Cap. 5 - Exclusão Mútua e Sincronização

5.1 - Princípios de Concorrência entre Processos

5.2 - Exclusão Mútua: Abordagem por Software

5.3 - Exclusão Mútua: Suporte por Hardware

5.4 - Mecanismo de Semáforos

5.5 - Mecanismo de Monitores

5.6 - Problema dos Escritores/Leitores

... Cap. 5 - Exclusão Mútua e Sincronização

- William STALLINGS; **Operating Systems: Internals and Design Principles**, New Jerssey, Prentice-Hall, 1998, ISBN: 0-13-887407-7

- Eleri CARDOZO; Maurício MAGALHÃES; Luís F. FAINA; **Sistemas Operacionais**, Dep. de Eng. de Computação e Automação Industrial, Fac. de Engenharia Elétrica e de Computação, UNICAMP, 1996.

... Cap. 5 - Exclusão Mútua e Sincronização

✯ No projeto de sistemas operacionais, o tema central está associado ao gerenciamento de processos e *threads* nas mais diferentes áreas, tais como: multiprogramação, multiprocessamento e computação distribuída;

- ... fundamentalmente em todas estas áreas e principalmente para o projeto do sistema operacional, o aspecto chave é a concorrência.

✯ **concorrência:** engloba aspectos de projeto do *host* incluindo comunicação entre processos, compartilhamento e competição por recursos, sincronização de atividades entre múltiplos processos e alocação de tempo do processador para processos.

... Cap. 5 - Exclusão Mútua e Sincronização

✯ O conceito de concorrência evidencia-se em 03 diferentes contextos:

- **aplicações múltiplas:** a multiprogramação é resultado do compartilhamento do tempo de processamento entre aplicações ativas;

- **aplicações estruturadas:** como uma extensão dos princípios da programação modular e estruturada, algumas aplicações podem ser efetivamente desenvolvidas como um conjunto de processos concorrentes;

- **estrutura do sistemas operacionais:** as mesmas vantagens da estruturação aplicam-se aos programadores de sistemas e, pode ser verificado que os sistemas operacionais são implementados como um conjunto de processos.

5.1 - Princípios da Concorrência

✡ Tanto em sistemas multiprogramados quanto nos multiprocessados, as técnicas de entrelaçamento e sobreposição podem ser vistas como exemplos de concorrência, e ambas apresentam os mesmos problemas:

- ✘ ... o compartilhamento de recursos globais está recheado de perigos - p. ex.: se 02 processos compartilham uma mesma variável global e ambos efetuam operações de escrita e leitura, então a ordem com que estas operações (escritas e leituras) ocorrem torna-se um elemento crítico;
- ✘ ... é difícil para o sistema operacional gerenciar a alocação de recursos de forma ótima - exemplo: se um processo requisita o uso de um canal de I/O e em seguida é suspenso, pode ser ineficiente para o S.O. simplesmente bloquear o canal e não permitir o seu uso por outros processos;
- ✘ ... torna-se difícil a localização de erros de programação, dado que os resultados são tipicamente não podem ser reproduzidos.

✡ Razão para estas dificuldades: a execução de processos não pode ser predita pois esta depende das atividades de outros processos, da maneira pela qual as interrupções são manipuladas e das políticas de escalonamento.

... 5.1 - Princípios da Concorrência

✡ Seja o exemplo abaixo que ilustra os elementos essenciais de um programa que "ecoa" em algum dispositivo de saída o caracter informado no dispositivo de entrada:

```
procedure echo;                Process P1                Process P2
var out, in: character;        ..                        ..
begin                          input( in, keyboard )     ..
  input( in, keyboard );       ..                        input( in, keyboard )
  out := in;                   out := in                 out := in
  output( out, display );      output( out, display )    ..
end                            ..                        output( out, display )
                               ..                        ..
```

- **aspecto chave:** faz-se necessário proteger variáveis e possivelmente recursos globais compartilhadas através do controle do código que dá acesso a esta variáveis.

... 5.1 - Princípios da Concorrência

✰ **região crítica:** a execução de uma região crítica (parte do código que manipula o recurso compartilhado) deve ser um procedimento controlado a fim de evitar que os recursos compartilhados atinjam estados inconsistentes.

✰ **solução:** não permitir que mais de um processo leia ou escreva dados compartilhados ao mesmo tempo, ou seja, deve-se garantir a **exclusão mútua**.

... 5.1 - Princípios da Concorrência

✰ Embora a **mútua exclusão** evite inconsistências, o mecanismo não garante eficiência na utilização dos recursos compartilhados e, assim, para assegurarmos uma boa solução faz-se necessário as seguintes considerações:

❶ garantir que 02 ou mais processos não estejam simultaneamente dentro de suas regiões críticas referentes ao mesmo recurso compartilhado;
❷ mútua exclusão deve ser contemplada de forma independente da velocidade relativa dos processos ou mesmo do número de processadores;
❸ nenhum processo executando fora da região crítica pode bloquear outro processo;
❹ nenhum processo espere um tempo arbitrariamente longo para executar uma região crítica (p. ex., na ocorrência de um bloqueio temporário).

