



Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos (4)

Monitores



Universidade Federal do Espírito Santo
Departamento de Informática

Monitores (1)

- Sugeridos por Dijkstra (1971) e desenvolvidos por Hoare (1974) e Brinch Hansen (1975), são estruturas de sincronização de alto nível, que têm por objetivo impor (forçar) uma boa estruturação para programas concorrentes.
- Motivação:
 - Sistemas baseados em algoritmos de exclusão mútua ou semáforos estão sujeitos a erros de programação. Embora estes devam estar inseridos no código do processo, não existe nenhuma reivindicação formal da sua presença. Assim, erros e omissões (deliberadas ou não) podem existir e a exclusão mútua pode não ser atingida.

Monitores (2)

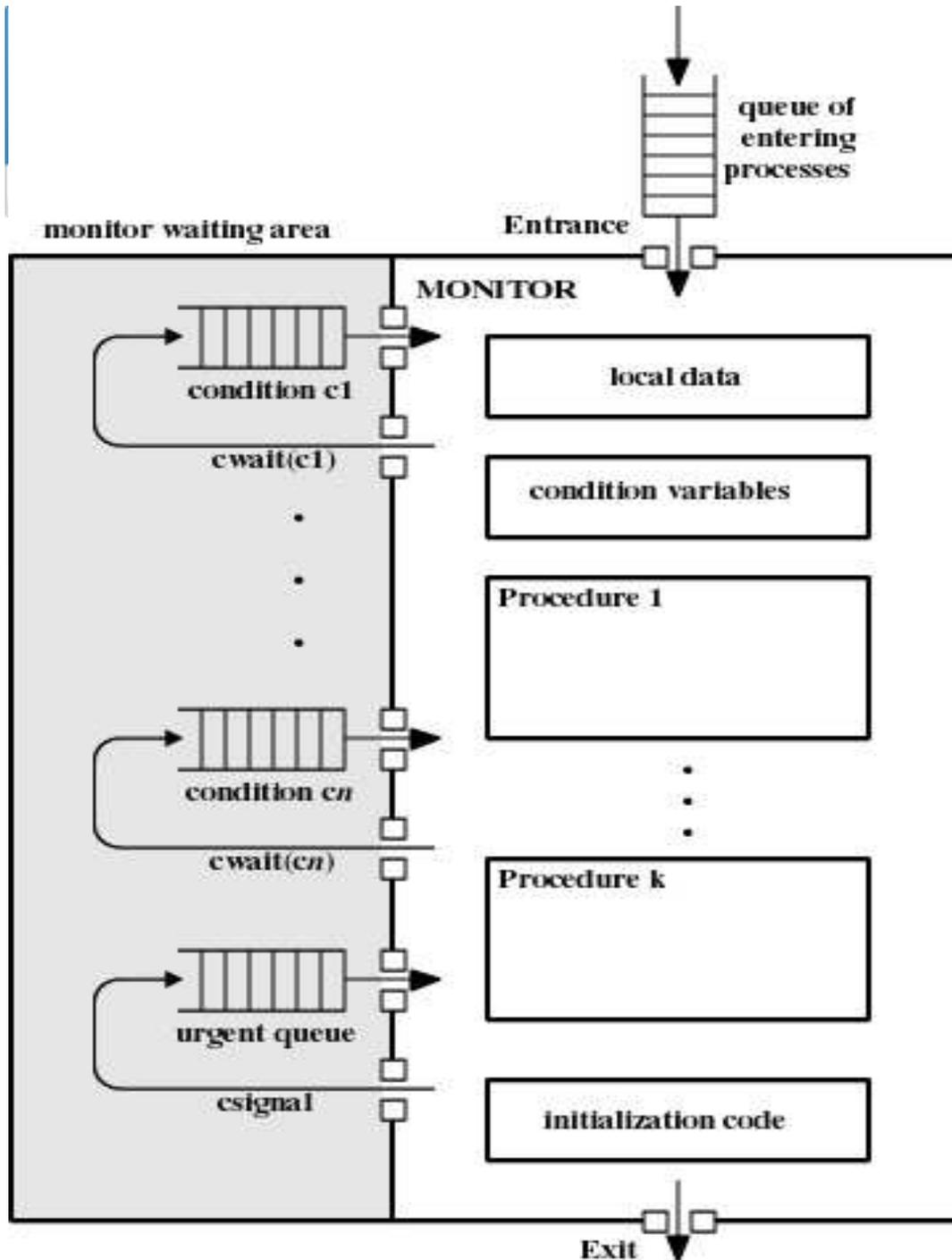
- Solução:
 - Tornar obrigatória a exclusão mútua. Uma maneira de se fazer isso é colocar as seções críticas em uma área acessível somente a um processo de cada vez.
- Idéia central:
 - Em vez de codificar as seções críticas dentro de cada processo, podemos codificá-las como procedimentos (*procedure entries*) do monitor. Assim, quando um processo precisa referenciar dados compartilhados, ele simplesmente chama um procedimento do monitor.
 - Resultado: o código da seção crítica não é mais duplicado em cada processo.

