Paradigmas de Linguagens de Programação

Concorrência

Cristiano Lehrer, M.Sc.

ybadoo

Introdução

- Níveis de concorrência:
  - Nível de instrução de máquina:
    - Executando duas ou mais instruções de máquina simultaneamente.
  - Nível de comando:
    - Executando dois ou mais comandos simultaneamente.
  - Nível de unidade:
    - Executando duas ou mais unidades de subprogramas simultaneamente.
  - Nível de programa:
    - Executando dois ou mais programas simultaneamente.

Evolução das Arquiteturas com Multiprocessadores

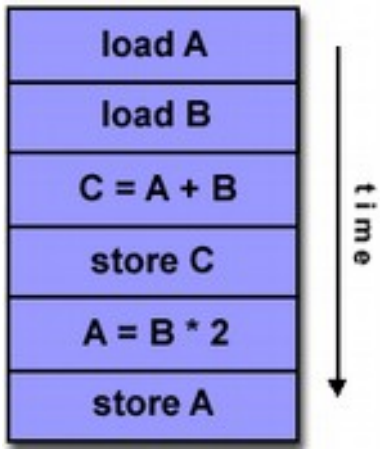
- Final da década de 50:
  - Um processador de propósitos gerais e um ou mais processadores especiais para operações de entrada e saída.
- Início da década de 60:
  - Vários processadores completos usados para concorrência no nível do programa.
- Meados da década de 60:
  - Vários processados parciais usados para concorrência no nível de instruções.

Taxonomia de Flynn (1/5)

- A taxonomia de Flynn abrange quatro classes de arquiteturas de computadores:
  - SISD (*Single Instruction Single Data*):
    - Fluxo único de instruções sobre um único conjunto de dados.
  - SIMD (*Single Instruction Multiple Data*):
    - Fluxo único de instruções em múltiplos conjuntos de dados.
  - MISD (*Multiple Instruction Single Data*):
    - Fluxo múltiplo de instruções em um único conjunto de dados.
  - MIMD (*Multiple Instruction Multiple Data*):
    - Fluxo múltiplo de instruções sobre múltiplos conjuntos de dados.

Taxonomia de Flynn (2/5)

- SISD (*Single Instruction Single Data*):
  - Fluxo único de instruções sobre um único conjunto de dados.



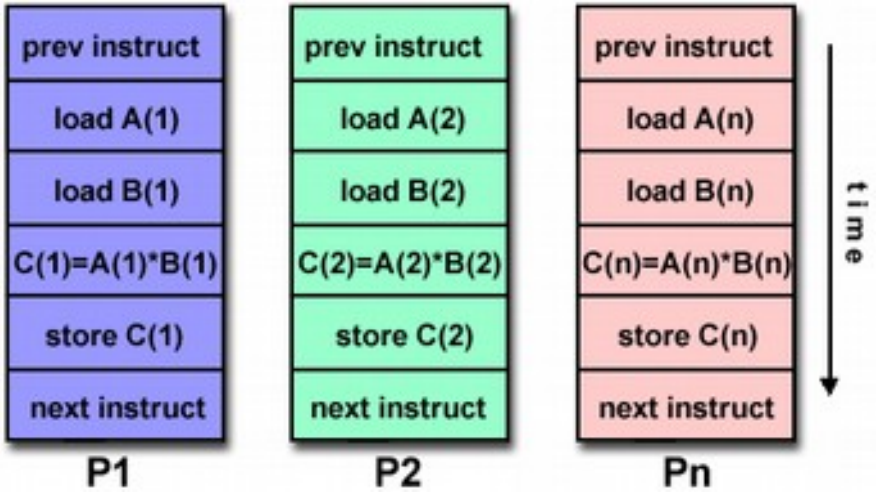
IBM 360

CRAY1

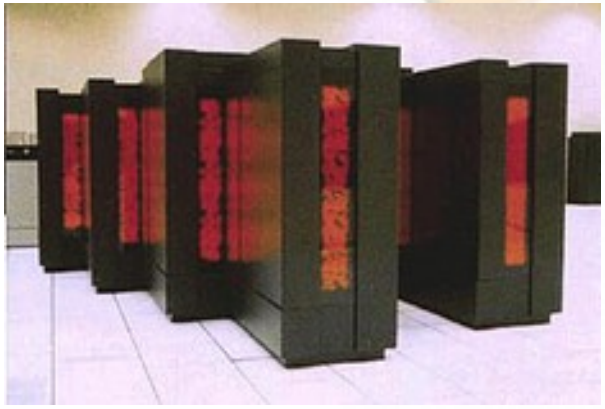
ybadoo

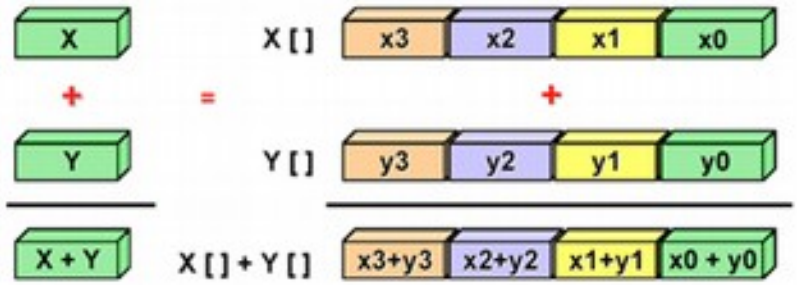
Taxonomia de Flynn (3/5)