- ... para garantir as propriedades acima propostas, os algoritmos de controle são classificados segundo o modo que esperam pela autorização de entrada em uma região crítica: competindo pelo processador durante a espera ou bloqueado.

5.2 - Exclusão Mútua: Abordagem por Software

✠ Um algoritmo de mútua exclusão possui invariavelmente duas partes:

❶ **enter_critical:** é evocada quando o processo deseja iniciar a execução de uma região crítica; ... ao retornar o processo está apto a executar a região crítica;

❷ **exit_critical:** é evocada para anunciar que a região crítica já foi executada.

```
program  mutual_exclusion;
const  n = ...;  /* number of processes */

procedure  P( i: integer );
begin
  repeat
    enter_critical( R );
    ... critical region ...
    exit_critical( R );
    ... ramainder ...
  forever
end;
```

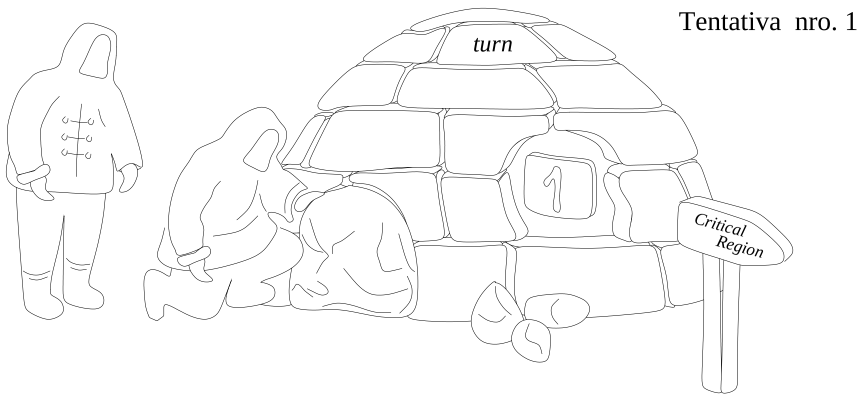
```
/* main program */
 begin
   parbegin
     P( 1 );
     P( 2 );
     ...
     P( n );
   parend
 end.
```


... 5.2 - Exclusão Mútua: Abordagem por Software

```
var  turn:  0 .. 1;
```

```
Process 0
  .
  while turn != 0 do { nothing; }
  ... critical region ...
  turn := 1;
  .
  .
```

```
Process 1
  .
  while turn != 1 do { nothing }
  ... critical region ...
  turn := 0;
  .
  .
```



... 5.2 - Exclusão Mútua: Abordagem por Software

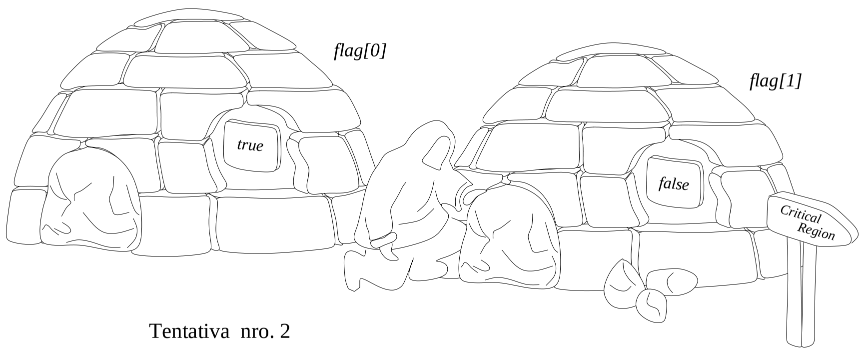
```
var  flag:  array [0 .. 1] of boolean;
```

Process 0

```
  .
  while flag[1] do { nothing; }
  flag[0] := true;
  ... critical region ...
  flag[0] := false;
  .
  .
```

Process 1

```
  .
while flag[0] do { nothing }
flag[1] := true;
... critical region ...
flag[1] := false;
.
.
```



Tentativa nro. 2

... 5.2 - Exclusão Mútua: Abordagem por Software

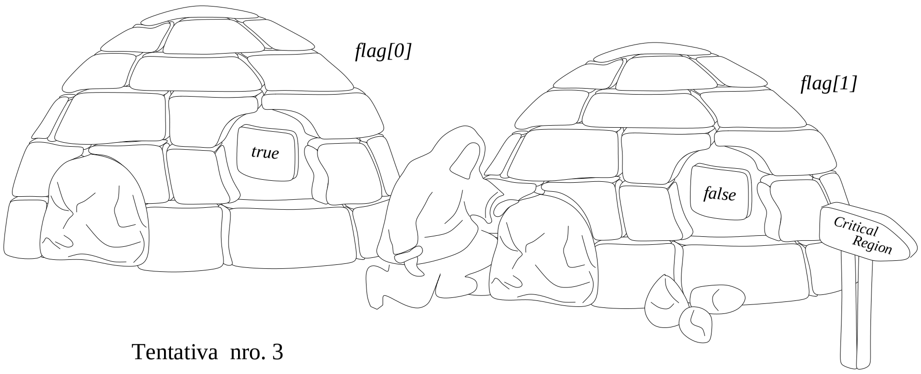
```
var  flag:  array [0 .. 1] of boolean;
```

