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.

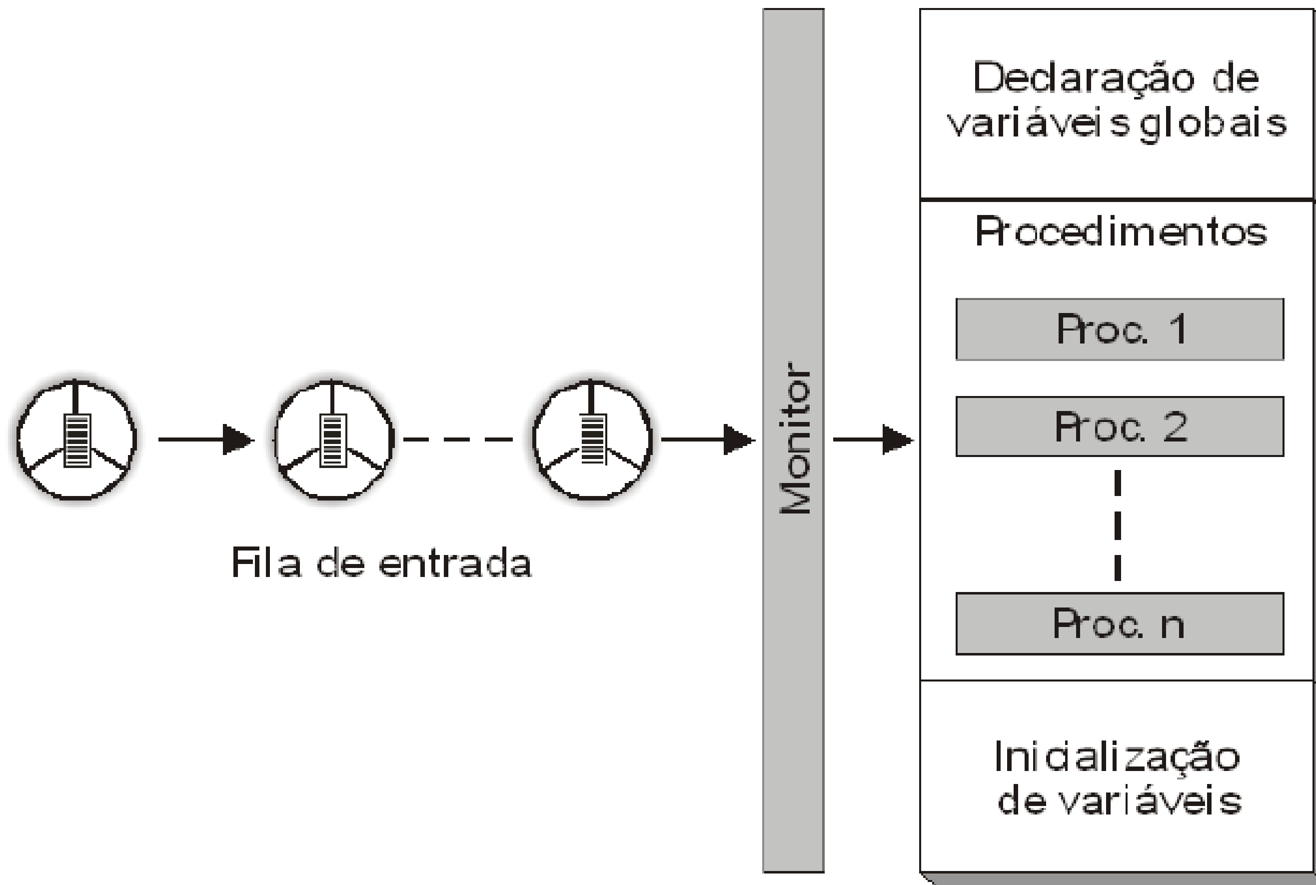
Monitores (3)

- 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 por meio 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:
  - Módula, Pascal Concorrente, Euclid Concorrente,Java.

