repeat  
      ProdutorConsumidor.Remove(dado);  
      ConsomeDado(dado);  
    Until False;  
  END;  
  
  VAR  
    ProdutorConsumidor : Monitor;  
  BEGIN  
    ProdutorConsumidor.Create;  
  PARBEGIN  
    Produtor;  
    Consumidor;  
  PAREND;  
END.
```

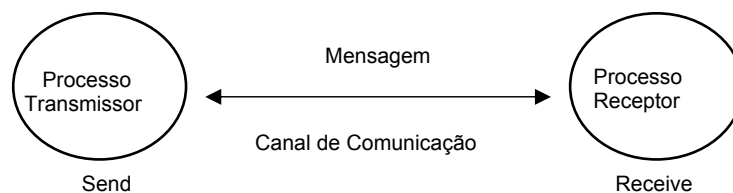
Para finalizar, lembramos que **Variáveis de Condição** não são contadores conforme semáforos, isto é, eles não tem a função de acumular o número de sinais de **wakeup / sleep** recebidos pelo processo. As instruções **wait** e **signal** se parecem com as primitivas **sleep** e **wakeup** mencionados anteriormente, porém com uma diferença crucial: **sleep** e **wakeup** podem falhar (conforme já mencionamos) quando um processo tentar se bloquear e o outro processo já estiver sinalizado para não fazê-lo anteriormente (perda do sinal de **wakeup**). Com Monitores, este problema não pode ocorrer, já que os comandos **sleep** e **wakeup** estariam dentro de procedimentos do **Monitor** (Explique!). Um outro problema do **Monitor**, e também de **Semáforos**, é que eles foram projetados para resolver Problemas de **Exclusão Mútua** em máquinas **monoprocessadas** ou **multiprocessadas** porém **com memória compartilhada**. Portanto a conclusão é que **Semáforos** são problemáticos para o programador e **Monitores** não estão disponíveis em qualquer linguagem, além disso ambas as soluções não se prestam para o caso de **arquiteturas com multiprocessadores de memória distribuída**. Dessa forma passaremos ao estudo de uma nova solução.

### 2.6.3 Troca de Mensagens

**Troca de mensagens** é um método de comunicação entre processos através de um canal de comunicação. A **Troca de Mensagens** usa duas primitivas **Send** e **Receive** que, da mesma forma que **Semáforos** e diferentemente de **monitores** são chamadas ao sistema (system calls) e não mecanismos da linguagem. Os formatos das rotinas é dado abaixo:

**Send** (receptor, mensagem);  
**Receive** (transmissor, mensagem);

Os procedimentos **Send** e **Receive**, são mecanismos de comunicação entre processos que apesar de não terem suas execuções mutuamente exclusivas, permitem tanto a implementação da **Exclusão Mútua** como a **Sincronização Condicional**. A sincronização ocorre porque um processo, ao receber uma mensagem (receptor), obtém dados enviados de outro processo (transmissor), e a sincronização é obtida porque uma mensagem somente pode ser lida após ter sido enviada, restringindo dessa forma, a ordem na qual dois eventos devem ocorrer.



A comunicação por **Troca de Mensagens** possui muitos problemas, como por exemplo, a perda de uma mensagem. Para garantir que uma mensagem não seja perdida, o processo receptor, ao recebê-la, deve enviar ao processo transmissor uma mensagem de confirmação (**acknowledgement - ACK**). Caso o transmissor não receba um ACK em um determinado tempo, ele retransmite a mensagem. Para que o receptor possa distinguir entre as mensagens normais e as retransmitidas associa-se a cada mensagem um número seqüencial identificador da mensagem, de forma que quando uma nova mensagem é recebida com o mesmo identificador de alguma mensagem previamente recebida o receptor sabe que pode descartá-la. A comunicação entre processos pode ser feita através do **endereçamento ( ou Comunicação ) direto** ou **endereçamento ( ou Comunicação ) indireto**.

No **endereçamento direto**, basta que o processo que deseja enviar ou receber uma mensagem enderece explicitamente o nome do processo receptor ou transmissor. O canal de comunicação formado neste esquema é normalmente bidirecional e é estabelecido automaticamente no momento em que os processos desejem se comunicar.

No **endereçamento indireto** as mensagens são recebidas e enviadas para um buffer de memória componente do contexto de software do processo, ou alternativamente para um buffer comum, chamado de **mailbox** (ou **portas**), de forma a ser compartilhado entre os diversos processos participantes da comunicação. Um **mailbox** pode ser visto como um objeto abstrato onde mensagens podem ser deixadas e recebidas nele. Cada **mailbox** contém um identificador único de forma que dois processos só podem se comunicar se eles partilharem o mesmo **mailbox**. Neste caso as primitivas **Send** e **Receive** devem receber como parâmetro o identificador do **mailbox** e a mensagem propriamente dita. Neste caso o canal de comunicação pode ser bidirecional ou unidirecional, e é formado entre dois processos somente se eles se referenciam e compartilham a mesma **mailbox**. Alternativamente, este canal pode ser estabelecido entre mais de dois processos.

Outro aspecto importante na **troca de mensagens** é quanto à **capacidade** do canal de comunicação, que é o número de mensagens que podem ser armazenadas temporariamente pelo canal de comunicação. Esta propriedade pode ser vista como uma fila de mensagens associada ao canal de comunicação. Existem três possíveis implementações para esta fila de mensagens, a saber:

- **Capacidade Zero** - Não existe fila de mensagens associada ao canal e portanto o processo Transmissor deve esperar, isto é, fica bloqueado até que o Receptor receba a mensagem. Desta forma a comunicação acontece somente quando os processos estão sincronizados. Este sincronismo é chamado de **rendezvous**.
- **Capacidade Limitada** - A fila de mensagens tem capacidade para armazenar um numero finito de mensagens. Enquanto a fila não estiver cheia toda vez que uma mensagem for enviada não existe o bloqueio do processo Transmissor permitindo que ele possa continuar a sua execução. Uma vez que a fila fica cheia o Transmissor sofrerá um pequeno atraso até que exista algum espaço na fila.
- **Capacidade Ilimitada** - Neste o tamanho do buffer da fila de mensagens é suficientemente grande de forma a ser considerado como infinito para efeitos práticos. Dessa forma o processo Transmissor nunca é bloqueado.

O esquema de fila de mensagens com **Capacidade Zero** é as vezes chamado de **troca de mensagens síncrona ou sem bufferização**; os outros esquemas são chamados de **troca de mensagens assíncrona ou com bufferização automática**. Observe que no esquema síncrono não é necessário o envio das mensagens ACK. **Porque?**

O mecanismo de pipe do **Unix**, é um exemplo de **mailbox**. A única diferença é que dentro do pipe as mensagens são consideradas de tamanho variável, cabendo aos processos participantes do pipe definir a forma como querem receber e enviar as mensagens, podendo ser inclusive diferentes.

Passamos agora a solução do Problema **Produtor x Consumidor** utilizando **Troca de Mensagens**. Apresentamos duas soluções, uma para o caso da **Comunicação Síncrona (Rendezvous)** e outra para o caso da **Comunicação Assíncrona**.

```
Program Produtor_Consumidor_4.1;      /* Comunicação Síncrona - Rendezvous */
    Type TIPO_DADO      = ...;        /* tipo do dado */
        TIPO_MENSAGEM = ...;        /* tipo da mensagem que contem o dado */
Procedure Produtor;
Var dado: TIPO_DADO;
    mensagem: TIPO_MENSAGEM;
    ultimaMensagem: boolean;
BEGIN
    Repeat
        ProduzDado(dado);
        FormataMensagem(dado, mensagem);
        Send(Consumidor, mensagem);
        If ultimaMensagem then
            break;
    Until false;
END;
```

```
Procedure Consumidor;
Var dado: TIPO_DADO;
    mensagem: TIPO_MENSAGEM;
    ultimaMensagem: boolean;
BEGIN
    Repeat
        Receive(Produtor, mensagem);
        DesmontaMensagem(mensagem, dado);
        ConsomeDado(buffer, dado);
        If ultimaMensagem then
            break;
    Until false;
END;

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

*Observe que estas soluções não apresentam Variáveis Globais. Desta forma não temos uma Região Crítica definida, e sim um sincronismo entre os processos através das primitivas Send e Receive. Este fato também nos mostra que esta solução pode ser usada também em máquinas com Memória Distribuída.*

```
Program Produtor_Consumidor_4.2;    /* Comunicação Assíncrona */
    Const TAMBUF = N;
    Type  TIPO_DADO      = ...;      /* tipo do dado */
          TIPO_MENSAGEM = ...;      /* tipo da mensagem que contem o dado */
          BUFFER_MENSAGENS= array[0..TAMBUF-1] of TIPO_MENSAGEM;

Procedure Produtor;
Var dado: TIPO_DADO;
    buffMensagem: BUFFER_MENSAGEM;
    ultimaMensagem: boolean;
    i : integer;
BEGIN
    i := 0;
    Repeat
        ProduzDado(dado);
        Receive(Consumidor, buffMensagem[i]); *espera chegada BuffMensagem vazio*
        FormataMensagem(dado, buffMensagem[i]);
        Send(Consumidor, buffMensagem[i]);    /* envia buffMensagem preenchido */
        i := (i + 1) mod TAMBUF;
        If ultimaMensagem then
            break;
    Until false;
END;
```

```
Procedure Consumidor;
Var dado: TIPO_DADO;
    mensagem: TIPO_MENSAGEM;
    ultimaMensagem: boolean;
    i : integer;
BEGIN
  For i := 0 to TAMBUF do    /* envia TAMBUF buffers vazios para Produtor */
  BEGIN
    Send(Produtor, buffMensagem[i]);
    i := i + 1;
  END;
  i := 0;
  Repeat
    Receive(Produtor, buffMensagem[i]); /*recebe buffer preenchido do Produtor*/
    DesmontaMensagem(buffMensagem[i], dado);
    Send(Produtor, buffMensagem[i]); /* devolve buffer vazio para Produtor */
    ConsomeDado(buffer, dado);
    If ultimaMensagem then
      break;
    Until false;
  END;

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

**OBS:**

Observe que a solução do programa Produtor\_Consumidor 4.1 também pode ser utilizada neste 2º cas, resultando, entretanto, em uma versão com pior performance.

## 2.7 Equivalência entre Primitivas de Sincronização

Nas seções anteriores apresentamos três primitivas de sincronização distintas: **Semáforos, Monitores e Troca de Mensagens**. Na realidade não existe nenhuma vantagem de uma em relação a outra, sendo todas semanticamente equivalentes, quando estamos considerando casos com uma única CPU. De fato, a partir de uma primitiva podemos simular as outras, conforme veremos a seguir. A escolha de uma particular primitiva deve ser feita pelo programador segundo o seu gosto e disponibilidade no ambiente computacional.

Nas próximas seções apresentaremos a equivalência entre as primitivas de sincronização apresentadas. Esta equivalência se torna útil para aumentar a compreensão das mesmas.

### 2.7.1 Usando Semáforos para Simulação de Monitores e Troca de Mensagens

No caso do SO prover a primitiva de sincronização **Semáforo**, qualquer compilador pode implementar facilmente o mecanismo de **Monitor** dentro da linguagem, conforme discutimos a seguir.

Associado a cada **Monitor** coloca-se um **Semáforo** binário para **Exclusão Mútua**, **mutex**, inicializado com 1, para controlar a entrada no **Monitor**, e um **Semáforo c**, inicializado com 0, para cada **variável de condição** do **Monitor** para implementar a **Sincronização Condicional**. Toda vez que um processo entra no **Monitor** o compilador gera código para a chamada de uma função **EntraMonitor** que realiza uma operação **Down** em **mutex**. Se o **Monitor** já estiver sendo usado por outro processo o processo que executar a função **EntraMonitor** ficará bloqueado até que o **Monitor** seja liberado. Ao sair do **Monitor**, se não existir nenhum processo esperando por um **signal** associado a alguma **variável de condição** o processo pode executar a função **SaidaNormalMonitor** que simplesmente executará um **Up** em **mutex** liberando o **Monitor**.

As coisas se tornam complicadas quando lidamos com **variáveis de condição**. A execução da função **Wait** em uma **variável de condição** deve ser implementada como uma operação de **Up** em **mutex** para a liberação do **Monitor**, seguida de uma operação **Down** no **Semáforo c** para bloquear o processo que executou a função **Wait**. A execução de um **Signal**, que por convenção, adotamos ser a última instrução dentro de um método do **Monitor** para evitar que dois processos executem métodos do **Monitor** ao mesmo tempo, deve ser implementada por uma função **SaidaSignalMonitor**. Esta função executará um **Up** no **Semáforo c** desbloqueando o processo preso pelo **Down** no **Semáforo c** executado na função **Wait**. Observe que dessa forma o controle da execução é passado imediatamente do processo que executa o **Signal** para o processo que executou o **Wait** sem que nenhum outro processo possa executar qualquer método do **Monitor** porque o **Semáforo mutex** continua valendo 0, e só será incrementado quando o processo que executou o **Wait** terminar normalmente através da função **SaidaNormalMonitor**.

A seguir apresentamos o código das rotinas mencionadas para implementação do **Monitor** com o uso de **Semáforo**. A título de exercício você deve tentar verificar a validade desta implementação para o Problema **Produtor x Consumidor**.

```
Var mutex := 1: Semáforo; /* 1 Semáforo para Exclusão Mútua entre os métodos do Monitor */
    c      := 0: Semáforo; /* 1 Semáforo para cada Variável de Condição do Monitor */

Procedure EntraMonitor;
BEGIN
    Down(mutex);

END;

Procedure SaidaNormalMonitor;
BEGIN
    Up(mutex);

END;

Procedure SaidaSignalMonitor( Var c: Semáforo );
BEGIN
    Up(c);

END;

Procedure Wait( Var c: Semáforo );
BEGIN
    Up(mutex);
    Down(c);

END;
```

Agora apresentamos a implementação da **Troca de Mensagens** com o uso de **Semáforo**. Associado a cada processo colocamos um **Semáforo c**, inicializado com 0, para implementar a **Sincronização Condicional** entre as funções **Send** e **Receive**. Uma área de memória deve ser compartilhada entre os processos de forma a implementar o conceito de **mailbox**. Cada **mailbox** pode ser representada pela seguinte estrutura de dados:

```

Const TAMBUF = N;
NP = ...; /* Numero de processos associados a uma mailbox */
Type TIPO_DADO = ...; /* tipo do dado */
TIPO_MENSAGEM = ...; /* tipo da mensagem que contem o dado */
BUFFER_MENSAGENS= array[0..TAMBUF-1] of TIPO_MENSAGEM;
Var C: array[0..NP] of Semáforo; /*Vetor Semáforos para Sincronização Condicional */
/*entre os Processos enviando Send e Receive */
/*Semáforo para Exclusão Mútua no Acesso a MAILBOX */
Mutex : Semáforo;
MAILBOX = record
    fullSlots: integer; /* Numero de slots preenchidos/vazios */
    emptySlots: integer; /* em BufferMsgs - vetor de mensagens */
    bufferMsgs: BUFFER_MENSAGENS;

    mensagens: Lista_Mensagens; /* Índices dentro de BufferMsgs */
/* de acordo com a ordem de chegada */
/* das mensagens na mailbox */

    filaSend: Lista_Processos; /* Fila de Processos bloqueados */
    filaReceive: Lista_Processos; /* na execução de Send/Receive */
/* para/de mailbox */
end;

```

Quando um **Send** ou **Receive** é feito para uma **mailbox** contendo pelo menos um slot vazio ou cheio respectivamente, a operação insere ou remove a mensagem na **mailbox** e atualiza os contadores **fullSlots** e **emptySlots** e remove ou acrescenta a mensagem na lista de mensagens da **mailbox**. O acesso a estrutura **mailbox** deve ser protegida pelo **Semáforo Mutex** para garantirmos a exclusão mútua na atualização da estrutura e evitarmos possíveis inconsistências também chamadas de **Condições de Corrida - Race Conditions**.

Quando um processo *i* executa um **Receive** para uma **mailbox** contendo apenas slots vazios, o processo que deseja receber a mensagem é colocado na **filaReceive** da **mailbox** indicando que este processo deseja receber uma mensagem da **mailbox**. Posteriormente o processo executa um **Up** em **mutex** e um **Down** no seu **Semáforo C[i]** pondo-se para dormir. Mais tarde quando um processo acordado através da execução de um **Up** no **Semáforo** associado **C** [ **proc\_filaReceive** ]. Após isto o processo que executou o **Send** sai da **Região Crítica** e o processo acordado pode continuar a sua execução.

Quando um processo executa um **Send** para uma **mailbox** e existe algum slot vazio na **mailbox**, a mensagem do transmissor é colocada no **bufferMsgs** em alguma posição livre, e esta posição é colocada na Lista de mensagens da **mailbox** para indicar a ordem de chegada da mensagem. Se houver algum processo pendente na **filaReceive** da **mailbox** este processo é então retirado da fila e é acordado através da execução de um **Up** no **Semáforo** associado **C** [ **proc\_filaReceive** ]. Após isto o processo que executou o **Send** sai da **Região Crítica** e o processo acordado pode continuar a sua execução.

Quando um processo *i* executa um **Send** para uma **mailbox** contendo apenas slots cheios, o processo que deseja enviar a mensagem é colocado na **filaSend** da **mailbox** indicando que este processo deseja enviar uma mensagem para a **mailbox**. Posteriormente o processo executa um **Up** em **mutex** e um **Down** no seu **Semáforo C[i]** pondo-se para dormir. Mais tarde quando um processo receptor retirar alguma mensagem da **mailbox** e notar que existe um processo pendente na **filaSend** da **mailbox** ele acordará o processo pendente removendo-o da fila e executando um **Down** no **Semáforo** associado **C** [ **proc\_filaSend** ]. Após isto o processo que executou o **Receive** sairá da **Região Crítica** e o processo acordado pode continuar a sua execução.

## 2.7.2 Usando Monitores para Simulação de Semáforos e Troca de Mensagens

A implementação de **Semáforos** usando **Monitores** é parecido com o caso anterior, porém um pouco mais simples porque **Monitores** são mecanismos de sincronização de mais alto nível do que **Semáforos**. Para implementar um **Semáforo** precisamos de um contador e uma lista encadeada de processos vinculada a cada **Semáforo**, conforme a própria definição de **Semáforos**. Para a **Sincronização Condicional** podemos utilizar as **variáveis de Condição** do **Monitor**.



Quando a função **Down**, implementada com o uso do **Monitor** é executada, ela verifica se o valor do contador dentro do **Monitor** é maior do que zero. Se for, o contador é decrementado e a função termina liberando o **Monitor**. Se o contador for zero, o processo que chamou a função **Down** é colocado na **lista de processos associada ao semáforo** e executa um **Wait** na **variável de condição** associada ao processo.

Quando a função **Up**, implementada com o uso do **Monitor** é executada, o valor do contador dentro do **Monitor** é incrementado. Se existir algum processo pendente na **lista de processos associada ao semáforo** ele é removido da fila um **Signal** na **variável de condição** associada ao processo retirado da fila é realizado. Em uma implementação mais sofisticada a **lista de processos associada ao semáforo** poderia conter também a prioridade de cada processo para que a retirada de processos seguisse esta prioridade.

A implementação de **Troca de Mensagens** usando **Monitores** pode ser feita da mesma forma que a implementação feita para **Troca de Mensagens** usando **Semáforos**, exceto que ao invés de um **Semáforo** por processo usaremos uma **variável de condição** para cada processo. A estrutura **MAILBOX** pode ser a mesma bem como as demais variáveis.

**Exercício:** Esboce usando a linguagem Pascal a simulação de Semáforos e de Troca de Mensagens usando Monitores conforme comentado no texto.

### 2.7.3 Usando Troca de Mensagens para Simulação de Semáforos e Monitores

A implementação de **Semáforos e Monitores** usando **Troca de Mensagens** pode ser feita usando-se um pequeno truque. O truque é a criação de um novo processo, chamado de **processo sincronizador**. A finalidade da existência deste processo é prover os mecanismos base para a implementação da **Exclusão Mútua** e da **Sincronização Condicional**.

Começamos pela implementação de **Semáforos**. Neste caso o **processo sincronizador** deve manter um **contador** e uma **lista-de-processos-em-espera** para cada **Semáforo** criado. Para fazer um **Up** ou **Down** o processo chamador executa a rotina correspondente implementada usando **Troca de Mensagens**. Estas rotinas enviam uma mensagem ( **Send** ) para o **processo sincronizador** especificando o tipo de operação desejada e qual o **Semáforo** a ser utilizado. Após o envio da mensagem a rotina executa um **Receive** para obter a resposta do **processo sincronizador** sobre o retorno da primitiva.

Quando o **processo sincronizador** recebe uma mensagem, ele verifica o contador associado ao **Semáforo** para concluir se a operação pode ou não ser terminada. Conforme já sabemos, operações de **Up** podem sempre ser completadas, entretanto operações **Down** devem bloquear o processo chamador se o contador estiver zerado. Para completar a operação o **processo sincronizador** envia de volta ao processo chamador uma mensagem vazia, desbloqueando-o. Entretanto, se a operação solicitada pelo processo chamador for um **Down** e o contador associado estiver zerado o **processo sincronizador** coloca o processo chamador na **lista-de-processos-em-espera** e não retorna nenhuma mensagem para o processo chamador mantendo-o dessa forma bloqueado. Mais tarde quando um **Up** for executado o **processo sincronizador** retira algum processo da **lista-de-processos-em-espera**, segundo alguma política do tipo FCFS, Prioridade, etc., e envia uma mensagem vazia de volta ao processo retirado da lista.

**Monitores** podem ser implementados usando o mesmo truque na implementação de **Semáforos**. Observe que no item 2.7.1 nós vimos como usar **Semáforos** para implementar **Monitores** e no parágrafo anterior nós vimos como usar **Troca de Mensagens** para implementar **Semáforos**, usando a transitividade, isto é, combinando as duas soluções podemos usar **Troca de Mensagens** para implementar **Monitores**. Observamos que esta não é a única solução possível ! Mas é sem sombra de dúvida uma boa solução.

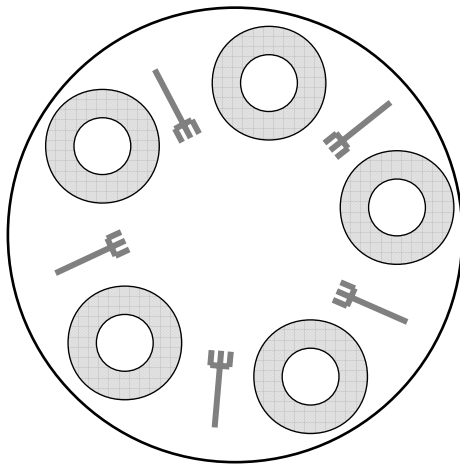
**Exercício:** Esboce usando a linguagem Pascal a simulação de Semáforos e de Monitores usando Troca de Mensagens conforme comentado no texto.

## 2.8 Problemas Clássicos de Sincronização

A literatura de Sistemas Operacionais esta repleta de exemplos interessantes sobre sincronização entre processos. A seguir apresentaremos três problemas clássicos muito importantes que são: **"The Dining Philosophers"**, **"Readers and Writers" (Leitores e Escritores)** e **"The Sleeping Barber"**. Para cada um destes problemas apresentaremos uma breve discussão e apresentaremos a solução utilizando **Semáforos**. Um bom exercício é o de implementar estas soluções usando as outras primitivas de sincronização vistas na seção 2.7

### 2.8.1 The Dining Philosophers Problem

Em 1965, **Dijkstra** propôs e resolveu o problema chamado **"The Dining Philosophers"**. Este problema é útil para modelar um conjunto de processos que estejam competindo pelo uso exclusivo a um número limitado de recursos, tais como o acesso a dispositivos de E/S. O problema pode ser colocado da seguinte forma:



#### **The Dining Philosophers Problem**

Cinco filósofos estão sentados ao redor de uma mesa redonda. Cada filósofo tem um prato de macarrão à sua frente. Para comer o macarrão cada filósofo tem que usar dois garfos, e existe um garfo entre cada prato da mesa. A vida de cada filósofo consiste na alternância de períodos em que está comendo ( EATING ) e períodos em que está pensando ( THINKING ). Quando um filósofo fica com fome ( HUNGRY ), ele tenta tomar posse de seus garfos esquerdo e direito, um de cada vez. Em caso de sucesso na aquisição dos dois garfos, ele passa a comer durante um período finito de tempo após o qual libera os dois garfos e volta a pensar.

**A questão que se coloca é se é possível escrever um programa para cada filósofo de tal forma que todos possam comer de forma alternada sem se bloquearem uns aos outros !**

Uma solução óbvia para tal problema pode ser como a mostrada abaixo, onde a rotina pegaGarfo fica bloqueada até que o garfo requisitado esteja disponível.

