

1 Introdução

O crescente interesse por concorrência é devido a maior disponibilidade de multiprocessador de custo mais baixo e à proliferação de aplicações gráficas, multimídia e Web, as quais são naturalmente representadas por várias linhas de controle distintas¹.

Entendemos programação concorrente como um paradigma de programação que visa desenvolver programas que possuem linhas de controle distintas, as quais podem executar de maneira simultânea. Um programa é dito *concorrente* se ele possui mais de um contexto de execução ativo, isto é, mais de um fluxo de execução, o qual é identificado por uma sequência de instruções[1]. Dessa forma, um programa dito concorrente se diferencia de um programa dito sequencial por conter mais de um contexto de execução ativo ao mesmo tempo.

Com base nas definições apresentadas, as co-rotinas² não são concorrentes, pois, mesmo representando um fluxo de execução independente, uma co-rotina deve explicitamente suspender sua execução para permitir que outra possa executar. Um sistema concorrente é dito *paralelo* se mais de uma tarefa pode estar “fisicamente” ativa, isto é, tarefas executando em diferentes processadores. Um sistema paralelo é dito *distribuído* se seus processadores estão fisicamente separados entre si no “mundo real”[1]. Note que o conceito de concorrência é mais abrangente que o de paralelismo.

Podemos citar, no mínimo, três grandes motivações para o uso do paradigma da programação concorrente:

1. **Capturar a estrutura lógica de um problema:** vários programas, especialmente servidores e aplicações gráficas, necessitam de várias tarefas, com propósitos diferentes, cooperando entre si para cumprir os objetivos da aplicação. Logo, é natural representar cada tarefa como uma linha de controle distinta e a aplicação precisa manter esses diferentes fluxos.
2. **Aumentar o desempenho das aplicações:** aproveita-se os recursos disponíveis para paralelizar, e consequentemente acelerar, as aplicações.
3. **Cooperar com dispositivos independentes:** algumas aplicações, *softwares* embarcados e até o próprio sistemas operacional, precisam lidar com eventos de diferentes dispositivos aos quais eles estão conectados.

¹Rossetto, Silvana. “Computação Concorrente (MAB-117) Cap. I: Introdução e histórico da programação concorrente.” (2012).

²As co-rotinas diferem dos subprogramas comuns na forma de transferência de controle realizada na chamada e no retorno. Elas possuem um único ponto de entrada, mas podem ter vários pontos intermediários de entrada e saída. Uma co-rotina pode voltar a executar de onde parou na última execução.

Todavia, nem tudo são flores, pois a programação concorrente traz alguns desafios:

- Dificuldade de programação e *debug*;
- Dificuldade de garantir consistência (consistência significa que “infinitas” execuções de um mesmo código sobre os mesmos dados geram sempre os mesmos resultados);
- Dificuldade de garantir sincronização (sincronizar é manter a ordem e coerência das ações entre os vários fluxos de um programa concorrente; a necessidade advém do fato da ordem de execução das unidades ser não-determinística – inerentemente assíncronos).

Na computação a concorrência pode ser tratada de formas diferentes e/ou em camadas diferentes na arquitetura. Neste estudo iremos abordar a concorrência no *nível de aplicação*, representada por construções que são visíveis ao programador.

2 Fundamentos

As tarefas (fluxos de execução, processos) podem ser independentes (não compartilha dados ou não é afetado por outros) ou cooperativos. Se não forem independentes é possível executar em paralelo desde que sincronizemos suas interações. A sincronização serve para eliminar as disputas entre diferentes fluxos controlando a forma como suas ações ocorrem no decorrer da execução[1].

Dizemos que uma *condição de corrida* é a tentativa de uso de dados compartilhados, onde o resultado final pode variar de acordo com a ordem de execução. Uma *região crítica* é a parte do código onde é feito o acesso a recursos compartilhados, e que podem levar a condições de corrida.

O mau gerenciamento da região crítica pode acarretar em alguns problemas:

- **Deadlock:** ocorre quando dois, ou mais, processos aguardam um evento que só pode ser produzido pelo outro.
- **Starvation:** ocorre quando um processo (de baixa prioridade) nunca obtém um recurso.