Process 0

```
  .
  flag[0] := true;
  while flag[1] do { nothing; }
  ... critical region ...
  flag[0] := false;
  .
  .
```

Process 1

```
  .
flag[1] := true;
while flag[0] do { nothing }
... critical region ...
flag[1] := false;
.
.
```



Tentativa nro. 3

... 5.2 - Exclusão Mútua: Abordagem por Software

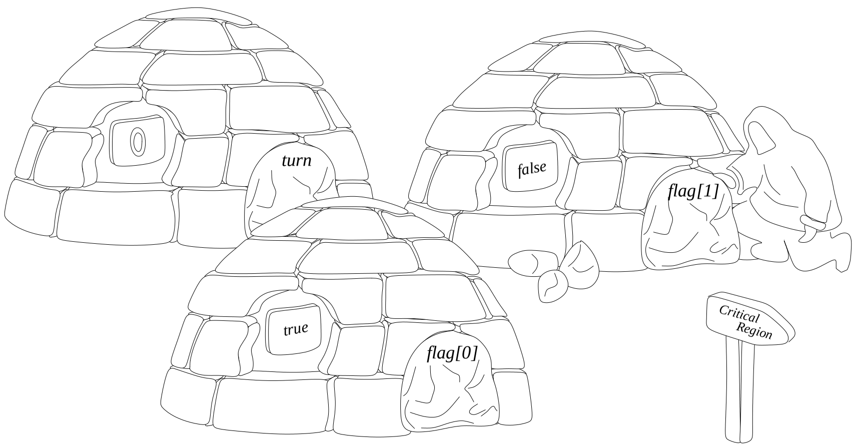
✠ Na tentativa anterior, *deadlock* ocorre pois cada processo insiste em entrar na sua região crítica e não abre a possibilidade de voltar atrás na sua posição.

```
var  flag:  array [0 .. 1] of boolean;

Process 0                                   Process 1;
  .                                         .
  .                                         .
  flag[0] := true;                          flag[1] := true;
  while flag[1] do                          while flag[0] do
  begin                                     begin
    flag[0] := false;                         flag[1] := false;
    ... delay for a short time ...            ... delay for a short time ...
    flag[0] := true;                          flag[1] := true;
  end;                                      end;
  ... critical region ...                   ... critical region ...
  flag[0] := false;                         flag[1] := false;
  .                                         .
```

- ... ao tentarmos corrigir o problema anterior, ainda assim há uma seqüência de instruções onde nenhum dos processos irá progredir no seu processamento.

5.2.1 - Exclusão Mútua: Algoritmo de Dekker

```
/* Main Program */
var  flag:  array [0 .. 1] of boolean;
     turn: 0 .. 1;

begin
  flag[0] := false;       flag[1] := false;       turn := 1;
  parbegin
    P0;  /* Procedure P0 */               P1;  /* Procedure P1 */
  parend;
end;
```



... 5.2.1 - Exclusão Mútua: Algoritmo de Dekker

- ... restante do código do Algoritmo de Dekker:

```
Procedure P0;
begin
  repeat
    flag[0] := true;
    while flag[1] do if turn = 1 then
    begin
      flag[0] := false;
      while turn = 1 do { nothing };
      flag[0] := true;
    end;
    ... critical region ...
    turn := 1;
    flag[0] := false;
    ... remainder ...
  forever;
end;
```

```
Procedure P1;
begin
  repeat
    flag[1] := true;
    while flag[0] do if turn = 0 then
    begin
      flag[1] := false;
      while turn = 0 do { nothing }
      flag[1] := true;
    end;
    ... critical region ...
    turn := 0;
    flag[1] := false;
    ... remainder ...
  forever;
end;
```


5.2.2 - Exclusão Mútua: Algoritmo de Peterson

```
var  flag:  array [0 .. 1] of boolean;
     turn: 0 .. 1;
```

```
Procedure P0;
begin
  repeat
    flag[0] := true;
    turn := 1;
    while flag[1] and turn = 1 do
      { nothing };
    ... critical region ...
    flag[0] := false;
    ... remainder ...
  forever;
end;
```

```
Procedure P1;
begin
  repeat
    flag[1] := true;
    turn := 0;
    while flag[0] and turn = 0 do
      { nothing };
    ... critical region ...
    flag[1] := false;
    ... remainder ...
  forever;
end;
```

```
/* Main Program - the same as before */
```

5.3 - Exclusão Mútua: Solução por Hardware

✰ Do ponto de vista da exclusão mútua, um processo continuará sendo executado até que uma operação do sist. oper. seja requerida ou até ser interrompido;

- ... a solução mais simples é o método de desabilitar todas as interrupções quando se está entrando na região crítica e reabilitá-la ao sair.

```
repeat
  ... disable interrupts ...
  ... critical region ...
  ... enable interrupts ...
  ... remainder ...
forever
```

- ... pelo fato da região crítica não poder ser interrompida, a exclusão mútua está garantida, embora o preço pago por esta abordagem seja alto.

