Universidade Federal de Itajubá – UNIFEI
Instituto de Engenharia de Sistemas e Tecnologias da Informação – IESTI
CCO 004 – Sistemas Operacionais – Prof. Edmilson Marmo Moreira

Capítulo 3 – Programação Concorrente

> *"Nunca encontrei um homem tão ignorante que eu não pudesse aprender algo com ele."*
>
> **Galileu Galilei**

3.1 Introdução

A *Concorrência* existe quando, em um determinado instante, dois ou mais processos começaram a sua execução, mas não terminaram. Ela pode existir com um único processador, quanto em sistemas com múltiplos processadores.

Afirmar que processos estão sendo executados em paralelo implica na existência de mais de um processador, ou seja, paralelismo (paralelismo físico) ocorre quando há mais de um processo sendo executado no mesmo intervalo de tempo.

Quando vários processos são executados em um único processador, sendo que somente um deles é executado a cada vez, tem-se um pseudo-paralelismo (paralelismo lógico).

Com base nas definições, é possível definir três tipos de estilo de programação dentro da computação:

- Programação Seqüêncial: caracteriza-se pela execução de várias tarefas uma após a outra;
- Programação Concorrente: caracteriza-se pela iniciação de várias tarefas, sem que as anteriores tenham necessariamente terminado;
- Programação Paralela: caracteriza-se pela iniciação e execução das tarefas em paralelo (sistemas multiprocessadores).

Exemplo de Algoritmos para a preparação de um jantar:

Seqüencial:

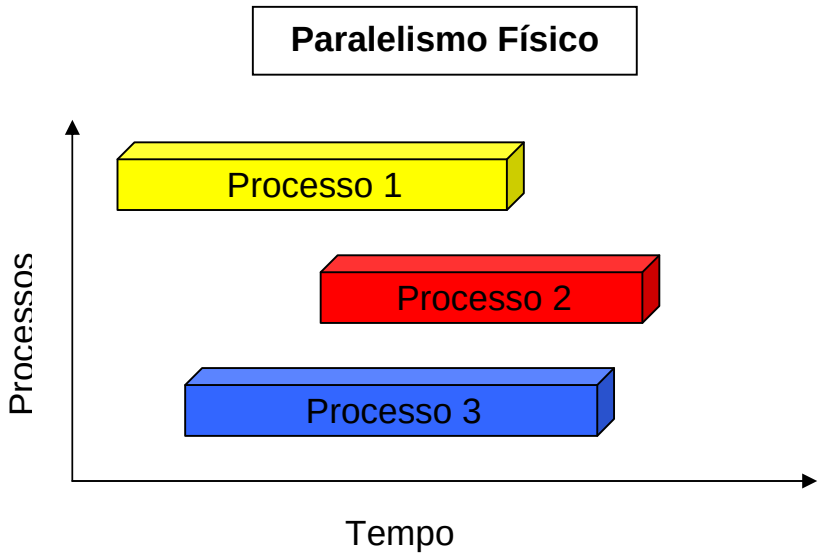
```
Abrir o refrigerador
Se estiver vazio
  Então vá ao restaurante
  Senão Preparar Salada
      Preparar Carne
      Preparar Sobremesa
      Comer.
```

