Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos (5)

Aula 14 – Monitores e Troca de Mensagens

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)

- São elementos de uma linguagem de programação que provêm funcionalidade equivalente a dos semáforos mas que são de controle mais fácil.
- 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.
- Apenas um processo pode "estar executando" em um monitor. Qualquer outro processo que tenha invocado o monitor é bloqueado (esperando pela disponibilidade do monitor).
- Monitor é um conceito incluído em algumas linguagens de programação:
  - Módula, Pascal Concorrente, Java.

Variáveis de Condição (1)

- Mecanismo usado por monitores para realizar sincronização
- 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*)

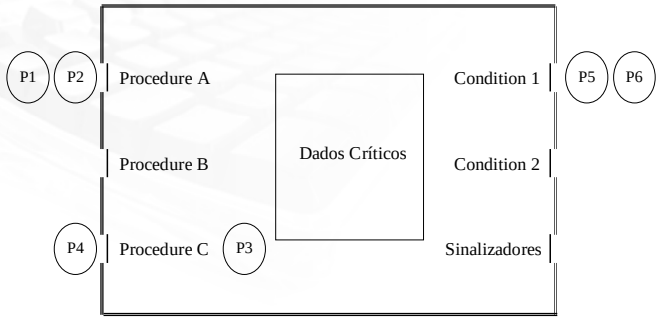
Variáveis de Condição (2)

- *Wait (condition)*
  - Faz com que o monitor suspenda o processo que fez a chamada (suspende processo na condição). O monitor armazena as informações sobre o processo suspenso em uma estrutura de dados (fila) associada à variável de condição. Monitor é liberado para outro processo.
- *Signal (condition)*
  - Faz com quer o monitor reative UM dos processos suspensos na fila associada à variável de condição. Se um processo der um signal e não tiver outra tarefa esperando na condição, o sinal é perdido.

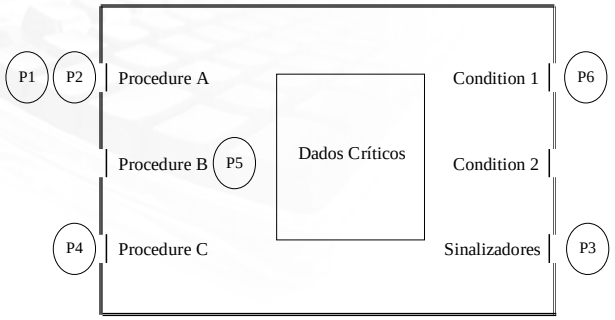
Variáveis de Condição (3)

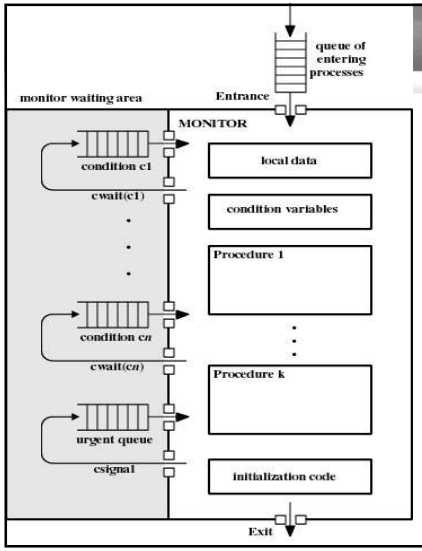
- O que acontece após um *Signal (condition)*?
  - *Hoare* propôs deixar o processo *Q* recentemente acordado executar, bloqueando o processo *P* sinalizador . *P* deve esperar em uma fila pelo término da operação de monitor realizada por *Q*.
    - Fila de Sinalizadores (pode ter prioridade)
  - *Brinch 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.

Exemplo (Abordagem de *Hoare*)



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





Visão da Estrutura 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 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

```
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

Problema do Produtor-Consumidor

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

procedure enter;
begin
   if count = N then wait(full);
   enter_item;
   count := count + 1;
   if count = 1 then signal(empty)
end;
procedure 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;
```


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;
  }
}
```

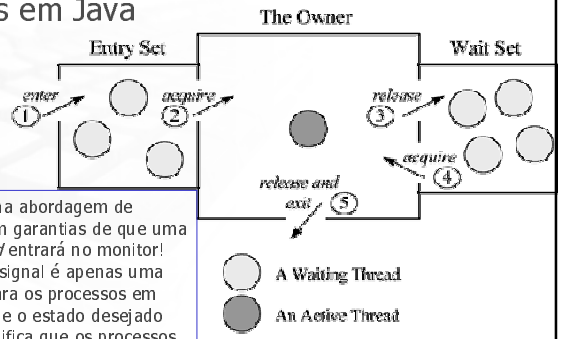

Filósofos Glutões (cont.)