Monitores (3)

- Um monitor pode ser visto como um bloco que contém internamente *dados* para serem compartilhados e *procedimentos* para manipular esses dados.
- Os dados declarados dentro do monitor são compartilhados por todos os processos, mas só podem ser acessados através dos procedimentos do monitor, isto é, a única maneira pela qual um processo pode acessar os dados compartilhados é indiretamente, por meio das *procedure entries*.

Monitores (4)

- As *procedure entries* são executadas de forma mutuamente exclusiva. A forma de implementação do monitor já garante a exclusão mútua na manipulação dos seus dados internos.
- Monitor é um conceito incluído em algumas linguagens de programação:
 - Módulo, Pascal Concorrente, Euclid Concorrente, Java.



Visão da Estrutura de um Monitor

Processo P1

Begin

...

Chamada a um procedimento do monitor

...

End

Processo P2

Begin

...

Chamada a um procedimento do monitor

...

End

Processo P3

Begin

...

Chamada a um procedimento do monitor

...

End

Chamada de procedimento do Monitor

```
MONITOR <NomeDoMonitor>;
```

Declaração dos dados a serem compartilhados pelos processos (isto é, das variáveis globais acessíveis a todos procedimentos do monitor);

Exemplos:

```
X,Y: integer;
```

```
C, D: condition;
```

```
Entry Procedimento_1(Argumentos_do_Procedimento_1)
  Declaração das variáveis locais do Procedimento_1
  Begin
    ...
    Código do Procedimento_1 (ex: X:=1; wait(C))
    ...
  End
```

```
Entry Procedimento_N(Argumentos_do_Procedimento_N)
  Declaração das variáveis locais do Procedimento_N
  Begin
    ...
    Código do Procedimento_N (ex: Y:=2; signal(C))
    ...
  End
```

```
BEGIN
```

```
...
```

```
Iniciação das variáveis globais do Monitor
```

```
...
```

```
END
```

Formato de um Monitor

Variáveis de Condição (1)

- São variáveis que estão associadas a condições que provocam a suspensão e a reativação de processos. Permitem, portanto, sincronizações do tipo *sleep-wakeup*.
- Só podem ser declaradas dentro do monitor e são sempre usadas como argumentos de dois comandos especiais:
 - *Wait* (ou *Delay*)
 - *Signal* (ou *Continue*)

Variáveis de Condição (2)

- *Wait (condition)*

- Faz com que o monitor suspenda o processo que fez a chamada. O monitor armazena as informações sobre o processo suspenso em uma estrutura de dados (fila) associada à variável de condição.

- *Signal (condition)*

- Faz com que o monitor reative UM dos processos suspensos na fila associada à variável de condição.

