Problemas da programação concorrente

Axell Brendow Batista Moreira

[1] Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUC-MG)
Caixa Postal 1.686 – 30.535-901 – Belo Horizonte – MG – Brazil

Porque sincronizar as threads?

Porque o acesso concorrente a dados e recursos partilhados pode criar uma situação de inconsistência desses mesmos recursos, isto porque as várias instâncias acedem e manipulam-nos de uma forma "simultânea", dando azo a situações de inconsistência e de falha. Então para que uma rotina ou programa seja consistente são precisos mecanismos que assegurem a execução ordenada e correta dos processos cooperantes.

Um dos mecanismos computacionais que podemos usar para que não haja inconsistência de recursos são os semáforos. Um semáforo é uma variável, um mecanismo de sincronização sem espera ativa. Esta variável pode ser manipulada através de duas primitivas atómicas, isto é, que não podem ser interrompidas por processos.

A sua função é controlar o acesso a recursos partilhados num ambiente multitarefa. A invenção desse tipo de variável é atribuída a Edsger Dijkstra.

Quando declaramos um semáforo, o seu valor indica quantos processos (ou threads) podem ter acesso ao recurso partilhado. Assim, para partilhar um recurso deve-se verificar qual o valor do semáforo para saber qual a ação a executar.

Assim, e de forma a consolidar os conhecimentos, as principais operações sobre semáforos são:

Inicialização

Recebe como parâmetro um valor inteiro indicando a quantidade de processos que podem aceder a um determinado recurso.

Operação wait

Decrementa o valor do semáforo. Se o semáforo está com valor zero, o processo é posto em espera. Para receber um sinal, um processo executa o wait e bloqueia se o semáforo impedir a continuação da sua execução.

Operação signal

Se o semáforo estiver com o valor zero e existir algum processo em espera, um processo será acordado. Caso contrário, o valor do semáforo é incrementado. Para enviar um sinal, um processo executa um signal.

As operações de incrementar e decrementar devem ser operações atômicas, ou indivisíveis, isto é, a instrução deve ser executada na totalidade sem que haja interrupções. Enquanto um processo estiver a executar uma dessas duas operações, nenhum outro processo pode executar outra operação nesse mesmo semáforo, devendo esperar que o pri-

meiro processo encerre a sua operação. Essa obrigação evita condições de disputa entre as várias threads.

Para evitar espera, o que desperdiça processamento da máquina, a operação wait utiliza uma estrutura de dados (geralmente uma FIFO). Quando um processo executa essa operação e o semáforo tem o seu valor a zero, este valor é posto na estrutura. Quando um outro processo executar a operação signal havendo processos nessa mesma estrutura, uma delas é retirada (no caso FIFO, é retirada a primeira que entrou).

Tendo em conta que há necessidade de garantir a consistência dos recursos, a utilização mais simples do semáforo é em situações na qual se utiliza o princípio da exclusão mútua, isto é, que só um processo é executado de cada vez. Para isso utiliza-se um semáforo binário, com inicialização em 1. Esse semáforo binário atua como um mutex.

O princípio da exclusão mútua ou mutex, é uma técnica usada em programação concorrente para evitar que dois processos ou threads tenham acesso simultaneamente a um recurso partilhado. Esta partilha dá origem à chamada secção crítica que pode ser protegida da seguinte forma:

```
pthread_mutex_lock()
[secção crítica]
pthread_mutex_unlock()
```

Então, um meio simples para exclusão mútua é a utilização do tal semáforo binário, isto é, que só pode assumir dois valores distintos, 0 e 1. O fecho do semáforo deve ser feito antes de utilizar o recurso e deve ser libertado só depois do uso o recurso. Tendo em conta que apenas um processo pode entrar de cada vez na secção crítica, então os outros processos devem esperar pela sua vez.

Contudo o uso do princípio da exclusão mútua, pode originar alguns problemas, como por exemplo deadlocks ou starvation (inanição).

Um **deadlock**, ou bloqueio, é referente a uma situação em que existe um impasse entre dois ou mais processos, isto é, no nosso contexto, quando dois ou mais processos tentam aceder ao mesmo recurso, bloqueiam-se mutuamente impedindo a boa continuação do algoritmo.

Ocorre quando várias threads do conjunto estão à espera da libertação de um recurso por uma outra thread que, por sua vez, aguarda a libertação de outro recurso alocado ou dependente da primeira thread.

A **starvation** por seu lado, ocorre quando uma thread nunca é executada por estar à espera de uma resposta que uma outra thread supostamente lhe dará, contudo esta última thread por algum motivo (a thread pode morrer ou ficar bloqueada) nunca lhe dá a resposta esperada. Diz-se que neste caso a thread se torna um zombie, esperando uma resposta que possivelmente nunca terá.

Quando o escalonamento não é feito adequadamente, pode haver **inanição**. Uma solução para esta situação é a existência de um tempo máximo de espera.

A ocorrência da inanição dá-se também quando os programas entram em ciclo infinito (razão pela qual também se dá o nome de preterição indefinida a esta situação)

e não fazem nenhum progresso no processamento, ao contrário do deadlock, que ocorre quando os processos permanecem bloqueados, dependendo da liberação dos recursos por eles alocados.