UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

RELATÓRIO DE ANÁLISE DE MECANISMOS DE SINCRONIZAÇÃO E PROBLEMAS DE CONCORRÊNCIA EM AMBIENTES *MULTITHREADED*

Sadi Júnior Domingos Jacinto

Professor orientador: Odorico Machado Mendizabal

Florianópolis
2019

Sadi Júnior Domingos Jacinto

RELATÓRIO DE ANÁLISE DE MECANISMOS DE SINCRONIZAÇÃO E PROBLEMAS DE CONCORRÊNCIA EM AMBIENTES *MULTITHREADED*

Análise de mecanismos de sincronização e problemas de concorrência em programação paralela em ambientes *multithreaded* requerido pelo professor da disciplina Programação Paralela e Distribuída, Odorico Machado Mendizabal, necessário para obtenção de nota.

Professor orientador: Odorico Machado Mendizabal

Florianópolis

2019

1 PROBLEMA DOS LEITORES E ESCRITORES

O problema dos leitores e escritores é um dos problemas clássicos da programação concorrente.

Este problema é uma abstração do acesso à base de dados, onde a leitura dos dados não apresenta perigos, mesmo se feita por vários processos, enquanto a escrita dos dados deve ter um tratamento diferente, onde a exclusão mútua deve ser aplicada para garantir a consistência dos dados.

Neste problema, divide-se os processos em duas classes:

- Leitores: Processos ou *threads*, os quais acessam mas não alteram a informação. Dessa forma pode-se ter vários leitores utilizando ao mesmo tempo a informação compartilhada, uma vez que nenhum deles realizará alterações na mesma.

- Escritores: Processos ou *threads*, os quais acessam a informação e fazem alterações na mesma, sendo, portanto, requerido o acesso exclusivo em detrimento aos outros processos ou *threads*, sejam eles leitores ou escritores, com o intuito de garantir a consistência do dado.

De forma resumida, o problema se caracteriza quando existem diversos leitores e escritores buscando ler/modificar um dado compartilhado. É necessário então que quando um escritor estiver atualizando o dado, nenhum outro leitor ou escritor possa acessar aquele dado, visto que os leitores podem acabar lendo um dado desatualizado e os escritores podem sobreescrever esse dado.

Analisando uma situação de um banco de dados localizado em um servidor, por exemplo, tem-se situações relacionadas ao caso do problema dos leitores e escritores. Supondo que existem usuários ligados a este servidor querendo ler dados em uma tabela chamada Estoque, a princípio todos os usuários terão acesso a esses dados. Supondo agora usuários querendo atualizar na mesma tabela de Estoque, informações de vendas realizadas, de fato esses dados serão atualizados. Mas para organizar esses acessos tanto de atualização quanto de leitura no banco de dados, algumas políticas devem ser seguidas, sendo que o mesmo se aplica ao problema dos leitores e escritores.

A solução ideal deve permitir que vários leitores possam acessar o dado ao mesmo tempo, mas quando um escritor esta atualizando o dado nenhum outro processo deve ter acesso.

O padrão de exclusão aqui pode ser chamado de exclusão mútua categórica. Um processo leitor na seção crítica não exclui necessariamente outros processos leitores, mas a presença de uma processo escritor na seção crítica exclui outras processos.

Este problema pode ser abordado de várias formas. Em uma das abordagens, dá-se preferência ao escritor, ou seja, o escritor, uma vez desejando acesso à memória compartilhada, a obtém, mesmo que processos leitores estivessem esperando antes.

Em outra abordagem, dá-se preferência aos leitores, ou seja, o escritor só obtém acesso à memória compartilhada quando um todos os escritores liberam o acesso. Esta situação é perigosa e pode levar a um *starvation* do escritor no caso em que existem vários processos leitores.

2 POSSÍVEIS RESOLUÇÕES DO PROBLEMA

Primeiramente é importante frisar que existem diversas soluções para esse problema, algumas utilizando *mutex*, outras utilizam semáforos, outros ainda utilizam os dois. Fora isso, existem soluções que compartilham o dado e promovem a concorrência através da criação de *threads* enquanto outros soluções utilizam processos separados acessando uma memória compartilhada ou um arquivo para tal.

Dito isso, e visando manter uma maior coerência deste relatório para com o leitor, não haverá um aprofundamento em todas as possíveis soluções para esse problema, ao invés disso, haverá uma explicação de qual deve ser a lógica utilizada pelo desenvolvedor para obter a melhor solução. Para tal, serão apresentadas duas soluções, melhor ao longo deste relatório.