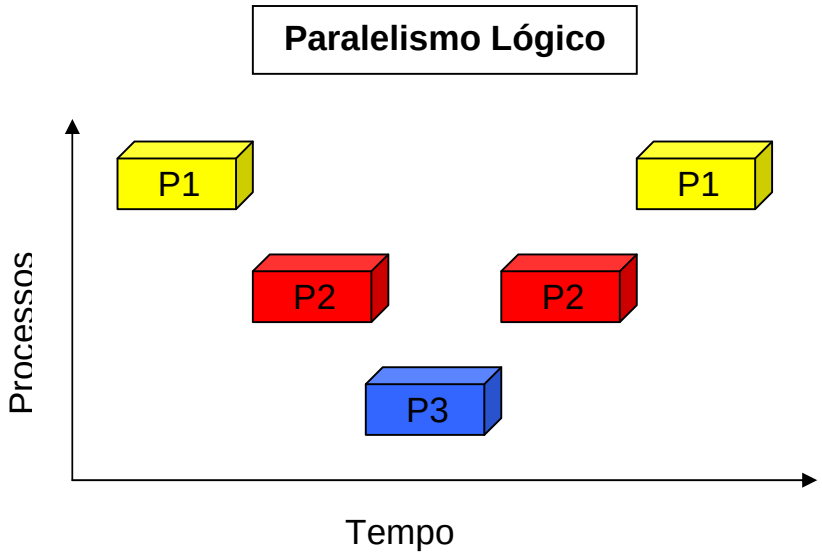
Concorrente:

```
Abrir o refrigerador
Se estiver vazio
  Então vá ao restaurante
  Senão lavar a alface
        Colocar de molho
        Temperar a carne
        colocar a carne para cozinhar
        preparar a sobremesa
        escorrer a alface
        temperar a alface
        retirar a carne do forno
      Comer.
```

Paralelo:

Fernando:	**Marina:**
Abrir o refrigerador	
Se estiver vazio	Se estiver vazio
Então	Então
vá ao restaurante	Vá ao restaurante
Senão	Senão
Temperar a carne	Preparar a Salada
Preparar a sobremesa	colocar a carne para Cozinhar
	Retirar a carne do Forno
Comer.	Comer.





Motivação para a Programação Concorrente

- Aumento do Desempenho.

Desvantagens da Programação Concorrente

- Programação mais complexa.
- Existência do não determinismo.

3.2 Especificação de Paralelismo

Várias construções para a ativação e término de processos concorrentes são discutidas na literatura, apresentando características e finalidades distintas.

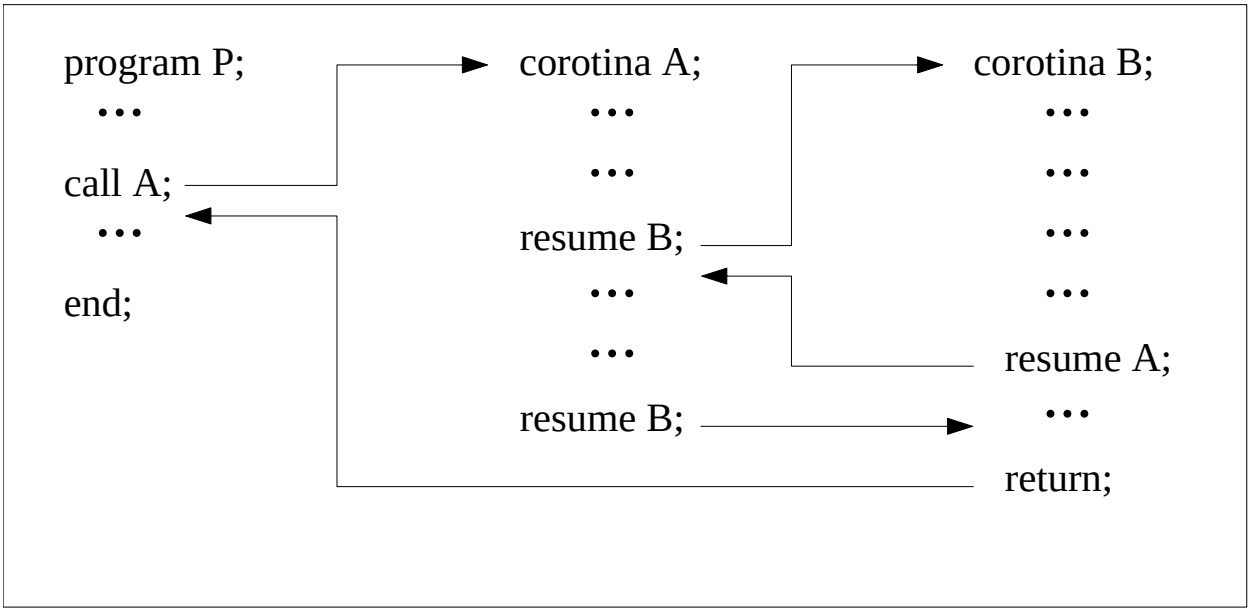
Os principais modelos serão discutidos a seguir.