```
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();
    }
}
```

Monitores em Java



Baseada na abordagem de *Hansen*, mas sem garantias de que uma *waiting thread* entrará no monitor! Portanto, um signal é apenas uma indicação (para os processos em espera) de que o estado desejado existe. Isto significa que os processos que estão esperando devem sempre checar a condição antes começar a executar no monitor.

Monitores em Java

Java Monitors

Java uses the synchronized keyword to indicate that only one thread at a time can be executing in this or any other synchronized method of the object representing the monitor. A thread can call wait() to block and leave the monitor until a notify() or notifyAll() places the thread back in the ready queue to resume execution inside the monitor when scheduled. A thread that has been sent a signal is **not** guaranteed to be the next thread executing inside the monitor compared to one that is blocked on a call to one of the monitor's synchronized methods. Also, it is **not** guaranteed that the thread that has been waiting the longest is the one woken up with a notify(); an arbitrary thread is chosen by the JVM. Finally, when a notifyAll() is called to move all waiting threads back into the ready queue, the first thread to get back into the monitor is **not** necessarily the one that has been waiting the longest.

Each Java monitor has a single nameless anonymous condition variable on which a thread can wait() or signal one waiting thread with notify() or signal all waiting threads with notifyAll(). This nameless condition variable corresponds to a lock on the object that must be obtained whenever a thread calls a synchronized method in the object. Only inside a synchronized method may wait(), notify(), and notifyAll() be called. Methods that are static can also be synchronized. There is a lock associated with the class that must be obtained when a static synchronized method is called.

Mais Informações

- Próxima aula
- Link prog concorrente em java
  - http://www.mcs.drexel.edu/~shartley/ConcProgJava/monitors.html

Implementando Monitores usando Semáforos

- Considere o monitor com apenas uma fila
- Variáveis: Semáforo X = 1; Condição C;
- nC: número de processos bloqueados em C;
- sC: implementa a fila de processos bloqueados.

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

P(x);
...
body of *F*;
...
v(x);

Implementando Monitores usando Semáforos (cont.)

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