```
Program DinningPhilosopher_0;  
Const N = 5;  
Type TESTADOS = (THINKING, EATING, HUNGRY);  
  
Funtion Left( i: integer ) : integer;  
BEGIN  
    Left := (i-1)%N  
END;  
  
Funtion Right ( i: integer ) : integer;  
BEGIN  
    Right := (i+1)%N  
END;
```

```
Procedure Philosopher( i: integer );
Var
  estado: TESTADOS;
BEGIN
  Repeat
    estado := THINKING;      /* Filósofo pensa */
    Think;
    estado := HUNGRY;        /* Filósofo tenta comer */
    pegaGarfo( i );          /* Obtém garfo esquerdo */
    pegaGarfo( Right(i) );   /* Obtém garfo direito */
    estado := EATING;        /* Filósofo come */
    Eat;
    liberaGarfo( i );         /* Libera garfo esquerdo */
    liberaGarfo( Right(i) ); /* Libera garfo direito */
  Until false;
END;
BEGIN
  PARBEGIN
    Philosopher(0); Philosopher(1); Philosopher(2);
    Philosopher(3); Philosopher(4);
  PAREND;
END.
```

Apesar de simples, esta solução não funciona pois se todos os 5 filósofos tentarem pegar os seus garfos esquerdos ao mesmo tempo, nenhum filósofo poderá comer pois nenhum deles poderá obter os dois garfos necessários. Neste caso nós entraremos numa situação de **Deadlock**.

Uma outra alternativa seria modificar o problema de forma que após pegar o garfo da esquerda, o filósofo verifica se o garfo da direita está disponível. Em caso negativo, o garfo da esquerda é liberado e após alguns instantes o processo deve ser repetido. Infelizmente esta estratégia também falha porque todos os filósofos podem estar sincronizados de tal forma que todos peguem os seus garfos esquerdos e verifiquem a disponibilidade do garfo da direita simultaneamente. Após verificarem a indisponibilidade do garfo da direita todos liberam os seus garfos esquerdos e passam esperar alguns instantes para repetir o processo novamente. Apesar de ser bastante improvável, esta situação pode se manter para sempre o que caracteriza uma situação de **Starvation**, onde os processos rodam indefinidamente mas não conseguem progredir. Este sincronismo poderia ser quebrado se cada filósofo esperasse um tempo aleatório antes de repetir o processo. Entretanto estamos interessados em soluções que funcionem sempre !

Uma solução livre dos problemas de **Deadlock** e **Starvation** seria proteger o trecho de código entre o momento em que os garfos são obtidos e liberados, com um **Semáforo** mutex, criando uma **Região Crítica**. Esta solução, entretanto, apresenta um problema de performance, pois apenas um filósofo irá comer a cada instante mas com cinco garfos disponíveis pelo menos dois filósofos podem comer a cada instante !

```
Down( mutex );
estado := HUNGRY;                      /* Filósofo tenta comer */
pegaGarfo( i ); pegaGarfo( Right(i) ); /* Obtém garfos esquerdo e direito */
estado := EATING;                      /* Filósofo come */
Eat;
liberaGarfo( i ); liberaGarfo( Right(i) ); /* Libera garfos esquerdo e direito */
Up( mutex );
```

A solução apresentada abaixo é correta e permite o máximo paralelismo para um número arbitrário de filósofos. Usamos um vetor de **Semáforos** para todos os filósofos de forma que os filósofos que estejam com fome ( estado = HUNGRY ) possam ficar bloqueados quando não conseguem obter os garfos necessários.

```
Program DinningPhilosophers;
Const N = 5;
Type TESTADOS = (THINKING, EATING, HUNGRY);
Var  Estados : array [ 0..N-1 ] of TESTADOS;
     Mutex   : Semáforo;
     S       : array [ 0..N-1 ] of Semáforo;

Function Left( i: integer ) : integer;
BEGIN
  Left := (i-1) mod N;
END;
Function Right ( i: integer ) : integer;
BEGIN
  Right := (i+1) mod N;
END;
```

```
Procedure Test( i: integer );
BEGIN
  if Estados[i] = HUNGRY AND
     Estados[Left(i)] <> EATING AND Estados[Right(i)] <> EATING then
  BEGIN
    Estados[i] = EATING;          /* Estado Filósofo indicando que vai comer */
    Up(S[i]);
  END;
END;

Procedure pegaGarfos( i: integer );
BEGIN
  Down(Mutex);
  Estados[i] = HUNGRY;           /* Estado Filósofo indicando que está HUNGRY */
  Test(i);
  Up(Mutex);
  Down(S[i]);
END;

Procedure liberaGarfos( i: integer );
BEGIN
  Down(Mutex);
  Estados[i] = THINKING;        /* Estado Filósofo muda para THINKING */
  Test( Left(i) );
  Test( Right(i) );
  Up(Mutex);
END;

Procedure Philosopher( i: integer );
BEGIN
  Repeat
    Think;                      /* Filósofo pensa */
    pegaGarfos( i );            /* Obtém garfos */
    Eat;                        /* Filósofo come */
    liberaGarfos( i );          /* Libera garfos */
  Until false;
END;
BEGIN
  Mutex.valor := 1;
  PARBEGIN
    Philosopher(0); Philosopher(1); Philosopher(2);
    Philosopher(3); Philosopher(4);
  PAREND;
END.
```

## 2.8.2 The Readers and Writers Problem

Um outro problema famoso é o **Leitores e Escritores ( Readers and Writers )** que modela o acesso a registros de um Banco de Dados. Um exemplo típico é o caso de um *Sistema de Controle de Reservas de Passagens Aérea*, onde diversos processos estão competindo para ler e atualizar informações sobre a disponibilidade de vagas nos vôos de uma determinada companhia aérea. Admite-se o acesso simultâneo de diversas leituras desde que nenhuma atualização esteja sendo feita. Além disso, toda atualização deve ser feita de forma exclusiva, para evitar que a leitura de dados obtenha dados inconsistentes.

Uma solução que coloque o acesso ao Banco de Dados como uma **Região Crítica** apresentará sérios problemas de performance, além de ser uma restrição muito forte para o problema. Observamos que este problema não pode ser visto como um caso especial do problema **Produtor x Consumidor**, porque o **Produtor** não pode ser visto apenas como um **Escritor** pois ele deve acessar as variáveis de controle do buffer de dados para saber qual a posição para gravar o próximo item produzido, além de se precaver quanto a possibilidade do buffer estar cheio. Analogamente o **Consumidor** também não pode ser visto como um **Leitor** porque ele também deve acessar as variáveis de controle do buffer de dados para indicar a remoção do buffer de um item consumido.

Para este problema apresentaremos duas soluções, onde em cada uma ou os **Leitores** ou os **Escritores** terão prioridade. Na solução **Leitores têm Prioridade ( Readers Have Priority )** os **Escritores** ficam parados até que todos os **Leitores** terminem, enquanto na solução **Escritores têm Prioridade ( Writers Have Priority )**, bem mais difícil, o oposto deve acontecer.

### 2.8.2.1 Leitores têm Prioridade - Readers Have Priority

```
Program ReadersWrites_ReadersHavePriority;
Var   readCount   : integer;
      MutexR, WSema: Semáforo;

Procedure Reader( i: integer );
BEGIN
  Repeat
    Down(MutexR);                /* Região Crítica para atualizar readCount */
    readCount := readCount + 1;
    if readCount = 1 then Down(WSema); /* Indica entrada do primeiro Leitor */
    Up(MutexR);

    ReadDataBase;                /* Lê dados do DataBase */

    Down(MutexR);                /* Região Crítica para atualizar readCount */
    readCount := readCount - 1;
    if readCount = 0 then Up(WSema);  /* Indica saída do último Leitor */
    Up(MutexR);

    ProcessaDados;                /* Processa dados lidos */
  Until false;
END;

Procedure Writer( i: integer );
BEGIN
  Repeat
    AtualizaDados;                /* Atualiza dados a serem gravados */

    Down(WSema);                  /* Região Crítica para gravação do DataBase */
    WriteDataBase;                /* Grava dados no Database */
    Up(WSema);
  Until false;
END;

BEGIN
  MutexR.valor := 1;
  WSema.valor  := 1;
  readCount := 0;
  PARBEGIN
    Reader; Reader; Reader;
    Writer; Writer;
  PAREND;
END.
```

### 2.8.2.2 Escritores têm Prioridade - Writers Have Priority

Na solução **Leitores têm Prioridade** os **Escritores** ficam parados até que todos os **Leitores** terminem, o que pode levá-los a uma situação de **Starvation**.

A solução **Escritores têm Prioridade** garante que nenhum novo **Leitor** tenha acesso ao DataBase uma vez que um **Escritor** manifeste a sua vontade de acessá-lo. Para permitir isto, criaremos para os **Escritores**, um novo Semáforo, **RSema**, de forma a inibir os **Leitores** enquanto existir pelo menos um **Escritor** desejando acessar o DataBase, além disso também criaremos um contador de escritores e o respectivo Semáforo, **MutexW**, para permitir a sua atualização. Para os **Leitores**, necessitamos de um Semáforo adicional, para evitar que tenhamos uma fila associada ao Semáforo **RSema** com mais de um **Leitor**, pois dessa forma estes bloqueariam os **Escritores**.

```
Program ReadersWrites_WritersHavePriority;
Var  readCount, writeCount : integer;
     MutexR, MutexW       : Semáforo;
     WSema, RSema, QueueR : Semáforo;

Procedure Reader( i: integer );
BEGIN
  Repeat
    Down(QueueR);
    Down(RSema);          /* Bloqueia Leitor Enquanto houver Escritor */
    Down(MutexR);         /* Região Crítica para atualizar readCount */
    readCount := readCount + 1;
    if readCount = 1 then Down(WSema); /* Indica entrada do primeiro Leitor */
    Up(MutexR);
    Up(RSema);
    Up(QueueR);

    ReadDataBase;         /* Lê dados do DataBase */

    Down(MutexR);         /* Região Crítica para atualizar readCount */
    readCount := readCount - 1;
    if readCount = 0 then Up(WSema);  /* Indica saída do último Leitor */
    Up(MutexR);

    ProcessaDados;        /* Processa dados lidos */
  Until false;
END;

Procedure Writer( i: integer );
BEGIN
  Repeat
    Down(MutexW);         /* Região Crítica para atualizar writeCount */
    writeCount := writeCount + 1;
    if writeCount = 1 then Down(RSema); /* Indica entrada do primeiro Escritor */
    Up(MutexW);

    AtualizaDados;        /* Atualiza dados a serem gravados */
    Down(WSema);          /* Região Crítica para gravação do DataBase */
    WriteDataBase;        /* Grava dados no Database */
    Up(WSema);

    Down(MutexR);         /* Região Crítica para atualizar writeCount */
    writeCount := writeCount - 1;
    if writeCount = 0 then Up(RSema);  /* Indica saída do último Escritor */
    Up(MutexR);
  Until false;
END;

BEGIN
  MutexR.valor := 1; MutexW.valor := 1;
  WSema.valor  := 1; RSema.valor  := 1;
  QueueR.valor := 1;
  readCount    := 0; writeCount   := 0;

  PARBEGIN
    Reader; Reader; Reader;
    Writer; Writer;
  PAREND;
END.
```

### 2.8.3 The Sleeping Barber Problem

O problema do **Barbeiro Dorminhoco ( Sleeping Barber )** consiste de uma barbearia onde se encontra um **Barbeiro**, uma cadeira de barbeiro e NCHAIRS cadeiras de espera para **Clientes**. Se não existem **Clientes** o **Barbeiro** se senta e dorme na sua cadeira. Quando um **Cliente** chega a barbearia, e o **Barbeiro** esta dormindo, ele acorda o **Barbeiro** para cortar o seu cabelo. Se um **Cliente** chega a barbearia, e o **Barbeiro** esta cortando o cabelo de algum **Cliente**, ele verifica se existe uma cadeira de espera disponível e se senta nessa cadeira, caso contrário, deixa a barbearia.

A solução emprega três Semáforos: Clientes, para contar o número de clientes em espera, excluindo o Cliente que esta cortando o cabelo; Barbers, para contar o número barbeiros dormindo. Quando o barbeiro chega a barbearia pela manhã ele executa a Procedure **Barber** causando o seu bloqueio até a chegada de algum **Cliente**. Quando o primeiro **Cliente** chega ele bloqueia o acesso a **Região Crítica** para atualização da variável waitingClientes que indica quantos clientes já estão esperando pelo barbeiro. Se não existem mais lugares vagos o Cliente vai embora sem Ter o seu cabelo cortado. Caso exista uma cadeira de espera vaga, o Cliente indica a sua chegada ao barbeiro através do Up(Clientes) e incremento da variável waitingClientes.

```
Program SleepingBarber;

Const NCHAIRS = 3;
Var   waitingClientes : integer;
      Clientes, Barbers, Mutex : Semáforo;

Procedure Barber;
BEGIN
  Repeat
    Down(Clientes);          /* Espera chegada de algum Cliente */

    Down(Mutex);             /* Região Crítica para atualizar waitingClientes */
    waitingClientes := waitingClientes - 1;
    Up(Barbers);             /* Acorda Barbeiro Dorminhoco */
    Up(Mutex);

    CortarCabelo;            /* Corta cabelo de cliente */
  Until false;
END;

Procedure Clientes;
BEGIN
  Down(Mutex);              /* Região Crítica para atualizar waitingClientes */
  if waitingClientes < NCHAIRS then
  BEGIN
    waitingClientes := waitingClientes + 1;
    Up(Clientes);           /* Avisa chegada de um novo cliente */
    Up(Mutex);

    Down(Barbers);          /* Vai dormir se não existem Barbeiros livres */
    EsperaCorteCabelo;      /* Fica na fila para cortar cabelo */
  END
  else
    Up(Mutex);
  END;

BEGIN
  Barbers.valor := 0; Clientes.valor := 0;
  Mutex.valor  := 1;
  waitingClientes := 0;