Corotinas

Corotinas são subrotinas que possuem um modo de transferência de controle não hierárquico.

Corotinas transferem o controle entre si de maneira livre, através do comando *resume corotina*. E sempre que são ativadas, executam a partir do ponto onde foi executado a última chamada à *resume*.

Sempre existe apenas uma corotina ativa em cada instante, o que implica na adequação desta estrutura para a organização de programas concorrentes que compartilhem uma única CPU.



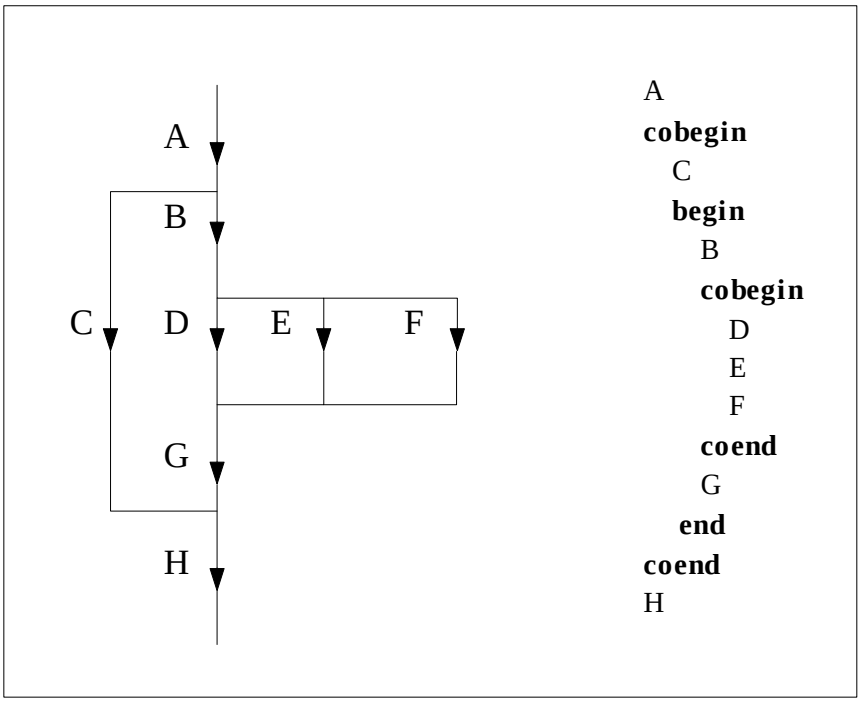
CoBegin/CoEnd

São também conhecidas como ParBegin e ParEnd, estes comandos oferecem uma maneira estruturada de ativação de um conjunto de instruções que devem ser executadas concorrentemente.

A execução concorrente das declarações S_1, S_2, ..., S_n pode ser ativada através da estrutura:

```
Cobegin
      S1
      S2
      ...
      Sn
Coend
```

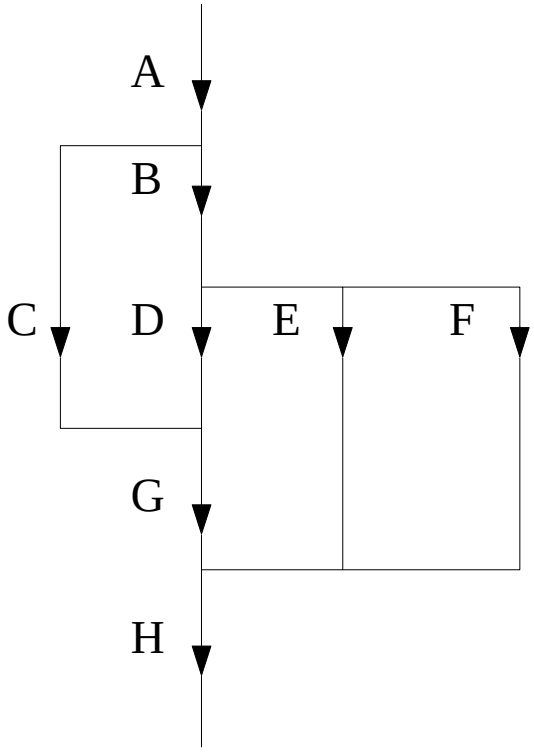
O processo pai será bloqueado até que S_1, S_2, ..., S_n estejam terminadas.



