5. PARADIGMA CONCORRENTE

Algumas vezes é conveniente estruturar os sistemas como um conjunto de unidades concorrentes que são executadas em paralelo. Isto ocorre quando o programa é executado em um computador com múltiplos processadores ou em um sistema distribuído composto de vários computadores que cooperam entre si. Um programa concorrente especifica dois ou mais processos concorrentes, onde cada um executa um programa seqüencial. Tais processos interagem através da comunicação, o que traz a necessidade de sincronização. A concorrência é uma importante área da Ciência da Computação estudada em diferentes contextos: arquitetura das máquinas, sistemas operacionais, sistemas distribuídos, banco de dados, etc.

Há pelo menos duas razões para se estudar concorrência. Primeiro, ela fornece uma solução diferente para resolução dos problemas. Diversos tipos de problemas levam à estruturação de um sistema como um conjunto de unidades concorrentes que executam em paralelo, da mesma maneira que a recursão é uma forma natural de projetar a solução para determinados problemas. Além disso a execução em paralelo, na maioria das vezes, aumenta a performance dos sistemas. Muitos programas são escritos para simular entidades físicas e atividades, e freqüentemente o sistema que está sendo simulado inclui mais do que uma entidade que realizam ações simultaneamente. A segunda razão para o estudo da concorrência é que os computadores com múltiplos processadores estão sendo muito utilizados, criando a necessidade de desenvolvimento de *software* para fazer uso efetivo das capacidades de *hardware*. Por isso, facilidades para implementação da concorrência devem ser desenvolvidas e incluídas nas linguagens de programação.

Hoje em dia, várias arquiteturas de computadores possuem mais do que um processador e podem suportar a execução concorrente de programas. Nos primeiros computadores que possuíam múltiplos processadores, um processador era usado para funções genéricas e os outros eram usados apenas para operações de entrada e saída. Assim, enquanto um programa era executado, outros podiam executar operações de entrada e saída. Na década de 60, as máquinas possuíam processadores que eram usados através de um organizador de *jobs* (ou tarefas), que distribuía a execução entre os processadores. Sistemas com esta estrutura suportavam a programação concorrente. Em meados da década de 60 surgiram computadores multiprocessados providos de diversos processadores que executavam apenas um conjunto de instruções. Por exemplo, algumas máquinas tinham dois ou mais multiplicadores de ponto-flutuante, enquanto outras tinham duas ou mais unidades aritméticas de ponto-flutuante completas. Neste caso, os compiladores tinham que determinar quais instruções poderiam ser executadas concorrentemente e em quais processadores.

Atualmente existem vários tipos de computadores multiprocessados, sendo que os mais comuns são SIMD e MIMD. O primeiro, *Single-Instruction Multiple-Data*, possui múltiplos processadores, cada um com sua própria memória local, que executam a mesma instrução simultaneamente com dados diferentes. Neste caso um processador controla a operação dos outros processadores. Talvez as máquinas SIMD mais populares pertençam à categoria de processadores vetoriais. O segundo tipo, *Multiple-Instruction Multiple-Data*, possui múltiplos processadores que operam independentemente, mas cujas operações podem ser sincronizadas. Cada processador em um computador MIMD executa seu próprio conjunto de instruções. Eles podem aparecer em duas configurações distintas: sistemas distribuídos e de memória compartilhada.

A concorrência é naturalmente dividida nos seguintes níveis:

- Instrução: execução de duas ou mais instruções de máquina simultaneamente (não envolve questões de linguagens de programação);
- Comando: execução de dois ou mais comandos simultaneamente;
- Unidade: execução de duas ou mais unidades, ou subprogramas, simultaneamente;
- Programa: execução de dois ou mais programas simultaneamente (não envolve questões de linguagens de programação).

Métodos de controle da concorrência a nível de subprogramas aumentam a flexibilidade de programação. Estes métodos foram criados para serem usados em problemas específicos de sistemas operacionais, mas também podem ser utilizados em uma variedade de outras aplicações. Por exemplo, muitas aplicações foram projetadas para simular sistemas físicos, e muitos deles consistem de múltiplos subsistemas concorrentes. Já a concorrência a nível de comando consiste em especificar como os dados devem ser distribuídos em memórias múltiplas e quais comandos podem ser executados concorrentemente [SEB 99].