  PARBEGIN
    Clientes; Clientes; Clientes; Clientes; Clientes; Clientes; Clientes;
    Barber;
  PAREND;
END.
```

### **3   Comunicação entre processos**

Em um ambiente Multiprogramado, vários processos podem competir por um número finito de recursos, e se estes



## 4 Deadlock

Em um ambiente Multiprogramado, vários processos podem competir por um número finito de recursos, e se estes recursos não estiverem disponíveis neste momento, o processo entra em estado de espera. Pode acontecer que estes processos em estado de espera não mudem de estado novamente, porque os recursos que ele requisitou estão alocados por outro processo em estado de espera. Por exemplo, esta situação pode ocorrer em um sistema com quatro drivers de fitas e dois processos. Se cada processo aloca dois drivers mais necessita de três, então cada um deles irá esperar que o outro libere seus drivers de fita. Esta situação é chamada **Deadlock**.

Para prevenir um **Deadlock**, ou recuperar um, se ele ocorrer, o sistema pode tomar algumas ações relativamente extremas, como a preempção de recursos de um ou mais processos envolvidos no **Deadlock**. Neste capítulo nós descrevemos alguns dos vários métodos que os sistemas operacionais podem usar para lidar com o problema de **Deadlock**.

### 4.1 O Problema de Deadlock

O problema de **Deadlock** não acontece somente nos ambientes de sistemas operacionais. Generalizando nossa interpretação de recursos e processos nós podemos ver que problemas de **Deadlock** podem ser parte de nosso ambiente diário. Por exemplo, considere o problema da travessia de um rio que tem um número de pedras que podem ser utilizadas como ponte. No máximo um pé pode estar em cada pedra por vez. Para atravessar o rio, uma pessoa deverá usar cada uma das pedras. Nós podemos ver cada pessoa atravessando o rio como um processo e cada pedra como um recurso. Um **Deadlock** ocorre quando duas pessoas iniciam a travessia de lados opostos e se encontram no meio.

Pisar em uma pedra pode ser visto como alocar um recurso, enquanto que remover o pé corresponde a liberar este recurso. A **Deadlock** ocorre quando duas pessoas tentam pisar na mesma pedra. O **Deadlock** pode ser resolvido se uma das pessoas voltar para o lado do rio de onde ela iniciou a travessia. Em termos de sistema operacional, esta volta é chamada **rollback**. Observe que se várias pessoas estão cruzando o rio partindo do mesmo lado, mais de uma pessoa poderá ter que voltar para resolver o problema de **Deadlock**. Se uma pessoa inicia a travessia do rio sem descobrir quando outra pessoa está tentando atravessar o rio partindo do outro lado, então um **Deadlock** sempre será possível.

O único meio de se ter certeza de que um **Deadlock** não irá ocorrer é fazer com que cada pessoa que deseje atravessar o rio siga um protocolo determinado. Este protocolo deverá fazer com que cada pessoa que queira atravessar o rio descubra quando alguém está cruzando o rio a partir da outra margem. Se a resposta é não, ela poderá prosseguir. Caso contrário, ela deverá esperar até que a outra pessoa tenha terminado sua travessia. Vários comentários deveriam ser feitos a cerca deste protocolo:

1. nós deveremos ter um mecanismo para determinar quando outro alguém está cruzando o rio. Se for sempre possível examinar o estado de todas as pedras, esta condição é suficiente. Se não (por exemplo, o rio pode ser muito largo ou uma neblina pode obscurecer a visão), outro mecanismo será necessário;
2. suponha duas pessoas querendo atravessar o rio de lados opostos ao mesmo tempo. Nosso protocolo não especifica o que deverá ser feito neste caso. Se ambos iniciam a travessia do rio, um **Deadlock** irá ocorrer. Se cada uma esperar pela outra iniciar, outra forma de **Deadlock** irá ocorrer. Uma solução para esta dificuldade será definir uma prioridade mais alta para uma das margens do rio, por exemplo, lado Este. Isto é, a pessoa do lado Este sempre irá atravessar primeiro, enquanto a pessoa do lado Oeste terá que esperar.
3. se este protocolo for observado, um ou mais processos poderão ter esperar indefinidamente para atravessar o rio. Esta situação é descrita como **Starvation**. Isto pode ocorrer no exemplo da travessia do rio se uma continua fila de pessoas estiver cruzando o rio a partir da margem de maior prioridade. Para evitar a situação de **Starvation**, este protocolo deverá ser estendido. Por exemplo, nós podemos definir um algoritmo que alterne a direção da travessia de tempos em tempos. O desenho deste algoritmo será deixado para o leitor como um exercício.

Um sistema consiste de um número finito de recursos para serem distribuídos entre um número de processos concorrentes. Os recursos são divididos em vários tipos, cada um consistindo de alguns números de instancias idênticas. Ciclos de CPU, espaço de memória, arquivos e dispositivos de I/O (como impressoras, drivers de fitas, e leitoras de cartões) são exemplos de tipos de recursos. Se um sistema tem duas CPU, então o tipo de recurso CPU tem duas instancias. Similarmente, o tipo de recurso impressora, pode ter cinco instancias.

Se um processo requisita uma instância de um tipo de recurso, a alocação de qualquer instância de um tipo irá satisfazer a requisição. Se este não é o caso, então as instâncias não são idênticas, e as classes de tipos de recursos poderão não estar definidas apropriadamente. Por exemplo, um sistema pode ter duas impressoras. Estas duas impressoras e estas duas impressoras podem ser definidas para estar na mesma classe de recursos se ninguém se preocupar que impressora irá imprimir suas saídas. Entretanto, se uma impressora está no nono andar e a outra no térreo, então pessoas no nono andar podem não ver ambas as impressoras como equivalentes, e classes de recursos separadas podem ser necessárias para definir cada impressora.

Um processo deverá requisitar um recurso antes de usá-lo e deverá liberá-lo após o uso. Um processo pode requisitar tantos recursos quantos necessários para realizar sua tarefa. Obviamente, o número de recursos requisitados não poderá exceder o número total de recursos disponíveis no sistema. Em outras palavras, um processo não pode requisitar três impressoras se o sistema tem somente duas. Sob condições normais de operação, um processo pode utilizar um recurso somente na seguinte seqüência:

1. **Requisição.** Se a requisição não pode ser garantida imediatamente (por exemplo, o recurso está sendo usado por outro processo), então o processo requisitante deverá esperar até poder alocar o recurso;
2. **Uso.** O processo pode operar com o recurso (por exemplo, se o recurso é uma impressora linear, o processo pode imprimir na impressora);
3. **Liberação.** O processo libera o recurso.

A requisição e liberação de recursos são *system calls*. Exemplos de *system calls* são Requisição/Liberação de Dispositivos, Abrir/Fechar Arquivos, Alocação/Liberação de Memória. O uso de recursos também pode ser feito somente através de *system calls* (por exemplo, para ler ou gravar um arquivo ou dispositivo de I/O). Consequentemente, para cada uso, o sistema operacional checka para certificar-se de que o processo que usa o recurso, requisitou e teve alocado o recurso solicitado. Uma tabela do sistema registra quando se cada recurso está livre ou alocado, e se alocado, para que processo. Se um processo requisita um recurso que está alocado para outro processo, ele poderá ser adicionado a uma fila de processos esperando por este recurso.

Um conjunto de processos está em estado de **Deadlock** quando cada processo do conjunto está esperando por um evento que somente pode ser causado por outro processo do conjunto. Os eventos que estamos nos referindo aqui são aquisição e liberação de recursos. Entretanto, outros tipos de eventos podem resultar em **Deadlock**.

Para ilustrar um estado de **Deadlock**, considere um sistema com três drivers de fitas. Suponha que existam três processos, cada um alocando um desses drivers de fitas. Se cada processo agora requisitar outro driver de fita, os três processos entrarão em um estado de **Deadlock**. Cada um está esperando por um evento "driver de fita foi liberado", que somente pode ser causado por um dos outros processos que estão em estado de espera. Este exemplo ilustra um **Deadlock** envolvendo processos competindo pelo mesmo tipo de recurso.

**Deadlocks** também podem envolver diferentes tipos de recursos. Por exemplo, considere um sistema com uma impressora e uma leitora de cartões. Suponha que o processo P está alocando a leitora de cartões e o processo Q está alocando a impressora. Se P agora requisita a impressora e Q requisita a leitora de cartões, um **Deadlock** ocorrerá.

#### 4.1.1 Caracterização de Deadlock

É óbvio que **Deadlocks** são indesejáveis, já que em uma situação de **Deadlock**, processos nunca terminam sua execução e os recursos do sistema ficam paralisados, impedindo que outros serviços sejam iniciados. Antes de discutirmos os vários métodos para tratar o problema de **Deadlock**, será muito útil descrever alguns aspectos que caracterizam o **Deadlock**.

Uma situação de **Deadlock** pode acontecer se e somente se as quatro condições seguintes acontecerem ao mesmo tempo em um sistema.

1. **Exclusão Mútua.** Pelo menos um recurso está alocado em um modo não compartilhado; isto é, somente um processo por vez pode usar o recurso. Se outro processo requisita este recurso, o processo requisitante deverá ser adiado até que o recurso tenha sido liberado.
2. **Hold and Wait.** Deverá existir um processo que está alocando pelo menos um recurso e está esperando para alocar recursos adicionais que estão sendo alocados por outros processos neste momento.

3. **Ausência de Preempção.** Recursos não podem sofrer preempção. Isto é, um recurso somente pode ser liberado voluntariamente pelo processo que o alocou, após este processo ter terminado sua tarefa.
4. **Espera Circular.** Deverá existir um conjunto  $\{P_0, P_1, \dots, P_n\}$  de processos em estado de espera em que o processo  $P_0$  está esperando por um recurso que está alocado pelo processo  $P_1$ ,  $P_1$  está esperando por um recurso que está alocado pelo processo  $P_2, \dots, P_{n-1}$  está esperando por um recurso que está alocado por  $P_n$ , e  $P_n$  está esperando por um recurso que está alocado pelo processo  $P_0$ .

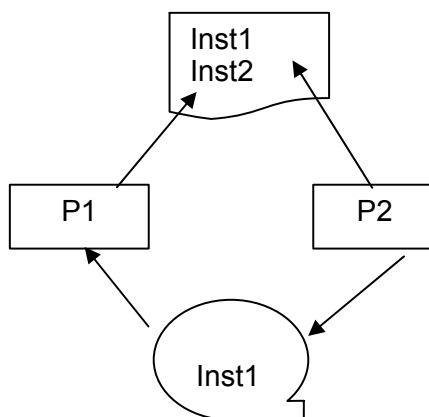
Enfatizamos que todas as quatro condições precisam necessariamente ocorrer ao mesmo tempo para que um **DeadLock** ocorra. A condição de espera circular implica que a condição alocar e esperar aconteça, então, a quatro condições não são completamente independentes. Entretanto, nos iremos ver que é muito comum considerar cada condição separadamente.

Podemos ver estas quatro condições em nosso exemplo da travessia do rio. Um **DeadLock** ocorre se e somente se duas pessoas de lados opostos do rio se encontram no meio. A exclusão mútua obviamente acontecem, desde que no máximo uma pessoa pode estar pisando em uma pedra de cada vez. A condição alocar e esperar é satisfeita, desde que cada pessoa pisando uma pedra está esperando para pisar na próxima. A condição não preempção acontece, desde que uma pedra que está sendo pisada não pode ser removida à força da pessoa que a está pisando. Finalmente, a condição de espera circular acontece, desde que a pessoa vindo do lado Este está esperando pela pessoa vinda do Oeste, enquanto que a pessoa vindo do Oeste, por sua vez, está esperando pela pessoa vinda do Este. Nenhuma das duas pode prosseguir enquanto uma está esperando que a outra remova seu pé de uma das pedras.

#### 4.1.2 Grafo de Alocação de Recursos

Um grafo pode mostrar a ocorrência ou não ocorrência de **DeadLock**. Alguns fatores devem ser levados em consideração: o número de processos que acessam um determinado recurso ao mesmo tempo e quantas são as instâncias existentes para este recurso.

Como exemplo podemos citar o seguinte sistema: 02 processos, 02 impressoras e 03 drives de fitas. Se temos um gráfico que mostra a seguinte situação:  $P_1$  e  $P_2$  acessando ao mesmo tempo uma impressora cada um e também ao mesmo tempo um drive de fita cada. Apesar de ser o mesmo recurso, impressora e drive de fita, o **deadlock** não acontece pois existem instâncias diferentes para cada recurso.



Agora, no caso de termos um grafo onde temos situação idêntica a anterior e, não tendo sido liberado nenhum recurso, um dos processos tenta alocar mais uma impressora além da já alocada, teremos um **DeadLock**. Neste caso, o **DeadLock** ocorre porque a impressora só tem duas instâncias e para que fosse evitado seria necessário que existisse mais uma instância.

#### 4.2 Prevenção de Deadlock

Foram discutidos algoritmos de prevenção de **DeadLock** na Seção 8.3 acima. Para prevenir **DeadLock**, é preciso garantir que pelo menos uma das quatro condições necessárias para sua ocorrência nunca se satisfaça. Um efeito colateral de prevenir **DeadLock** por este método, porém, é possivelmente a baixa utilização de dispositivos e capacidade de processamento do sistema reduzida.

Um método alternativo para evitar **DeadLock** é requerer informação adicional sobre como os recursos serão solicitados. Por exemplo, em um sistema com um leitor de cartão e uma impressora de linha, poderia nos ser informado que o processo P solicitará primeiro o leitor de cartão, e depois a impressora de linha, antes de ceder ambos os recursos. Por outro lado, o processo Q solicitará primeiro a impressora de linha, e depois o leitor de cartão. Com este conhecimento da sucessão completa de solicitações e liberações para cada processo, nós podemos decidir qual dos dois deve prosseguir ou esperar. Cada solicitação requer que o sistema considere os recursos atualmente disponíveis, os recursos alocados atualmente a cada processo, e as solicitações e liberações futuras de cada processo, decidir se a solicitação atual pode ser satisfeita ou se tem que esperar para evitar um possível **DeadLock** futuro.

Há vários algoritmos que diferem na quantidade e tipo de informações requeridas. O modelo mais simples e mais útil requer que cada processo declare o *número de máximo* de recursos de cada tipo que pode precisar. Dado *uma* informação *a priori*, para cada processo, sobre o número de máximo de recursos de cada tipo que podem ser solicitados, é possível construir um algoritmo que assegure que o sistema nunca entrará em um estado de deadlock. Este algoritmo define como evitar uma situação de deadlock. Um algoritmo de prevenção de deadlock examina dinamicamente o estado de alocação dos recursos para assegurar que nunca tenhamos o que chamamos de espera circular (circular wait). O estado de alocação dos recursos é definido pelo número de recursos disponíveis e alocados, e o máximo de recursos exigidos pelos processos.

O sistema está em estado seguro se pode alocar os recursos para cada processo (até seu máximo) em alguma ordem e ainda evita uma situação de deadlock

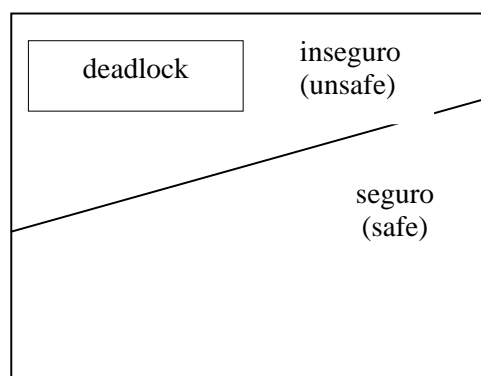
Mais formalmente, um sistema só está no estado seguro se nele existe *uma sucessão segura*. Uma sucessão de processos  $\langle p_1, p_2, \dots, p_n \rangle$  é uma sucessão segura para a alocação de recursos se para cada  $p_i$ , os recursos que  $p_i$  ainda pode solicitar podem ser satisfeitos pelos recursos atualmente disponíveis mais todos recursos segurados por um processo  $p_j$ , com  $j < i$ . Nesta situação, se os recursos necessários para o processo  $p_i$  não estão imediatamente disponíveis, então  $p_i$  poderia esperar até  $p_j$  terminar. Quando eles terminar,  $p_i$  pode obter todos os seus recursos necessários, completar suas tarefas designadas, devolver seus recursos alocados, e terminar. Quando  $p_i$  terminar,  $p_{i+1}$  pode obter seus recursos necessários, e assim por diante. Se nenhuma tal sucessão existe, então é dito que o estado do sistema é inseguro.

Um estado seguro não é um estado de deadlock. Reciprocamente, um estado de deadlock é um estado inseguro. Nem todos os estados inseguros são estados de deadlock, porém (Figura 8.6), um estado inseguro pode conduzir a um deadlock. Contanto que o estado esteja seguro, o sistema operacional pode evitar estados inseguros (e deadlocks). Em um estado inseguro, o sistema operacional não pode impedir que os processos de solicitem recursos de tal um modo que aconteça uma situação de deadlock: o comportamento dos processos controlam os estados inseguros.

Para ilustrar, considere um sistema com doze drives de fita magnética e três processos:  $P_0$ ,  $P_1$ , e  $P_2$ . O processo  $P_0$  requer dez drives de fita, o processo  $P_1$  pode precisar de no máximo quatro, e o processo  $P_2$  pode precisar até de nove drives de fita. Suponha que em um momento  $T_0$  o processo  $P_0$  está segurando cinco drives de fita, o processo  $P_1$  está segurando dois, e o processo  $P_2$  está segurando dois. (Assim há três drives fita disponíveis).

	<u>Necessidades</u> <u>Máximas</u>	<u>Necessidades</u> <u>Atuais</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