Visão da Estrutura de um Monitor



```
Processo P1
   Begin
   ...
   myMonitor.proc1(...)
   ...
   End
```

```
Processo P2
   Begin
   ...
      myMonitor.proc2(...)
   ...
   End
```

```
Processo P3
   Begin
   ...
   myMonitor.proc1(...)
   ...
   End
```

Chamada de Procedimentos do Monitor

Implementação de um 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 proc1(Argumentos_do_proc1)
  Declaração das variáveis locais do proc_1
  Begin
    ...
    Código do proc_1 (ex: X:=1; wait(C);)
    ...
  End
...
Entry procN(Argumentos_do_procN)
  Declaração das variáveis locais do procN
  Begin
    ...
    Código do procN (ex: Y:=2; signal(C);)
    ...
  End

BEGIN
...
Iniciação das variáveis globais do Monitor
...
END
```

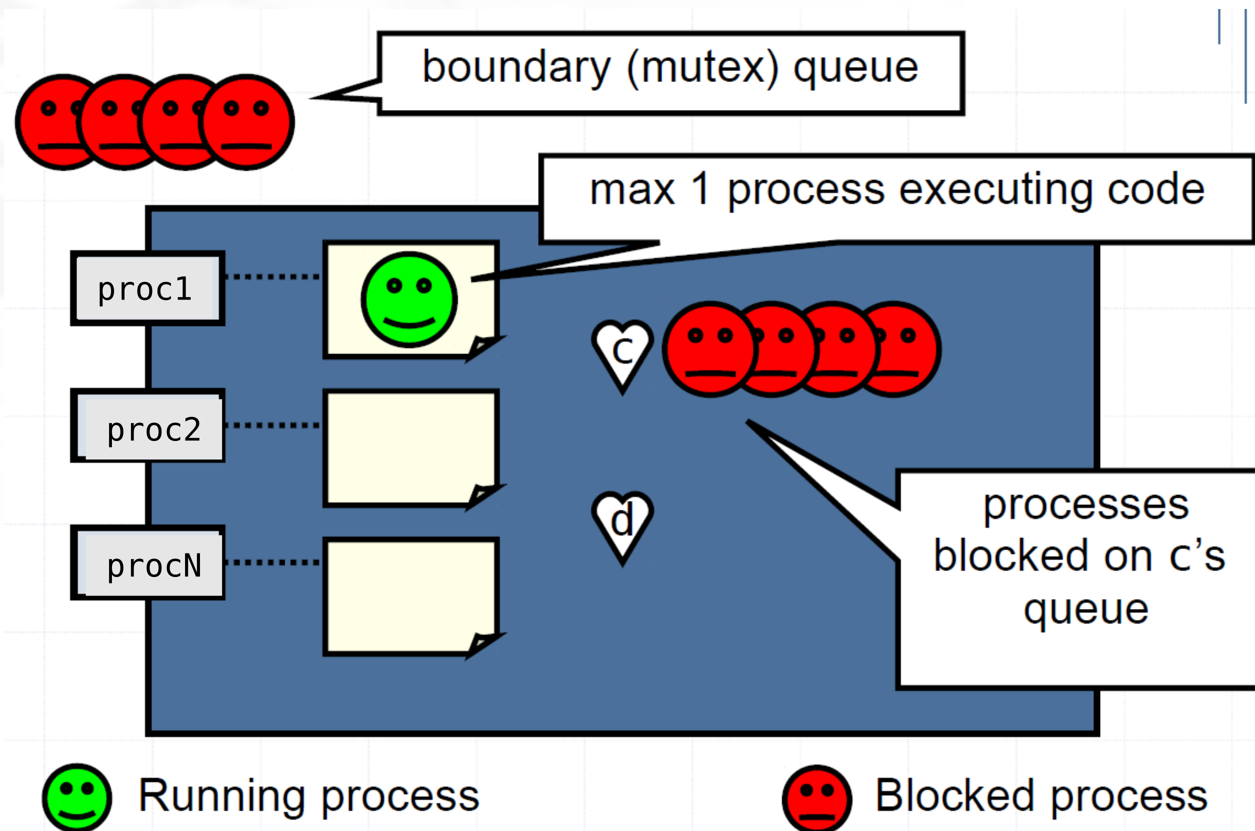
Variáveis de Condição (1)

- São variáveis que estão associadas a condições que provocam a suspensão e a reativação de processos. Permitem, portanto, sincronizações do tipo *sleep-wakeup.*
- Em geral, são declaradas dentro do monitor e são sempre acessadas por meio de dois comandos especiais:
  - *Wait* (ou *Delay*)
  - *Signal* (ou *Continue*)

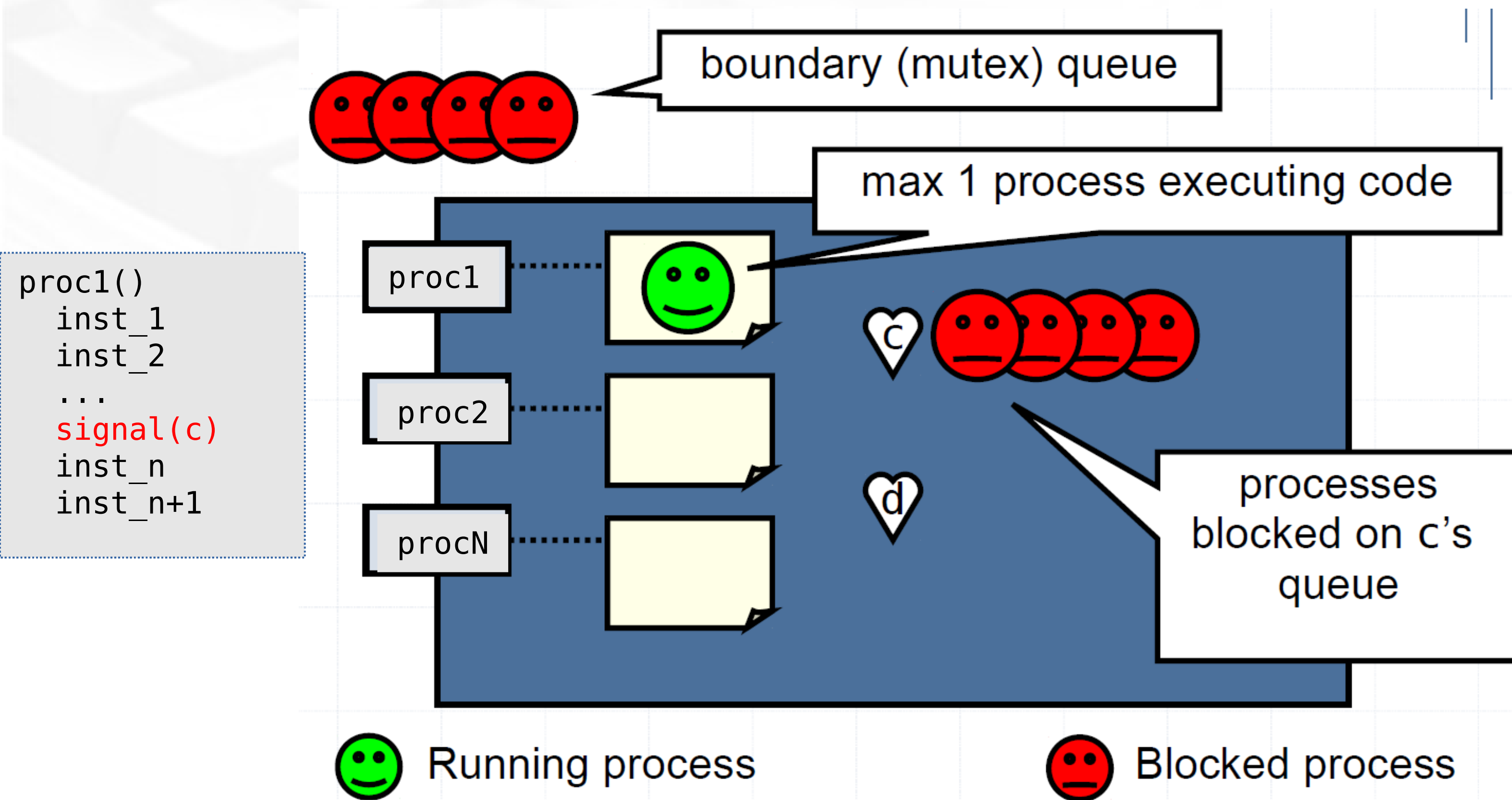