Fork/Join

O comando Fork implica que um determinado conjunto de instruções (processo filho) deve iniciar a sua execução em paralelo com o processo que o executa (processo pai).

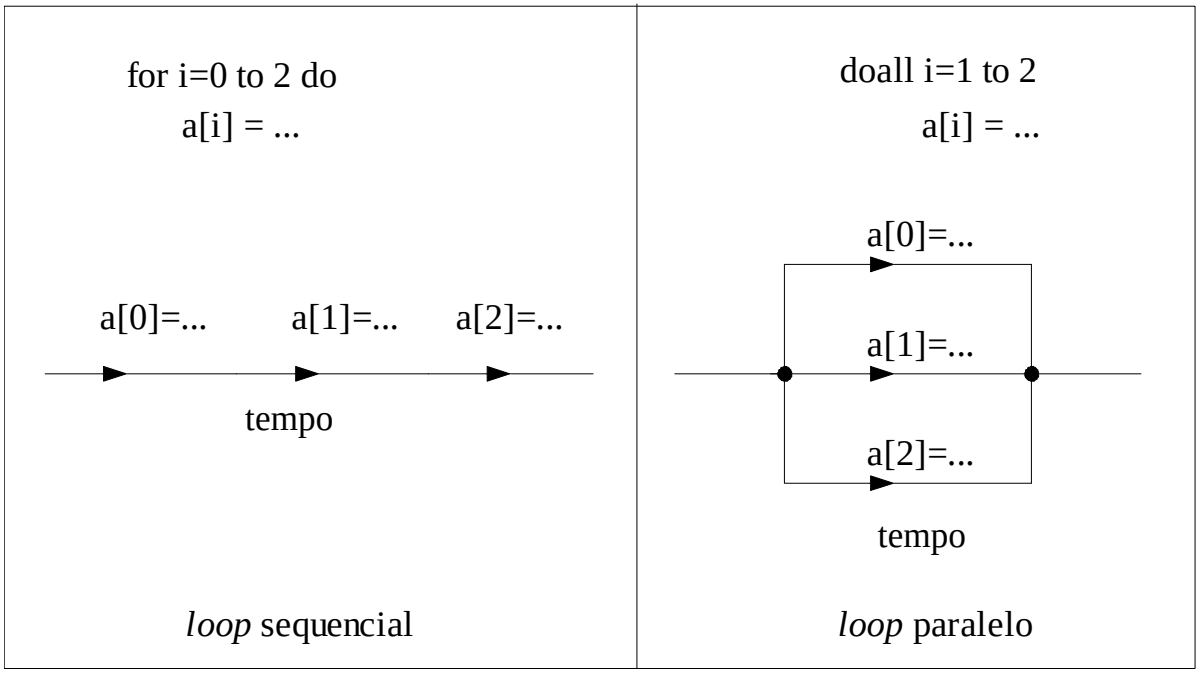
O comando Join é utilizado para a sincronização do processo pai com os filhos gerados.




```
   A
   m=2
   fork c
   B
   n=3
   fork e
   fork f
   D
   join m,g; quit
g:G
   join n,h; quit
h: H
   quit
c: C
   join m,g; quit
e: E
   join n,h; quit
f: F
   join n,h; quit
```

DoAll

Este comando pode ser visto como um comando CoBegin/CoEnd onde as instruções executadas em paralelo são as diversas instâncias de um bloco de comandos dentro de um comando de iteração. Alguns comandos de função semelhante são: forall, pardo e doacross.



3.2 Problema da Seção Crítica

A seção crítica ou região crítica é uma parte do código de um processo que faz acesso a recursos compartilhados.