As políticas seguidas no caso dos leitores e escritores para acesso a região critica são as seguintes:

- processos ou *threads* leitores somente leem o valor da variável compartilhada (não alteram o valor da variável compartilhada), podendo ser de forma concorrente;
- processos ou *threads* escritores podem modificar o valor da variável compartilhada, para isso necessita de exclusão mutua sobre a variável compartilhada;
- durante escrita do valor da variável compartilhada a operação deve ser restrita a um único escritor;
- para a operação de escrita não se pode existir nenhuma leitura ocorrendo, ou seja, nenhum leitor pode estar com a região critica bloqueada;
- em caso de escrita acontecendo, nenhum leitor conseguirá ter acesso ao valor da variável.

Uma das soluções para esse problema, que se encontra no arquivo **first.c**, consiste na adição de garantias de acesso à região crítica pelos processos ou *threads*. Para obter o acesso à base de dados, o primeiro leitor faz um *down* no semáforo *db*. Os leitores subsequentes meramente incrementam um contador, *rc*. Conforme saem, os leitores decrescem o contador em uma unidade e o último leitor a sair faz um *up* no semáforo, permitindo que um eventual escritor bloqueado entre.

De forma simplificada o funcionamento dessa solução pode ser interpretado da seguinte forma:

```
1 sem_t mutex;  /* controla o acesso a 'rc', que será usada posteriormente
    para verificar se existem leitores na seção crítica */
2 sem_t db; /* controla o acesso a base de dados. Neste caso se trata de um
    acesso direto na memória, mas poderia ser um acesso à um arquivo ou
    banco de dados */
3 int rc = 0; /* número de processos lendo ou querendo ler */
4
```

```
void reader(void) {
  while(TRUE) { // repete para sempre
    down(&mutex); // obtém acesso exclusivo a região critica
    rc = rc + 1; // um leitor a mais agora
    if (rc == 1)
      down(&db); //se este for o primeiro leitor bloqueia a base de dados

    up(&mutex) // libera o acesso a região critica

    read_data_base(); //acesso aos dados

    down(&mutex); // obtém acesso exclusivo a região critica
    rc = rc -1; // menos um leitor
    if (rc == 0)
      up(&db); // se este for o último leitor libera a base de dados

    up(&mutex) // libera o acesso a região critica
  }
}
```

```
void writer(void) {
  while (TRUE) { // repete para sempre
  down(&db); // obtém acesso exclusivo
  write_data_base(); // atualiza os dados
  up(&db); // libera o acesso exclusivo
  }
}
```

Tal solução, apesar de evitar a espera ocupada[1] e ter um bom desempenho, contém um problema sutil que vale a pena ser comentado. Suponha que, enquanto um leitor está usando a base de dados, um outro leitor chegue. Como ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Um terceiro leitor e outros subsequentes podem também ser admitidos se chegarem.

Agora, imagine que chegue um escritor. O escritor não pode ser admitido na base de dados, pois escritores devem ter acesso exclusivo. O escritor é então suspenso. Leitores adicionais chegam. Enquanto houver pelo menos um leitor ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver um fluxo estável de leitores chegando, todos entrarão assim que chegarem. O escritor permanecerá suspenso até que nenhum leitor esteja presente. Se um novo leitor chegar - supondo a cada dois segundos - e cada leitor levar cinco segundos para fazer seu trabalho, o escritor nunca entrará, sofrendo, então, de *livelock* ou *starvation*.

O programa poderia ser escrito de um modo um pouco diferente: se um leitor chegar quando um escritor estiver esperando, o leitor será suspenso logo depois do escritor, em vez de ser admitido de imediato. Essa solução, assim como as soluções do problema convencional, possuem diversas formas de ser implementada, como, por exemplo, ao usar um *mutex* que garante que o escritor poderá acessar a zona crítica, ou com uma variável global que indica se um escritor está querendo entrar na zona crítica. Independente de como a solução seja implementada, o escritor, para entrar na zona crítica, precisa esperar

[1] que é um dos maiores problemas de comunicação entre processos ou *threads*

apenas pelos leitores que estavam ativos quando ele chegou, mas não por leitores que chegaram depois dele. A desvantagem dessa solução é que se consegue menos concorrência e, portanto, um desempenho menor. Tal solução está presente no arquivo **second.c**.

No demais, segue, em anexo, uma Rede de Petri que apresenta uma solução genérica para o problema.