✰ Eficiência de execução pode notadamente ser degradada como conseqüência da limitação imposta ao processador no que tange a habilidade de entrelaçar processos.

... 5.3 - Exclusão Mútua: Solução por Hardware

✰ **solução alternativa:** proposição de instruções de máquina que suportam ações de forma atômica, tais como: ler e escrever ou ler e testar uma posição da memória com um único ciclo de busca de instrução.

✰ **test and set instruction:** é uma operação atômica não sujeita a interrupções que testa o valor de seu argumento, modificando-o sob certa condição e retornando *true* ou *false* conforme a operação realizada.

```
function testset ( var i: integer ) : boolean;
  begin
    if i = 0 then
      begin
        i := 1;
        testset := true;
      end;
  else
    testset := false; end.
```

... 5.3 - Exclusão Mútua: Solução por Hardware

- Usando esta instrução, a solução do problema de exclusão mútua torna-se razoavelmente simples quando comparada a qualquer umas das soluções anteriores:

```
const n = ...;  /* number os processes */
var bolt: integer;

procedure P( i : integer );
begin
  repeat
    repeat { nothing } until testset( bolt );
    ... critial region ...
    bolt := 0;
    ... remainder ...
  forever
end;

begin /* Main Program */
  bolt := 0;
  parbegin
    P(1);  P(2);  ...  P(n);
  parend
end.
```


... 5.3 - Exclusão Mútua: Solução por Hardware

- **exchange instruction:** é uma operação atômica não sujeita a interrupções que permuta os conteúdos de uma registrador e de uma posição da memória.

```
procedure exchange  ( var r: register; var m: memory )
var  temp;
begin
  temp := m;
  m := r;
  r := temp;
end.
```

... 5.3 - Exclusão Mútua: Solução por Hardware

✰ Solução para o problema da exclusão mútua usando **exchange instruction**:

```
const n = ...;  /* number os processes */
var bolt: integer;

procedure P( i : integer );
var keyi: integer;
begin
  repeat
    keyi := 1;
    repeat exchange( keyi, bolt ) until keyi = 0;
    ... critial region ...
    exchange( keyi, bolt );
    ... remainder ...
  forever
end;

begin /* Main Program */
  bolt := 0;
  parbegin
    P(1);  P(2);  ...  P(n);
  parend
end.
```


... 5.3 - Exclusão Mútua: Solução por Hardware

✰ Algumas propriedades da abordagem utilizando instruções de máquina:

- é aplicável a um grande número de processos (p. ex., mono e multiprocessados).
- a abordagem por instruções de máquina é simples e fácil de ser verificada.
- pode ser usada para suportar múltiplas regiões críticas, com cada região crítica definindo sua própria variável.

✰ Algumas desvantagens da abordagem utilizando instruções de máquina:

- espera ocupa é empregada, o que gera consumo de processador;
- possibilidade de ocorrer *starvation*, pois alguns dos processos esperando para entrar na região crítica podem ter que esperar indefinidamente;
- possibilidade de *deadlock*, quando processos com prioridades diferentes estiverem um acessando a região crítica e o outro solicitando a região crítica.

5.4 - Mecanismo de Semáforos

- **idéia:** processos podem se comunicar por meio de sinais, de modo que um processo possa ser obrigado a parar em um dado ponto de seu processamento antes que tenha recebido um sinal em particular, assim, uma coordenação poderá ser estabelecida.

- **semáforo:** é uma variável especial para sinalização que possibilita o envio de um sinal a um processo através da primitiva *signal(s)* e, recebimento de um sinal de um processo através da primitiva *wait(s)*;

- Variável semáforo pode ser vista como uma variável que contém um inteiro sobre o qual 03 operações são definidas e uma fila do semáforo:
  - o semáforo pode ser inicializado com valor negativo;
  - a operação *wait(s)* decrementa o valor do semáforo e caso o valor tenha se tornado negativo o processo executando a operação é bloqueado;
  - a operação *signal(s)* incrementa o valor do semáforo e testa se o valor é 0 ou negativo e em caso afirmativo um processo bloqueado é desbloqueado.

