# Sistemas Concorrentes e Distribuídos

## Monitores



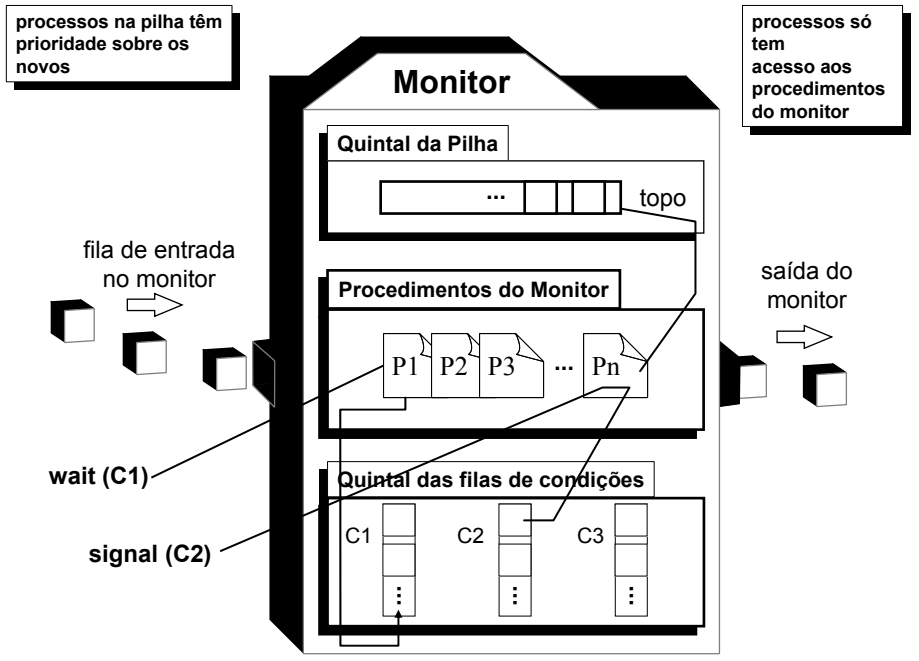
## Sincronização - Soluções por Software

- **Brich Hansen e Hoare**
- **Monitores**
  - **solução estruturada**
  - **concentra partes críticas de sincronização**
  - **funções e procedimentos declarados pelo programador estão protegidas e encapsuladas dentro do monitor**
  - **além das funções o monitor encapsula dados**
  - **procedimentos dentro do monitor são executados em exclusão mútua**
  - **o monitor possui uma fila (FIFO) de entrada onde, processos desejando executar um procedimento do monitor, aguardam sua vez**

# Monitores: programação

- **Variável de Condição (um tipo primitivo dos monitores, na verdade são os nomes de uma fila FIFO)**
- **dois procedimentos e uma função atuam sobre a variável de condição**
  - **wait (condição)**
    - » **processo é suspenso e vai para o fim da fila "condição"**
  - **signal (condição)**
    - » **primeiro processo suspenso na fila "condição" retorna a sua execução**
  - **NonEmpty (condição)**
    - » **TRUE se a fila "condição" não estiver vazia**
- **semântica das funções wait e signal dos monitores é diferente do semáforo (não tem memória)**



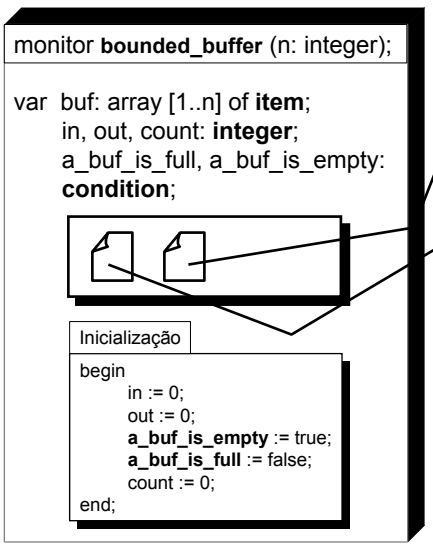


# Monitores: Interface de Programação

```
monitor exemplo;
    < declara variáveis locais ao monitor>;
procedure entry proc_name_1;
Begin
    <corpo do procedimento 1>
End;
procedure entry proc_name_2;
Begin
    <corpo do procedimento 2>;
End;
...
Begin
    < inicialização das variáveis do monitor >;
End exemplo;
```