A *exclusão mútua* consiste de técnicas para garantir que apenas um dos fluxos possa estar em sua região crítica num determinado momento. Na próxima subseção serão discutidas algumas dessas técnicas.

2.1 Técnicas de Controle e Sincronismo

As técnicas discutidas aqui são dependentes da linguagem de programação que o programador irá utilizar e do contexto da aplicação (se é local, distribuída). É importante também entender o conceito de *operação atômica*. Uma operação é dita atômica quando ela é *indivisível no tempo*, isto é, uma vez que a operação tenha sido iniciada, ela não pode ser interrompida. Desta forma, operações atômicas ou são totalmente executadas ou não iniciam sua execução.

2.1.1 Semáforos

Um semáforo é uma variável que não assume valores negativos e só pode ser modificada através de duas primitivas atômicas (executadas no *kernel* do sistema operacional):

- *down*, também conhecida como P (de *proberen*); É caracterizada por decrementar o valor do semáforo.
- *up*, também conhecida como V (de *verhogen*); É caracterizada por incrementar o valor do semáforo.

Há dois tipos de semáforos: contador e binário. A ideia geral da semáforo contador é indicar quantos recursos de uma determinada região estão disponíveis para que os processos os utilizem. Assim, quando um determinado recurso é disponibilizado para uso, o semáforo é incrementado por meio da primitiva *up* e quando um recurso é alocado por algum processo o semáforo é decrementado com a primitiva *down*. Daí, enquanto o semáforo for maior que zero, os processos podem alocar recursos, caso contrário eles devem esperar até que algum seja disponibilizado e o semáforo incrementado.

Um semáforo binário admite apenas dois valores (0 e 1) e funciona basicamente como um sinalizador, de maneira que toda vez que um processo entrar em sua região crítica deve sinalizar tal ocorrência para que haja garantia de exclusão mútua (através da primitiva *down* – valor do semáforo vai para 0) e, analogamente, quando ele sair da região crítica (sinalizando através da primitiva *up* – valor do semáforo vai para 1).

O mal uso de semáforos pode causar *deadlock*.

2.1.2 Monitores

Os monitores podem ser vistos como regiões de controle que possuem um conjunto de procedimentos, variáveis e estruturas de dados encapsulados em um mesmo módulo (semelhante a uma classe). São baseados e dependentes de construções da linguagem. Ademais, duas condições básicas são impostas:

1. Os processos podem solicitar procedimentos presentes em monitores, mas não podem manipular alguns atributos diretamente.
2. Somente um processo pode estar ativo em um monitor em qualquer momento.

Quando um processo realiza uma chamada de algum procedimento interno, ocorre uma verificação de se algum outro processo está ativo dentro do monitor. Caso positivo, o acesso é bloqueado e o processo permanece em espera. Caso contrário, o procedimento requerido pode ser executado. Dentro dos monitores são utilizadas **variáveis de condições** para bloquear ou ativar processos. Essas variáveis são modificadas por meio de dois comandos: *wait* e *signal*. Em geral, o comando *wait* faz com que o processo aguarde na fila de espera, pois já há um processo ativo dentro do monitor. O comando *signal* faz com que um processo que aguarda na fila retome sua execução.

2.1.3 Troca de Mensagens

Os dois métodos vistos anteriormente só são aplicáveis em sistemas centralizados (com memória compartilhada). Em sistemas distribuídos³ pode-se utilizar *troca de mensagens*. Nesta estratégia a comunicação se dá através de duas primitivas (que como os semáforos, e ao contrário dos monitores, são chamadas de sistema em vez de construções da linguagem):

- `send`: envia uma mensagem para um determinado destino.
- `receive`: recebe uma mensagem de uma fonte (determinada ou qualquer). Se nenhuma mensagem estiver disponível ela pode bloquear até que chegue alguma.

É importante destacar que essas mensagens podem ser trocadas através de uma rede ou dentro de uma única máquina. Mensagens trocadas através da rede devem preocupar-se com a perda/duplicação de mensagens e ordem de envio e chegada.

Referências

[1] M. L. Scott, *Programming language pragmatics*. Morgan Kaufmann, 2000.

³Segundo Andrew S. Tanenbaum: “Um sistema distribuído é uma coleção de computadores independentes que aparecem para os usuários do sistema como um único computador.”