Variáveis de Condição (2)

- *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 quer o monitor reative UM dos processos suspensos na fila associada à variável de condição.

Visão da Estrutura de um Monitor e Suas Variáveis de Condição



Visão da Estrutura de um Monitor e Suas Variáveis de Condição



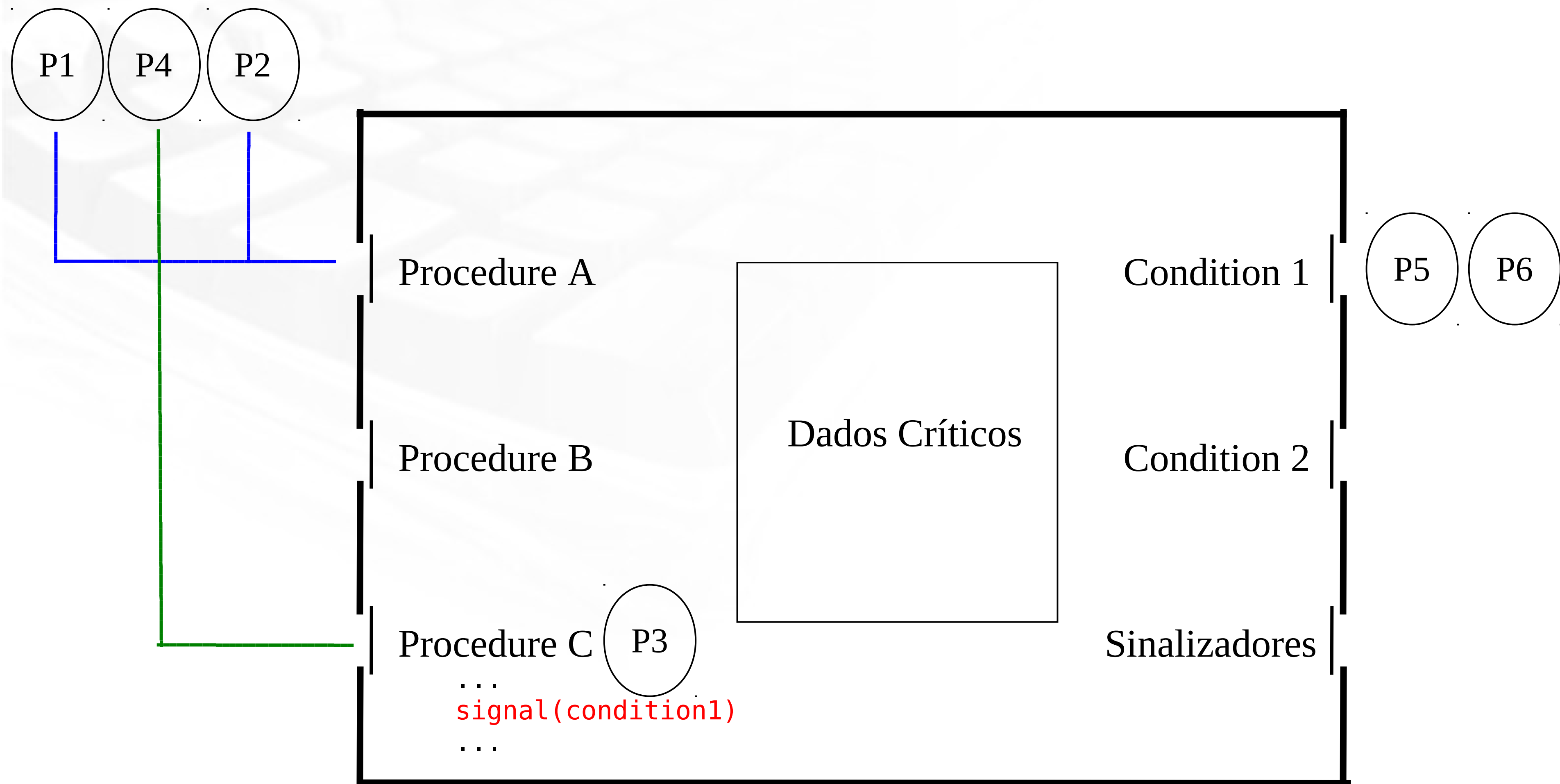
Variáveis de Condição (3)

- 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.

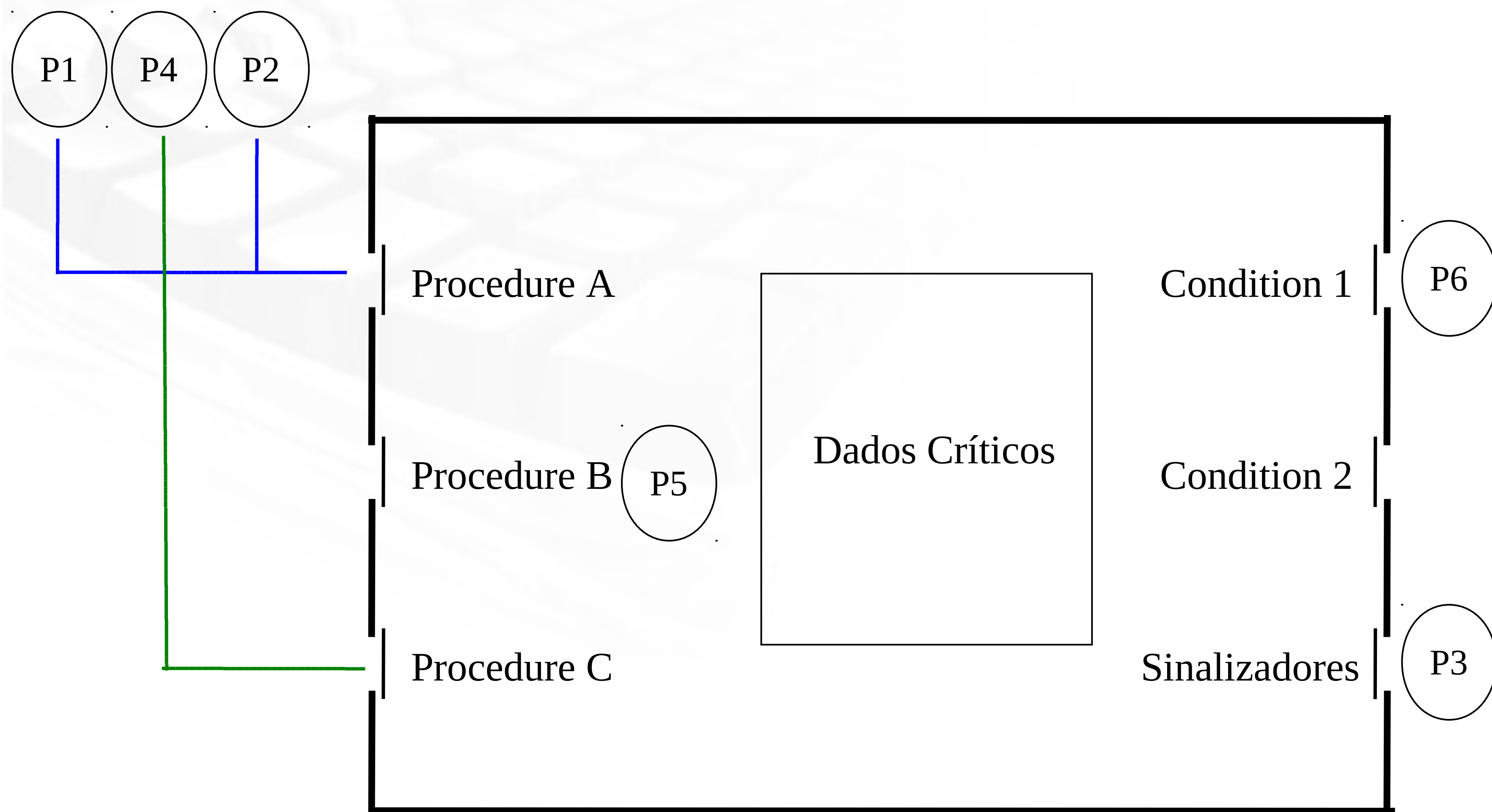
Variáveis de Condição (4)

- 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 (recentemente acordado) 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*).