No momento  $T_0$ , o sistema está funcionando em estado de segurança. A sucessão  $\langle P_1, P_0, P_2 \rangle$  satisfaz a condição de segurança, desde de que o processo  $P_1$  possa alocar imediatamente todos os seus drives de fita e depois devolve-los (o sistema terá então cinco drives de fita disponíveis), o processo  $P_0$  pode adquirir os todos os seus drives de fita e os devolver (o sistema terá então dez drives de fita disponíveis), e finalmente o processo  $P_2$  poderia adquirir todos os seus drives de fita e os devolver (O sistema terá então todas os doze drives de fita disponíveis).



Note que é possível o sistema ir de um estado seguro para um estado inseguro. Suponha que em um momento  $T1$ , o processo  $P2$  solicite e aloque mais um drive de fita. O sistema já não está mais em um estado seguro. Nesse instante, só o processo  $P1$  pode alocar todos os seus drives de fita. Quando os devolve o sistema terá apenas quatro drives de fita disponíveis. Se o processo  $P0$  estiver alocando cinco drives de fita, e solicitar mais cinco, o processo  $P0$  terá que esperar, pois o sistema não tem cinco drives de fita disponíveis. O mesmo ocorre para o processo  $P2$ , se ele solicitar mais seis drives de fita, ele terá que esperar, o que leva o sistema a uma situação de deadlock.

Nosso erro está em garantir ao processo  $P2$  mais um drive de fita. Se o processo  $P2$  for colocado em estado de espera até que um dos outros processos tenham terminado e liberado seus recursos, então nós poderíamos Ter evitado a ocorrência de deadlock.

Dado o conceito de estado seguro, nós podemos definir algoritmos de prevenção que assegurem a um sistema a não ocorrência de deadlock. A idéia simplesmente é assegurar que o sistema sempre permanecerá em um estado seguro. Inicialmente o sistema está em um estado seguro. Sempre que um processo pede um recurso que está atualmente disponível, o sistema tem que decidir se o recurso pode ser alocado imediatamente ou se o processo tem que esperar. O pedido só é concedido se deixar o sistema em um estado seguro.

Note que neste esquema se um processo pedir um recurso que está atualmente disponível, ainda pode ter que esperar. Assim a utilização do recurso pode ser mais baixa que sem um algoritmo de prevenção de deadlock.

O algoritmo de prevenção de deadlock descrito abaixo é comumente conhecido como o **algoritmo do banqueiro (banker's algorithm)**. O nome foi escolhido desde que este algoritmo pudesse ser usado em um sistema bancário para assegurar que o banco nunca aloca seu dinheiro vivo disponível de tal um modo que já não possa satisfazer as necessidades de todos seus clientes.

Quando um processo novo entrar no sistema, tem que declarar o número máximo de instâncias de cada tipo de recurso que poderá precisar. Este número não poderá exceder o número total de recursos no sistema. Quando um usuário pedir um conjunto de recursos, deve ser determinado se a alocação destes recursos deixará o sistema em um estado seguro. Nesse caso, os recursos são alocados; caso contrário, o processo tem que esperar até algum outro processo libere recursos suficientes.

Devem ser mantidas várias estruturas de dados para implementar o algoritmo do banqueiro. Estas estruturas de dados codificam o estado de alocação dos recursos do sistema. Considere  $n$  como o número de processos no sistema e  $m$  como número de tipos de recursos. Nós precisamos das seguintes estruturas de dados:

- Disponível:** Um vetor de tamanho  $m$  que indica o número de recursos disponíveis de cada tipo. Se  $Disponível[j] = k$ , há  $k$  instâncias do tipo de recurso  $r_j$  disponíveis.
- Max:** Uma matriz  $n \times m$  que define a demanda máxima de cada processo. Se  $Max(i,j) = k$ , então o processo  $P1$  pode pedir no máximo  $k$  instâncias do tipo de recurso  $r_j$ .
- Alocação:** Uma matriz  $n \times m$  que define o número de recursos de cada tipo atualmente alocados a cada processo. Se  $Alocação[i,j] = k$ , então, o processo  $P1$  está atualmente alocando  $k$  instâncias do tipo de recurso  $r_j$ .
- Necessidade:** Uma matriz  $n \times m$  que indica a necessidade de recursos restante de cada processo. Se  $Necessidade(i,j) = k$ , então o processo  $P1$  pode precisar de mais  $k$  instâncias do tipo de registro  $r_j$  para completar sua tarefa. Note que  $Necessidade(i,j) = Max(i,j) - Alocação(i,j)$ .

Estas estruturas de dados variam em tamanho e valores no decorrer do tempo. Para simplificar a apresentação do algoritmo, nos deixe estabelecer alguma anotação. Considere  $X$  e  $Y$  como vetores de duração  $n$ . Nós dizemos que  $X \leq Y$  se e somente se  $X[i] \leq Y[i]$  para todo o  $i = 1, 2, \dots, n$ , por exemplo, se  $X = (1, 7, 3, 2)$  e  $Y = (0, 3, 2, 1)$  então  $Y \leq X$ .  $Y < X$  se  $Y \leq X$  e  $Y \neq X$ .

Nós podemos tratar cada linha das matrizes Alocação e Necessidade como vetores e referir-nos a elas como  $Alocação(i)$  e  $Necessidade(i)$ , respectivamente.  $Alocação(i)$  especifica os recursos atualmente alocados para processar  $p_i$ , enquanto  $Necessidade(i)$  especifica os recursos adicionais que o processo  $p_i$  pode solicitar para completar sua tarefa.

Considere  $Solicitação(i)$  como um vetor de solicitações do processo  $p_i$ . Se  $Solicitação(i)[j] = k$ , então o processo  $p_i$  requer  $k$  instâncias do tipo de recurso  $r$ . Quando uma solicitação de recursos é feita pelo processo  $p_i$ , as seguintes ações são tomadas:

1. Se  $Solicitação(i) \leq Necessidade(i)$  então prossiga para o passo 2. Caso contrário nós temos um erro, desde que o processo tenha excedido sua reivindicação máxima.
2. Se  $Solicitação(i) \leq Disponível(i)$  então prossiga para o passo 3. Caso contrário os recursos não se encontram disponíveis e  $p_i$  tem que esperar.
3. O sistema pretende alocar os recursos solicitados pelo processo  $p_i$  modificando o estado, como descrito a seguir:

$Disponível(i) := Disponível(i) - Solicitação(i);$   
 $Alocação(i) := Alocação(i) + Solicitação(i);$   
 $Necessidade(i) := Necessidade(i) - Solicitação(i)$

Se o estado de alocação recursos resultante está seguro, a transação é completada e o processo  $p_i$  pode alocar seus recursos. Porém, se o novo estado é inseguro, então  $p_i$  tem que esperar por  $Solicitação(i)$  e o estado de alocação de recursos anterior é restabelecido.

O algoritmo para descobrir se um sistema está em estado seguro ou não pode ser descrito como segue:

1. Considere **Trabalho** e **Fim** como vetores de tamanho  $m$  e  $n$ , respectivamente.  
**Inicialize Trabalho** := **Disponível** e **Fim**[ $i$ ] := **false** para  $i = 1, 2, \dots, n$ .
2. Ache um  $i$  tal que:  
**Fim**[ $i$ ] = **false**, e  
**Necessidade**( $i$ )  $\leq$  **Trabalho**.
3. **Trabalho** := **Trabalho** + **Alocação**;  
**Fim**[ $i$ ] := **true**;  
vá para o passo 2.
4. Se **Fim**[ $i$ ] = **true** então para todo o  $i$ , o sistema está em um estado seguro,

Vejamos um exemplo. Considere um sistema com cinco processos **P0**, **P1**, ..., **P4** e três tipos de recursos **A**, **B**, **C**. O tipo de recurso **A** tem 10 instâncias, o tipo de recurso **B** tem 5 instâncias, e o tipo de recurso **C** tem 7 instâncias.

	Alocação			Necessidade			Disponível		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

É definido o conteúdo da matriz **Necessidade = Max – Alocação**, como:

Necessidade				
	A	B	C	
P0	7	4	3	
P1	1	2	2	
P2	6	0	0	
P3	0	1	1	
P4	4	3	1	

Supondo que o sistema está atualmente em um estado seguro, realmente, a sequência **<P1, P3, P4, P2, P0>** satisfaz o critério de segurança.

	Alocação			Necessidade			Disponível		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	1	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Suponha agora que o processo *P1* solicite uma instância do tipo de recurso A adicional e duas instâncias de tipo de recurso C, assim *Solicitação(i) = (1,0,2)*. Para decidir se esta solicitação pode ser atendida imediatamente, nós primeiramente conferimos se *Solicitação(i) ≤ Disponível(i)* (quer dizer,  $(1,0,2) \leq (3,3,2)$ ) que é verdade. Nós simulamos então que esta solicitação foi cumprida e foi chegado ao seguinte novo estado:

Nós temos que determinar se este novo estado do sistema está seguro. Fazendo assim nós executamos nosso algoritmo de segurança e descobrimos que a sucessão **<P1, P3, P4, P0, P2>** satisfaz nossa exigência de segurança. Conseqüentemente nós podemos conceder as solicitações do processo *P1* imediatamente.

Você deveria poder ver, entretanto, que neste estado, uma solicitação para (3,3,0) por *P4* não pode ser concedida desde que os recursos não estejam disponíveis. Uma solicitação para (0,2,0) por *P0* não pode ser concedido embora os recursos estejam disponíveis, desde que o estado resultante seja inseguro.

#### 4.2.1 Exclusão Mútua

#### 4.2.2 Condição Hold and Wait

#### 4.2.3 Ausência de Preempção

#### 4.2.4 Espera Circular

### 4.3 Algoritmos de Banker

Algoritmo de Segurança – Garante ausência de deadlock pois vai se certificar que tem recurso para todos os processos pegarem independente da ordem em que cada processo vai pegar o recurso.

O nome foi escolhido desde que este algoritmo pudesse ser usado em um sistema bancário para assegurar que o banco nunca aloca seu dinheiro vivo disponível de tal um modo que já não possa satisfazer as necessidades de todos seus clientes.

Quando um processo novo entrar no sistema, tem que declarar o número máximo de instâncias de cada tipo de recurso que poderá precisar. Este número não poderá exceder o número total de recursos no sistema. Quando um usuário pedir um conjunto de recursos, deve ser determinado se a alocação destes recursos deixará o sistema em um estado seguro. Nesse caso, os recursos são alocados; caso contrário, o processo tem que esperar até algum outro processo libere recursos suficientes.

Devem ser mantidas várias estruturas de dados para implementar o algoritmo do banqueiro. Estas estruturas de dados codificam o estado de alocação dos recursos do sistema. Considere  $n$  como o número de processos no sistema e  $m$  como número de tipos de recursos. Nós precisamos das seguintes estruturas de dados:

**Disponível:** Um vetor de tamanho  $m$  que indica o número de recursos disponíveis de cada tipo. Se  $Disponível[j] = k$ , há  $k$  instâncias do tipo de recurso  $r_j$  disponíveis.

**Max:** Uma matriz  $n \times m$  que define a demanda máxima de cada processo. Se  $Max(i,j) = k$ , então o processo  $P_i$  pode pedir no máximo  $k$  instâncias do tipo de recurso  $r_j$ .

**Alocação:** Uma matriz  $n \times m$  que define o número de recursos de cada tipo atualmente alocados a cada processo. Se  $Alocação[i,j] = k$ , então, o processo  $P_i$  está atualmente alocando  $k$  instâncias do tipo de recurso  $r_j$ .

**Necessidade:** Uma matriz  $n \times m$  que indica a necessidade de recursos restante de cada processo. Se  $Necessidade(i,j) = k$ , então o processo  $P_i$  pode precisar de mais  $k$  instâncias do tipo de registro  $r_j$  para completar sua tarefa. Note que  $Necessidade(i,j) = Max(i,j) - Alocação(i,j)$ .

(1) Supõe atendimento da requisição

(2) Processos requisitam  $Max R_j; j=1...m$

(3) Aplica Algoritmo Detecção de Deadlock

(4) Se não há deadlock, entrega recursos requisitados, caso contrário entrega os recursos e bloqueia o processo.

O algoritmo:

Entrega recursos requisitados:

Requisição  $(i,j) = k$  //  $P_i$  requer  $k$  instâncias do recurso  $R[j]$

Se  $Requisição(i,j) > Necessidade(i,j)$  --> Erro

Senão

Se  $Requisição(i,j) \leq Disponível(i,j)$  então

$Disponível(i,j) -= Requisição(i,j)$      $Alocação(i,j) += Requisição(i,j)$

$Necessidade(i,j) -= Requisição(i,j)$

Senão

$P_i$  --> Espera

Fim Se

Conclusão: Se sistema está seguro --> Aloca recursos  $P_i$

Senão -->  $P_i$  deve esperar

OBS:  $Necessidade(i,j) = (Max(i,j) - Alocação(i,j))$ .



#### 4.4 Detecção de Deadlock

Algoritmo de Detecção:

Sejam *Trabalho* e *Fim* vetores de tamanho *m* e *n*, respectivamente. Inicialize:

(a) *Trabalho*<sub>*i*</sub> = *Disponível*<sub>*i*</sub>

(b) For *i* = 1, 2, ..., *n*, se *Alocação*<sub>*i*</sub> ≠ 0, então *Fim*<sub>*i*</sub> = *false*; caso contrário, *Fim*<sub>*i*</sub> = *true*.

2. Encontre um índice *i* tal que :

(a) *Fim*<sub>*i*</sub> == *false*

(b) *Requisição*<sub>*i,j*</sub> ≤ *Trabalho*<sub>*j*</sub> para *j*=1, 2,...*m*

Se não houver tal *i*, vá para a etapa 4

*Trabalho*<sub>*j*</sub> = *Trabalho*<sub>*j*</sub> + *Alocação*<sub>*i,j*</sub> e *Fim*<sub>*i*</sub> = *true* vá para a etapa 2.

4. Se *Fim*<sub>*i*</sub> == *false* para algum *i*,  $1 \leq i \leq n$ , então o sistema está em um estado de Deadlock. Além do mais, se *Fim*<sub>*i*</sub> == *false*, então o *Pi* está em um Deadlock.

O algoritmo exige uma ordem de  $O(m \times n^2)$  operações para detectar se o sistema está no estado de Deadlock.

## 4.5 Remoção do Deadlock

Quando um algoritmo de detecção determina que existe um problema de deadlock, o sistema deve corrigir este problema. Isto pode ser feito de duas formas. Uma solução é simplesmente matar um ou mais processos de forma a quebrar o ciclo de espera. A segunda opção é desalocar os recursos de um ou mais processos em deadlock.

### Término do Processo

Se optarmos por esta solução, poderemos matar todos os processos em deadlock ou matar os processos um após o outro até que o deadlock seja eliminado. A primeira opção é mais cara, pois estes processos podem estar sendo executados a muito tempo, o que resultará em tempo perdido e uma provável re-execução mais tarde. O segundo método requer um overhead considerável, pois após matar cada processo, o algoritmo de detecção de deadlock deverá ser chamado.

Eliminar um processo pode não ser fácil, dependendo do tipo de recurso envolvido. Se o processo estiver no meio de uma atualização de arquivo, abortá-lo poderá causar inconsistências no arquivo. O sistema deve garantir que esses recursos sejam liberados sem problemas.

Se o método de término parcial for utilizado, deveremos determinar quais processos dentre os em deadlock deverão ser abortados. Basicamente, deveremos abortar os processos que acarretarão menor custo.

### Preempção de Recursos

Esta solução consiste em liberar sucessivamente recursos alocados por processos em deadlock e entregar esses recursos a outros processos, até que o ciclo de espera termine. Para que essa solução seja realmente eficiente, o sistema precisará ser capaz de :

- determinar a ordem de preempção de forma a minimizar os custos,
- suspender um processo, liberar seus recursos e, após a solução do problema, retornar à execução do processo, sem perder o processamento já realizado (rollback),
- garantir que não serão sempre os mesmos processos a terem recursos desalocados (starvation). Caso contrário, o processo selecionado poderá nunca completar sua execução. A solução mais comum neste caso é incluir o número de rollbacks no fator custo.

## 4.6

# *Parte II*

# *Sistemas Distribuídos*

## 5 Sistemas Distribuídos

Caracterizaremos um **Sistema Distribuído (SD)** através da seguinte definição:

### **Sistema Distribuído**

**Um Sistema Distribuído é uma coleção de computadores independentes que se apresentam para os usuários do sistema como sendo um único sistema de computação.**

Esta definição contempla dois aspectos importantes de um SD.

**Software** - Os usuários vêem o SD como um único computador;

**Hardware** - Os processadores são autônomos;

Para tornar clara esta definição vejamos alguns exemplos:

#### **( I ) Rede de estações de trabalho de um departamento de uma companhia.**

Se neste sistema além das estações de trabalho dos usuários, tivermos um sistema de arquivos compartilhado ( com todos os arquivos sendo acessados da mesma maneira e com mesmo "**path**" por todas as máquinas ) e além disso sempre que o usuário digitar um comando, o sistema procurar qual o melhor lugar ( máquina ) para executá-lo ( seja na própria máquina ou em alguma máquina menos sobrecarregada ) estaremos diante de um **SD**. Neste caso nós poderíamos dizer que o sistema se comporta como um sistema time-sharing virtual.

#### **( II ) Banco com centenas de agências espalhadas**

Cada agência tem um computador central para armazenar dados dos clientes da agência e cuidar das transações bancárias. Além disso cada agência pode se comunicar com um computador central da matriz e com os computadores de outras agências.

Se para este sistema as transações puderem ser realizadas independentemente da origem da conta do cliente e para os clientes tudo se passa como se estivessem se comunicando com o antigo computador central ( mainframe ) então nós teremos aqui um SD.

### **5.1 Sistema Distribuído (SD) x Sistema Centralizado (SC)**

Nem sempre a construção de um SD é uma boa idéia. Existem vantagens e desvantagens na utilização de um Sistema Distribuído que iremos avaliar daqui em diante.

#### **5.1.1 Vantagens dos SD sobre SC**

Vantagens	Descrição
Econômicas	Microprocessadores oferecem atualmente uma melhor relação preço/performance que os mainframes;
Velocidade	Um <b>SD</b> pode ter um poder computacional bem superior ao de um mainframe, já que a velocidade deste está limitada a velocidade da luz ( Teoria da relatividade geral de Einstein );
Inerentemente Distribuído *	Algumas aplicações envolvem máquinas geograficamente separadas;
Segurança - Tolerância a Falhas	No caso de queda de uma máquina o sistema como um todo ainda pode funcionar, a custo de uma perda de performance;