# Produtores/Consumidores com Monitor




- encapsulamento
- exclusão mútua

```
proc entry insert (it: item);
begin
    if count = n then a_buf_is_empty.wait;
    proc entry remove (it: item);
    begin
        if count = 0 then a_buf_is_full.wait;
        it := buf [out];
        out := (out mod n) + 1;
        count := count -1;
        a_buf_is_empty.signal;
    end;
end;
```

Tarefa produtor:
**bounded_buffer**.insert (p);

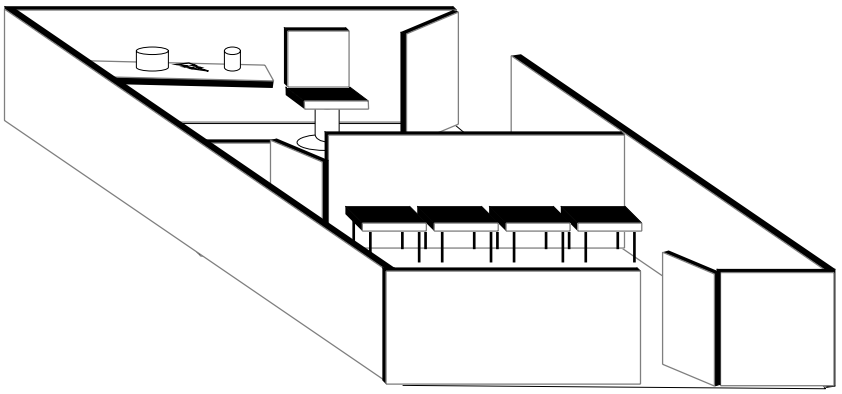
Tarefa consumidor:
**bounded_buffer**.remove (p);

# Monitores e Semáforos

- **podemos simular semáforos do tipo FIFO com monitores**
  - **exercício para turma**
- **podemos construir um monitor com semáforos**



# Barbeiro Dorminhoco



# Barbeiro Dorminhoco

- A barbearia consiste numa sala de espera com n cadeiras mais a cadeira do barbeiro
- Se não existirem clientes o barbeiro fica a dormir
- Ao chegar um cliente:
  - » Se todas as cadeiras estiverem ocupadas, este vai-se embora
  - » Se o barbeiro estiver ocupado, mas existirem cadeiras então o cliente senta-se e fica à espera
  - » Se o barbeiro estiver a dormir o cliente acorda-o e corta o cabelo



# SOLUÇÃO: Barbeiro Dorminhoco

**Barbeiro**

```
do {
   wait(cliente); //barbeiro a dormir
   ...
   Corta cabelo
   ...
   signal(barbeiro)
 } while (1)
```

**Cliente**

```
wait(mutex)
if (lugares_vazios==0)
    exit();
else
    lugares_vazios--;
signal(mutex);
signal(cliente);
wait(barbeiro); //corta cabelo
wait(mutex);
lugares_vazios++;
signal(mutex);
```

# Leitores e Escritores

- **vários processos escritores e leitores competindo por um recurso comum**
- **O acesso ao recurso comum por um escritor deve ser feito em exclusão mútua**
- **entre leitores, exclusão mútua não é exigida**
- **se há um escritor escrevendo, escritores esperando para escrever e leitores esperando para ler**
  - » **o primeiro escritor na fila não começará a a escrever até que todos os leitores que estiverem esperando o escritor escrever tenham acabado de ler**
  - » **se há leitores lendo**
  - – **chega primeiro um escritor para esperar para escrever**
  - – **novos leitores que chegarem ficarão suspensos**
- **se há leitores lendo, escritores esperando e leitores esperando**
  - – **todos os leitores esperando para ler não vão começar a ler até que o primeiro escritor que está esperando para escrever acabe de escrever**