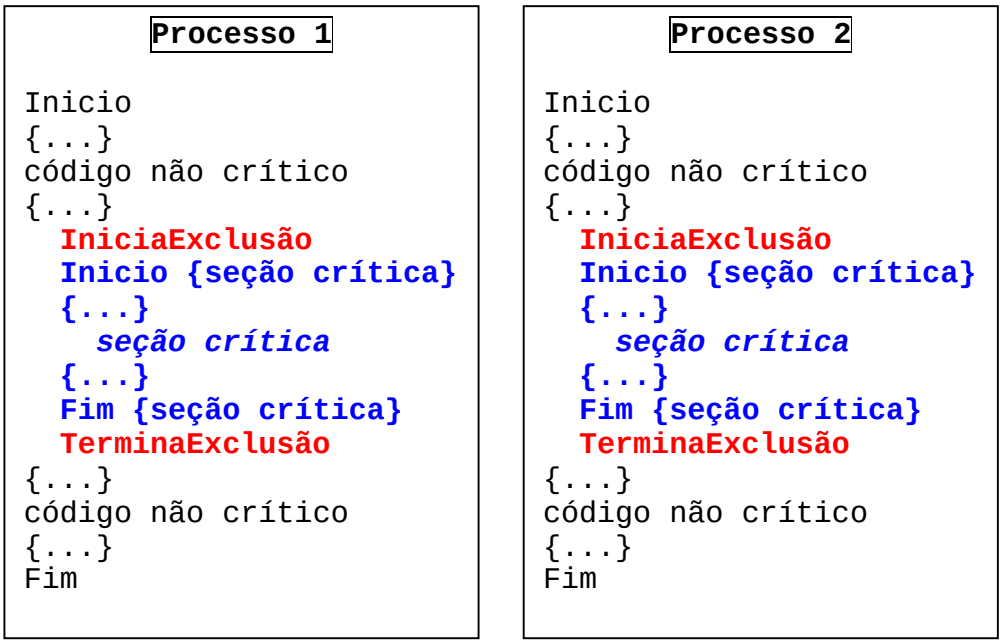
Muitos problemas podem surgir quando vários processos fazem acessos a dados ou recursos compartilhados de forma paralela ou concorrente.

Os mecanismos de sincronização devem evitar a concorrência nas regiões críticas, permitindo que somente um processo esteja executando sua região crítica de cada vez.

Essa idéia de exclusivismo de acesso é denominada Exclusão mútua.

Exclusão Mútua

Se um processo estiver sendo executado em sua seção crítica, é preciso impedir que todos os outros processos entrem em suas próprias seções críticas. Reciprocamente, não se pode permitir que um processo entre em sua seção crítica se qualquer outro processo estiver em sua própria seção crítica.



O problema da exclusão mútua leva as seguintes perguntas:

- Como garantir a exclusão mútua?
- O que pode ser feito antes que um processo entre em sua seção crítica para garantir a exclusão mútua?
- Deve-se fazer alguma coisa quando um processo termina a sua seção crítica?

Na Figura anterior, cada processo faz referência a um comando chamado IniciaExclusão antes de sua seção crítica, e cada um se refere a um comando chamado TerminaExclusão depois de sua seção crítica.

IniciaExclusão deve fazer o seguinte:

- Verificar se qualquer outro processo está em sua própria seção crítica e esperar se houver algum.
- Passar à execução da seção crítica se nenhum outro processo estiver em sua própria seção crítica.

TerminaExclusão deve informar a todos os outros processos que um processo terminou a execução de sua própria seção crítica.

Soluções de Hardware

As soluções de hardware são importantes pois criam mecanismos que permitem a implementação das soluções de software.

Desabilitação de Interrupções

A solução mais simples para o problema da exclusão mútua é fazer com que o processo, antes de entrar em sua região crítica, desabilite todas as interrupções externas e as reabilite após deixar a seção crítica.

Como a mudança de contexto só pode ser realizada através de interrupções, o processo que as desabilitou terá acesso exclusivo garantido.