- SIMD (*Single Instruction Multiple Data*):
  - Fluxo único de instruções em múltiplos conjuntos de dados.



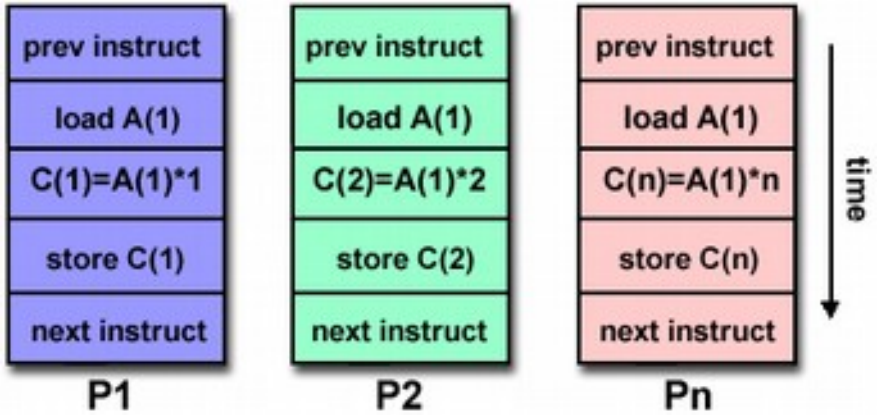
Thinking Machines CM-2





Taxonomia de Flynn (4/5)

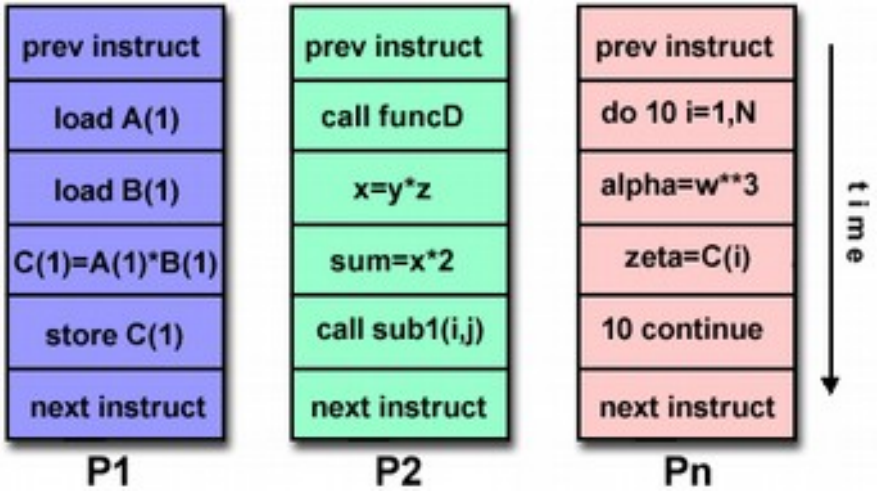
- MISD (*Multiple Instruction Single Data*):
  - Fluxo múltiplo de instruções em um único conjunto de dados.



ybadoo

Taxonomia de Flynn (5/5)

- MIMD (*Multiple Instruction Multiple Data*):
  - Fluxo múltiplo de instruções sobre múltiplos conjuntos de dados.



AMD Opteron

Categorias de Concorrência

- Concorrência Física:
  - Vários processadores independentes.
- Concorrência Lógica:
  - *Time-sharing* – compartilhamento de tempo.

Razões para Estudar Concorrência

- Concorrência envolve uma diferente maneira de projetar software que pode ser muito usual:
  - Muitas situações reais envolvem concorrência.
- Atualmente computadores capazes de realizar concorrência física são largamente utilizados.

Concorrência no Nível do Subprograma

- Tarefas:
  - Uma tarefa é uma unidade de programa que pode ser executada concorrentemente com outras unidades.
  - Tarefas são diferentes de programas ordinários:
    - Uma tarefa pode ser implicitamente iniciada.
    - Quando uma unidade de programa inicia a execução de uma tarefa, a unidade não é necessariamente suspendida.
    - Quando a execução de uma tarefa termina, o controle pode não retornar para o chamador.