# Escr/Leit: monitor

```
program LeitoresEscritores;

monitor MonitorLeitoresEscritores;
var
  Leitores : integer;
  Escrevendo : boolean;
  PodeLer, PodeEscrever : condition;

procedure ComecarALer;
begin
  if (Escrevendo or
     NonEmpty(PodeEscrever)) then
     wait(PodeLer);
  Leitores:=Leitores+l;
  signal (PodeLer);
end;

procedure AcabarDeLer;
begin
  Leitores:=Leitores-1;
  if Leitores=0 then
     signal(PodeEscrever);
end;

procedure ComecarAEscrever;
begin
  if ((Leitores<>0) or
     Escrevendo) then
     wait(PodeEscrever);
  Escrevendo:=true;
end;

procedure AcabarDeEscrever;
begin
  escrevendo := false;
  if NonEmpty(PodeLer) then
     signal(PodeLer)
  else
     signal (PodeEscrever);
end;

begin
 Leitores:=0;
 Escrevendo:=false;
 end;
```

```
procedure Leitor;
begin
 repeat
  ComecarALer;
  LerDado;
  AcabarDeLer;
  ConsumirDado;
 until false;
end;

procedure Escritor;
begin
 repeat
  ProduzirDado;
  ComecarAEscrever;
  EscreverDado;
  AcabarDeEscrever;
 until false;
end;

Begin
 cobegin
   Leitorl ; Escritorl; ...;
 coend;
end.
```

# Sincronização

- Troca de Mensagens
- send / receive
- endereçamento
  - direto
  - caixa postal
- API
  - System V IPC Messages
  - sockets
  - RPC
  - CORBA

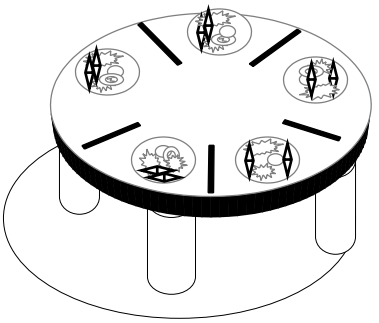


# Verificação de Programas Concorrentes

- Provas formais de corretude
- lógica temporal
- linguagens que permitem verificações formais de propriedades
  - ADA (Rendevouz)
  - CCS
  - CSP
  - Estelle
  - Lotus
  - Redes de Petri
  - Pi Calculus

# Problema dos Filósofos

- problema clássico de exclusão mútua
- filósofos pensam e comem em um tempo arbitrário
- para comer: 2 pauzinhos
- evitar
  - starvation (esfomeação)
  - deadlock
- soluções
  - semáforo
  - monitor





# SOLUÇÃO POR MONITOR

**Pascal**

```
type dining-philosophers = monitor
  var state : array [0..4] of :(thinking, hungry, eating);
  var self : array [0..4] of condition;
  procedure entry pickup (i: 0..4);
    begin
              state[i] := hungry,
              test (i);
              if state[i] != eating then self[i], wait,
    end;
  procedure entry putdown (i: 0..4);
    begin
              state[i] := thinking;
              test (i+4 mod 5);
              test (i+1 mod 5);
    end;
  procedure test(k: 0..4);
    begin
              if state[k+4 mod 5] != eating
              and state[k] = hungry
              and state[k+1 mod 5] ] != eating
              then begin
                  state[k] := eating;
                  self[k].signal;
              end;
    end;
begin
              for i := 0 to 4
              do state[i] := thinking;
end.
```

**JAVA**

```
monitor DiningPhilosophers {
  int state = new int [5];
  static final int thinking = 0;
  static final int hungry= 1;
  static final int eating= 2;
  condition[] self = condition[5];
  public dinningPhilosophers {
              for(int i=0;i<5;i++)
                  state[i] = thinking;

  public entry pickup (int i) {
              state[i] := hungry;
              test (i);
              if (state[i] !=eating)
                  self[i].wait;
   }
   public entry putdown (int i) {
              state[i] := thinking;
              test (i+4 mod 5);
              test (i+1 mod 5);
   }
   private test(int i) {
              if (state[k+4 mod 5] != eating)
              && (state[k] == hungry)
              && (state[k+1 mod 5] != eating) {
                  state[k] := eating;
                  self[k].signal;
              }
   }
}
```