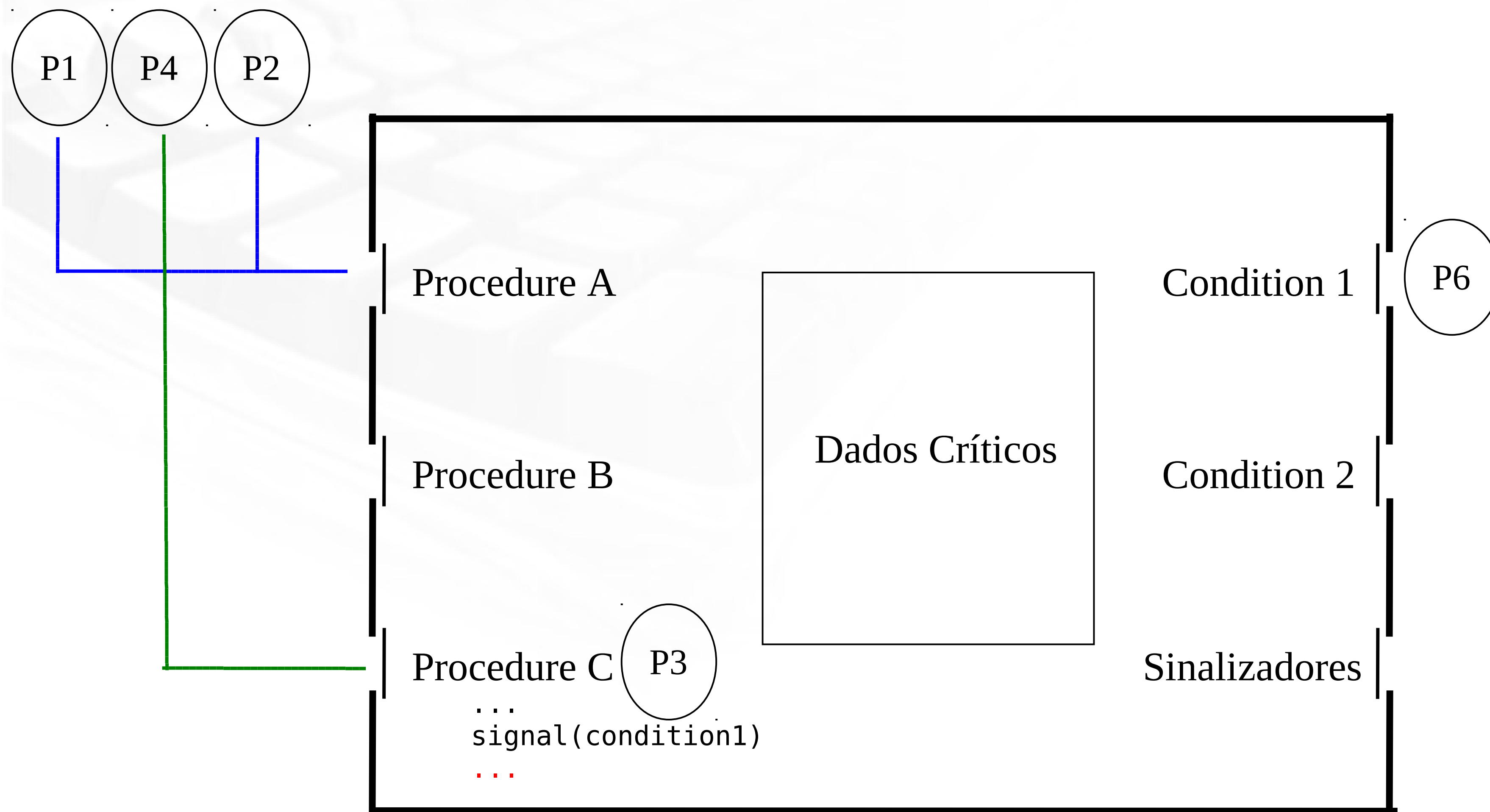
Exemplo (Abordagem de *Hoare*)



Exemplo (Abordagem de *Hoare*)



Exemplo (Abordagem de *Hoare*)



LPRM

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] := AlgumDado;
  j := (j+ 1) MOD n ;
  ocupado:= ocupado + 1;
  signal(bufvazio);
End

Procedure Entry Retira(AlgumDado: coisa)
Begin
  if ocupado = 0 then wait(buffvazio);
  remove AlgumDado 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, semaf. next
  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;
```