Tarefas

- Tarefas pesadas (*heavyweight*) executam em seu próprio espaço de endereçamento.
- Tarefas leves (*lightweight*) executam em um mesmo espaço de endereçamento.
- Uma tarefa é disjunta se ela não comunica com ou afeta a execução de outra tarefa do programa de alguma maneira.

Comunicação entre Tarefas

- Comunicação entre tarefas é necessária para sincronização.
- A comunicação pode ser feita através de:
  - Variáveis compartilhadas.
  - Parâmetros.
  - Passagem de mensagem.

ybadoo

Tipos de Sincronização

- Cooperação:
  - Tarefa A deve esperar a tarefa B completar alguma atividade específica antes de poder continuar sua execução:
    - Problema do produtor-consumidor.
- Competição:
  - Quando duas ou mais tarefas devem usar algum recurso que não pode ser simultaneamente usado:
    - Um contador compartilhado.

Tarefas e Sincronização

- Competição é usualmente provida através da exclusão mútua.
- Prover sincronização requer um mecanismo para atrasar a execução de uma tarefa.
- O controle da execução das tarefas é mantido pelo programa chamado escalonador, que mapeia as tarefas aos processadores disponíveis.

Estados das Tarefas

- Nova:
  - Criada mas não inicializada.
- Pronta:
  - Pronta para execução.
- Executando.
- Bloqueada:
  - Já foi executada, porém não pode continuar sua execução:
    - Usualmente esperando que algum evento ocorra.
- Morta:
  - Não mais ativa.

Liveness

- Característica que uma unidade do programa pode ou não ter:
  - Em código sequencial, isto significa que a unidade irá eventualmente completar sua execução.
  - Em um ambiente concorrente, uma tarefa pode facilmente perder esta característica:
    - Se todas as tarefas em um ambiente concorrente perdem esta característica, ocorre um *deadlock*.

Questões de Projeto

- Como a sincronização de cooperação é provida.
- Como a sincronização de competição é provida.
- Como e quando as tarefas iniciam e terminam sua execução.
- As tarefas são criadas estaticamente ou dinamicamente.

Métodos para Prover Sincronização

- Semáforos.
- Monitores.
- Passagem de mensagem.

Semáforos

- Propostos por Dijkstra em 1965.
- Um semáforo é uma estrutura de dados consistindo de um contador e uma fila para armazenar descritores de tarefas.
- Semáforos podem ser usados para implementar guardas no código que acessam estruturas de dados compartilhadas.
- Semáforos tem apenas duas operações, *wait* e *release*:
  - Originalmente chamadas de P e V por Dijkstra.
- Semáforos podem ser usados para prover sincronização de competição e de cooperação.

Sincronização de Cooperação com Semáforos (1/6)

- Exemplo de um *buffer* compartilhado:
  - O *buffer* é implementado como um TAD com as operações `DEPOSIT` e `FETCH`:
    - Como meios únicos de acessar o *buffer*.
  - Uso de dois semáforos para cooperação:
    - `emptyspots` e `fullspots`.

Sincronização de Cooperação com Semáforos (2/6)

- `DEPOSIT` deve primeiro checar `emptyspots` para verificar se existe espaço no *buffer*:
  - Se existe espaço, o contador de `emptyspots` é decrementado e o valor é inserido.
  - Se não existe espaço, o chamador é armazenado na fila de `emptyspots`.
  - Quando `DEPOSIT` termina, ele deve incrementar o contador de `fullspots`.

Sincronização de Cooperação com Semáforos (3/6)

- `FETCH` deve primeiro checar `fullspots` para ver se existe um valor:
  - Se existe um espaço cheio, o contador de `fullspots` é decrementado e o valor é removido.
  - Se não existe valores no *buffer*, o chamador deve ser colocado na fila de `fullspots`.
  - Quando `FETCH` termina, ela decrementa o contador de `emptyspots`.
- As operações `FETCH` e `DEPOSIT` são acompanhadas por duas operações sobre semáforos a saber:
  - `wait` e `release`.

Sincronização de Cooperação com Semáforos (4/6)

```
wait(semaforo)
  if semaforo.counter > 0 then
    decremente o contador de semaforo
  else
    coloque o chamador na fila de semaforo
    tente transferir o controle para alguma tarefa pronta
    {se a fila de tarefas prontas estiver vazia, ocorrerá
    um deadlock}
end
```

Sincronização de Cooperação com Semáforos (5/6)

```
release(semaforo)
  if fila de semaforo estiver vazia then
    incremente o contador de semaforo
  else
    coloque a tarefa de chamada na fila pronta
    transfira o controle para uma tarefa da fila de semaforo
end
```