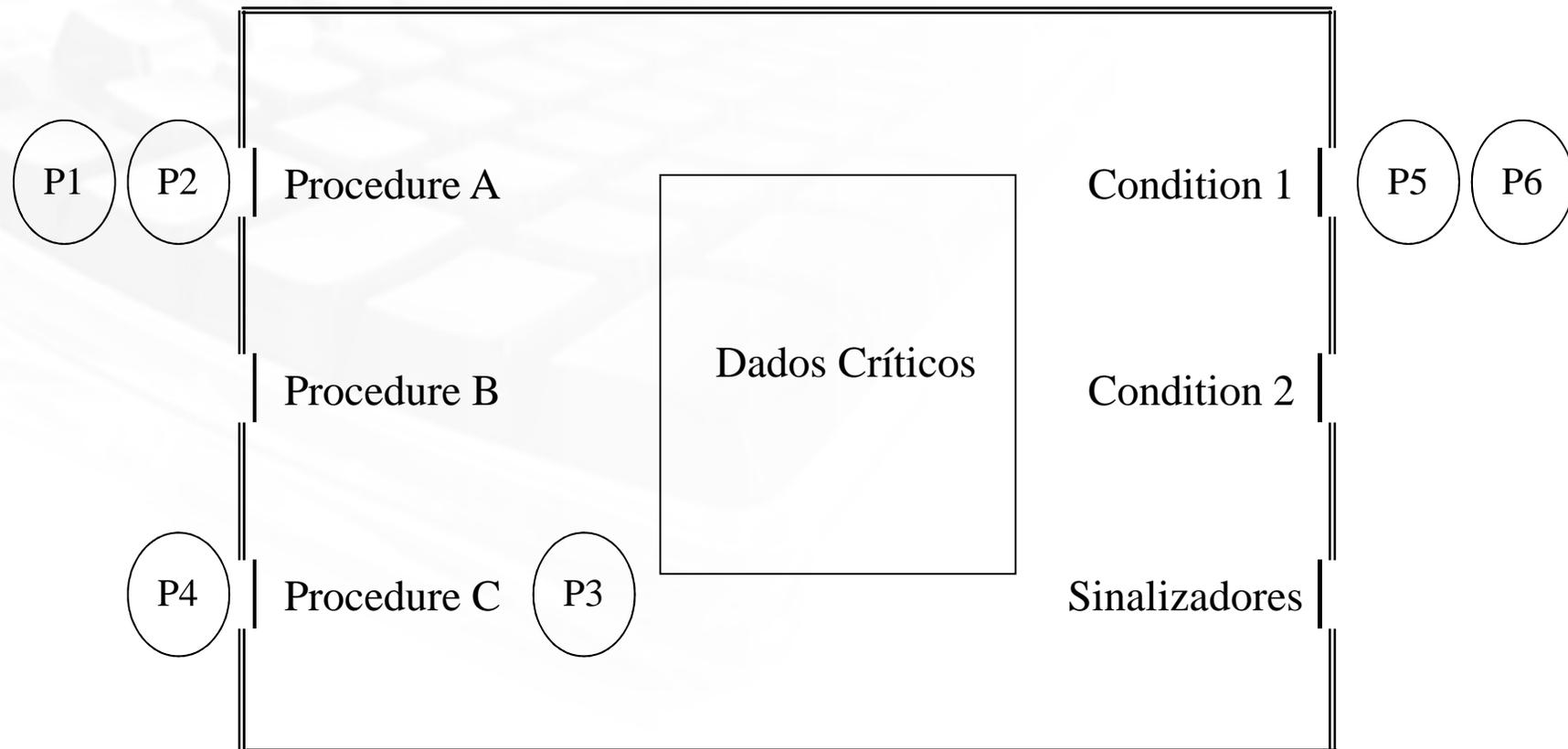
Variáveis de Condição (3)

- O que acontece após um *Signal (condition)*?
 - *Hoare* propôs deixar o processo Q recentemente acordado executar, bloqueando o processo P sinalizador. P deve esperar em uma fila pelo término da operação de monitor realizada por Q .
 - Fila de Sinalizadores
 - *Brinch Hansen* propôs que o processo P conclua a operação em curso, uma vez que já estava em execução no monitor (i.e., Q deve esperar). Neste caso, a condição lógica pela qual o processo Q estava esperando pode não ser mais verdadeira quando Q for reiniciado.
 - Simplificação: o comando *signal* só pode aparecer como a declaração final em um procedimento do monitor.

