Revisão

Introdução

Modelos de programação

Máquinas/arquiteturas paralelas

Sincronização

- Spin locks
- Barreiras
- Sincronização baseada em escalonador
- Problemas tradicionais em sincronização

Hoje

Passagem de mensagens

- Conexão entre processos
- Envio e recepção de msgs
- RPC e Rendezvous
- Projeto de progs que passam msgs

Passagem de Mensagens

Conexão entre processos (naming):

- 1-pra-1, naming explíicito (Occam)
- vários-pra-1, servidor recebe "nomes" (Demos-MP)
- 1-pra-vários (multicast), naming explíicito
- vários-pra-vários, não existem nomes (Linda)

Envio e Recepção de Msgs

Diferentes estilos:

- Síincrono = bloqueante

- Assíncrono = não-bloqueante

Passagem de Msgs Síincrona

Processos só continuam qdo transmissor e receptor alcançam o comando de comunicação (precisa de acks)

Ex: Servidor de máximo divisor comum (naming de processos)

```
Server::
  do true -> Client ? args(x,y)  # com guard
     do x > y -> x := x - y      # com guard
        x < y -> y := y - x      # com guard
     od
     Client ! result(x)
  od

Client::
  Server ! args(v1, v2); Server ? result(r)
```

Passagem de Msgs Síincrona

Como podem existir vários clientes, precisamos de comunicação guardada

B; C $\rightarrow$ S

onde: B = expressão booleana; C = comando de comunicação; e S = conjunto de comandos

Resultados possíveis para guarda:

- Falha se B falso
- Sucede se B verd e C roda sem atraso
- Bloqueia se B verd e C atrasa

Passagem de Msgs Síncrona

Ex: MDC

```
Server::
  do (i:1..N) Client[i] ? args(x,y) ->
     do x > y -> x := x - y
        x < y -> y := y - x
     od
     Client[i] ! result(x)
  od
```

Passagem de Msgs Síncrona

Ex: alocação de recursos

```
Allocator::
  do (c:1..n) avail > 0;
        Client[c] ? acquire() ->
        avail := avail - 1;
        unitid := remove(units);
        Client[c] ! reply(unitid);
     (c:1..n) Client[c] ? release(unitid) ->
        avail := avail + 1;
        insert(units,unitid);
  od

Client[i:1..n]::
  Allocator ! acquire();
  Allocator ? reply(unitid);
  # use resource unitid
  Allocator ! release(unitid);
```

Passagem de Msgs Assíncrona

Processos não bloqueiam em sends e às vezes em receives (precisa de bufferização)

Apesar de máxima concorrência, bufferização e controle de fluxo são problemas. Erros (detecção e correção) são problemas ainda maiores

Receives podem ser explícitos ou implícitos

Passagem de Msgs Assíncrona

Ex: alocação de recursos (naming de canais)

```
Allocator::
  do true ->
    receive request(index, kind, unitid)
    if kind = ACQUIRE ->
          if avail > 0 ->
                avail := avail - 1;
                unitid := remove(units);
                send reply[index](unitid);
             avail = 0 ->
                insert(pending,index);
          fi
       kind = RELEASE ->
          if empty(pending) ->
                avail := avail + 1;
                insert(units,unitid);
             not empty(pending) ->
                index := remove(pending);
                send reply[index](unitid);
          fi
    fi
  od
```

RPC e Rendezvous

Ideais para interações entre servidor e clientes

Combinam aspectos de monitores (métodos e recepção implícita) e passagem de msgs síncrona

Sintaxe de RPC similar a chamada de procedimento. Software implementa a montagem (parâmetros) e transferência das msgs

Diferença entre RPC e rendezvous: em rendezvous, os processos se encontram num comando *in*

Rendezvous

Ex: Bounded buffer

```
Buffer:: op deposit(data: T),
            fetch(var result: T)
         var buf[1:n]: T
         var front:=1, rear:=1, count:=0
  do true ->
    in deposit(data) and count < n ->
          buf[rear] := data;
          rear := rear+1 mod n;
          count := count + 1;
       fetch(result) and count > 0 ->
          result := buf[front];
          front := front+1 mod n;
          count := count - 1;
    ni
  od
Producer:: Buffer.deposit(3);
Consumer:: Buffer.fetch(result);
```

Projeto de Programas

Fases:

1. Partição: computação e dados decompostos em tasks (decomposição de domínio e decomposição funcional)

2. Comunicação: comunicação necessária determinada, estruturas de comunicação e algoritmos definidos

3. Aglomeração: tasks e comunicação avaliadas em termos de desempenho e custo de implementação. se necessário, tasks são combinadas

4. Mapeamento: atribuição de tasks a processadores (estático ou dinâmico)

Exemplo: Rede Neural

1. Partição: decomp funcional – nó = task

2. Comunicação: comunicação com nós vizinhos

3. Aglomeração: vários nós por task (reduzir comunicação)

4. Mapeamento: estático – 1 task/proc

Próxima Aula

Capítulo 2 do livro do Foster

Discussão em aula

Aluno responsável por esclarecer dúvidas