... 5.4 - Mecanismo de Semáforos

- **semáforo:** assume valores positivos, zero e negativos.

```
type semaphore = record                         var s: semaphore;
    count: integer;
    queue: list of process;
  end;

wait( s ):
  s.count := s.count -1;
  if s.count < 0
    then  begin
      place this process in s.queue;
      block this process;
    end;

signal( s ):
  s.count := s.count + 1;
  if s.count <= 0
    then  begin
      remove a process P from s.queue;
      place process P on ready list;
    end;
```

... 5.4 - Mecanismo de Semáforos

✯ **semáforo binário:** assume somente dois valores, 0 e 1.

```
type binary_semaphore = record                    var s: binary_semaphore;
    value: ( 0, 1 );
    queue: list of process;
  end;

waitB( s ):
  if s.value = 1;
    then  s.value := 0;
    else  begin
      place this process in s.queue;
      block this process;
    end;

signalB( s ):
  if s.queue is empty
    then  s.value := 1;
    else  begin
      remove a process P from s.queue;
      place process P on ready list;
    end;
```


... 5.4 - Mecanismo de Semáforos

✯ Exclusão Mútua utilizando Semáforos:

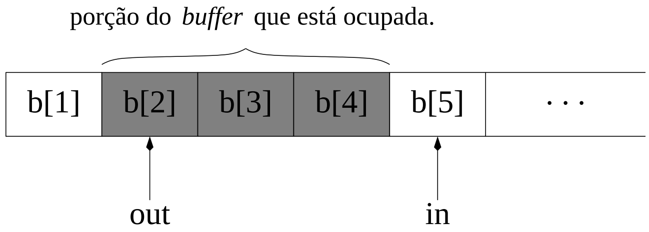
```
program  mutual_exclusion;

const n = ...;  /* number of processes */
var s: semaphore (:= 1);

procedure P( i: integer )
begin
  repeat
    wait( s );
    ... critical region ...
    signal( s );
    ... remainder ...
  forever;
end;

begin  /* Main Program */
  parbegin
    P( 1 );   P( 2 );    ...    P( n );
  parend;
end.
```

5.4.1 - Problema do Produtor/Consumidor

✰ **problema do produtor/consumidor:** há um ou mais produtores gerando algum tipo de dado e armazenando-o em um *buffer* e um único consumidor retirando ítens, um por vez, deste *buffer* e consumindo.

```
producer:
   repeat
      produce item v;
      b[in] := v;
      in := in + 1;
   forever;
```

```
consumer:
   repeat
      while in <= out do { nothing };
      w := b[out];
      out := out + 1;
      consume item w;
   forever;
```



... 5.4.1 - Problema do Produtor/Consumidor

✰ **solução utilizando semáforos:** em vez de rastreamos *in* e *out*, rastreia-se simplesmente o número de elementos no *buffer* ($n = in - out$).

```
var  n: integer;
     s: binary_semaphore( := 1 );
     delay: binary_semaphore( := 1 );
```

```
begin  /* Main Program */
   n := 0;
   parbegin
      producer;     consumer;
   parend
end.
```

```
procedure producer;
begin
   repeat
      produce;
      waitB( s );
      append;
      n := n + 1;
      if n = 1 then signalB( delay );
      signalB( s );
   forever
end;
```

```
procedure consumer;
begin
   waitB( delay );
   repeat
      waitB( s );
      take;
      n := n – 1;
      signalB( s );
      consume;
      if n = 0 then signalB( delay );
   forever
end;
```

... 5.4.1 - Problema do Produtor/Consumidor

- ... há no entanto um erro neste programa - quando o consumidor consome todo o *buffer*, ele necessita reinicializar o semáforo *delay* de modo que ele possa ser forçado a esperar até que o produtor tenha colocado mais ítens no *buffer*.

	Ação	*n*	*delay*
1	Inicialmente	0	0
2	Produtor: região crítica	1	1
3	Consumidor: waitB(delay)	1	0
4	Consumidor: região crítica	0	0
5	Produtor: região crítica	1	1
6	Consumidor: if n = 0 then waitB(delay)	1	1
7	Consumidor: região crítica	0	1
8	Consumidor: if n = 0 then waitB(delay)	0	0
9	Consumidor: região crítica	-1	0

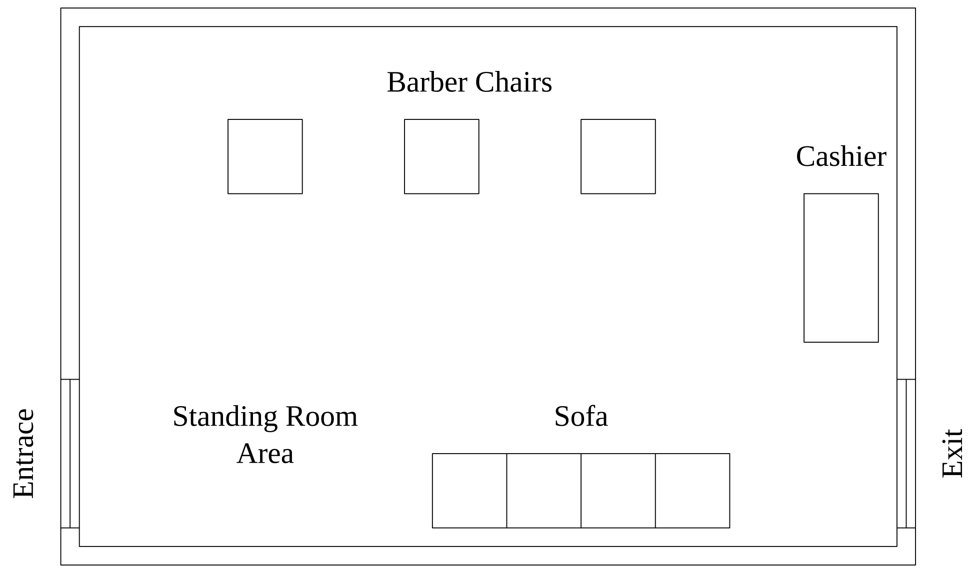
... 5.4.1 - Problema do Produtor/Consumidor