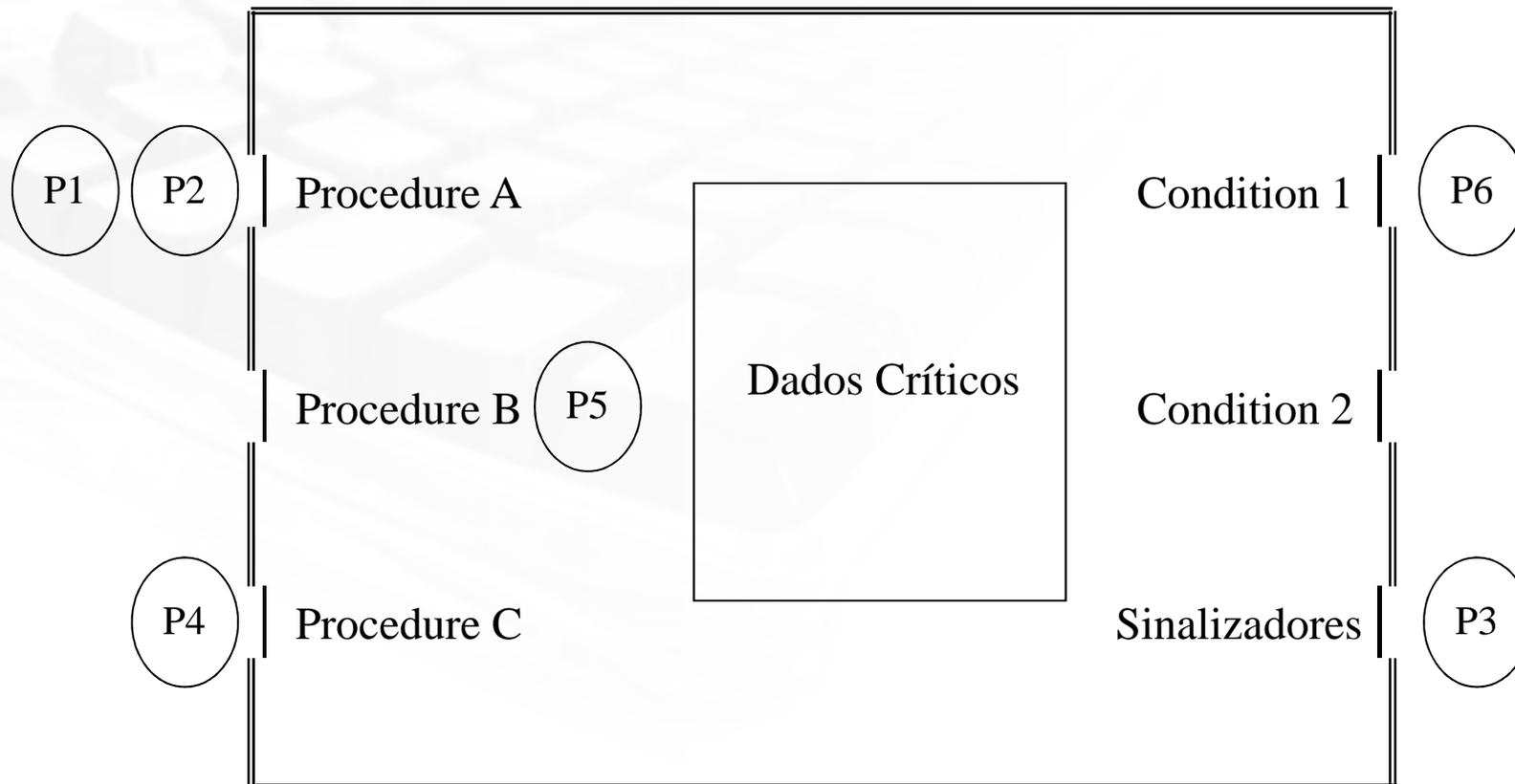
Variáveis de Condição (4)

- A linguagem Concurrent Pascal adota um meio-termo entre essas duas possibilidades:
 - Quando P executa um *signal*, a operação do monitor que ele estava executando termina imediatamente, sendo a execução de Q imediatamente reiniciada.
 - Nesta solução, um processo não pode realizar duas operações *signal* durante a execução de uma chamada de procedimento de monitor (ou seja, é uma solução menos poderosa que a proposta por *Hoare*).