Crescimento Incremental	A capacidade computacional pode ser incrementada gradativamente a medida que for necessário;

\* Exemplos de sistemas inerentemente distribuídos são:

**CSCW ( Computer-Supported Cooperative Work )** - neste sistema um grupo de pessoas, geograficamente separadas, trabalham juntas utilizando recursos do SD para, por exemplo, confeccionarem um relatório ou algum documento.

**CSCG ( Computer-Supported Cooperative Games )** - jogadores de diferentes regiões jogam uns contra os outros em tempo real.

### 5.1.2 Vantagens dos SD sobre conjunto independente de PCs

Mesmo os microprocessadores apresentando uma relação preço/performance bastante boa, não é necessário dedicar-se um PC para cada usuário porque sempre existirá a necessidade de comunicação e compartilhamento conforme mostra o quadro abaixo.

Vantagens	Descrição
Compartilhamento de Dados e Equipamentos	Permite que muitos usuários compartilhem dados e equipamentos/periféricos reduzindo os custos da instalação
Comunicação	Facilita interação entre as pessoas através, por exemplo, de Correio Eletrônico
Flexibilidade	Permite a distribuição da carga de processamento entre as máquinas disponíveis da forma mais econômica possível

### 5.1.3 Desvantagens dos SDs

Desvantagens	Descrição
Software	Existem poucos softwares no presente para ambientes distribuídos
Redes de Comunicação	A rede de comunicação pode saturar e outros problemas de performance e perda de mensagens podem ocorrer
Segurança	A facilidade para compartilhamento de dados pode permitir o acesso a dados secretos à pessoas desautorizadas

Apesar destes problemas potenciais, admite-se que as vantagens dos **SDs** superam as desvantagens. Acreditamos que nos próximos anos muitas organizações conectarão a maioria dos seus computadores em grandes SDs para prover um melhor, mais barato e mais conveniente serviço para os seus usuários.

## 5.2 Arquiteturas utilizadas em Sistemas Distribuídos

Os Sistemas Distribuídos são constituídos por diversos processadores que podem ser organizados de diferentes maneiras, especialmente em termos da forma que são interconectados e da forma que se comunicam.

Diversas classificações para sistemas com múltiplos processadores foram propostos, porém nenhuma dessas classificações conseguiu contemplar todas as possibilidades existentes. Provavelmente, a mais frequentemente citada é a classificação de Flynn (Flynn's Taxonomy – 1972). Nesta classificação, Flynn selecionou duas características que ele considerou essencial, que são:

Número de fluxos de instruções executadas;

Número de fluxos de dados sendo processados;

O produto cartesiano destas duas características produziu a seguinte caracterização:

<b>Número de fluxos de instrução</b>	<b>Número de fluxos de dados</b>	
	<b>Single</b>	<b>Multiple</b>
<b>Single</b>	SISD	SIMD
<b>Multiple</b>	MISD	MIMD

Usaremos a seguinte notação para as figuras representativas de cada classe de máquinas:

**FI** - Fluxo de Instruções

**FD** - Fluxo de Dados

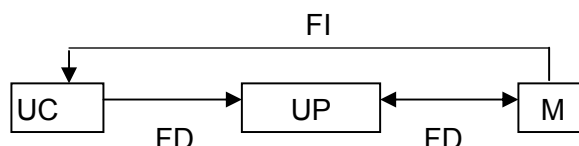
**UP** - Unidade de Processamento

**M** - Memória

**UC** - Unidade de Controle

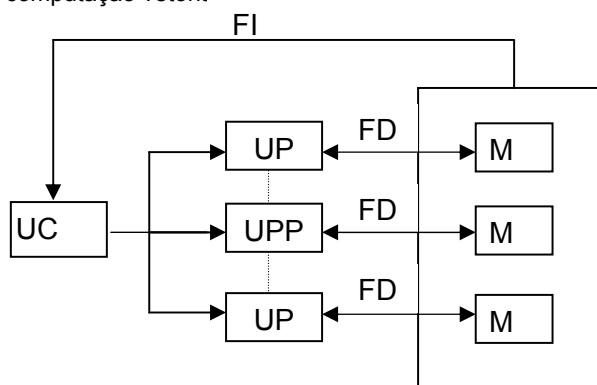
### **SISD – Single Instructions Stream Single Data Stream**

São os computadores tradicionais com uma CPU desde o PC até os Mainframes.

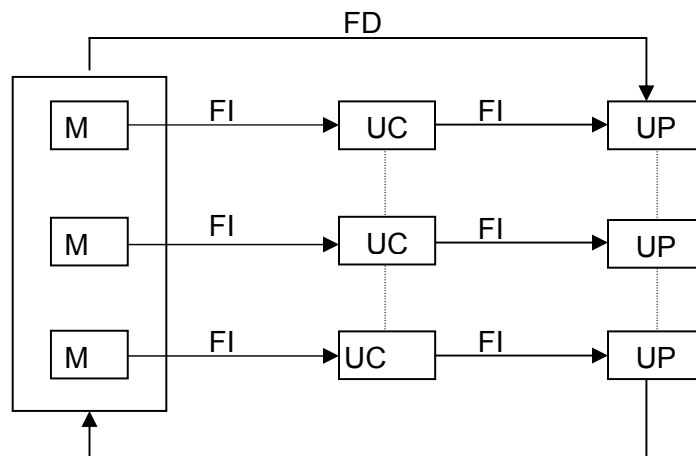


### **SIMD – Single Instructions Stream Multiple Data Stream**

Array Processors, computadores vetoriais e alguns supercomputadores. Nesta classificação enquadram-se as máquinas em cuja uma única instrução é aplicada em diversos dados em paralelo (todo dado residente em um processador). Estas máquinas são úteis para máquinas que fazem a mesma computação para muitos conjuntos de dados, como por exemplo computação vetorial

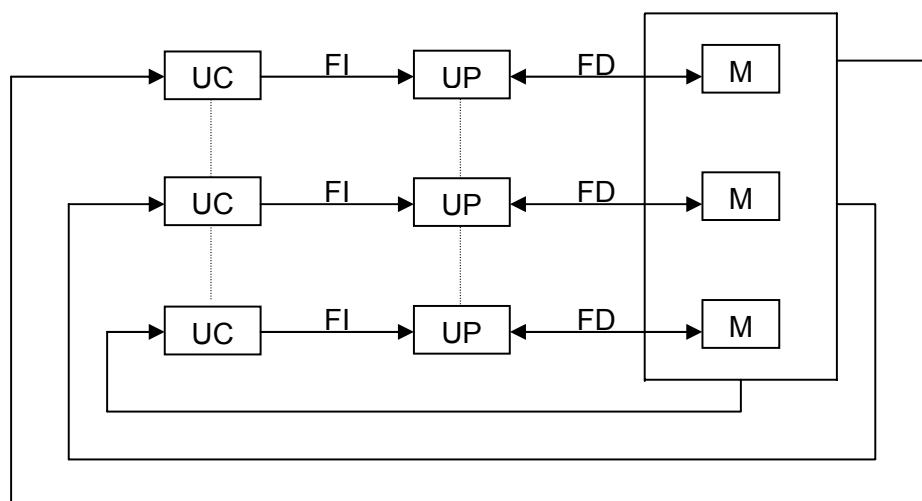


Não se conhece nenhum computador nesta categoria.

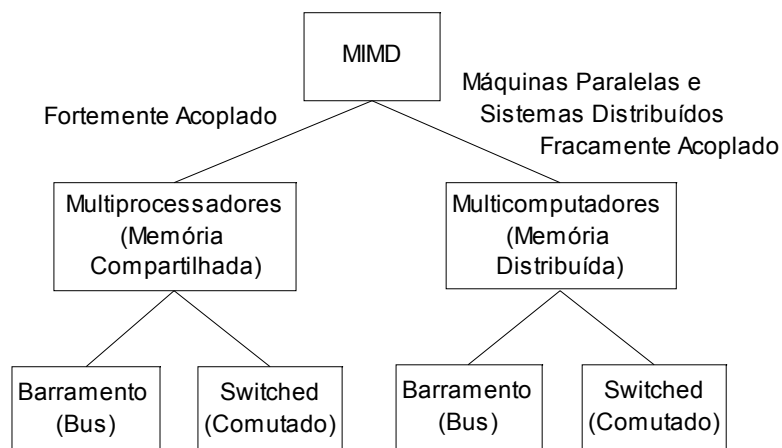


**MIMD – Multiple Instructions Stream Multiple Data Stream**

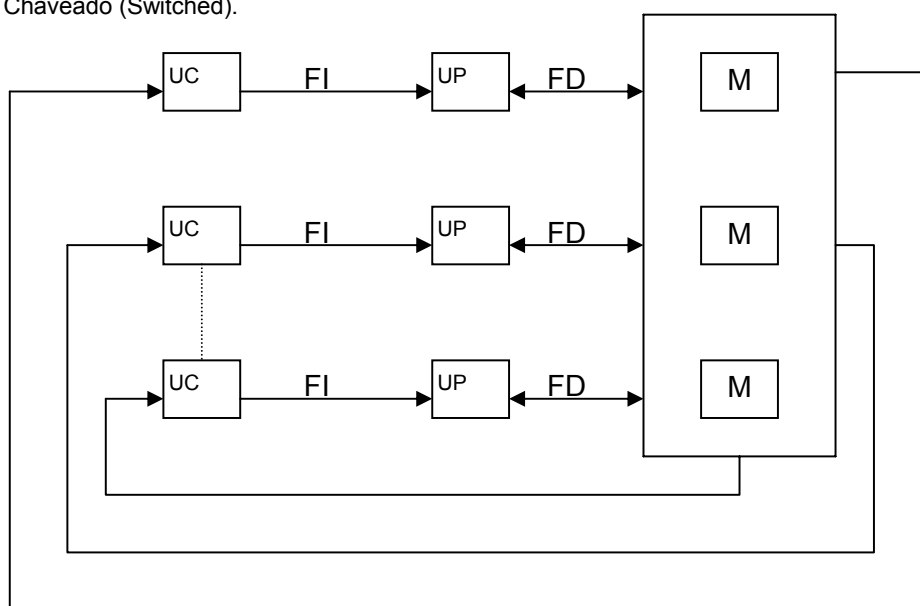
Nesta categoria enquadram-se os grupos de computadores independentes, cada um com seus respectivos programas e dados. Dessa forma, todos os SDs se enquadram nesta categoria, o que torna esta classificação muito geral para nós.



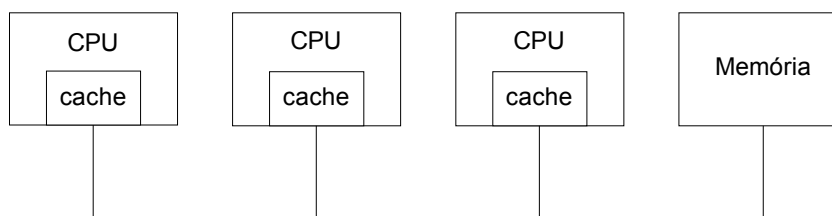
Entretanto, podemos ainda subdividir as máquinas MIMD em dois grupos: Multiprocessadores (memória compartilhada) e Multicomputadores (memória distribuída). A diferença fundamental entre estas duas categorias é que nos Multiprocessadores existe um único espaço de endereçamento virtual compartilhado pelas diversas CPUs, isto é, todas as máquinas compartilham a mesma memória caracterizando um sistema fortemente acoplado. Já em um Multicomputador, toda a máquina tem sua memória privativa o que caracteriza um sistema fracamente acoplado, pois a comunicação entre programas rodando nestas máquinas é, em geral, feita através de troca de mensagens (Ex.: rede de PCs conectados em rede local).



Cada uma destas categorias podem ser subdivididas de acordo com a arquitetura de interconexão: Barramento (Bus) e Comutado ou Chaveado (Switched).



### 5.2.1 Multiprocessadores com Barramento - Bus-Based Multiprocessors



Nesta arquitetura todas as CPUs são conectadas a um barramento comum juntamente com a memória. Um barramento típico é composto de barramentos de dados e de endereços de 32 ou 64 bits, cada leitura ou escrita de um dado na memória é feita pela CPU acessando o barramento e enviando um pedido para a memória que se encarrega de receber e executar o pedido disponibilizando o resultado para a CPU requerente através do Barramento.

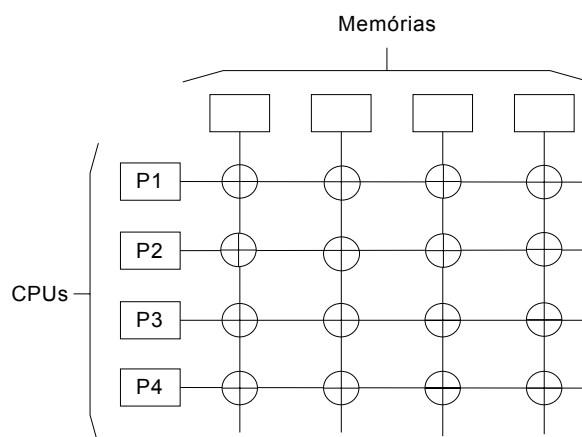


Um problema importante nesta arquitetura é que a medida que conectamos mais processadores a performance cai drasticamente devido a sobrecarga de tráfego causada no Barramento. A solução é a adoção de memórias cache para cada CPU de forma a diminuirmos o tráfego no barramento (devido ao princípio da localidade). Normalmente tem se utilizado caches de 64 KB a 1MB, o que proporciona uma taxa de acerto (hit rate) no cache da ordem de 90% ou mais. A introdução do cache, entretanto, acarreta um novo problema que é chamado Gerência de Memória, pois agora, em cada processador podemos ter diferentes versões de um mesmo dado. Uma possível solução deste problema é a estratégia chamada write – through cache, onde todo dado que é escrito no cache é escrito também na memória. Assim, cache hits de leitura não causam tráfego na rede, enquanto cache misses para leitura e cache hits ou misses para write causam.

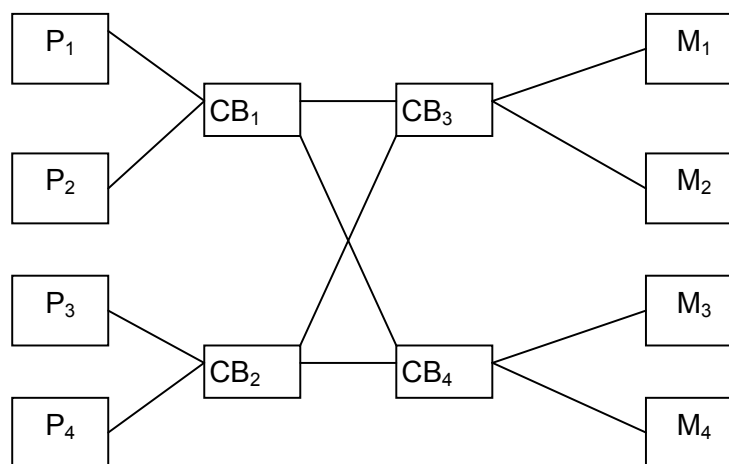
Além disso, todos os caches devem monitorar constantemente o Barramento, para invalidar ou atualizar qualquer informação do cache que esteja sendo gravada por outro processador (Cache-Invalidate, Cache-Update). Tais caches são chamados de Snoopy cache e a política de write-through resolve o problema de coerência de memória (memory coherence).

### 5.2.2 Multiprocessadores Chaveados - Switched Multiprocessors

A construção de Multiprocessadores com mais de 64 processadores é impraticável devido a elevada largura de banda que deveria ser atendida pelo barramento para permitir a conexão dos processadores à memória. Um método alternativo é a criação de circuitos chaveados (ou comutados), idéia equivalente a da Rede Telefônica, para permitir a conexão entre as CPUs. Esta rede de interconexão é chamada de Barramento cruzado (crossbar switch). O problema do switch é que para o caso de  $n$  CPUs e  $n$  memórias teremos  $n^2$  conexões, o que torna proibitivo o seu uso para valores de  $n$  elevados.



Uma solução alternativa ao barramento cruzado foi a criação de um barramento cruzado com múltiplos estágios. No caso do barramento anterior, poderíamos utilizar 4 barramentos cruzados de 2x2 ao invés de um barramento cruzado de um estágio. Neste caso, teríamos a necessidade de  $\log_2^n$  estágios cada um contendo  $n/2$  conexões, perfazendo um total de  $n/2 \log_2^n$  conexões, que apesar de ser melhor que  $n^2$ , ainda é um número bastante elevado.



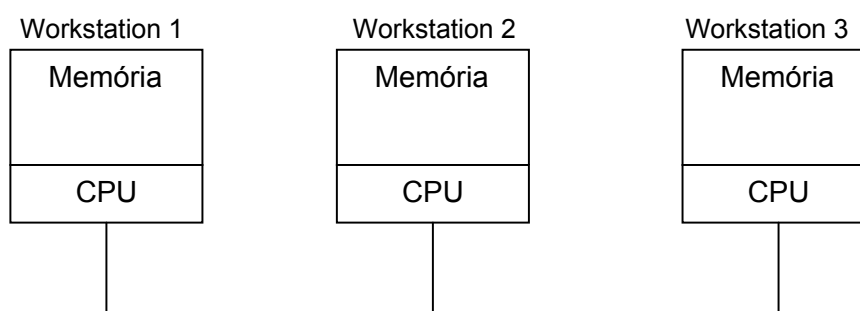
**Barramento Cruzado com 2 Estágios**

Para completar, mencionamos mais um problema: atraso devido a comunicação. Para  $n=1024$ , toda mensagem passaria por 10 estágios de switch da CPU para a memória e mais 10 para voltar fazendo um total de 20 estágios. considerando-se uma CPU RISC rodando a 100 mips teríamos uma execução executando a cada 10 nsec, o que exigiria um tempo de switch do barramento cruzado de 0.5 nsec, o que tornaria seu custo bastante elevado.

Finalizando, concluímos que a construção de um multiprocessador com memória compartilhada em uma arquitetura fortemente acoplada é possível, porém difícil e cara.

### 5.2.3 Multicomputadores com Barramento - *Bus-Based MultiComputers*

Ao contrário dos "Switched Multiprocessors", a construção destes multicomputadores é fácil. Cada CPU tem uma memória local, restando apenas definir-se como uma CPU acessa uma memória remota. Observe que agora o tráfego é bem menor, já que só teremos tráfego na rede para comunicações entre CPUs. Na figura abaixo, vemos que, topologicamente, este esquema se parece com o caso "Bus-Based Multiprocessor", mas como o tráfego na rede é menor, não há necessidade de utilizarmos um switch de alta velocidade. Em alguns casos, são usadas LAN cuja taxa de transferencia se situa na faixa dos 10-100 mbps, bem inferior aos 300 mbps de um switch comercial.

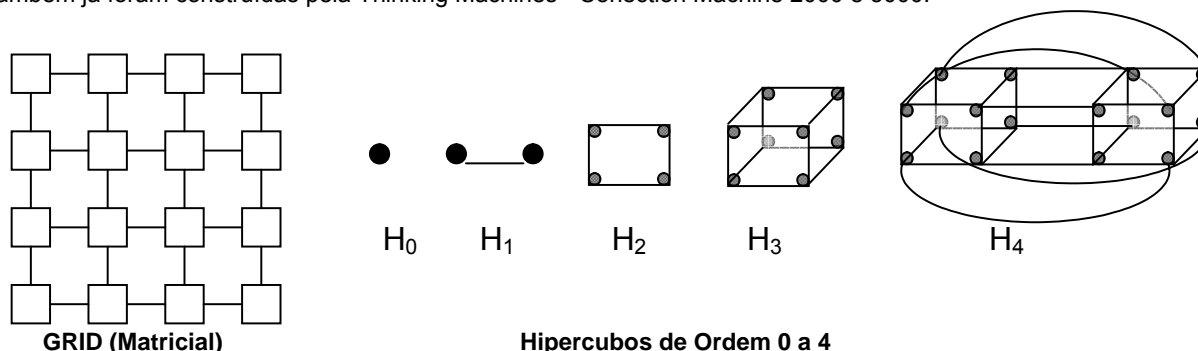


### 5.2.4 Multicomputadores Chaveados - Switched Multicomputers

Neste caso, vários esquemas de conexão foram propostos, todos eles colocando a CPU com acesso direto e exclusivo a sua memória local. Duas categorias se popularizaram: foi a GRID e a Hipercúbica.

A categoria GRID, se presta a solução de problemas cuja estrutura tenha natureza bidimensional. Tais problemas são típicos da área de grafos e visão computacional.

As máquinas hipercúbicas são formadas por cubos n-dimensionais, conforme a figura. Observe que para um hipercubo n-dimensional, cada CPU tem n conexões para as outras 2n CPUs. Dessa forma, a complexidade das conexões aumenta de forma logarítmica em relação ao número de processadores. Como cada nó se conecta apenas aos seus vizinhos, uma mensagem entre CPUs não vizinhas tem que percorrer diversas arestas até chegar ao seu destino. Felizmente, o caminho máximo a ser percorrido por uma mensagem também cresce de forma logarítmica. (explique porquê!), o que representa uma grande vantagem em relação as máquinas com arquitetura GRID (matricial), cujo caminho máximo cresce com a raiz quadrada do número de CPUs. Máquinas hipercúbicas com 1024 processadores (IPSC 860 Intel) já estão sendo comercializadas a bastante tempo e hipercubos com 16k-64k CPUs também já foram construídas pela Thinking Machines - Connection Machine 2000 e 5000.



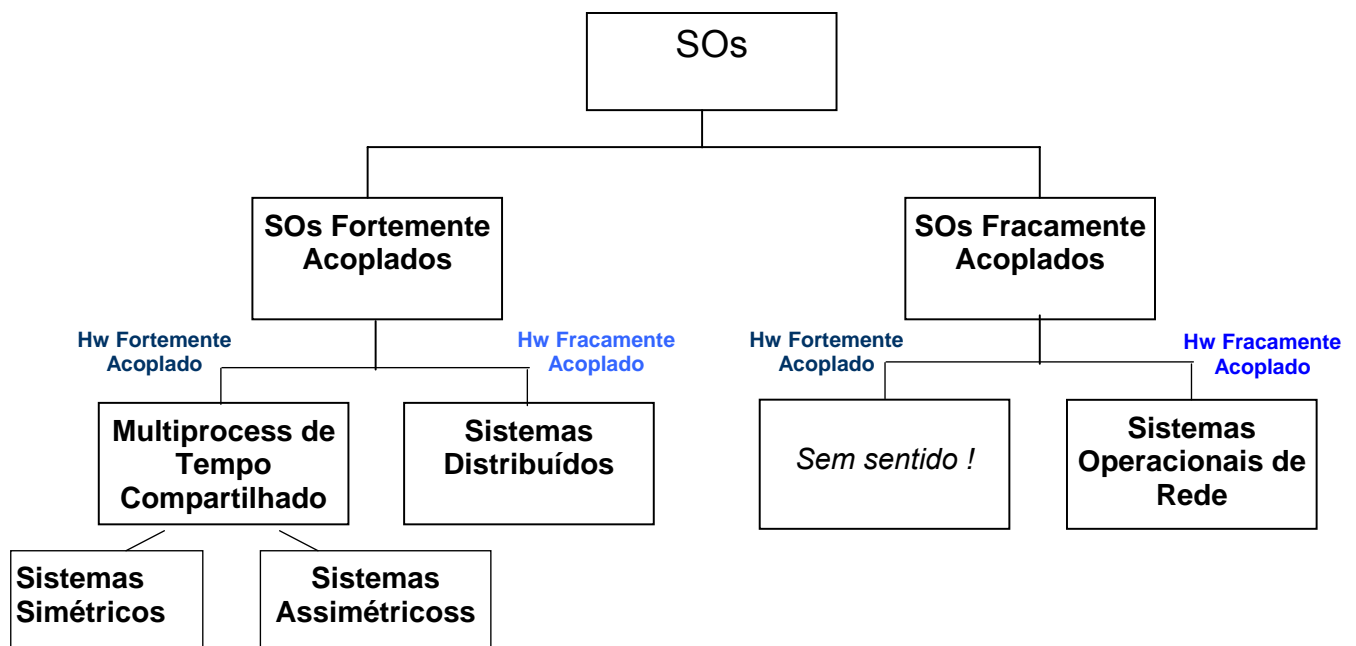
### 5.3 Classificação dos Sistemas Operacionais

Nesta seção definiremos uma classificação para os sistemas operacionais para multiprocessadores e multicomputadores. Podemos distinguir basicamente dois tipos de SOs: SOs Fracamente Acoplados e SOs Fortemente Acoplados.

**SOs Fracamente Acoplados** permitem que as máquinas e os usuários de um sistema distribuído sejam completamente independentes. Este é o caso, por exemplo, de uma rede de PCs que compartilham alguns recursos como impressoras laser, servidores de Banco de Dados, via uma rede local (LAN).

**SOs Fortemente Acoplados** permitem que os programas dos usuários sejam divididos em sub-programas para execução simultânea em mais de um processador.

Até agora, nós vimos então 4 tipos de Hardware distribuídos e dois tipos de SOs Distribuídos. A combinação destes diferentes tipos geraria 8 diferentes combinações. Como, do ponto de vista do usuário a tecnologia de interconexão (barramento ou switch) é transparente, nós acabamos tendo apenas 4 combinações de interesse:



### 5.3.1 Sistemas Operacionais de Rede - SOR

SOR se caracterizam pelo fato de serem sistemas operacionais fracamente acoplados operando sobre HW também fracamente acoplados. Isto é, cada nó possui um próprio SO, além de um HW e SW que possibilitam ter acesso a outros componentes da rede, compartilhando seus recursos.

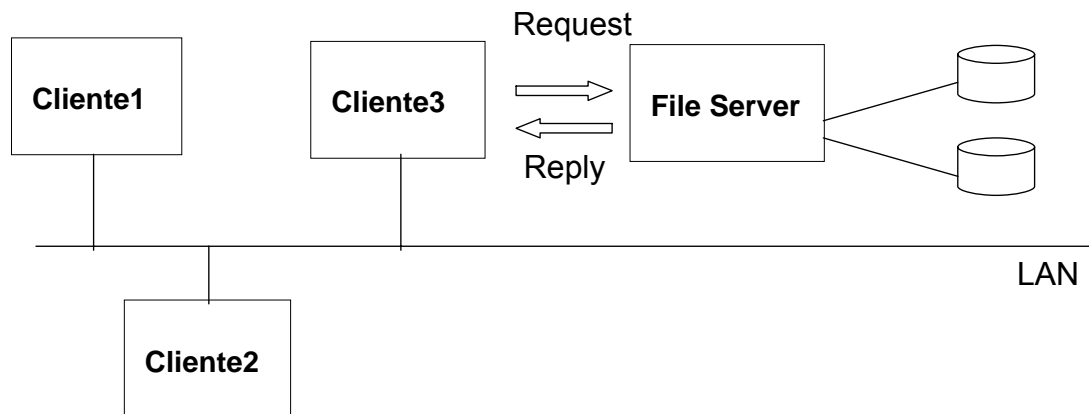
Algumas das funções permitidas pelo SOR são:

- . cópia remota de arquivos
- . emulação de terminal
- . impressão remota
- . gerência remota
- . correio eletrônico

Cada nó é totalmente independente do outro, podendo inclusive possuir sistemas operacionais diferentes. Caso a conexão entre os nós sofra qualquer problema, os sistemas podem continuar operando normalmente, apesar de alguns recursos se tornarem indisponíveis.

O melhor exemplo da utilização dos SOR são as redes locais (LAN). Nesse ambiente, cada estação pode compartilhar seus recursos com o restante da rede. Caso uma estação sofra qualquer problema, os demais componentes da rede podem continuar o processamento, apenas não dispondo de todos os recursos oferecidos.

A utilização de File Servers ( servidores de arquivos ) é uma tentativa de se prover um sistema de arquivos compartilhado e independente da estação onde está logado o usuário. Entretanto, como cada estação funciona de forma independente das demais, não podemos garantir que todas as estações apresentem a mesma hierarquia de diretórios ( e drives no Windows ).



### 5.3.2 Sistemas Operacionais Distribuídos - SOD

SOR são SOs fracamente acoplados sobre HW fracamente acoplado. Apesar da existência de um sistema de arquivos compartilhado, o usuário tem a perfeita noção da existência de diversos computadores, cada um rodando o seu próprio SO sem nenhuma coordenação das ações dos mesmos.

O passo seguinte é a implementação de um SO fortemente acoplado em um HW fracamente acoplado. O objetivo deste sistema é criar a ilusão para o usuário que todos os componentes da rede cooperam para prover um único Sistema Operacional de Tempo Compartilhado criando a idéia para os usuários, de que todos os computadores da rede cooperam para prover um único Sistema Operacional de Tempo Compartilhado, criando a idéia, para os usuários, de que toda a rede se comporta como um único SO de tempo-compartilhado. Criando o conceito de "single-system image"- Imagem Única de Sistema, onde o SD é um sistema que executa em um conjunto de máquinas interligadas por uma rede fazendo com que elas se comportem para o usuário como uma máquina virtual monoprocessada. O objetivo é fazer com que o usuário não tenha o conhecimento sobre o número de CPUs presentes no sistema. Infelizmente, nenhum dos SDs atualmente disponíveis, tais como: AMOEBA, MACH, CHORUS, implementam, ainda, todas estas características.

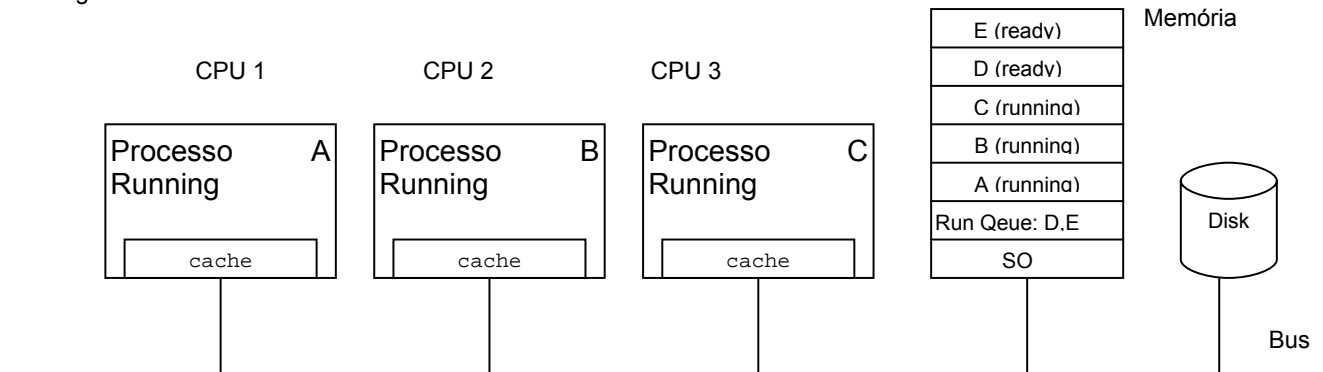
As características importantes de um SD são:

- existência de um mecanismo global para comunicação inter-processos (GLOBAL IPC Mechanism);
- existência de um esquema global de proteção de recursos compartilhados (arquivos, diretórios, impressoras, etc);
- existência de um gerenciamento de memória único e uniforme em todas as máquinas. A criação, destruição, iniciação e parada de processos não deve variar de máquina para máquina;
- existência de uma visão uniforme do sistema de arquivos. Todo arquivo deve ser visível em todas as máquinas, sujeito, é claro, as restrições de proteção.

Uma consequência lógica do fato de termos uma mesma camada system call em todas as máquinas é que normalmente nos temos também o mesmo kernel rodando em todas as CPUs do sistema. Dessa forma, o SOD tem sua tarefa de coordenação de atividades globais facilitada. Por exemplo, quando um processo deve ser iniciado, todos os kernels devem cooperar para encontrar o melhor lugar onde o processo deverá ser executado. .

### 5.3.3 Multiprocessadores de Tempo Compartilhado - MTC

A combinação de SO Fortemente Acoplados sobre Hardware Fracamente Acoplado não apresenta nenhum aspecto importante que não exista na combinação SO Fortemente Acoplado sobre Hardware Fortemente Acoplado. Estes sistemas são normalmente chamados de **Multiprocessadores de Tempo Compartilhado** (Multiprocessor Time Sharing Systems). A característica principal dessa classe de sistemas é a existência de uma única fila de processos prontos, monitorada pelo SO que ocupa uma parte da memória compartilhada pelos processadores, conforme mostra a figura abaixo:



Toda vez que um processo deixar a CPU, seja após o fim do time-slice ou execução de uma SVC, o código do SO rodando na CPU deve, após salvar o contexto do processo, entrar numa região crítica para rodar o escalonador para encontrar um novo processo a ser executado. É essencial que o scheduler execute como uma região crítica para evitar que duas CPUs escalonem o mesmo processo da fila de processos prontos. A exclusão mútua, neste caso, pode ser conseguida através do uso de primitivas de sincronização, tais como semáforos, monitores ou troca de mensagens (que seria necessária neste caso, por que?).