✰ Como mencionado, é imperativo que as operações *wait* e *signal* sejam atômicas, por exemplo, através de *hardware* ou *firmware*.

```
var  n: integer;
     n: semaphore( := 0 );
     s: semaphore( := 1 );

begin  /* Main Program */
   n := 0;
   parbegin
      producer;    consumer;
   parend
end.

procedure producer:
begin
   repeat
      produce;
      wait( s );
      append;
      signal( s );
      signal( n );
   forever
end;

procedure consumer:
begin
   repeat
      wait( n );
      wait( s );
      take;
      signal( s );
      consume;
   forever
end;
```

5.4.2 - Problema da Barbearia

✰ Um outro exemplo do uso de semáforos na implementação de concorrência é o problema encontrado no corte de cabelo em uma barbearia são similares aos encontrados num sistema operacional real.



... 5.4.2 - Problema da Barbearia

✰ Implementação usando semáforos com filas com política *first-in-first-out*.

```
program barbershop1;
var     max_capacity: semaphore (:= 20)
        sofa: semaphore (:= 4);
        barber_chair, coord: semaphore (:= 3);
        cust_ready, finished, leave_b_chair, payment, receipt: semaphore (:= 0);

procedure customer;
begin
  wait( max_capacity );
  enter shop;
  wait( sofa );
  sit on sofa;
  wait( barber_chair );
  get up from sofa;
  signal( sofa );
  sit in barber chair;
  signal( cust_ready );
  wait( finished );
  leave barber chair;
  signal( leave_b_chair );
  pay;
  signal( payment );
  wait( receipt );
  exit shop;
  signal( max_capacity );
end;

procedure barber;
begin
  repeat
    wait( cust_ready );
    wait( coord );
    cut hair;
    signal( coord );
    signal( finished );
    wait( leave_b_chair );
    signal( barber_chair );
  forever;
end;

procedure cashier;
begin
  repeat
    wait( payment );
    wait( coord );
    accept pay;
    signal( coord );
    signal( receipt );
  forever;
end;

begin (* main programa *)
  parbegin
    customer; ... 50 times; ... customer;
    barber; barber; barber;
    cashier;
  parend;
end.
```

... 5.4.2 - Problema da Barbearia

✰ Propósito dos semáforos na solução apresentada:

Semaphore	Wait Operation	Signal Operation
max_capacity	Customer waits for room to enter shop.	Exiting customer signals customer waiting to enter.
sofa	Customer waits for seat on sofa.	Customer leaving sofa signals customer waiting for sofa.
barber_chair	Customer waits for empty barber chair.	Barber signals when that barber's chair is empty.
cust_ready	Barber waits until a customer is in the chair.	Customer signals barber that customer is in the chair.
finished	Customer waits until his haircut is complete.	Barber signals when done cutting hair of this customer.
leave_b_chair	Barber waits until customer gets up from the chair.	Customer signals barber when customer gets up form chair.
payment	Cashier waits for a customer to pay.	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for payment	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to perform either the hair cutting or cashing function.	Signal that a barber resource is free.

... 5.4.2 - Problema da Barbearia

✰ Embora haja um bom esforço na solução apresentada, há um problema de temporização que pode levar um tratamento injusto dos clientes, para tanto, considere o cenário no qual 03 clientes estão sentados nas 03 poltronas:

- ... neste caso, os clientes muito provavelmente serão bloqueados em *wait(finished)* e conforme a organização da fila serão liberados na ordem que entraram;
- ... se, no entanto, um dos barbeiros for mais rápido que os outros, o primeiro cliente será forçado a liberar a poltrona e pagar pelo corte completo, enquanto um outro não poderá deixar a poltrona embora seu cabelo já tenha sido cortado.

✰ O problema pode ser resolvido com a adição de mais semáforos (*homework:* estudar esta solução - Fig.5.21 do Livro Texto).

5.5 - Mecanismo de Monitores

✰ Semáforos provêem uma solução flexível e forte, embora primitiva, para garantir a exclusão mútua e coordenação entre os processos;

- ... entretanto, pode ser difícil produzir um programa correto utilizando semáforos — esta dificuldade decorre do fato de que as operações *wait* e *signal* dispersas no programa dificultam a visualização do efeito (global) esperado no semáforo.