```
wait (C):
    nC ++;
    V(X);
    P(sC);
```

```
signal (C)
    if (nC>0)
    {
        nC--;
        V(sC);
        P(X);
    }
```


Passagem (Troca) de Mensagens

- Motivação:
  - Semáforos e algoritmos de exclusão mútua são baseados no compartilhamento de variáveis. Isso implica no compartilhamento de memória física. Entretanto, em sistemas distribuídos, as máquinas formam unidades independentes (nós de uma rede), não existindo memória compartilhada entre os vários processos.
- Solução:
  - Novo paradigma de programação ("*message passage*"), onde a sincronização e comunicação entre processos é baseada em (duas) novas primitivas: *send* e *receive*.

Primitivas

- *send (destination, message_buffer)*
  - Envia a mensagem armazenada em "*message_buffer*" para um processo destino ou para uma caixa postal.
- *receive (source, message_buffer)*
  - Recebe uma mensagem de uma fonte específica, ou de qualquer fonte, e armazena-a em "*message_buffer*".

Questões de Projeto

- Uma série de novas questões surgem no cenário de troca de mensagens:
  - Sincronização entre os processos;
  - Endereçamento;
  - Formato das mensagens;
  - Disciplinas de filas;
  - Etc.

Sincronização (1)

- "*Blocking send, blocking receive*":
  - Emissor e receptor são bloqueados até que a mensagem seja entregue.
  - Mecanismo conhecido como "*rendezvous*" (encontro).
- "*Nonblocking send, blocking receive*":
  - Emissor continua processando normalmente (p.ex., enviando novas mensagens).
  - Receptor é bloqueado até a recepção da mensagem (ex: servidor).

Sincronização (2)

- "*Nonblocking send, blocking receive*" (cont.)
  - Esquema mais usado.
  - Erros podem levar a uma situação onde um processo gera mensagens repetidamente, consumindo os recursos do sistema, incluindo tempo do processador e memória.
  - "*Nonblocking send*" coloca no programador a responsabilidade de determinar se a mensagem foi ou não recebida (uso de mensagens de reconhecimento – "*acknowledgment*").
  - Com "*blocking receive*" o processo receptor pode ficar bloqueado eternamente se a mensagem enviada se perder (soluções: uso de *timeout*, uso de mais de uma fonte, etc.)

Sincronização (3)

- "*Nonblocking send, nonblocking receive*"
  - Nenhuma parte envolvida na comunicação precisa esperar.
  - Com "*nonblocking receive*" o receptor nunca fica bloqueado eternamente, entretanto, existe o perigo da mensagem ser enviada após o processo receptor ter executado o *receive* correspondente, ou seja, a mensagem será perdida.

Endereçamento (1)

- Endereçamento Direto:
  - A primitiva *send* inclui um identificador específico do processo destino.
  - A primitiva *receive* pode operar de duas formas:

    (i) requerendo, a priori, a identificação do processo transmissor; ou

    (ii) usando um parâmetro ("*source*") para retornar a identificação do processo transmissor após o término da operação (ex: servidor de impressão, que aceita pedidos de impressão de quaisquer outros processos).

Endereçamento (2)

- Endereçamento Indireto:
  - As mensagens não são enviadas diretamente do transmissor para o receptor mas sim um local intermediário (filas) que temporariamente armazena as mensagens enviadas.
  - Tal local é conhecido como *mailbox*. Assim, processos enviam mensagens para *mailboxes* e outros retiram mensagens dos *mailboxes*.
  - Uma vantagem do endereçamento indireto é que, ao desacoplar o transmissor e receptor, uma grande flexibilidade no uso das mensagens é conseguida.

Endereçamento (3)

- Endereçamento Indireto: relacionamentos entre transmissor e receptor
  - *Um-para-um*: permite comunicação privativa.
  - *Muitos-para-um*: útil para interação cliente-servidor. Neste caso, o *mailbox* é geralmente referenciado como "*port*".
  - *Um-para-muitos*: útil para aplicações de grupo (*broadcast* e *multicast*).

Endereçamento Indireto

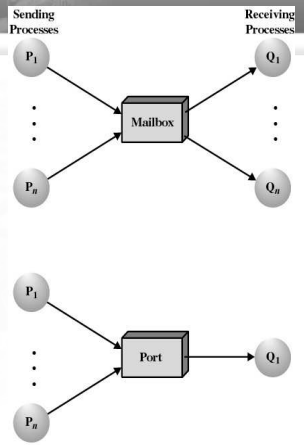


Figure 5.24 Indirect Process Communication

Formato da Mensagem

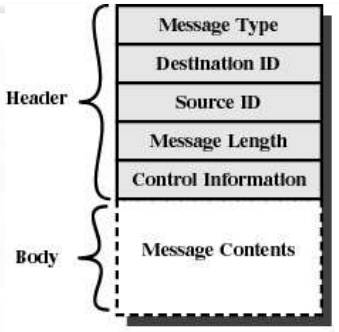


Figure 5.25 General Message Format

Exclusão Mútua usando Mensagens

```
const int n = //numero de processos
void P (i){
    message msg;
    while (true){
        receive(box, msg); //blocking
        //critical section
        send(box, msg); //nonblocking
        //resto
    }
    void main(){
        create_mailbox(box);
        send(box, null);
        parbegin (P(1), P(2), ... P(n));
    }
```


Problema do Produtor-Consumidor com Troca de Mensagens

```
#define N 100                       /* number of slots in the buffer */
void producer(void) {
   int item;
   message m;                       /* message buffer */
   while (TRUE) {
      produce_item(&item);          * generate someth. to put in buffer */
      receive(consumer, &m);        /* wait for an empty to arrive */
      build_message(&m, item);      /* construct a message to send */
      send(consumer, &m);  }}       /* send item to consumer */

void consumer(void) {
   int item, i;
   message m;
   for (i = 0; i < N; i++) send(producer, &m);      /* send N empties */
   while (TRUE) {
      receive(producer, &m);        /* get message containing item */
      extract_item(&m, &item);      /* extract item from message */
      send(producer, &m);           /* send back empty reply */
      consume_item(item);  }}       /* do something with the item */
```