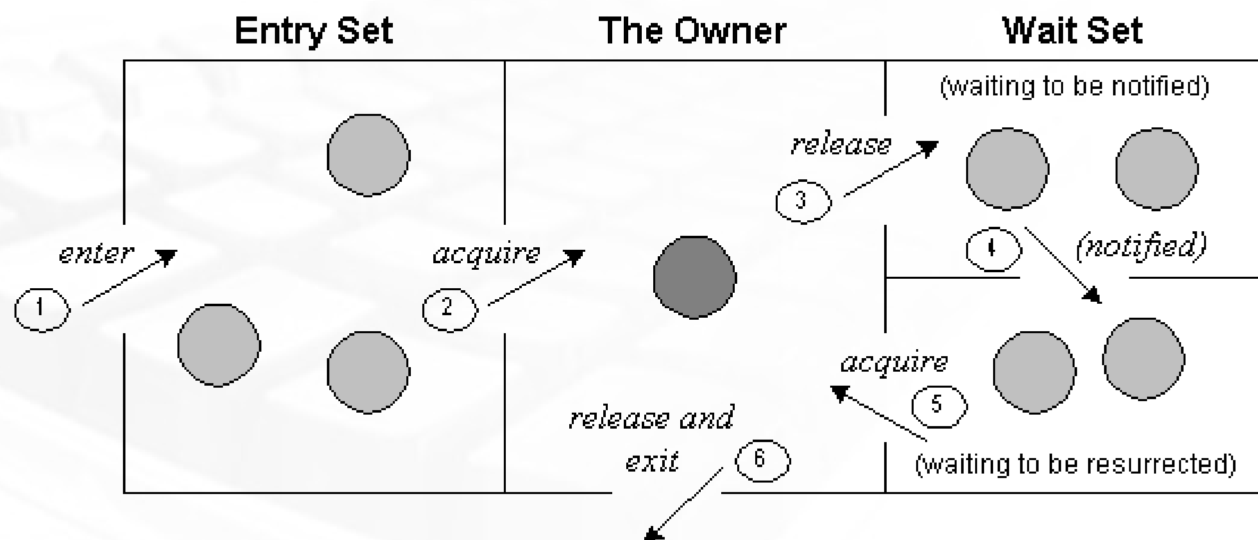
- As operações *wait* e *signal* podem ser implementadas da seguinte forma:

```
//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--;
}
```