Unidades concorrentes podem ser vistas como outra forma de abstração procedural. A principal característica de distinção é que uma vez que são invocadas, a unidade que fez a invocação pode prosseguir com sua execução sem esperar pelo término da unidade invocada (figura 4.1). Consequentemente, unidades concorrentes requerem todas as mesmas considerações das outras abstrações procedurais, nome, definição, invocação e compartilhamento de dados. Outras considerações incluem a necessidade de comunicação e sincronização entre as execuções concorrentes.

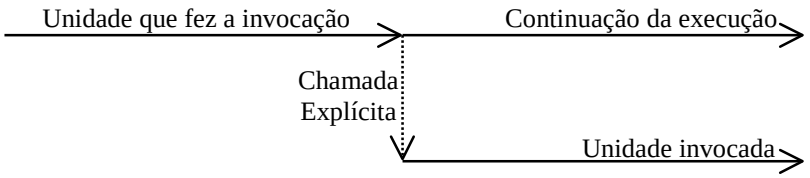

Figura 5.1 – Processamento concorrente

Existem dois modelos para a definição de unidades de programa concorrentes: o primeiro segue a construção da definição de um procedimento e o segundo de um tipo. No primeiro caso, quando a unidade concorrente é declarada, uma amarração é feita entre o nome o corpo executável. A unidade pode então ser invocada concorrentemente através da referência apropriada ao seu nome. No segundo modelo a definição cria um tipo e variáveis que são declaradas como sendo daquele tipo são então amarradas aos nomes das unidades executáveis, e múltiplas execuções do mesmo tipo de processo podem ser referenciadas pelos seus nomes distintos.

A invocação de unidades concorrentes pode ser implícita ou explícita. A primeira assume que o processo concorrente pertence a unidade de programa na qual ele foi definido. Sempre que uma unidade de programa começa a execução, todas as unidades concorrentes que pertencem a esta unidade (são definidas nesta unidade) começam suas execuções simultaneamente. Quando a unidade "mestre" termina, ela espera até que todas as unidades concorrentes terminem, antes de retornar para a unidade que fez a invocação. Este processo de invocação implícita é ilustrado na figura 4.2. Já as unidades concorrentes que são invocadas explicitamente, são invocadas como procedimentos, como uma abstração de comandos através de uma referência ao nome ao qual a unidade é amarrada através de sua definição. Isto permite a invocação da unidade concorrente em qualquer ponto dentro da unidade de programa na qual foi definida [DER 90].

Também deve-se considerar que a execução de unidades de programa concorrentes pode ocorrer fisicamente em processadores separados, ou logicamente em fatias de tempo diferentes em um único processador. Na concorrência física, assume-se que mais de um processador está disponível e vários subprogramas de um mesmo programa estão literalmente executando simultaneamente. Na concorrência lógica, tanto o programador como o sistema assumem que existem múltiplos processadores fornecendo concorrência, quando, na verdade, a execução atual do programa é feita em intervalos de tempos diferentes no mesmo processador. Isto é similar a ilusão de execução simultânea que é fornecida a usuários diferentes de sistemas multiprogramáveis. Do ponto de vista do programador, a concorrência lógica é igual à física [SEB 99].