Processo 1

```
Inicio
{...}
código não crítico
{...}
  Desabilita_Interrupções
  {...}
    seção crítica
  {...}
  Habilita_Interrupções
{...}
código não crítico
{...}
Fim
```

Esse mecanismo é inconveniente por vários motivos. O maior deles acontece quando o processo que desabilitou as interrupções não torna a habilitá-las. Nesse caso o sistema, provavelmente, terá seu funcionamento seriamente comprometido.

A desabilitação das interrupções é útil ao SO quando este necessita manipular estruturas de dados compartilhadas do sistema, como lista de processo. Assim, o SO garante que não ocorrerão problemas de inconsistência em seus dados.

Instrução Test-And-Set

Muitos processadores possuem uma instrução especial, que permite ler uma variável, armazenar seu conteúdo em uma outra área e atribuir um

novo valor a essa variável. Esse tipo de instrução é denominado instrução *test-and-set*.

A principal característica dessa instrução é ser sempre executada sem interrupção, ou seja, trata-se de uma instrução indivisível (atômica).

A instrução *test-and-set* possui o seguinte formato:

```
Test-and-Set (X, Y);
```

Na execução dessa intrução o valor lógico da variável Y é copiado para X, sedo atribuído à variável Y o valor lógico verdadeiro.

Para coordenar o acesso concorrente a um recurso, a instrução *test-and-set* utiliza uma variável lógica global. Quando essa variável for falsa, qualquer processo poderá alterar seu valor para verdadeiro, através da instrução *test-and-set* e, assim, acessar o recurso de forma exclusiva.

Ao terminar o acesso o processo deve simplesmente retornar o valor da variável para falso, liberando o acesso ao recurso.

```
Program Programa_Test_and_Set;
Var Bloq : Boolean;
```

```
Procedure ProcessoA;
var PodeA : Boolean;
begin
  repeat
   PodeA := True;
   while PodeA do
    Test_and_Set(PodeA, Bloq);
   {...}
   {seção crítica}
   {...}
   Bloq := False;
end;
```

```
Procedure ProcessoB;
var PodeB : Boolean;
begin
  repeat
   PodeB := True;
   while PodeB do
    Test_and_Set(PodeB, Bloq);
   {...}
   {seção crítica}
   {...}
   Bloq := False;
end;
```

```
begin
     Bloq := False;
     CoBegin
          ProcessoA;
          ProcessoB;
     CoEnd;
end;
```

Soluções de Software

Além da exclusão mútua, que soluciona os problemas de compartilhamento de recursos, três fatores fundamentais para a solução dos problemas de sincronização deverão ser atendidos:

- O número de processadores e o tempo de execução dos processos concorrentes devem ser irrelevantes;
- Um processo, fora de sua região crítica, não pode impedir que outros processo entrem em suas próprias regiões críticas;
- Um processo não pode permanecer indefinidamente esperando para entrar em sua região crítica.

As primeiras soluções de software para o problema da exclusão mútua entre processos possuiam, de forma geral, duas inconviniências:

provocavam *starvation* e utilizavam um mecanismo de espera ocupada (*busy wait*).

Além da exclusão mútua, que soluciona os problemas de compartilhamento de recursos, três fatores fundamentais para a solução dos problemas de sincronização deverão ser atendidos:

3.3 Semáforos

O conceito de *semáforo* foi proposto por Dijkstra, como uma solução mais geral e simples de ser implementada, para os problemas de sincronização entre processos concorrentes.

Um semáforo é uma variável inteira, não negativa, que só pode ser manipulada por duas instruções atômicas: DOWN e UP, também chamadas, originalmente, instruções P e V.

No caso da exclusão mútua, as instruções DOWN e UP funcionam como protocolos de entrada e saída, respectivamente, para que um processo possa entrar e sair de sua região crítica.

O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes. Se seu valor for maior que 0, nenhum processo está utilizando o recurso; caso contrário, o processo fica impedido do acesso.

Sempre que deseja entrar na sua região crítica, um processo executa uma instrução DOWN. Se o semáforo for maior que 0, este é decrementado de 1, e o processo que solicitou a operação pode executar sua região crítica. Entretanto, se uma instrução DOWN é executada em um semáforo cujo valor seja igual a 0, o processo que solicitou a operação ficará no estado de espera, em uma fila associada ao semáforo.