A criação dos sistemas Fortemente Acoplados se deveu, em grande parte, ao elevado custo de desenvolvimento de processadores cada vez mais rápidos, utilizados nas máquinas monoprocessadas. De fato, estamos chegando muito rapidamente aos limites na fabricação de chips com a tecnologia MOS (Metal Oxide Semiconductor). Nesta tecnologia, se o clock da CPU é incrementado acima do aceitável o chip passa a se comportar como um emissor de ondas (antena), o que gera uma grande quantidade de calor. De forma semelhante, se a espessura dos fios metálicos dos transistores do CI é reduzida ao extremo, passamos a conviver com os efeitos da Física Quântica, tornando o comportamento do transistor completamente imprevisível (efeito conhecido como Tunelamento de Elétrons).

Uma consequência do multiprocessamento foi o surgimento dos computadores voltados, principalmente, para processamento científico em áreas como: pesquisa aeroespacial, prospecção de petróleo, simulações, processamento de imagens, CAD/CAM, também para processamento comercial com os servidores de bancos de dados.

Os sistemas Fortemente Acoplados podem ser divididos conforme a simetria existente entre seus processadores. Os sistemas Fortemente Acoplados Assimétricos (SFAA) caracterizam-se por possuir um processador primário, responsável pelo controle dos demais (chamados de processadores secundários) e pela execução do SO. Os processadores secundários apenas processam programas dos usuários e, sempre que necessitam de um serviço do sistema, solicitam ao processador primário. Já nos sistemas Fortemente Acoplados Simétricos (Symmetric Multiprocessing - SMP), todos os processadores têm as mesmas funções, podendo executar o SO independentemente.

Nos sistemas SMP, como nenhuma CPU tem memória local e todos os programas estão armazenados na Memória Compartilhada, não importa em que processador o programa roda. Em geral, processos longos (elevado tempo de parede - "wall clock time"), que tenham sido muitas vezes escalonados, tendem a executar o mesmo período de tempo em cada CPU, o que proporciona um melhor balanceamento de carga (processamento e operações de E/S). A maioria dos fornecedores de SO, tais como Microsoft, Novel, Linux, OSF, etc, fornecem versões com suporte a SMP.

A figura abaixo apresenta as diferenças principais entre os tipos de Sistemas Operacionais examinados anteriormente.

Característica ou Função	SOR	SOD	MTC (SMP)
Se comporta como um sistema monoprocesso virtual	Não	Sim	Sim
Todas as máquinas devem rodar o mesmo SO	Não	Sim	Sim
Quantas cópias do SO existem	N	N	1
Tipo de comunicação entre os processos	Arquivos compartilhados	Troca de Mensagens	Memória Compartilhada (Shared Memory)
Existe uma única fila de processos prontos	Não	Não	Sim
Uso de protocolos e rede de comunicação compatíveis	Sim	Sim	Não há necessidade de rede
O compartilhamento de arquivos tem uma semântica bem definida	Usualmente não	Sim	Sim

## 5.4 Requisitos Principais dos Sistemas Distribuídos

### 5.4.1 Transparência

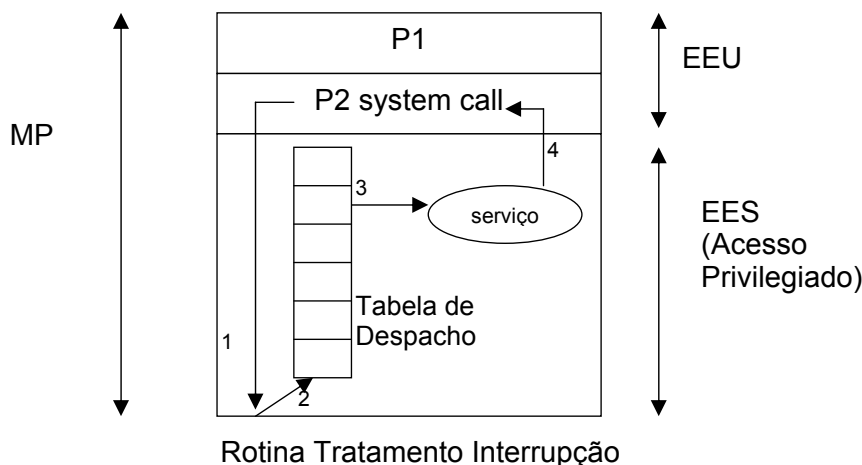
Um sistema operacional distribuído (SOD) é um SO que enxerga os usuários da mesma forma que um SO centralizado tradicional o faria, porém é executado através de múltiplas CPUs independentes. O principal conceito envolvido no projeto destes sistemas é o conceito de transparência, buscando-se fornecer aos usuários "um único ambiente virtual", e não um conjunto de recursos distribuídos e heterogêneos.

O conceito de transparência pode ser aplicado a vários aspectos relacionados aos SDs, como mostrado na figura abaixo. O nível de transparência implementado depende da qualidade do serviço de nomes disponível. Este serviço mapeia nomes lógicos em nomes físicos, para os recursos computacionais espalhados pelo ambiente distribuído.

Tipo	Significado
Transparência de Localização	Os usuários não precisam ter conhecimento sobre a localização dos recursos na rede.
Transparência de Migração (independência de localização)	Os recursos podem ser migrados de um local para outro na rede, sem precisar trocar seus nomes a cada mudança. Um esquema de nomes independente de localização implementa um mapeamento dinâmico entre nomes lógicos e nomes físicos de recursos em uma rede.
Transparência de Replicação	O sistema operacional fornece cópias adicionais de recursos, sem precisar informar este fato ao usuário. Réplicas fornecem um mecanismo eficiente para implementar funcionalidades de tolerância a falhas.
Transparência de Concorrência (controle de acesso a recursos compartilhados)	Controle necessário para manter a integridade de recursos compartilhados, usados ao mesmo tempo.
Transparência de Paralelismo	Possibilidade de uso de várias CPUs, executando processos que cooperam para solucionar um problema computacional comum a todos. Para disponibilizar esta funcionalidade, é necessário que computadores, sistemas run-time e SODs estejam aptos a aproveitar o paralelismo em potencial existente em SDs.

### 5.4.2 Flexibilidade

Os principais serviços disponibilizados por um SO incluem a execução de programas, o acesso a arquivos, o controle de entrada/saída de dados, a alocação de memória principal para processos e o controle de compartilhamento de recursos entre os diversos processos concorrentes. Tipicamente, o SO divide sua área de memória em duas partes: a parte de endereços mais baixos, reservada para o próprio SO, chamada de Espaço de Endereçamento do Sistema (EES); a parte dos endereços mais altos, reservada para os programas dos usuários, chamada de Espaço de Endereçamento do Usuário (EEU). Programas dos usuários só podem ter acesso ao Espaço de Endereçamento do Sistema (EES) via mecanismos de "chamada ao sistema" (*system call*), que é invocado sempre que um processo precisa ter acesso a algum serviço disponível.



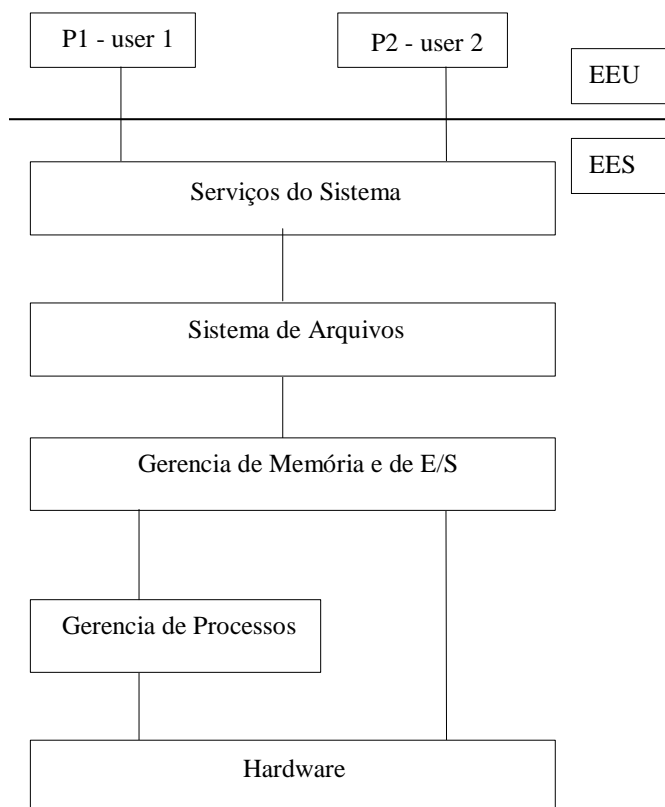
Existem, basicamente, 3 tipos de organização diferentes para os SOs, dependendo da forma como se administra o acesso ao EES: **Monolítico**, **Em Camadas** e **Micro-Núcleo** ( **Micro Kernel** ).

Nos SOs **monolíticos**, o código do SO reside no EES, que é protegido, os programas dos usuários residem no EEU. Sempre que um processo necessitar de um serviço, ele o requisita através de uma chamada ao sistema (*system call*). Para tratar essa chamada, a rotina de tratamento de interrupções do SO interpreta o pedido através de uma tabela de despacho (*Dispatch Table*). Esta tabela fornece o endereço para a rotina de serviço pedido e o SO passa o controle da execução para a mesma que, após o seu final, retorna o controle para o programa do usuário. O núcleo **Monolítico** é bastante popular e foi o primeiro modelo a ser implementado. O SO fornece todos os serviços que os programas dos usuários necessitam, incluindo sistemas de arquivos, serviços de diretórios, gerência de processos e de memória bem como o mecanismo para tratamento das chamadas ao sistema. Normalmente, os serviços de rede e serviços remotos também estão integrados ao sistema.

O acesso ao núcleo do SO a partir dos programas dos usuários pode causar problemas, particularmente em sistemas onde os processos individuais podem comunicar-se uns com os outros. Um conflito entre processos, que concorrem pelo mesmo serviço, pode causar a queda da máquina hospedeira devido a ocorrência de **deadlocks**.

Para resolver este problema, foram projetados os **SOs Em camadas**, que fornecem proteção aos endereços críticos de serviços pelo estabelecimento de uma hierarquia como a estabelecida na figura abaixo:

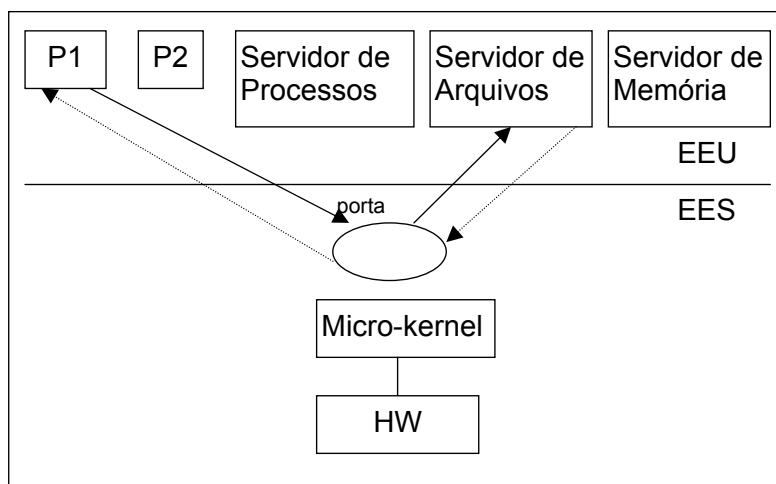




Nesta estrutura, os programas dos usuários fazem chamadas às camadas mais altas do núcleo, as quais por sua vez, passam adiante o pedido para o serviço apropriado. Um usuário não pode, desta forma, ter acesso a uma posição arbitrária de memória ou a uma interface de hardware. Exemplos deste sistema são o **Multics**, precursor do **UNIX** e o **VMS** da **DEC**. Apesar de uma perda de desempenho do SO como um todo, devido a estrutura em camadas, garante-se um processamento seguro e robusto.

A Estrutura baseada em **Micro-Núcleos** representa o que há de mais adequado para as plataformas de computação distribuída, devido à sua modularidade e rapidez de processamento. Sua principal função é fornecer uma interface para todo o hardware e gerenciar toda a comunicação entre os serviços que residem no **EEU**, e não mais no **EES** como nos outros casos.

Um processo cliente obtém um serviço pela troca de mensagens com processos servidores através de portas de comunicação (mailbox), mantidos no EES. Quando um processo cliente faz um pedido por um serviço, ele envia a mensagem para o kernel, que a armazena em uma porta específica para aquele processo cliente. A seguir, o núcleo notifica o processo servidor requisitado para que o mesmo estabeleça uma comunicação com a porta do cliente, de modo que se possa atender ao pedido.



Os **Micro-Núcleos** disponibilizam uma quantidade mínima de serviços chamados serviços essenciais, que incluem, dentre outras, a comunicação entre processos; a gerência de memória básica; gerência de processos de baixo nível e escalonamento; e entrada/saída de baixo nível, incluindo serviços para comunicação remota. Geralmente, não estão incluídas no **Micro-Núcleo** os chamados serviços não essenciais, como, por exemplo, os sistemas de arquivos, os serviços de diretórios, a gerência completa de processo, etc. Esses serviços não essenciais são implementados como processos em nível de EEU. Tal enfoque conduz ao projeto de um núcleo menor, mais confiável, que fornece uma maior facilidade para a adição, alteração e

teste de novos serviços. Além disso, a instalação, implementação e a depuração de novos serviços não essenciais é muito mais simples nas arquiteturas em **Micro-Núcleos**, pois, para adicionar ou modificar um desses serviços não é necessário que se pare o sistema, ou que se reinicie o núcleo, como ocorre nas arquiteturas monolíticas.

Esta é a grande **flexibilidade** apresentada por estes sistemas. Além do mais, usuários insatisfeitos com serviços fornecidos podem implementar seus próprios serviços da maneira que considerarem mais apropriada.

A única vantagem potencial dos sistemas monolíticos é em relação a performance, já que a execução de uma *system call* (que envolve o desvio para a área do sistema operacional e execução da rotina de serviço no *modo kernel*) é mais rápida que enviar mensagens para servidores remotos.

### 5.4.3 Tolerância a Falhas

Um sistema tolerante a falhas tem a capacidade de continuar operacional mesmo que ocorra uma falha em um de seus subsistemas, apesar de poder ocorrer uma degradação no desempenho e no seu nível de funcionalidade. Exemplos de falhas que poderiam ser toleradas incluem as falhas de comunicação, as falhas em discos e outras. tolerância a falhas envolve dois aspectos principais:

- (i) **Confiabilidade** - é representada pelo fato do sistema não se corromper ou não danificar seus dados durante a execução de aplicações;
- (ii) **Disponibilidade** - é representada pela capacidade de manter o sistema sempre operacional, sempre que for necessário.

Um sistema pode ser altamente confiável, no sentido de que ele não danifica seus dados, porém pode ter, ao mesmo tempo, uma baixa disponibilidade, por estar inoperante na maior parte do tempo.

Falhas podem ser classificadas em dois tipos: falhas previsíveis e falhas não previsíveis. Uma falha previsível, por exemplo, é a perda de pacotes em protocolos de transporte não-confiáveis. Uma falha não previsível pode ser um erro de memória em uma interface de rede, que pode gerar tempestades de difusão (broadcast storms), causando o congestionamento da rede.

Duas técnicas principais usadas para implementar a tolerância a falhas previsíveis são:

- (i) **Técnicas de redundância;**
- (ii) **Uso de transações atômicas.**

### 5.4.4 Escalabilidade

Escalabilidade é a capacidade que um sistema apresenta de poder adaptar-se facilmente a uma carga crescente de serviços. A escalabilidade é uma propriedade relativa, pois um sistema escalável deve reagir suavemente a cargas crescentes de processamento. Um sistema escalável deve apresentar o potencial de poder crescer sem problemas. Em um SD, esta possibilidade de crescimento suave é de especial importância, porque a expansão da rede, seja pela adição de novas estações, seja pela inter-conexão de segmentos dispersos, é sempre necessária. Assim, o projeto de um sistema escalável deve prever o suporte altas cargas de processamento, acomodando o crescimento da comunidade de usuários e permitindo a adição facilitada de novos recursos.

Embora, hoje em dia, pouco seja conhecido sobre SD de larga escala, alguns princípios básicos são essenciais no projeto deste tipo de sistema, tais como: evitar o uso de componentes centralizados, tabelas centralizadas e algoritmos centralizados. As razões para evitar tais elementos encontram-se dispostos na figura abaixo.

<b>Componentes Centralizados</b>	Um único servidor para um número grande de processos clientes, mesmo que o servidor tenha poder de processamento e capacidade de armazenamento, não parece ser uma boa idéia, pois o tráfego da rede representaria um grande problema. Além disso, as falhas não seriam toleradas de maneira adequada.
<b>Tabelas Centralizadas</b>	Se os dados de uma aplicação ficarem localizados em uma única Base de Dados, a saturação de pedidos e a vulnerabilidade a falhas seriam preocupantes. Além de elevados custos de comunicação.
<b>Algoritmos Centralizados</b>	Algoritmos para fazer o roteamento de informações baseado em informações completas sobre o estado do sistema podem levar a saturação do nó de processamento de roteamento, vulnerabilidade a falhas e custos elevados de comunicação. Dessa forma, o ideal é que os algoritmos distribuídos tenham as seguintes características: <ul style="list-style-type: none"><li>(I) Nenhuma máquina deve possuir informação completa sobre o estado do sistema;</li><li>(II) Os computadores só devem tomar decisões com base em informações disponíveis localmente;</li><li>(III) Falhas em uma estação não devem prejudicar os algoritmos, a ponto de impedir de rodá-los;</li><li>(IV) Os algoritmos não devem assumir a existência de um relógio global, que sincronize as operações de todas as máquinas</li></ul>

Obs.: Em uma rede de computadores é impossível sincronizarmos todos os relógios de forma precisa, o máximo que se consegue é a sincronização ( com erro da ordem de milissegundos ) de relógios dentro de uma LAN com uso de algum algoritmo de ***Distributed Clock Sincronization (DCS)***. E quanto maior o tamanho da rede maior o erro de sincronização.

## 6 Comunicação em Sistemas Distribuídos

### 6.1 Introdução

A principal diferença entre um SD e um sistema monoprocesso é a comunicação entre processos (IPC). Em máquinas monoprocessadas a maioria das primitivas de IPC assume a existência de uma única memória.

Um exemplo típico é o problema produtor-consumidor, no qual um processo (produtor) escreve informações em um buffer compartilhado e outro processo (consumidor) lê estas informações do buffer. Mesmo a mais básica forma de sincronização, conhecida como Semáforo, requer que a estrutura de dados representativa do Semáforo seja compartilhada.

Em um SD não existe memória compartilhada e portanto os mecanismos de IPC, Semáforos e Monitores, não podem ser utilizados, somente troca de mensagens. Dessa forma, novos mecanismos de IPC precisaram ser criados, sendo talvez o mais importante o mecanismo de PRC (Remote Procedure Call).

### 6.2 Padrões e Sistemas Abertos

Até recentemente, a indústria de computadores foi guiada mais pela tecnologia que pelas necessidades do mercado. Muitas empresas fornecedoras de sistemas de computadores consideravam seu diferencial de competitividade baseados em tecnologias proprietárias. A migração de aplicações entre plataformas, neste contexto, implicava em custos extras para treinar pessoal e redesenvolver software. Esta política de sistemas proprietários permitia que os fabricantes exagerassem em seus preços, gerando lucros fabulosos.

Felizmente, com o advento do microprocessador os fabricantes de computadores viram seu poder começar a ser contestado.

As contestações ocorreram a partir de grupos de interesse (usuários, desenvolvedores de software, fabricantes de hardware), que passaram a exigir padrões para os produtos de Tecnologia de Informação. Dessa forma, os padrões foram definidos para permitir que se incluíssem inovações no projeto de sistemas, sem que se destruísse a compatibilidade básica entre computadores de diferentes fabricantes. Os sistemas padronizados são chamados de Sistemas Abertos (SA). Podemos definir os AS como:

#### **Sistema Abertos**

Conjunto de padrões internacionais e funcionais para a Tecnologia de Informação, que especificam interfaces, serviços e suporte a formatos que possam atender aos requisitos de interoperabilidade e portabilidade de aplicações, dados e pessoal.

Os SA podem ser vistos também como uma metodologia para a integração de tecnologias divergentes, permitindo que se crie um ambiente flexível para resolver os problemas de negócios de uma organização, através do uso de software e hardware abertos.

### 6.3 Modelo de Referência OSI/ISO

A solução para garantir a interoperabilidade está no estabelecimento de padrões que sejam independentes de fabricantes, possibilitando o desenvolvimento de componentes de hardware e software que implementam Redes de Arquitetura de Referência definidas. A definição de uma arquitetura aberta - conhecida como RM/OSI (Reference Model/Open Systems Interconnection) é um trabalho que vem sendo desenvolvido pela ISO (International Standards Organization).