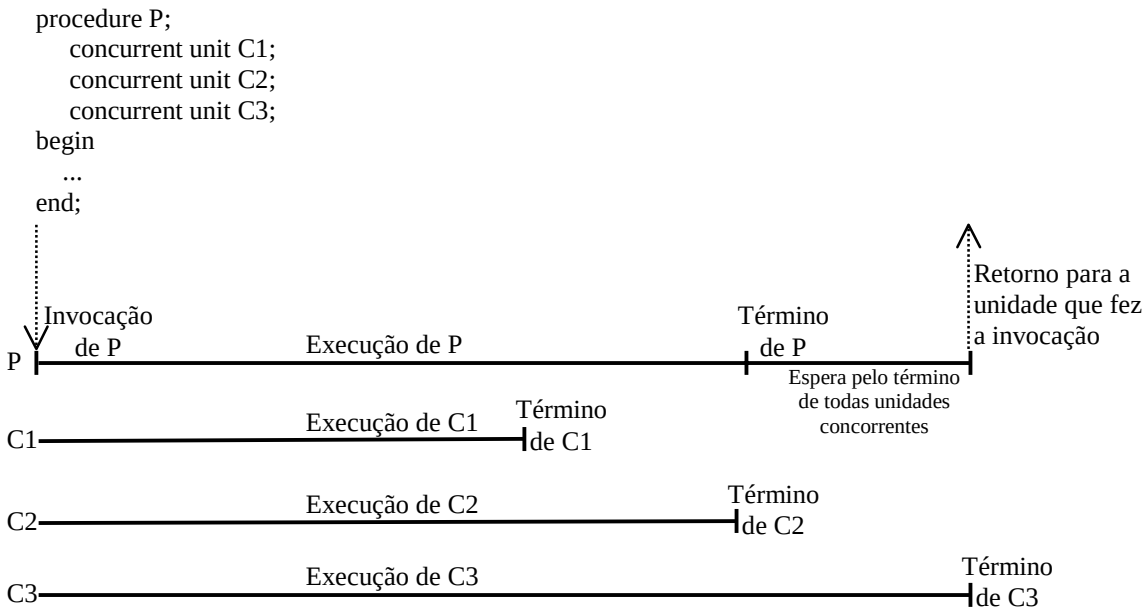

Figura 5.2 – Processamento concorrente

Freqüentemente é necessário que as unidades concorrentes compartilhem dados, sendo que os dois modelos geralmente usados são: memória compartilhada e troca de mensagens. No primeiro caso, existem dados que são globais a todas unidades no sentido que várias unidades possuem acesso a estes dados. Tais dados estão na forma de variáveis que todas as unidades podem acessar, ou em estruturas de dados mantidas em memória para todas unidades usarem. Isto é ilustrado no próximo exemplo em *Pascal Concorrente*, onde "*P1*" e "*P2*" usam a variável global "*N*", com um modelo de invocação explícita. Os resultados da execução do programa podem variar dependendo dos tempos das execuções das unidades concorrentes. Se "*P1*" é executado exatamente no mesmo tempo de "*P2*", "*N*" resulta em 4. Por outro lado, se a execução de "*P1*" é seguida pela execução de "*P2*", "*N*" resulta em 6.

```
program Concurrent;
        var N: integer;
        concurrent unit P1;
        begin
                N := N + 1;
        end;
        concurrent unit P2;
        begin
                N := N + 2;
        end;
begin
        N := 3;
        cobegin
                P1;
                P2;
        coend;
        writeln(N);
end.
```

A segunda forma de compartilhar dados, através da troca de mensagens de uma unidade para outra, é tipicamente referenciada como comunicação entre processos. Sendo assim, é necessário um emissor e um receptor de mensagens, o que resulta em dois modelos para troca de informações entre unidades: *mail* e telefone. A diferença entre eles está em quando ou não a unidade receptora precisa atender a mensagem antes da unidade

emissora prosseguir. No primeiro modelo, a mensagem é enviada da unidade "*S*" para "*R*" e colocada em um *mailbox*. A unidade "*S*" então prossegue com a execução e a unidade "*R*" recupera a mensagem mais tarde. Existem duas formas de troca de informações através do modelo de telefone que variam na espera: o emissor espera somente pela notificação do receptor de que a mensagem foi recebida e continua a executar; ou o emissor espera que a mensagem seja recebida e processada pelo receptor.

Uma técnica útil para visualizar o fluxo de execução através de um programa é imaginar uma *thread* para cada um dos comandos do programa fonte. Cada instrução atingida em uma execução particular é coberta pela *thread* que representa essa execução. Seguir visualmente a *thread* ao longo do programa fonte possibilitará rastrear o fluxo de execução pela versão executável do programa. Uma *thread* de controle em um programa consiste numa seqüência de pontos do programa atingidos à medida que o controle flui por ele.

Programas que possuem co-rotinas, que as vezes são chamados de "quase concorrentes", possuem uma única *thread* de controle. Programas executados com concorrência física podem ter múltiplas *threads* de controle, onde cada processador pode executar uma das *threads*. Apesar de que programas com concorrência lógica podem possuir apenas uma *thread* de controle, tais programas podem somente ser projetados e analisados imaginando-se que eles possuem múltiplas *threads* de controle. Quando um programa *multithreaded* executa numa máquina com um único processador, suas *threads* são mapeadas numa única *thread*, tornando-se, neste caso, um programa *multithreaded* virtual.

A concorrência a nível de comando é um conceito relativamente simples. Laços que incluem comandos que operam em vetores são desfeitos para que o processamento possa ser distribuído em múltiplos processadores. Por exemplo, um laço que executa 500 repetições e inclui um comando que opera em um dos 500 elementos do vetor pode ser desfeito de tal maneira que cada um dos dez processadores diferentes possa simultaneamente processar 50 elementos do vetor [SEB 99].