Exemplo (Abordagem de *Hoare*)



Exemplo (Abordagem de *Hoare*) (cont.)



Problema do Produtor-Consumidor

```

monitor ProducerConsumer
  condition full, empty;
  integer count;

procedure entry enter
begin
  if count = N then wait(full);
  enter_item;
  count := count + 1;
  if count = 1 then signal(empty)
end;
procedure entry remove
begin
  if count = 0 then wait(empty);
  remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
end;

count := 0;
end monitor;

```

```

//Processo Produtor
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end;

//Processo Consumidor
procedure consumer;
begin
  while true do
  begin

ProducerConsumer.remove;
    consume_item
  end
end;

```



Produtor-Consumidor com Buffer Circular

```

Monitor buffercircular;
  buffer matriz(0..n) of "coisa";
  i: integer;
  j: integer;
  buffcheio: condition;
  buffvazio: condition;
  ocupado: integer;

```

```

Procedure Entry Coloca(AlgumDado: coisa)

```

```

Begin
  if ocupado = n then wait(buffcheio);
  buffer[j] := AlgunDado;
  j := (j+ 1) MOD n ;
  ocupado:= ocupado + 1;
  signal(buffvazio);
End

```

```

Procedure Entry Retira(AlgumDado: coisa)

```

```

Begin
  if ocupado = 0 then wait(buffvazio);
  remove AlgunDado de buffer[i];
  i := (i+ 1) MOD n ;
  ocupado:= ocupado - 1;
  signal(buffcheio);
End

```

```

Begin
  i := 0; j :=0; ocupado := 0
End

```

```

Processo Produtor;
Begin
  ...
  Coloca(AlgumDado)
  ...
End

```

```

Processo Consumidor;
Begin
  ...
  Retira(AlgumDado)
  ...
End

```

Filósofos Glutões

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)           // prox. slide
    void putdown(int i)         // prox. slide
    void test(int i)            // prox. slide
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```

Filósofos Glutões (cont.)

```
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}
```

```
void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
}
```

```
void test(int i) {
    if ( (state[i] == hungry) &&
        (state[(I + 4) % 5] != eating) &&
        (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}
```

```
void take_forks(int i)
{ down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]); }

void put_forks(i)
{ down(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(&mutex); }

void test(i)
{ if (state[i] == HUNGRY &&
     state[LEFT] != EATING &&
     state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

Implementando Monitores usando Semáforos

- Variáveis

```
semaphore mutex; // (inicialmente = 1)
//Para implementar a fila de sinalizadores
semaphore next; // (inicialmente = 0)
int next-count = 0;
```

- Cada *entry procedure* F será implementada da seguinte forma

```
down(mutex);
...
body of  $F$ ,
...
if (next-count > 0)
up(next)
else
up(mutex);
```

Implementando Monitores usando Semáforos (cont.)

- Para cada variável de condição, temos:

```
semaphore x-sem; // (inicialmente = 0)  
int x-count = 0;
```

- As operações *wait* e *signal* podem ser implementadas da seguinte forma:

```
//wait  
x-count++;  
if (next-count > 0)  
    up(next);  
else  
    up(mutex);  
    down(x-sem);  
    x-count--;
```

```
//signal  
if (x-count > 0) {  
    next-count++;  
    up(x-sem);  
    down(next);  
    next-count--;  
}
```

Referências

- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 2a. Edição, Editora Prentice-Hall, 2003.
 - Seções 2.3.7
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
 - Seção 7.7
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; "Sistemas Operacionais", 3ª. Edição, Editora Prentice-Hall, 2005
 - Seções 6.2 e 6.3
- Monitores em Java
 - Link prog concorrente em java
 - <http://www.mcs.drexel.edu/~shartley/ConcProgJava/monitors.html>
 - Capitulo sobre java monitors
 - <http://java.sun.com/developer/Books/performance2/chap4.pdf>