✰ **Monitor:** é uma construção em linguagem de programação que provê funcionalidade equivalente a do semáforo, mas mais fácil de ser controlada.

- ... o conceito de monitores tem sido implementado em um grande conjunto de linguagens, incluindo: Pascal Concorrente, Pascal-plus, Modula-2 e Modula-3.

... 5.5 - Mecanismo de Monitores

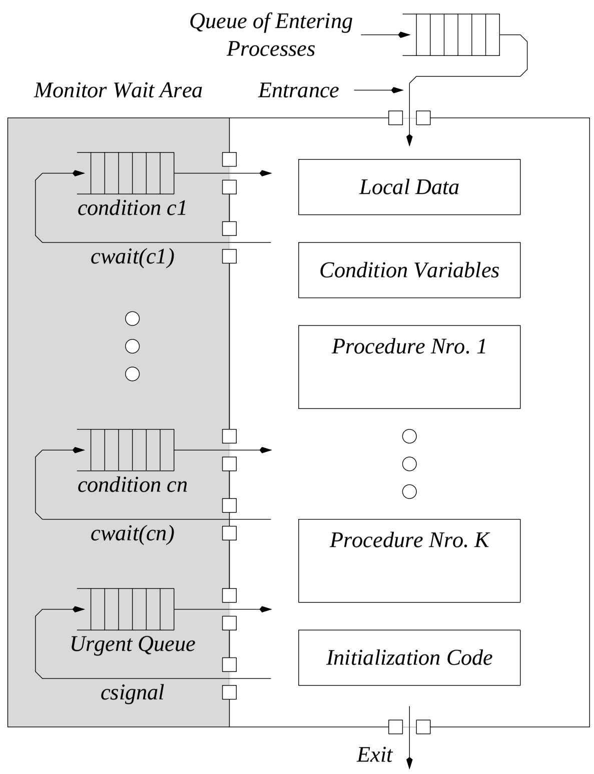
- monitor: é um módulo de *software* consistindo de um ou mais procedimentos, uma seqüência de inicialização e dados locais.

- constituem características do monitor:

  * os dados de variáveis locais são acessíveis somente pelos procedimentos do monitor;

  * um processo tem acesso ao monitor invocando um de seus procedimentos;

  * somente um processo pode executar um monitor em um dado instante, ou seja, qualquer outro processo que invocar o monitor será suspendido.



... 5.5 - Mecanismo de Monitores

✰ Dado que a disciplina de acesso ao monitor permite que apenas um processo o acesse por vez, o monitor garante a exclusão mútua;

- ... as variáveis no monitor podem ser acessadas por um único processo em um dado momento, desde modo, a estrutura de dados compartilhada pode ser protegida;

✰ Monitor oferece mecanismos de sincronização através das variáveis de condição e estando dentro do monitor são acessíveis somente no seu interior:

- *cwait(c)* - suspende a execução do processo que executa esta chamada e torna o monitor disponível para uso por outro processo;
- *csiganl(c)* - recupera a execução de algum processo (p. ex., lista de processos suspensos) suspendido após o *cwait* sobre alguma condição.

- note que as operações *wait/signal* do monitor são diferentes daquelas apresentadas no semáforo - se um processo no monitor invoca uma operação de *signal* e nenhuma tarefa está esperando aquela condição, o sinal é perdido.

... 5.5 - Mec. de Monitores - Prob. do Produtor/Consumidor

```
monitor  bounded_buffer;
   buffer: array[ 0 .. 1 ] of char;        /* space for N items */
   nextin, nextout: integer;               /* buffer pointers */
   count: integer;                         /* number of items in buffer */
   notfull, notempty: condition;           /* for synchronization */

procedure  append( x: char );
begin
   if count = N then cwait(notfull);       /* buffer is full; avoid overflow */
   buffer[nextin] := x;
   nextin := nextin + 1 mod N;
   count := count + 1;                     /* one more item in buffer */
   csignal(notempty);                      /* resume any wait consumer */
end;

procedure  take( x: char );
begin
   if count = 0 then cwait(notempty);      /* buffer is empty; avoid underflow */
   x := buffer[nextout];
   netxout := nextout + 1 mod N;
   count := count –1;                      /* one fewer item in buffer */
   csignal( notfull );                     /* resume any waiting producer */
end;

begin                                      /* Monitor Body */
   nextin := 0;  nextout := 0;  count := 0;/* buffer initially empty */
end;
```

```
begin           /* Main Program */
   parbegin
      producer; consumer;
   parend;
end.

procedure  producer;   /* Procedure Producer */
var  x: char;
begin
   repeat
      produce(x);
      append(x);
   forever;
end;

procedure  consumer;  /* Procedure Consumer */
var  x: char:
begin
   repeat
      take(x);
      consume(x);
   forever;
end;
```

5.6 - Mecanismo de Passagem de Mensagem

✰ Quando processos interagem uns com os outros, 02 requisitos devem ser satisfeitos:

- sincronização: exclusão mútua é garantida através da sincronização;
- comunicação: cooperação de processos exige troca de informações.