O processo que está acessando o recurso, ao sair de sua região cr´tica, executa uma instrução UP, incrementando o semáforo de 1 e liberando o acesso ao recurso. Se um ou mais processos estiverem esperando o sistema escolhe um processo na fila de espera e muda seue estado para pronto.

```
Type Semaforo = record
       Valor : integer;
       Fila_Espera :
         (*Lista de Proc.pendentens *);
end;
```

```
Procedure DOWN
           (var S : Semaforo);
begin
  if S.Valor = 0 then
    Insere_Proc.Fila_Espera
  else
    S.Valor := S.Valor - 1;
end;
```

```
Procedure UP
           (var S : Semaforo);
begin
  if (Tem_Proc_Fila) then
    Retira_Proc.Fila_Espera
  else
    S.Valor := S.Valor + 1;
end;
```

3.4 Monitores

O uso de semáforos exige dos programadores bastante atenção, pois os problemas que podem resultar dos uso incorreto dos semáforos podem ser difícieis de reproduzir devido ao não determinismo das tarefas. Dessa forma foi proposto por Hoare o mecanismo denominado monitor.

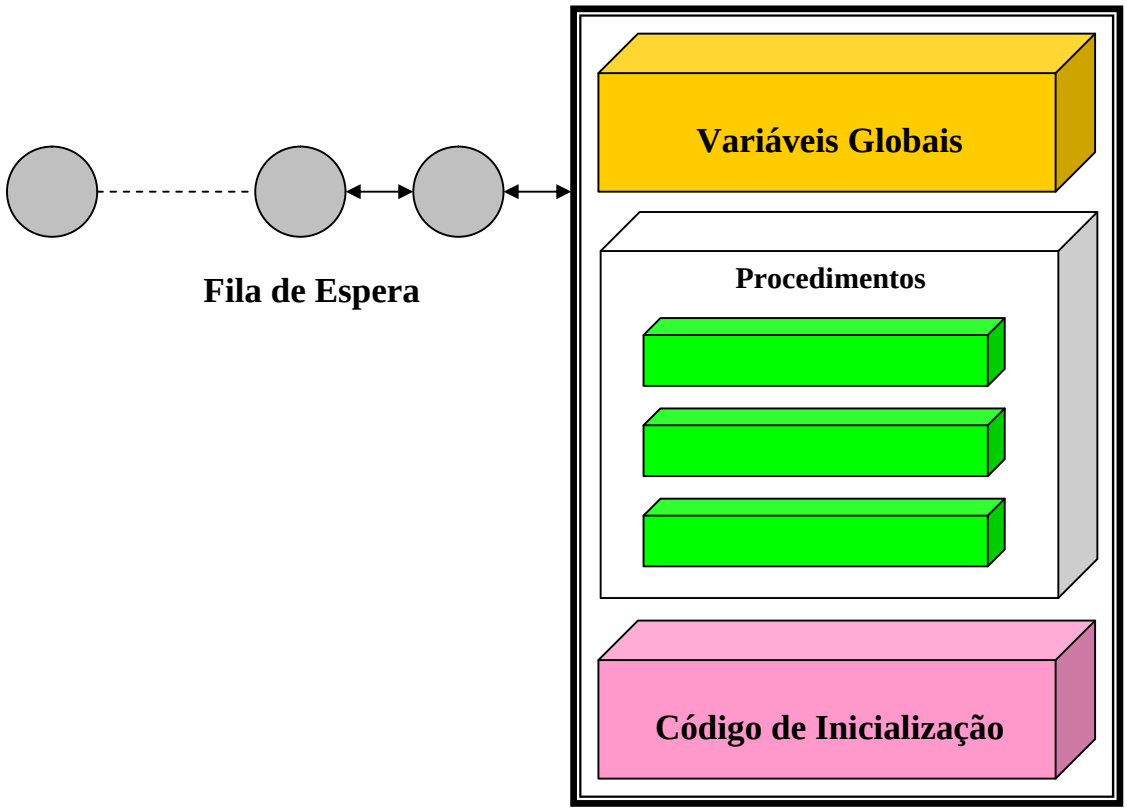
Um monitor é um conjunto de procedimentos, variáveis e estruturas de dados definido dentro de um módulo (semelhante a uma classe, da programação orientada à objetos).