O RM/OSI foi desenvolvido com o objetivo de facilitar a elaboração de aplicações distribuídas que pudessem ser executadas em equipamentos de diferentes fornecedores, permitindo a intercomunicação de forma transparente. O principal enfoque do modelo é o conceito de camada, onde cada camada executa uma função específica, fornecendo um serviço de qualidade para as camadas superiores, através de adição de funcionalidades das camadas inferiores.

O conceito de camadas habilita a implementação de mecanismos de multiplexação, permitindo que vários fluxos de dados lógicos sejam agrupados e despachados através de um único fluxo físico. O modelo habilita também a implementação de mecanismos de splitting, permitindo que um grande volume de dados seja subdividido para poder, a seguir, ser despachado através de várias conexões físicas de capacidade limitada.

Dois modos de transferência de dados são suportados pelo modelo OSI: interações orientadas a conexões e interações sem conexão. As interações orientadas à conexão permitem que uma relação temporária seja estabelecida entre dois computadores, e que uma rota entre eles seja traçada previamente, de modo que os custos para a determinação e o estabelecimento dessa rota ocorram uma única vez, permitindo que todos os dados trafegados entre os dois computadores possam fluir rapidamente. Por outro lado, as interações sem conexão não estabelecem esta relação, e cada item de dados trafegados representa uma transferência única, roteadas e despachadas de forma independente das demais. O telefone é um exemplo de comunicação orientada a conexão enquanto o envio de uma carta à uma caixa postal é um exemplo de uma interação sem conexão.

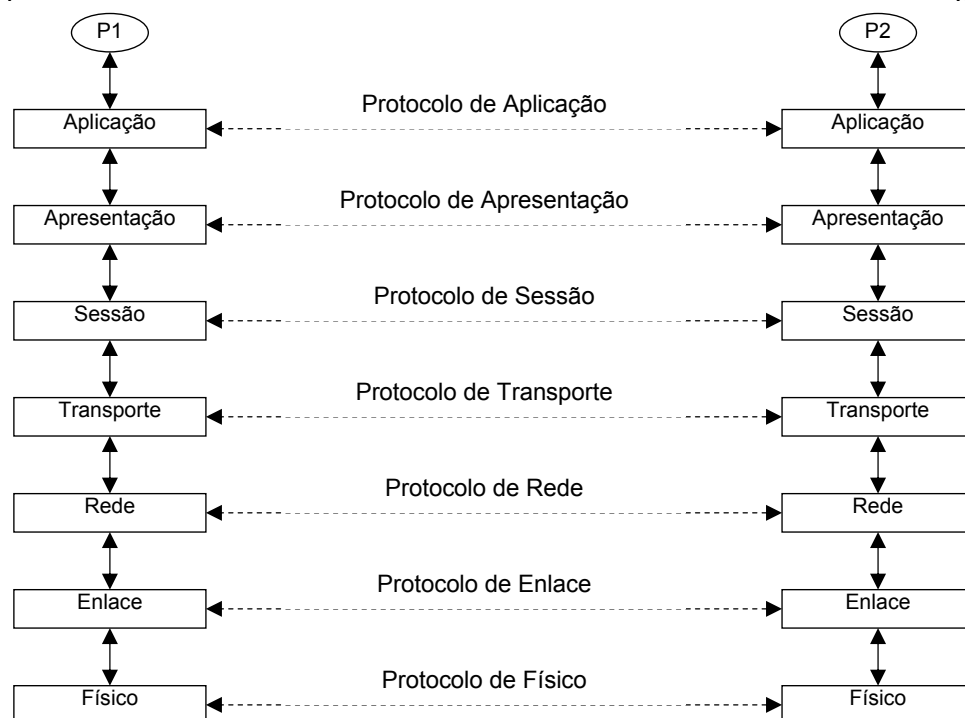
Outro conceito importante do RM/OSI diz respeito à qualidade de serviços. Os usuários OSI podem requisitar a qualidade de transmissão que desejam, em termos de limites para atrasos aceitáveis, e níveis de confiabilidade, prioridade e segurança desejados. Dependendo do nível de qualidade de serviço desejado, as entidades das camadas OSI selecionam os protocolos e as opções de serviços apropriados.

### 6.3.1 Cada uma OSI

O padrão RM/OSI possui sete camadas, numeradas em ordem decrescente de baixo para cima. A camada mais em ordem decrescente, de baixo para cima. A camada mais alta é a responsável pelos serviços que possibilitam aos programas de aplicação o acesso aos recursos de interconexão do sistema, enquanto as camadas mais baixas tratam da movimentação de bits entre a memória e o meio físico de transmissão utilizado.

Máquina 1

Máquina 2



#### 6.3.1.1 Camada Física

Esta camada trata as características físicas e elétricas das conexões que implementam a Rede. Neste nível, estão o cabeamento (através do qual o sinal se move), os repetidores (utilizados para se estender a distância máxima entre dois nós), concentradores e hubs eventualmente utilizados. Esta camada descreve como os bits devem ser enviados e recebidos, mas não provê o reconhecimento de dados. Um exemplo de um componente da camada física é a porta serial padrão IEEE RS-232C.

Alguns protocolos que têm sido usados são:

- (i) CSMA/CD (Carrier Sensing with Multiple Access/with Collision Detection) - redes locais padrão Ethernet (ISO 8802.3);
- (ii) Token-Ring - redes locais padrão token-ring (ISO 8802.5);
- (iii) Fibra Ótica - redes locais FDDI (ISO 93/4.)

### **6.3.1.2 Camada de Enlace de Dados**

Nesta camada, faz-se o controle do fluxo de dados que trafega entre os sistemas interconectados. Os vários protocolos para a camada de enlace de dados levam em consideração a diferença na qualidade da transmissão do meio físico utilizado, caracterizando os ambientes de redes locais e de redes de longa distância. A camada de enlace de dados baseia-se nos serviços fornecidos pela camada física, disponibilizando os meios que irão estabelecer, manter e liberar as conexões de enlace de dados. O fluxo de bits é arbitrário, sendo ordenado em quadros (frames) de tamanho variável. Além do controle de fluxo, são incluídas a detecção e correção de erros, o início e término da conexão e o controle da sequência de endereçamento. Em geral, estes protocolos são implementados em hardware e assistidos pelo software de comunicação.

Alguns protocolos que têm sido usados são:

- (i) HDLC (High-Level Data-Link Control) - redes de longa distância ponto a ponto;
- (ii) SDLC (Synchronous Data-Link Control) - redes de longa distância na Arquitetura IBM SNA (IBM Systems Network Architecture).

### **6.3.1.3 Camada de Rede**

Esta camada é responsável pela determinação física do roteamento da informação através da Rede. Ela contém as funções de endereçamento, de segmentação de grandes blocos de dados em pacotes, de determinação da rota a ser seguida para atingir o destinatário (roteamento), de despacho de mensagens e de controle do fluxo resultante.

O problema de roteamento torna-se complicado pelo fato de que a menor rota em distância nem sempre é a melhor rota possível. O importante é evitar os atrasos nas transmissões gerados pelo excesso de tráfego de mensagens. Alguns algoritmos tentam se adaptar dinamicamente com a variação da carga, enquanto outros se baseiam em estatísticas de longo prazo. O software para essa camada normalmente reside em dispositivos como roteadores ou pontes, que são responsáveis pela implementação dos mecanismos de roteamento.

Dois protocolos de rede são largamente utilizados, um orientado a conexão, chamado X.25, e outro sem conexão, chamado IP (Internet Protocol). O protocolo X.25 é utilizado por operadores de redes públicas telefônicas (no Brasil a Embratel opera a RNP - Rede Nacional de Pacotes, usada na telefonia). O protocolo IP é o padrão recomendado pelo DOD (Departamento de Defesa dos EUA) para o nível de rede.

### **6.3.1.4 Camada de Transporte**

Esta camada é responsável pela garantia da integridade e confiabilidade em conexões ponto a ponto (peer to peer). A camada de transporte e aquelas abaixo dela fornecem serviços de interconexão de redes completos, removendo das camadas superiores as preocupações com a entrega dos dados, sem perda.

As principais funções dessa camada são o mapeamento de endereços da camada de transporte para endereços da camada de rede, multiplexação, estabelecimento e liberação das conexões, detecção e recuperação de erros, controle de fluxo e supervisionamento da qualidade dos serviços.

O estabelecimento de conexões confiáveis pode ser feito tanto sobre X.25 quanto sobre protocolo IP. No caso do X.25 todos os pacotes chegarão ao seu destino (se chegarem) na sequência correta em que foram enviados (Por quê?), ao passo que no caso do IP é possível que um pacote trafegue por uma rota diferente e chegue antes do pacote previamente enviado. É função da camada de transporte ordenar os pacotes no destinatário para garantir a integridade da conexão a camada de transporte.

### 6.3.1.5 Camada de Sessão

Esta camada permite que duas aplicações se comuniquem pela rede, disponibilizando funções para reconhecimento de nomes, segurança, logging e administração. Ela sincroniza o diálogo entre duas aplicações para garantir que elas possam se comunicar na sequência correta. No início de uma sessão, por exemplo, os processos precisam estabelecer o modo de conversação, half-duplex ou full-duplex. Em uma sessão podem ser trocadas várias mensagens contendo requisições de vários tipos de serviço, como o acesso a dados, a execução de procedimentos de segurança, execução de procedimentos de administração e execução de programas de aplicação.

### 6.3.1.6 Camada de Apresentação

Esta camada assegura que cada um dos pares de uma conexão lógica esteja utilizando uma representação comum para a informação que está sendo transferida. Um exemplo de função desta camada é a conversão dos dados codificados em EBCDIC para ASCII, no processo de transferência de um arquivo no formato texto de um mainframe IBM para um microcomputador.

### 6.3.1.7 Camada de Aplicação

Esta é a camada dos processo de aplicação, fornecendo serviços como a transferência de arquivos, ou troca de mensagens de correio eletrônico. A camada de aplicação é empregada como uma janela para processos de usuários, possibilitando que os mesmos tenham acesso aos serviços e funcionalidades do modelo OSI. Os protocolos mais conhecidos neste nível são o correio eletrônico X.400, padrão MHS (Message Handling System), e o serviço de diretórios X.500, acesso a terminais remotos virtuais padrão VT, acesso remoto a dados, padrão RDA (Remote Database Access).

## 6.3.2 Camadas: Visão Funcional

As camadas do modelo OSI podem ser agrupadas em 3 grandes grupos, a saber:

(i) **Infra-estrutura da Rede:**

Camadas 1 a 4 fornecem toda a infra-estrutura para a rede, garantindo uma transferência de dados confiável. Neste nível são usados equipamentos como pontes ou roteadores para formar subredes de uma grande rede. A principal preocupação nesse nível é fornecer uma interface comum para os serviços de aplicações no nível acima.

(ii) **Serviço de Aplicações:**

As camadas 5 a 7 são voltadas para as questões de interoperabilidade de aplicações. Os serviços de aplicações disponibilizam os serviços que utilizam a infra-estrutura da rede como por exemplo, serviços de transferência de arquivos ou serviços de correio eletrônico. Os serviços de aplicações OSI disponíveis são:

- transferência, acesso e gerenciamento de arquivos (padrão FTAM – File Transfer and Access Management);
- tratamento de mensagens (padrão MHS – Message Handling System) X-400;
- acesso a terminais virtuais (padrão VT – Virtual Terminal);
- acesso remoto a bancos de dados (padrão RDA – Remote Database Access);
- processamento de transações distribuídas (padrão DTP – Distributed Transaction Processing);
- operações remotas (padrão JTM – Job Transfer and Manipulation)

(iii) **Formato de Aplicações:**

Serviços comuns de alto nível tais como EDI (Eletronic Data Interchange) e serviços de transferência de documentos por exemplo.

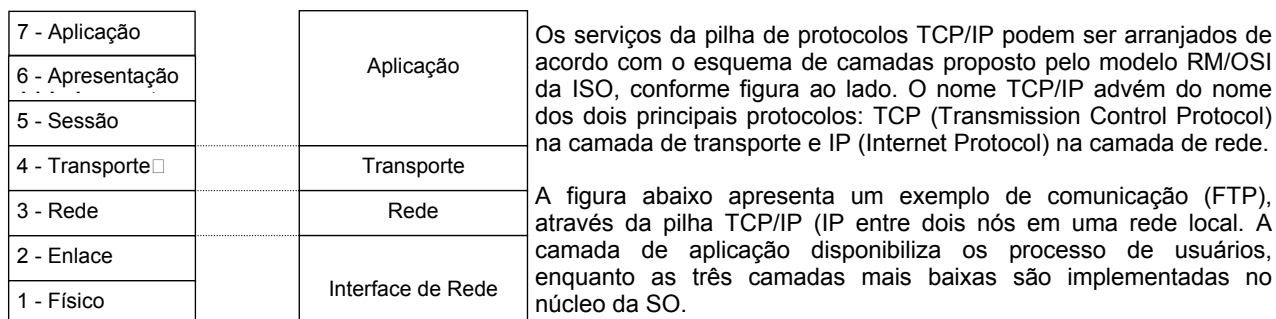
## 6.4 O Protocolo TCP/IP

O protocolo TCP/IP foi desenvolvido pelo DOD (Departament of Defence, EUA) para resolver o problema da interconexão de computadores heterogêneos, tendo evoluído a partir de trabalhos iniciados no MIT (Massachusetts Institute of Technology). A Agência de Pesquisas Avançadas de Defesa (DARPA – Defense Advanced Rescharch), ligada ao DOD, iniciou no começo dos anos 70 a busca pela interligação de diferentes computadores, possibilitando a comunicação entre laboratórios, universidades e instituições governamentais.

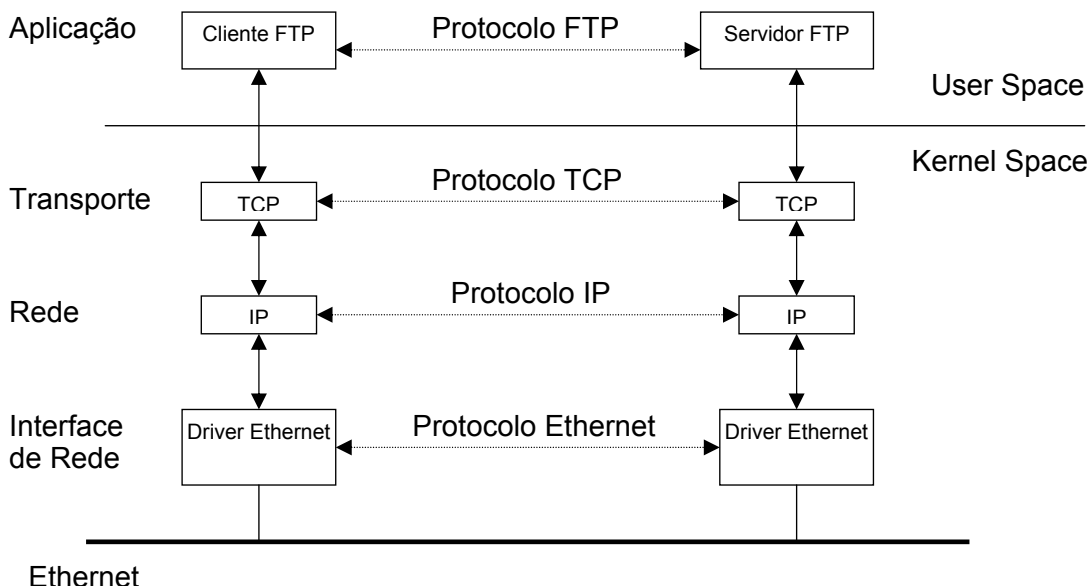
Em 1980, a DARPA instalou os principais módulos TCP/IP em computadores de sua rede, a ARPANET (atual Internet). A partir de janeiro de 1983, ela passou a exigir que todos os computadores ligados a sua rede utilizassem o protocolo TCP/IP.

A pilha de protocolos TCP/IP representa a mais usada forma de comunicação entre computadores remotos, disponível atualmente, fornecendo um sistema aberto de fato, pois suas especificações e muitas de suas implementações estão disponíveis publicamente. Atualmente o TCP/IP é adotado como padrão pela IBM e pela Novell.

### 6.4.1 Pilha de Protocolos do TCP/IP



A camada de interface de rede cuida dos detalhes relativos ao meio de comunicação, enquanto a camada de aplicação trata de serviços específicos tais como: FTP, Telnet, SMTP, etc.



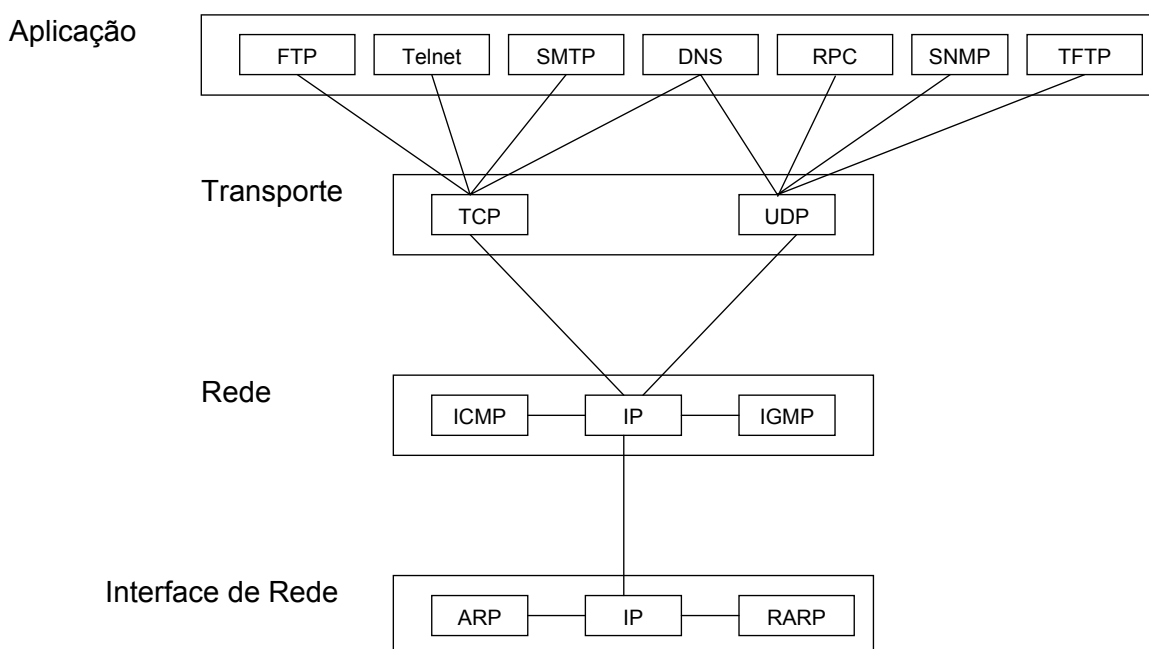


### 6.4.1.1 Camada de Aplicação TCP/IP

A camada de aplicação descreve as tecnologias usadas para fornecer serviços especializados para os usuários finais. Existem diversos serviços TCP/IP disponibilizados nessa camada. Os principais são:

- (i)SMTP (Simple Mail Transfer Protocol) – Fornece um serviço de correio eletrônico global que suporte diferentes sistemas de correio eletrônico locais;
- (ii)DNS (Domain Name Service) – Fornece serviços de nomes e diretórios. É um serviço distribuído complexo que executa a tradução de nomes simbólicos na rede para endereços IP, e vice-versa.
- (iii)FTP (File Transfer Protocol) – Fornece troca de arquivos entre computadores.
- (iv)Telnet (Telecommunications Network) – Fornece serviços de terminal virtual, possibilitando o acesso interativo através de terminais.
- (v)SNMP (Simple Network Management Protocol) – Fornece serviços sofisticados de gerência de redes de computadores.

As especificações para os protocolos da camada de aplicação da pilha TCP/IP estão agrupados em uma única camada, enquanto tais especificações são distribuídas através de três camadas no modelo OSI (sessão, apresentação e aplicação).



### 6.4.1.2 Camada de Transporte TCP/IP

A camada de transporte descreve as tecnologias para o estabelecimento de conexões fim-a-fim, e suporta o fluxo de dados entre dois computadores hospedeiros, garantindo a qualidade do serviço para a camada de aplicação. TCP e UDP são os protocolos mais importantes na camada de transporte, e ambos dependem dos serviços fornecidos pelo protocolo IP na camada de rede.

## TCP – Transmission Control Protocol

Este protocolo fornece um fluxo de dados confiável e trata de questões como a retransmissão de pacotes perdidos, eliminação de pacotes duplicados, avisos de “acknowledgments” para pacotes recebidos com sucesso. O fato do fluxo de dados ser confiável libera a camada de aplicação de tratar todos os detalhes para garantir a confiabilidade. É interessante observar que o TCP fornece um serviço de transporte confiável, apesar de o serviço na camada de rede (IP) ser não confiável. Alguns protocolos de aplicação utilizam as interações orientadas a conexão do TCP. Estes protocolos são implementados no espaço de endereçamento de usuários e incluem, dentre outros, o Telnet, o FTP e o SMTP. O TCP divide o fluxo de dados em unidades discretas, chamadas segmentos. Pacotes individuais podem ser usados para transmitir esses segmentos. Os principais mecanismos implementados no nível de transporte através do TCP são:

- mecanismos de gerência das conexões de transporte;
- mecanismos de reconhecimento positivo de retransmissão PAR (Positive Acknowledgment Retransmission), para permitir a recuperação de dados danificados nas camadas inferiores;
- mecanismos de controle de fluxo de dados, para permitir o controle da quantidade de dados enviados e recebidos;
- mecanismos de multiplexação, para permitir que múltiplos processos possam usar simultaneamente o TCP;
- mecanismo de sincronização (three-way handshake), para garantir a integridade dos dados trafegados.

## UDP– User Datagram Protocol

Este protocolo, por sua vez, fornece um serviço mais simples para a camada de aplicação. Ele envia pacotes, chamados datagramas, através dos nós de uma rede, porém sem garantir que os mesmos cheguem ao seu destino. Nesse caso, qualquer garantia de confiabilidade deve ser disponibilizada pela própria camada de aplicação. Diferentemente do TCP, o UDP fornece uma interação sem conexão, isto é, não-confiável. Os protocolos da aplicação que utilizam o UDP são o DNS, SNMP, TFTP e RPC. Os protocolos DNS e RPC podem também utilizar o TCP como suporte.

De maneira geral, servidores que implementam serviços de transporte baseados no TCP são do tipo iterativos. Como o TCP, o UDP emprega campos especiais para identificar os processos emissores e os receptores para cada transação (esses campos são chamados “portas”).

### 6.4.1.3 Camada de Rede TCP/IP

A camada de rede (também chamada de Internet) descreve as tecnologias para interligação de redes, administrando o fluxo de pacotes através das mesmas. Os serviços de transferência de dados fornecidos pela camada de interface da rede são muito limitados e, portanto, a camada de rede fornece os procedimentos necessários para melhorar a qualidade dos mesmos. O protocolo IP (Internet Protocol) é o protocolo mais importante da pilha TCP/IP nessa camada. Os outros Protocolos importantes dessa camada são o ICMP e o IGMP. As características principais são:

- ✓ **IP (Internet Protocol)** – Protocolo na camada de rede, usado tanto pelo TCP quanto pelo UDP. Cada parte dos pacotes TCP e UDP que trafegam na rede passa pela camada de rede em ambas as máquinas, origem e destino (sistemas finais), passando também por sistemas intermediários;
- ✓ **ICMP (Internet Control Message Protocol)** – Protocolo subordinado ao IP, utilizado para intercambiar mensagens de erro e outras informações com a camada de rede em outro computador ou em um roteador;
- ✓ **IGMP (Internet Group Management)** – É usado em operações multicasting, para enviar datagramas UDP através de múltiplos nós na rede.

O protocolo IP não fornece um serviço confiável. Ele permite a interconexão de uma ou mais redes para formar inter-redes (internets). Este serviço é fornecido para as camadas superiores, escondendo das mesmas as particularidades das tecnologias empregadas na camada de interface de rede.