✰ **passagem de mensagem:** é uma abordagem que provê ambas funcionalidades e, adicionalmente, possibilita seu uso em sistemas distribuídos bem como em sistemas mono e multiprocessados com memória compartilhada;

- ... abordagem passagem de mensagem pode aparecer em várias formas, mas normalmente é oferecida como um par de primitivas: *send(destination, message)* e *receive(source, message)*.

- ... este conjunto é o conjunto mínimo de operações necessárias para que processos possam utilizar a abordagem de passagem de mensagens.

... 5.6 - Mecanismo de Passagem de Mensagem

✰ Dentre as características de projeto em sistemas que empregam o mecanismo de passagem de mensagens na sincronização e comunicação entre processos, destacamos:

❶ **sincronização:** *send* bloqueante e não bloqueante; *receive* bloqueante, não bloqueante, com teste para ver se há alguma mensagem esperando;
❷ **endereçamento:** forma de especificar quem estará recebendo a mensagem, onde dois esquemas são possíveis: endereçamento direto e indireto;
❸ **formato:** pode ser orientado a conteúdo ou atreladas ao tamanho, podendo serem mensagens de tamanho fixo ou variável;
❹ **disciplina na fila de espera:** mensagens podem ser enfileiradas segundo uma lista FIFO (*First In First Out*) ou por alguma outra política;

- ... por prioridade, caso em que algumas mensagens são mais urgentes que outras;
- ... ou permitir que o receptor inspecione a fila de mensagens e selecione qual será a próxima mensagem a ser recebida com base em algum outro critério.

5.7 - Problema dos Leitores/Escritores

✰ **caracterização:** há uma área compartilhada (p. ex., bloco da memória principal, um arquivo ou até mesmo um banco de registradores) entre um número de processos, alguns deles lendo e outros apenas escrevendo na área de dados.

✰ As seguintes condições devem ser satisfeitas:

❶ qualquer número de leitores pode simultaneamente ler o arquivo;
❷ somente um escritor por vez pode escrever no arquivo;
❸ se um escritor está escrevendo no arquivo, nenhum leitor pode lê-lo.

✰ !? podemos considerar o problema produtor/consumidor como um caso especial do problema dos leitores/escritores com um único escritor (produtor) e um único leitor (consumidor) !? ... justifique ...

... 5.7 - Problema dos Leitores/Escritores

✰ Apresentamos soluções que descrevem uma instância para cada leitor e escritor do problema dos leitores/escritores e que utilizam-se dos semáforos para garantir a exclusão mútua.

✰ Duas soluções serão analisadas para este problema:

❶ leitores têm prioridade;
❷ escritores têm prioridade.

... 5.7 - Problema dos Leitores/Escritores

✰ Na solução, leitores com prioridade, possível que os leitores mantenham controle da área de dados enquanto um leitor estiver lendo.

```
program  readers_and_writers;
   var  readcount: integer;
   x, wsem: semaphore (:=1);

procedure  reader;    /* Reader Procedure */
begin
   repeat
      wait(x);
         readcount := readcount + 1;
         if readcount = 1 then wait(wsem);
      signal(x)
      READ_UNIT;
      wait(x);
         readcount := readcount  – 1;
         if  readcount = 0  then  signal(wsem);
      signal(x);
   forever;
end;
```

```
begin                    /* Main Program */
   readcount := 0;
   parbegin
      reader; writer;
   parend;
end.

procedure  writer;     /* Writer Procedure */
begin
   repeat
      wait(wsem);
         WRITE_UNIT;
      signal(wsem);
   forever;
end;
```


... 5.7 - Problema dos Leitores/Escritores

✰ Na solução, escritores com prioridade, nenhum leitor terá permissão para ler dados se ao menos um escritor manifestou o intenção de alterá-los.

```
program  readers_and_writers;
   var  readcount, writecount: integer;
   x, y, z, wsem, rsem: semaphore (:=1);

procedure  reader;    /* Reader Procedure */
begin
   repeat
      wait( z );
         wait( rsem );
            wait( x );
               readcount := readcount + 1;
               if readcount = 1 then wait(wsem);
            signal(x)
         signal( rsem );
      signal( z );
      READ_UNIT;
      wait( x );
         readcount := readcount  – 1;
         if  readcount = 0  then  signal( wsem );
      signal( x );
   forever;
end;
```

```
begin       /* Main Program */
   readcount := 0;
   parbegin
      reader;   writer;
   parend;
end.

procedure  writer;     /* Writer Procedure */
begin
   repeat
      wait( y );
         writecount := writecount – 1;
         if  writecount = 1  then  wait( rsem );
      signal( y );
      wait( wsem );
      WRITE_UNIT;
      signal( wsem );
      wait( y );
         writecount := writecount –1;
         if  writecount = 0  then  signal( rsem );
      signal( y );
   forever;
end;
```