A característica mais importante de um monitor é a implementação automática da exclusão mútua, pois somente um processo pode estar executando os procedimentos do monitor em um determinado instante.

Se um processo requisitar um procedimento do monitor, será verificado se já existe outro processo executando algum procedimento do monitor. Se isto ocorre o processo fica aguardando até o monitor estar disponível novamente.

A principal diferença entre monitores e semáforos é que a exclusão mútua, no caso do monitor, é implementada automaticamente pelo compilador e não pelo programador, como é o caso dos semáforos.

Estrutura do Monitor



A implementação da sincronização condicional não é tão simples quanto a da exclusão mútua. Para implementá-la, é necessário utilizar variáveis de condição e duas instruções que operam sobre elas: *WAIT* e *SIGNAL*.

Uma variável de condição é uma estrutura de dados do tipo fila, onde os processos esperam por algum evento. Sempre que o monitor descobre que alguma condição impede a continuação da execução de um processo, ele realiza um WAIT, fazendo com que o processo fique no estado de espera na fila associada a essa condição.

Uma característica do monitor é permitir que um processo possa executar um de seus procedimentos, mesmo que um ou mais processos estejam no estado de espera dentro do monitor.

O processo bloqueado só poderá prosseguir sua execução quando um outro processo executar um SIGNAL, sobre a mesma condição que o colocou no estado de espera.

3.5 Troca de Mensagens

Em sistemas com memória distribuída, se torna inviável o uso de semáforos ou monitores, pois os processos que executam em processadores diferentes não possuem acesso ao mesmo endereçamento de dados.

Nesse caso a única maneira de haver comunicação é através da troca de mensagens que é realizada pelas primitivas *SEND* e *RECEIVE*.

Existem, basicamente, duas formas de comunicação entre processos pela troca de mensagens: comunicação síncrona e comunicação assíncrona.

A comunicação síncrona acontece quando um processo que envia uma mensagem fica aguardando até que o processo receptor leia a mensagem ou quando o processo receptor espera pelo envio de uma mensagem por algum processo.

A comunicação assíncrona ocorre quando nenhum processo fica bloqueado, nem o processo emissor nem o processo receptor.

Nesse caso é necessária a utilização de *buffers* para o armazenamento temporário das mensagens.

A vantagem da comunicação assíncrona é o maior paralelismo entre as tarefas. Já a comunicação síncrona permite a realização de sincronização entre os processos.

Bibliografia:

DEITEL, H. M., DEITEL, P. J. & CHOFFNES, D. R.
Sistemas Operacionais
3a. Edição: Person (2005) – São Paulo / SP

MACHADO, F. B. & MAIA, L. P.
Arquitetura de Sistemas Operacionais
3a. Edição: LCT (2002) – Rio de Janeiro / RJ

SANTANA, R. H. C.; et. alli.
Computação Paralela
Apostila – ICMC - USP

SILBERSCHATZ, ABRAHAM & GALVIN, PETER B.
Operating System Concepts
5th Edition: John Wiley (1999) – Massachusetts

STALLINGS, WILLIAM
Operating Systems
2nd Edition: Prentice-Hall (1995)

TANENBAUM, ANDREW S.
Sistemas Operacionais Modernos
2ª. Edição. Person (2003) – São Paulo / SP

TANENBAUM, ANDREW S. & WOODHULL, ALBERT S.
Operating Systems: Design and Implementation
2nd Edition: Prentice-Hall (1997) – Upper Saddle River / NJ

TOSCANI, S. S.; OLIVEIR, R. S. & CARISSIMI, A. S.
Sistemas Operacionais e Programação Concorrente
Editora Sagra-Luzzatto (2003) – Porto Alegre / RS