As redes impõem limites ao tamanho de seus pacotes, devido a fatores como limitações de hardware e/ou de software, limitações do protocolo em uso, restrições de especificações padrão, dentre outros. Esse limite é chamado MTU (Maximum Transfer Unit) e representa uma característica da camada de interface da rede. Se o protocolo IP desejar enviar um pacote pela rede, e esse pacote for maior que o MTU da camada de interface de rede, cabe ao IP executar a fragmentação do pacote, quebrando-o em pequenas partes (chamadas fragmentos), de modo que cada fragmento seja menor que o MTU. A tabela abaixo lista valores típicos de MTU:

Tipo de Rede	MTU
TokenRing (16 Mbps) IBM	17914
TokenRing (4 Mbps) IEEE 802.5	4464
FDDI	4352
Ethernet Padrão	1500
IEEE 802.3	1492
X.25	576

É exatamente o IP que acomoda quaisquer restrições de tamanho a pacotes em subredes que compõem os sistemas intermediários, pela disponibilização de um mecanismo para fragmentação/montagem de pacotes. O envio de pacotes é feito da seguinte forma:

Quando um computador que representa um sistema final envia um pacote pela rede, o protocolo IP verifica onde se encontra o sistema final destinatário. No caso dele se encontrar no mesmo segmento de rede (do sistema final emissor), o módulo de SW que implementa o IP no sistema final envia esse pacote diretamente para o destinatário. Se o destinatário estiver em um segmento de rede remoto, então o protocolo envia o pacote diretamente para o roteador. O roteador, por sua vez, envia o pacote para o sistema final destinatário, ou então, o enviará para outro roteador na próxima subrede.

### Protocolos de Roteamento

Em uma rede local, são normalmente empregados algoritmos de roteamento estático, que se baseiam em informações mantidas em uma tabela de roteamento. As entradas em uma tabela de roteamento, no kernel do SO na estação hospedeira, são criadas por default quando se instalam e configuram as placas de interface de rede em cada estação. Roteadores estáticos são interessantes para redes pequenas, ou para redes com um único ponto de conexão em outras camadas. Quando for necessário que roteadores falem com roteadores adjacentes utiliza-se o roteamento dinâmico, que se utiliza de um daemon para a atualização das tabelas de roteamento no núcleo do SO da estação hospedeira, a partir de informações recebidas de roteadores vizinhos.

### Endereços IP

Cada nó em uma rede deve ter um endereço Internet (ou endereço IP) único, e os nós devem ser ajustados para as diversas configurações de redes possíveis. As classes de endereço IP estão divididas da seguinte maneira:

Classe	Padrão das Bits Iniciais	Escopo	Característica
A	0XXXX	0.0.0.0 – 127.255.255.255	Muitos computadores e poucas redes
B	10XXX	128.0.0.0 – 191.255.255.255	Número médio de computadores e redes
C	110XX	192.0.0.0 – 223.255.255.255	Poucos computadores e muitas redes
D	1110X	224.0.0.0 – 239.255.255.255	Multicast para grupos definidos
E	11110	240.0.0.0 – 247.255.255.255	Reservado para uso futuro

Um endereço IP completo é composto por 32 bits contendo a identificação do nó e da rede à qual o nó pertence. Esses endereços são representados por quatro valores decimais separados por um ponto, com cada valor tendo um byte reservado para representar o endereço (notação dotted decimal). Existem três tipos de endereços IP: unicast (identifica um único nó), broadcast (identifica todos os nós de uma rede) e multicast (identifica um conjunto predefinido de nós em uma rede).

O órgão InterNIC (Internet Network Information Center) é encarregado de atribuir um endereço IP único para cada rede conectada à rede mundial Internet. A atribuição de endereços IP para os nós individuais em uma rede é de responsabilidade do próprio administrador da rede.

#### 6.4.1.4 Camada de Interface da Rede TCP/IP

A camada de interface de rede é responsável pela transmissão de dados através de uma facilidade física comumente chamada meio de comunicação (tecnologias de redes locais: Ethernet e Token-Ring). Os principais protocolos nessa camada são o ARP (Address Resolution Protocol), RARP (Reverse ARP), SLIP (Serial Line IP) e o PPP (Point to Point Protocol).

##### ARP – Address Resolution Protocol

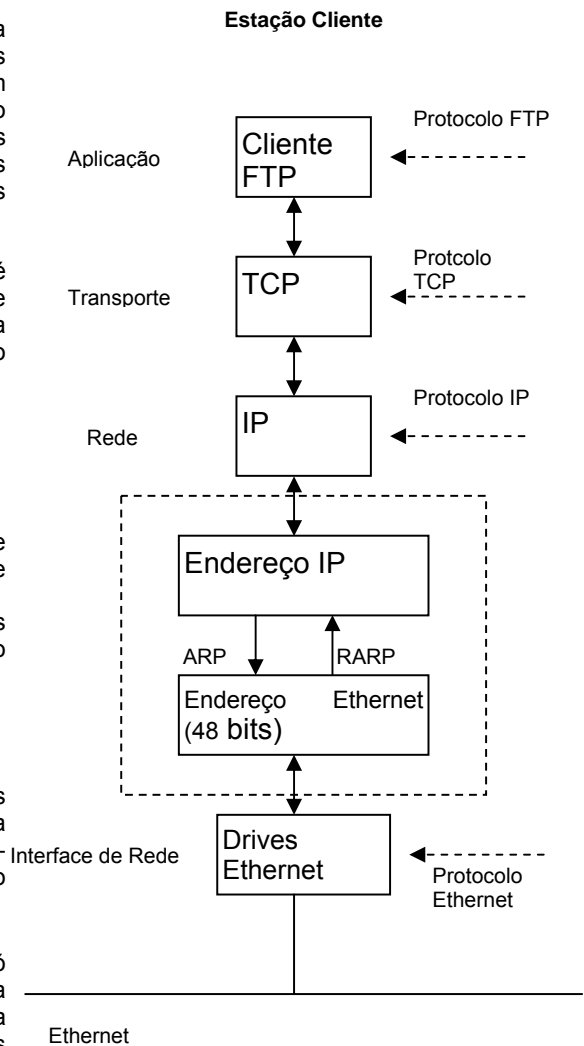
Um problema que surge com o uso de endereços IP é que eles só fazem sentido se usados na própria pilha TCP/IP. Tecnologias da camada de Interface da Rede, como por exemplo, Ethernet ou Token-Ring, possuem seus próprios esquemas de endereçamento (geralmente com endereços físicos de 48 bits).

Quando um frame Ethernet, por exemplo, é enviado de um nó para outro em uma rede local, é o endereço Ethernet que determina para qual nó exatamente esse frame é endereçado. O driver da camada de Interface da Rede não reconhece endereços IP contidos em pacotes IP. Deste modo, devem existir mecanismos de tradução que mapeiem essas duas formas de endereçamento: do endereços IP com 32 bits, para qualquer tipo de endereço físico usado na camada de Interface de Rede, e vice-versa. O ARP fornece a tradução dinâmica de endereços IP para endereços físicos, sendo disponibilizado em quase todas as implementações da pilha TCP/IP. Esta tradução é considerada dinâmica, porque acontece automaticamente e de maneira transparente para o usuário final. O ARP só pode localizar nós que estejam na mesma rede ou sub-rede do remetente.

##### RARP – Reverse ARP

É usado em estações sem disco (diskless), terminais X e network computers (NC), porém o mesmo precisa ser configurado manualmente pelo administrador da rede. Quando uma estação com disco é ativada, ela obtém seu endereço IP a partir de um arquivo de configuração residente no HD da máquina. Em estações diskless, o endereço IP é obtido de um outro computador através do protocolo RARP.

Cada estação em uma rede possui um endereço de HW único, atribuído pelo fabricante da placa de interface de rede. O princípio do RARP é permitir que uma estação diskless possa obter um endereço físico a partir da interface da rede e que possa enviá-lo, através de um frame RARP especial (um frame de difusão – broadcast), perguntando a algum servidor (ou a alguma outra estação) qual o seu endereço IP correspondente. A resposta é devolvida através de um frame especial de resposta (RARP reply).



## **SLIP – Serial Line IP**

Desenvolvido e implementado em 1984 por Reck Adams em uma versão do SO Unix 4.2 BSD. Constitui uma forma simples de encapsulamento de pacotes IP através de interfaces seriais, tendo se tornado popular por conectar microcomputadores domésticos à Internet, pelo uso da porta serial RS-232, através de modems de alta velocidade. Devido a sua simplicidade, o SLIP apresenta diversas deficiências que devem ser citadas:

- Cada sistema final deve conhecer com antecedência o endereço IP de seu par. Não existe um método que permita a um sistema final informar ao seu par qual o seu endereço IP;
- Se uma linha serial estiver sendo usada pelo SLIP, ela não poderá ser compartilhada ao mesmo tempo por outros protocolos;
- Não são fornecidos mecanismos de checksum. As camadas mais altas deverão detectar e corrigir este problema. Já existem modems modernos que fazem a detecção e a correção de erros.

## **PPP – Point to Point Protocol**

O PPP é um protocolo que corrige todas as deficiências do SLIP, consistindo de três componentes:

- (i) Um método para encapsular pacotes IP através de linhas seriais, suportando conexões assíncronas com oito bits e sem paridade (interface serial) quanto conexões síncronas orientadas a bits;
- (ii) Um protocolo de suporte LCP (Link Control Protocol), para estabelecer, configurar e testar as conexões da camada da interface da rede, permitindo que os sistemas finais possam negociar várias opções;
- (iii) Protocolos NCP (Network Control Protocol) específicos para diferentes protocolos de camada de rede.

O PPP apresenta as seguintes vantagens em relação ao SLIP:

- Suporte simultâneo a múltiplos protocolos em uma conexão serial, e não somente a pacotes IP;
- Mecanismo CRC (Cyclic Redundancy Chec) para cada frame;
- Negociação dinâmica do endereço IP em cada sistema final (através do protocolo NCP para IP);
- Compressão de cabeçalhos em pacotes IP e TCP.

O preço pago por estas vantagens é representado por 3 bytes adicionais por frame, alguns frames de negociação para o estabelecimento da conexão e uma implementação mais complexa.

## 7 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor (C/S) é um modelo conceitual, adotado para disciplinar e orientar o projeto e a implementação de aplicações que estão funcionalmente separadas em processos distintos, que por sua vez podem ou não ser distribuídos em plataformas distintas. A essas aplicações damos o nome de Aplicações Distribuídas (AD). As ADs são formadas a partir de diversos processos que interoperam para resolver um problema computacional comum. Na arquitetura C/S, uma aplicação distribuída é conceitualmente modelada para ser composta por somente dois processo cooperantes: o processo cliente e o processo servidor. Dessa forma, disciplina-se a construção de ADs que podem ser facilmente implementadas. Observe que uma AD, composta de várias partes cooperantes, pode causar muito facilmente um conflito (deadlock). Em uma AD C/S, a prevenção de deadlocks é trivial, pois a comunicação é exclusiva e síncrona entre somente um processo cliente e somente um processo servidor de cada vez.

No modelo C/S, a origem da comunicação entre os processos cooperantes define se um processo é cliente ou servidor. Em geral, o processo que inicia a comunicação é chamado cliente. Os usuários finais invocam processos clientes quando requisitam serviços da Rede. A cada vez que uma aplicação é executada, o processo cliente gerado contata um processo servidor (geralmente remoto), envia um pedido por serviço e fica bloqueado (suspendido) aguardando por uma resposta. Quando a resposta chega do processo servidor, o processo cliente pode, então prosseguir com a sua execução. Este protocolo simples e não orientado a conexão de Pedido/Resposta é usado para se evitar o overhead considerável causados pelos protocolos orientados a conexão tais como o RM/OSI e TCP/IP. As razões para adoção desse protocolo são:

- (i)     **Simplicidade** - nenhuma conexão tem que ser estabelecida antes das mensagens serem enviadas e a Resposta serve como um acknowledgement para o Pedido.
- (ii)    **Eficiência** - a pilha do protocolo é menor e portanto mais eficiente. Assumindo que tenhamos máquinas com uma mesma arquitetura, apenas 3 níveis do protocolo são necessários, conforma figura abaixo:

7	
6	
5	Pedido/Resposta
4	
3	
2	Enlace
1	Físico

As camadas Física e Enlace se encarregam de levar os protocolos entre cliente e servidor e vice-versa. Esta tarefa é gerenciada pelas placas de Rede, padrão Ethernet ou Token-Ring, por exemplo. Nenhum roteamento é necessário e nenhuma conexão é estabelecida, logo as camadas de Rede (3) e de Transporte (4) são dispensadas. A camada 5 é a camada do Protocolo Pedido/Resposta, a qual define um conjunto de Pedidos e Respostas válidas para serem usadas pelas aplicações C/S. Também a camada 6 não é necessária pois não existe a necessidade do gerenciamento de uma sessão.

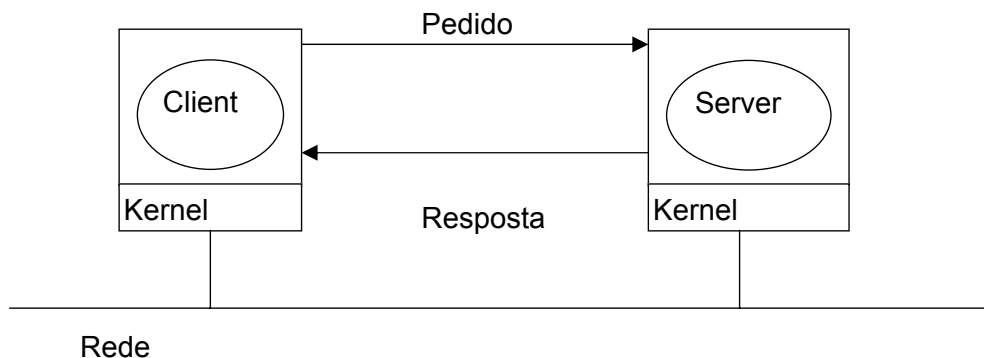
Devido a estrutura simplificada do protocolo, os serviços de comunicação providos pelo Kernel ( ou Micro Kernel ) podem ser reduzidos a duas system calls, uma para envio e outra para o recebimento das mensagens:

- (i)     **Send (destino, msg)** - Envia a mensagem "msg" para o processo "destino";
- (ii)    **Receive(origem,msg)** - Recebe a mensagem "msg"vinda do processo "origem";

No modelo C/S as rotinas Send e Receive são bloqueantes atendendo o modelo de Troca de Mensagens Síncrona ( rendezvous ), veja Apostila SOII.

Programas clientes são mais fáceis de serem escritos do que programas servidores e, normalmente, não exigem privilégios especiais do SO para poderem operar. Por sua vez, um programa servidor é bem mais trabalhoso de ser programado e trata de toda a complexidade envolvida nos pedidos enviados por clientes. Um processo servidor recebe um pedido de um processo cliente, executa todas as operações necessárias e retorna um resultado para ele.

No modelo de aplicações C/S, normalmente, cliente e servidor rodam em máquinas com o mesmo Kernel ( ou Micro Kernel ), com os processo cliente e servidor rodando na área de processos do usuário.



### 7.1 Componentes de uma Aplicação Distribuída

Uma AD possui funções que podem ser agrupadas em componentes voltados para o processamento da lógica do negócio, para a manipulação dos dados, e para os serviços de acesso aos dados. Esta distinção entre os componentes é apenas conceitual, na prática, algumas funções podem abranger mais de uma categoria. Observamos que nas aplicações convencionais (centralizadas) estes componentes sempre estiveram presentes, o que muda agora, no ambiente C/S é o interesse em analisá-los separadamente para efetuarmos a sua distribuição pelos diferentes processadores.

As funções de Lógica de Interface com o Usuário (LIU) são aquelas relacionadas com a apresentação e as atividades de Entrada/Saída de dados do usuário final. Tarefas como formatação de telas, leitura e exibição de dados em telas, gerência de diálogos são exemplos da LIU.

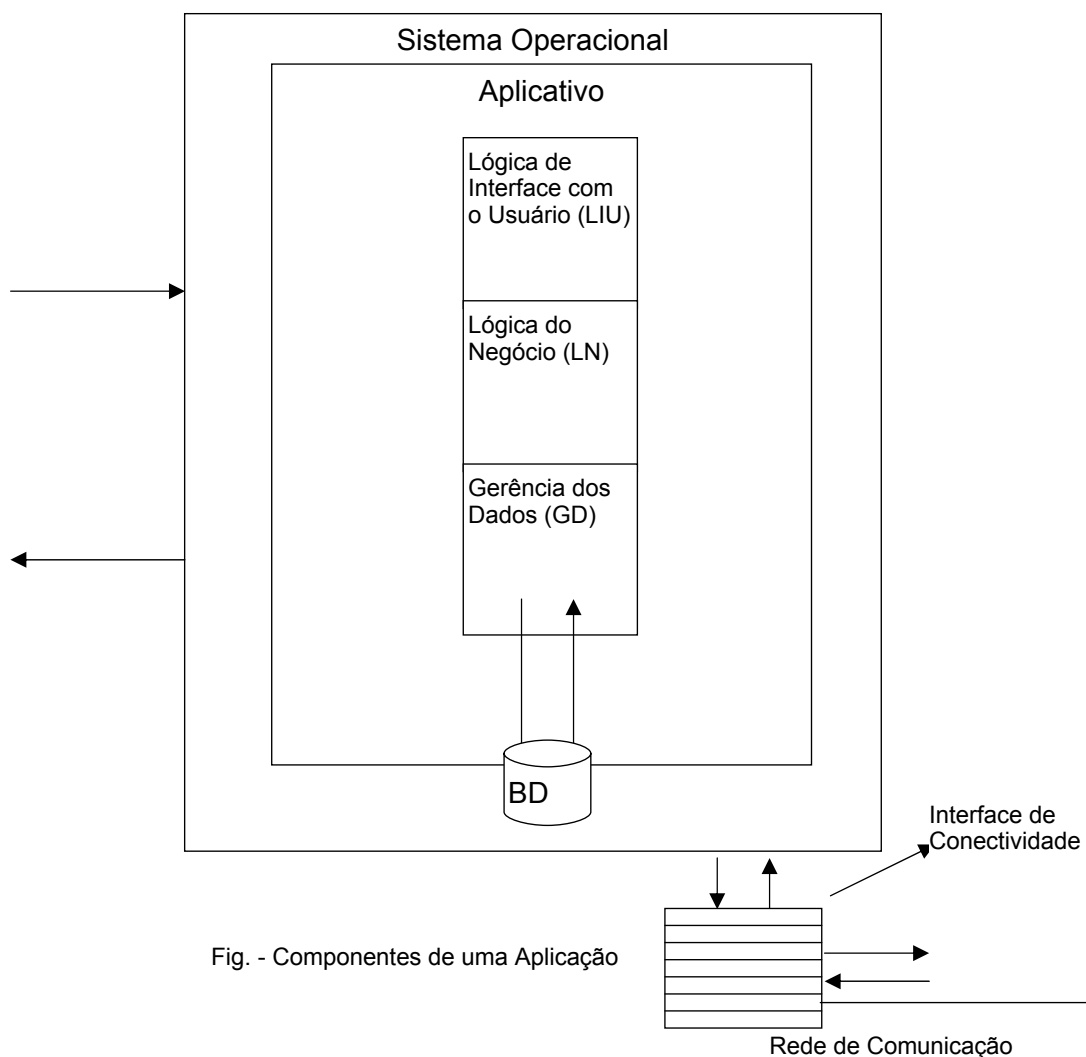


Fig. - Componentes de uma Aplicação

As funções da Lógica de Negócio (LN) implementam as regras do negócio e as práticas administrativas de uma organização, variando de organização para organização, e em geral não são passíveis de serem passadas a gerenciadores de uso geral, como as funções da LIU e o serviço de acesso aos dados (SAD).

As funções de gerência de dados (GD) compreendem as funções de manipulação de dados e as funções que realizam os serviços de acesso aos dados. As funções de manipulação de dados compõe a parte da aplicação que é responsável pelo armazenamento e pela recuperação de dados existentes em Bancos de Dados ou em sistemas de arquivos, conforme o caso. No caso de Bancos de Dados, estas funções são implementadas pelo uso dos comandos em SQL (Structured Query Language) submetidos ao sistema de gerência de Banco de Dados (SGBD).

As funções que realizam os serviços de acesso a dados provêm o acesso físico aos dados, como o acesso a índices, localização de registros e a sua recuperação.

## 7.2 Distribuição dos Componentes da Aplicação

Os componentes de uma aplicação devem ser distribuídos entre o cliente e os servidores de maneira a possibilitar o processamento cooperativo. A distribuição dos componentes é resultado de um processo complexo, onde diversos fatores são considerados, buscando-se o balanceamento da carga do sistema como um todo e o atendimento dos requisitos de desempenho e segurança especificados para a aplicação. Como resultado, temos as seguintes formas de distribuição dos componentes da Lógica de Interface com o Usuário (LIU) e da Lógica de Negócio (LN):

- (i) Apresentação Distribuída
- (ii) Apresentação Remota
- (iii) Função Distribuída

Com relação a distribuição das Funções de Gerência de Dados podemos ter as seguintes formas:

- (i) Gerência Remota dos Dados
- (ii) Gerência Distribuída dos Dados

### 7.2.1 Apresentação Distribuída

A apresentação distribuída ocorre quando as funções relacionadas com o processamento da LIU são divididas entre o cliente e o servidor. Basicamente, as funções relacionadas com a manipulação dos dispositivos físicos - vídeo, mouse, teclado - são alocadas ao cliente, enquanto as funções de apresentação que podem ser compartilhadas entre várias aplicações são alocadas no servidor.

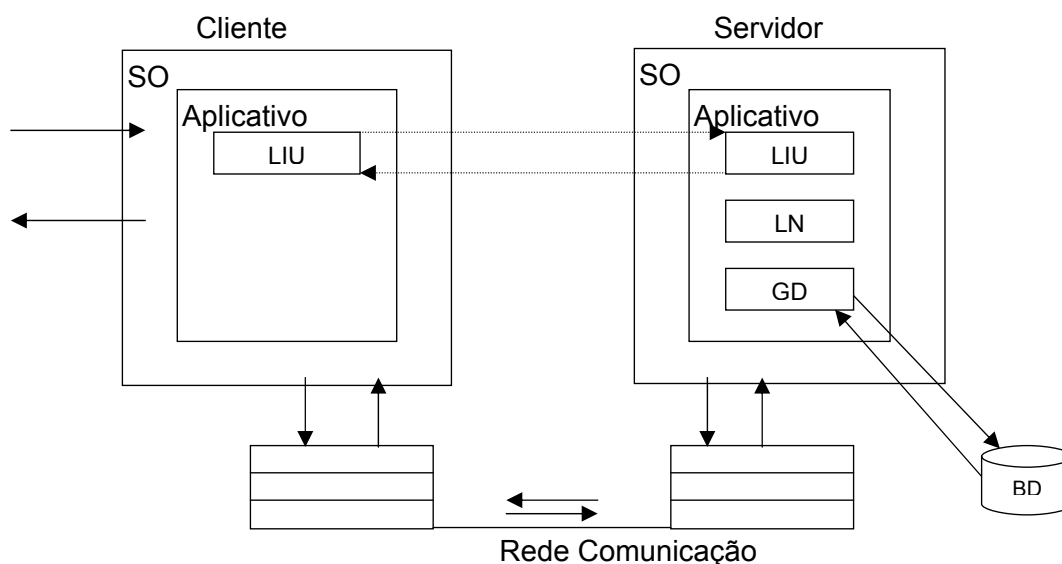


Fig. - Apresentação Distribuída  
Ex.: Interface Gráfica em Micros para aplicações tradicionais baseadas em mainframes



### 7.2.2 Apresentação Remota

As funções relacionadas com o funcionamento da LIU são consideradas remotas em relação ao resto da aplicação quando são colocadas integralmente no cliente, e os demais componentes alocados aos servidores.

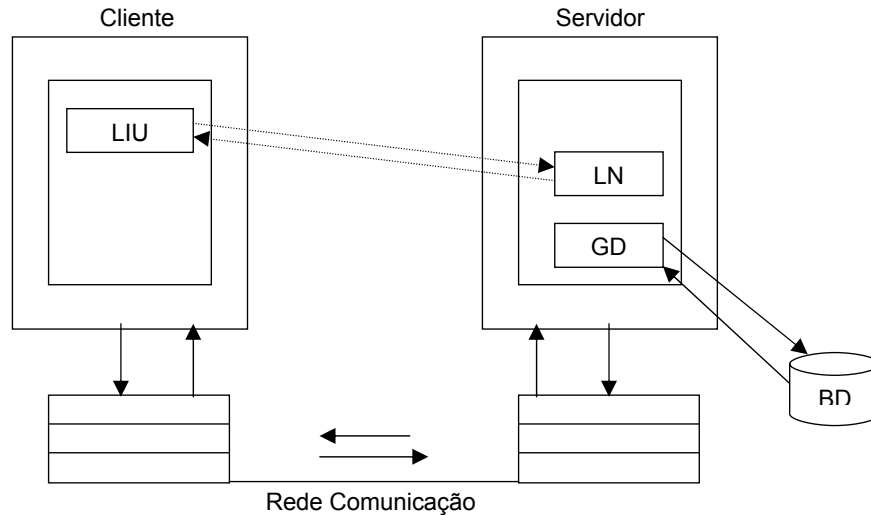


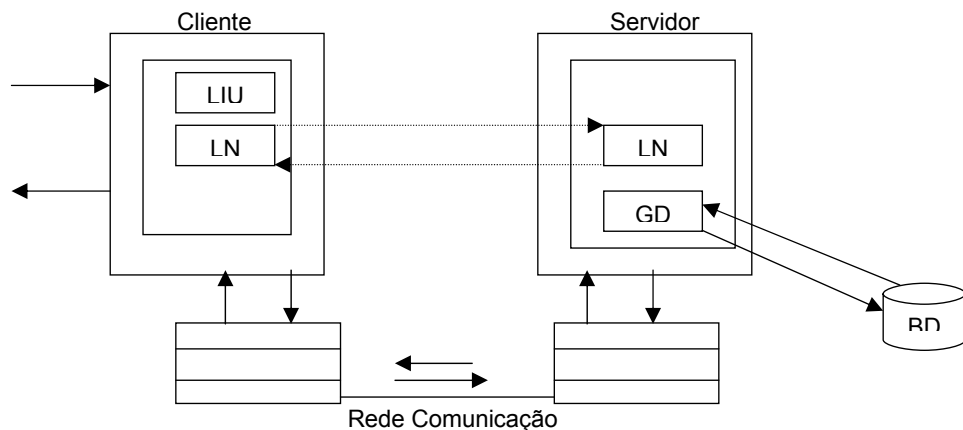
Fig. - Apresentação Remota

Ex.: Aplicações UNIX rodando sobre o Xwindows ( sistema gráfico de janelas em rede )

### 7.2.3 Função Distribuída

As funções de processamento da LN podem ficar integralmente no cliente ou no servidor, e podem, ainda, ser distribuídas entre o cliente e o servidor. A distribuição deve ser considerada quando:

- O conjunto de funções contiver funções que são compartilhadas com outras aplicações. O que nos indica a possibilidade de construção de um servidor:
- O conjunto de funções contiver funções que demandam capacidade de processamento não compatível com a capacidade do cliente.
- A aplicação possuir alto grau de interação com o usuário e, também, um volume elevado de acesso a dados gerenciados por um servidor de banco de dados. Neste caso, a solução para minimizar o fluxo de mensagens entre o servidor e o cliente pode ser obtido pela distribuição das funções: no cliente ficariam as funções de apresentação e as funções da lógica de negócio diretamente relacionadas, enquanto que no servidor poderiam ser alocadas as funções da LN relacionadas diretamente com o acesso aos dados.



## **8 Coordenação em Sistemas Distribuídos**

### **8.1 xxx**

### **8.2**

## **9 Sincronização em Sistemas Distribuídos**

### **9.1 xxx**

### **9.2**

## **10 Gerência de Processos em Sistemas Distribuídos**

### **10.1**

### **10.2**

## **11 Gerência de Arquivos em Sistemas Distribuídos**

### **11.1**

### **11.2**