Conceitos Básicos de Programação Concorrente

Rubem Kalebe 13 de dezembro de 2015

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 distintas[1].

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-rotinas[2] 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.

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

[2] As 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 distribuídos[3] 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.

3 Processos e Threads

As tarefas, ou fluxos de execução, citados até agora são implementados em forma de *processos* (ou *subprocessos*) e *threads*.

Quando um programa é executado, o sistema operacional cria um *processo* contendo o código e os dados do programa, e passa a gerenciar tal processo até que o programa termine sua execução. Processos de usuário são criados por programas de usuário e processos de sistema, por programas de sistema. Um processo de usuário tem seu próprio espaço de endereçamento lógico, separado do espaço de outros processos de usuário e do espaço (*kernel space*) dos processos de sistema. Isso significa que dois processos podem referenciar o mesmo endereço lógico, mas tal endereço será mapeado para posições diferentes na memória física. Portanto, processos não compartilham memória, a menos que eles comuniquem ao sistema operacional um espaço de memória compartilhado[2].

Algumas das informações de um processo mantidas pelo sistemas operacional são:

- O estado do processo (pronto para executar, executando, esperando, parado);
- Contador de programa (*Program Counter* – PC);
- Valores dos registradores;

Mas, isso são apenas alguns exemplos. Processos geralmente armazenam muitas outras informações que não foram citadas e esses dados fazem com que as tarefas de criação e gerenciamento de processos seja bem custosa. Por conta disso também temos as *threads*.

Uma *thread* é uma unidade de controle presente em um processo. Quando uma *thread* executa, ela roda um fluxo de execução do programa. O processo associado a um processo em execução começa

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

com uma *thread* em execução, a qual é chamada de *thread* principal (ou *main thread*, se preferir), a qual executa a função principal de um programa. Em sistemas concorrentes, a *thread* principal cria outras *threads*, que executam outros trechos de código. Essas outras *threads* podem ainda criar outras, e por aí vai. As *threads* são criadas a partir de construções da linguagem de programação ou funções da API (*Application Programming Interface*). Cada *thread* possui sua própria pilha e sua própria cópia de registradores, incluindo o *ponteiro da pilha* e o contador de programa, que juntos descrevem o estado de execução da *thread*[2].

Uma grande vantagem das *threads* é que elas compartilham alguns dados entre outras pertencentes ao mesmo processo. Por compartilharem alguns dados e armazenarem poucos, elas são menos custosas para criar e gerenciar, tornando-se a forma ideal para representar tarefas em sistemas concorrentes.

O sistema operacional deve decidir como alocar a CPU entre os processos e *threads* do sistema. Alguns selecionam o processo a ser executado, que por sua vez escolhe qual das suas *threads* irá executar. Todavia, as *threads* são escalonadas diretamente pelo sistema operacional. Em um dado momento, vários processos, cada um com suas *threads*, podem estar executando, porém algumas *threads* podem não estarem prontas para executar. Por exemplo, algumas delas podem estar esperando um evento de I/O. A *política de escalonamento* determina qual das *threads* prontas irá executar[2].

Geralmente, cada *thread* pronta para execução recebe um período de tempo (conhecido como *quantum*) durante o qual poderá executar na CPU. Se a *thread* decidir esperar por algo, ela cede sua vez voluntariamente. Caso contrário, um *timer* controlado por *hardware* determina quando uma *thread* completou sua execução, lança uma interrupção e a *thread* é interrompida (*preemptada* – sofre *preempção*) para permitir que outra que esteja pronta possa executar. Se existirem várias CPUs, várias *threads* podem executar ao mesmo tempo. Em computador com uma única CPU, *threads* apenas aparentam executar simultaneamente, enquanto na verdade elas se revezam na execução e podem não receber o mesmo *quantum*. Por isso, algumas *threads* aparentam serem executadas mais rapidamente que outras[2].

A política de escalonamento pode considerar ainda a *prioridade da thread* e o tipo de processamento que ela executa, dando preferência a umas *threads* em relação a outras. Trocar um processo, ou *thread*, por outro na CPU é o ato conhecido *troca de contexto* e requer o armazenamento do estado do antigo processo (ou *thread*) e o carregamento do estado do novo fluxo. Como podem ocorrer centenas de trocas de contexto por segundo, essas trocas podem potencialmente aumentar, significamente, o *overhead* de execução[2].

Referências

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

[2] R. H. Carver and K.-C. Tai, *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs*. John Wiley & Sons, 2005.