SISTEMAS OPERACIONAIS

Comunicação e Sincronização de Processos

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - **Monitores**;
  - Passagem de Mensagem;

Comunicação de Processos – Monitores

- Semáforos estão sujeitos a erros de programação.
- Up e Down devem estar inseridos no código do processo e não existe nenhuma reinvindicação formal da sua presença.
- Erros e omissões (deliberadas ou não) podem existir e a exclusão mútua pode não ser atingida.

Comunicação de Processos – Monitores

- Idealizado por Hoare (1974) e Brinch Hansen (1975)
- **Monitor**: primitiva (unidade básica de sincronização) de alto nível para sincronizar processos:
  - Conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote;
- Somente um processo pode estar ativo dentro do monitor em um mesmo instante; outros processos ficam bloqueados até que possam estar ativos no monitor;

Comunicação de Processos – Monitores

- Solução:
  - Tornar obrigatória a exclusão mútua. Uma maneira de se fazer isso é colocar as seções críticas em uma área acessível somente por 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) no monitor.
  - Assim quando um processo precisa referenciar dados compartilhadas ele simplesmente chama um procedimento do monitor.
  - Resultado: o código da seção crítica não é mais duplicado em cada processo.

Comunicação de Processos – Monitores

```
monitor example
   int i;
   condition c;

   procedure A();
   .
   end;
   procedure B();
   .
   end;
end monitor;
```

Estrutura básica de um Monitor

Dependem da linguagem de programação → Compilador é que garante a exclusão mútua.

- JAVA
- Pascal Concorrente
- Módula

Todos os recursos compartilhados entre processos devem estar implementados **dentro** do **Monitor**;

Comunicação de Processos – Monitores

- Execução:
  - Chamada a uma rotina do monitor;
  - Instruções iniciais → teste para detectar se um outro processo está ativo dentro do monitor;
  - Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;
  - Caso contrário, o processo novo executa as rotinas no **monitor**;

- A forma de implementação do monitor garante exclusão mútua na manipulação de regiões críticas.

Comunicação de Processos – Monitores

- *Condition Variables* (`condition`): variáveis que indicam uma condição; e
- Operações Básicas: *`WAIT` e `SIGNAL`*
  - `wait (condition)` → bloqueia o processo;
    - O monitor armazena as informações sobre o processo suspenso em uma estrutura de dados (fila) associada à variável de condição.
  - `signal (condition)` → "acorda" o processo que executou um `wait` na variável `condition` e foi bloqueado;
    - Tira processo associado a condição da fila

Comunicação de Processos – Monitores

- Variáveis condicionais não são contadores, portanto, não acumulam sinais;
- Se um sinal é enviado sem ninguém (processo) estar esperando, o sinal é perdido;
- Assim, um comando `WAIT` deve vir antes de um comando `SIGNAL`.

Comunicação de Processos – Monitores

- Como evitar dois processos ativos no monitor ao mesmo tempo?

  (1) Hoare → colocar o processo mais recente para rodar, suspendendo o outro!!! (*sinalizar e esperar*)

  (2) B. Hansen → um processo que executa um `SIGNAL` deve deixar o monitor imediatamente;

  - O comando `SIGNAL` deve ser o último de um procedimento do monitor;

A condição (2) é mais simples e mais fácil de se implementar.

Comunicação de Processos – Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Comunicação de Processos – Monitores

Se parar aqui

Consumidor não pode entrar no monitor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Comunicação de Processos – Monitores

- Devido a exclusão mútua automática dos procedimentos do monitor, tem-se:
  - o produtor dentro de um procedimento do monitor descobre que o buffer está cheio
  - produtor termina a operação de `WAIT` sem se preocupar
  - consumidor só entrará no monitor após produtor dormir

Comunicação de Processos – Monitores

- Limitações de semáforos e monitores:
  - Ambos são boas soluções somente para CPUs com memória compartilhada. Não são boas soluções para sistema distribuídos;
  - Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas;
  - Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores;

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - **Passagem de Mensagem**;

Comunicação de Processos – Passagem de Mensagem

- Provê troca de mensagens entre processos rodando em máquinas diferentes;
- Utiliza-se de duas primitivas de chamadas de sistema: `send` e `receive`;

Comunicação de Processos – Passagem de Mensagem

- Podem ser implementadas como procedimentos:
  - `send (destination,&message);`
  - `receive (source,&message);`
- O procedimento `send` envia para um determinado destino uma mensagem, enquanto que o procedimento `receive` recebe essa mensagem em uma determinada fonte; Se nenhuma mensagem está disponível, o procedimento `receive` é bloqueado até que uma mensagem chegue.

Comunicação de Processos – Passagem de Mensagem

- Problemas desta solução:
  - Mensagens são enviadas para/por máquinas conectadas em rede; assim mensagens podem se perder ao longo da transmissão;
  - Mensagem especial chamada ***acknowledgement*** → o procedimento `receive` envia um ***acknowledgement*** para o procedimento `send`. Se esse ***acknowledgement*** não chega no procedimento `send`, esse procedimento retransmite a mensagem já enviada;

Comunicação de Processos – Passagem de Mensagem

- Problemas:
  - A mensagem é recebida corretamente, mas o ***acknowledgement*** se perde.
  - Então o `receive` deve ter uma maneira de saber se uma mensagem recebida é uma retransmissão → cada mensagem enviada pelo `send` possui uma identificação – seqüência de números; Assim, ao receber uma nova mensagem, o `receive` verifica essa identificação, se ela for semelhante a de alguma mensagem já recebida, o `receive` descarta a mensagem!

Comunicação de Processos – Passagem de Mensagem

- Problemas:
  - Desempenho: copiar mensagens de um processo para o outro é mais lento do que operações com semáforos e monitores;
  - Autenticação → Segurança;

Comunicação de Processos – Passagem de Mensagem

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something 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 */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Comunicação de Processos
Outros mecanismos

- RPC – *Remote Procedure Call*
  - Rotinas que permitem comunicação de processos em diferentes máquinas;
  - Chamadas remotas;
- MPI – *Message-passing Interface*;
  - Sistemas paralelos;
- RMI Java – *Remote Method Invocation*
  - Permite que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais;
- Web Services
  - Permite que serviços sejam compartilhados através da Web

Comunicação de Processos
Outros mecanismos

- *Pipe*:
  - Permite a criação de filas de processos;
  - Saída de um processo é a entrada de outro;
  - Existe enquanto o processo existir;
- *Named pipe*:
  - Extensão de pipe;
  - Continua existindo mesmo depois que o processo terminar;
  - Criado com chamadas de sistemas;
- *Socket*:
  - Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes;
  - Host X (192.168.1.1:1065) Server Y (192.168.1.2:80);