Baseada na abordagem de *Hansen*, mas sem garantias de que uma *waiting thread* entrará no monitor!

Monitores em Java



- Todo objeto tem um monitor associado
  - Funciona como uma “tranca” nesse objeto;
- A “Regiao Crítica é declarada usando a palavra chave synchronized
  - Pode ser um método completo, ou apenas um bloco de código

Monitores em Java

```
void minhaClasse() {
     // código não sincronizado
     ...
     MeuMetodo1(...){
          ...
          synchronized(objCujoMonitorÉParaSerUsado) {
               // bloco sincronizado
               ...
          }
     }

     // mais código não sincronizado
     ...

     // Declarando um método todo sincronizado:
     synchronized meuMetodo(...) {
          // bloco sincronizado
          ...
     }


     // mais código não sincronizado
     ...
}
```

Produtor-Consumidor em Java

```
public synchronized void put(Object o) {
    while (buf.size()==MAX_SIZE) {
        wait(); // called if the buffer is full (try/catch removed for brevity)
    }
    buf.add(o);
    notify(); // called in case there are any getters or putters waiting
}

public synchronized Object get() {
    // Y: this is where C2 tries to acquire the lock (i.e. at the beginning of the method)
    while (buf.size()==0) {
        wait(); // called if the buffer is empty (try/catch removed for brevity)
        // X: this is where C1 tries to re-acquire the lock (see below)
    }
    Object o = buf.remove(0);
    notify(); // called if there are any getters or putters waiting
    return o;
}
```

- C1 (Consumidor 1) entra no bloco sincronizado e o buffer está cheio. C1 se bloqueia, entrando no “wait set” do objeto.
- C2 está tentando entrar no método “synchronized” (ponto Y), mas P1 (Produtor 1) está colocando um objeto no buffer, e chama na sequência o notify(). A única thread no “wait set” é C1, então ela é acordada e tenta reobter o lok do objeto (ponto X). .
- Agora, C1 e C2 estão tentando obter o lock. Um deles (não-determinístico) é escolhido e obtém o lock, enquanto o outro é bloqueado (não no “wait set”, apenas bloqueado esperando obter o lock do objeto).
- Suponha que C2 obtenha, então C1 é bloqueado (tentando obter o lock no ponto X).
- C2 completa o método e libera o lock. Agora C1 obtém o lock, e (que sorte!) volta no teste do while, para evitar que ele tente remover um elemento do buffer caso o buffer esteja vazio.... o que será o caso já que C2 já removeu o elemento que P1 havia inserido!

Referências

- A. S. Tanenbaum, ''Sistemas Operacionais Modernos'', 3a. Edição, Editora Prentice-Hall, 2010.
  - 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