Sincronização de Cooperação com Semáforos (6/6)

```
semaphore fullspots, emptyspots;
fullspots.count := 0;
emptyspots.count := TAMBUF;

task produtor;
  loop
    -- produza VALOR --
    wait(emptyspots);  {espere por um espaço}
    DEPOSIT(VALOR);
    release(fullspots);   {aumente os espaços preenchidos}
  end loop;
end produtor;

task consumidor;
  loop
    wait(fullspots);   {certifique-se de que não está vazio}
    FETCH(VALOR);
    release(emptyspots);  {aumente os espaços vazios}
    -- consuma o VALOR --
  end loop;
end consumidor;
```


ybadoo

Sincronização de Competição com Semáforos (1/2)

- Um terceiro semáforo, chamado `access`, é usado para controlar o acesso:
  - Sincronização de competição.
- O contador de `access` terá apenas os valores 0 e 1:
  - Tal semáforo é chamado de semáforo binário.

Sincronização de Competição com Semáforos (2/2)

```
semaphore access, fullspots, emptyspots;
access.count := 1;
fullspots.count := 0;
emptyspots.count := TAMBUF;

task produtor;
  loop
    -- produza VALOR --
    wait(emptyspots);          {espere por um espaço}
    wait(access);              {espere por acesso}
    DEPOSIT(VALOR);
    release(access);           {renuncie ao acesso}
    release(fullspots);        {aumente os espaços preenchidos}
  end loop;
end produtor;

task consumidor;
  loop
    wait(fullspots);           {certifique-se de que não está vazio}
    wait(access);              {espere por acesso}
    FETCH(VALOR);
    release(access);           {renuncie ao acesso}
    release(emptyspots);       {aumente os espaços vazios}
    -- consuma VALOR --
  end loop;
end consumidor;
```


ybadoo

Avaliação de Semáforos

- O mau uso de semáforos podem ocasionar falhas na sincronização de cooperação:
  - Deixar a instrução `wait(emptyspots)` fora da tarefa do produtor resultaria em *overflow*.
- O mau uso de semáforos podem ocasionar falhas na sincronização de competição:
  - Deixar a instrução `release(access)` em qualquer uma das tarefas resulta em *deadlock*.

Monitores

- Inicialmente suportado por Concurrent Pascal.
- Atualmente suportado por Ada, Java e C#.
- Objetivo:
  - Encapsular os dados compartilhados e suas operações para um acesso restrito.
- Um monitor é um tipo abstrato de dados para compartilhar dados.

Sincronização com Monitores

- Sincronização de competição:
  - O acesso ao dado compartilhado no monitor é limitado pela implementação a um simples processo por vez; então, acesso mutuamente exclusivo é inerente da definição semântica do monitor.
  - Múltiplas chamadas são enfileiradas.
- Sincronização de cooperação:
  - Cooperação é ainda requerida:
    - Feita com semáforos.

Avaliação de Monitores

- Suporte a sincronização de competição é notável.
- Suporte a sincronização de cooperação é muito similar com semáforos, ocasionando os mesmos problemas.

Passagem de Mensagem

- Proposto originalmente por Brinch Hansen e Hoare, em 1978.
- Uso de não determinismo para escolher entre múltiplas mensagens simultâneas para o mesmo endereço.

Passagem de Mensagem

- Passagem de mensagem pode ser síncrona ou assíncrona:
  - Assíncrona significa que tarefas podem interromper outras para se comunicar.
  - Síncrona significa que tarefas devem esperar até que a outra tarefa esteja pronta para receber a mensagem.
- Comunicação síncrona utiliza Rendez-Vous para se comunicar:
  - Bloqueio da thread interessada no encontro, até a finalização de uma thread qualquer em execução.
- Passagem de mensagem é um modelo geral para concorrência:
  - Ele pode modelar semáforos e monitores.
  - Ele não é apenas para sincronização de competição.

ybadoo

Java Threads (1/3)

- As unidades concorrentes em Java são os métodos runs.
- Método `run` é herdado e sobrecarregado.
- Threads são processos leves.
- Métodos:
  - `start()` - inicia a thread.
  - `yield()` - deixa a thread em execução temporariamente inativa e, quando possível, promove outra thread de mesma prioridade ou maior.
  - `sleep(time)` – deixa a thread corrente inativa por no mínimo tempo e promove outra thread.
  - `join()` - aguarda outra thread para encerrar.

Java Threads (2/3)

- Terminação:
  - Um método `run` termina quando sua execução chega ao fim de seu código.
  - Uma thread pode ser pedido para terminar.

Java Threads (3/3)

- Sincronização de Competição:
  - Um método que inclui o modificador `synchronized` desabilita qualquer outro método de acessar o objeto, enquanto ele está em execução.
  - Se apenas um parte de um método deve executar sem interferência, ele pode ser sincronizado.
  - Um objeto cujos métodos são todos sincronizados é efetivamente um monitor.
- Sincronização de Cooperação:
  - Utilização dos métodos `wait` e `notify/notifyall`.