Comunicação e sincronização de *threads*
Sistemas Operacionais

Daniel Terenzi Carvalho

Ciências da Computação – Pontifícia Universidade Católica de Minas Gerais (PUCMinas)
Belo Horizonte – MG – Brasil

1. Introdução

Foi escolhido para este trabalho um dos problemas clássicos de concorrência, o problema dos leitores e escritores. Este problema consiste em diferentes *threads,* ou trabalhadores, realizarem operações de escrita ou leitura em uma mesma base de dados, gerando então uma competição pelos mesmos recursos. Sendo assim, é necessário implementar soluções para manter a consistência da base de dados, visto que há possibilidade de uma *thread* interferir no funcionamento de outra.

2. Algoritmo

Para resolver o problema, decidiu-se implementar a solução de Courtois, Heymans e Parnas (1971), que permite que sejam realizadas leituras simultaneamente, pois estas não causam interferências entre si, e somente uma escrita é realizada por vez, visto que estas interferem tanto em escritas quanto leituras. E, caso haja um escritor esperando, requisições de leitura que chegarem deverão esperar a escrita, evitando assim que o escritor espere indefinidamente no caso de sempre chegarem novas requisições de leitura.

3. Implementação

A solução foi implementada de duas formas, primeiramente na linguagem C++, utilizando dos recursos *mutex* e semáforos do Windows, contidos na biblioteca Windows.h. E posteriormente foi feita uma implementação em Java, utilizando dos monitores disponibilizados pela linguagem através dos métodos *synchronized*.

Para fins de comparação, ambas implementações seguem a mesma lógica de funcionamento, uma *thread* principal inicia uma *thread* despachante, responsável por processar requisições de escrita ou leitura e despachar *threads* trabalhadoras para completar tais requisições. A proteção de acesso simultâneo é implementada dentro da base de dados, organizando a espera das *threads* trabalhadoras pelos recursos, e, quando uma *thread* trabalhadora finaliza sua execução, os recursos utilizados pela *thread* são liberados, uma notificação é enviada ao despachante utilizando o padrão observador e o trabalhador é colocado em uma lista de trabalhadores disponíveis.

Ao fim das requisições, a *thread* despachante espera pela finalização de todas as *threads* trabalhadoras ainda pendentes, e a *thread* principal espera a finalização da despachante para concluir a execução do programa.

4. Análise de resultados

Ambas implementações foram testadas em um computador com processador Intel® Core™ i3-4000M 2.40 GHz, 8,00 GB de memória RAM, no sistema operacional Windows 8.1 Pro. Para manter a consistência do ambiente de testes, ambos os programas foram executados de forma isolada, somente com processos do sistema operacional em execução, e utilizando o mesmo arquivo de entrada “database.txt” fornecido junto à proposta do trabalho.

O tempo de execução de cada programa foi obtido da média de 20 medições realizadas em sequência. Para o programa em C++, as medições dos tempos de execução foram realizadas utilizando a classe high_resolution_clock da biblioteca *chrono* presente no padrão C++11 e a média obtida foi 103,02 segundos. Em Java as medições foram feitas usando a função System.nanoTime() e a média obtida foi 94,81 segundos.

Em virtude da JVM e de comparativos de performance entre Java e C++, o resultado esperado era que o programa em C++ apresentasse melhor desempenho, porém, observa-se o contrário. Possivelmente o resultado obtido deve-se à forma como o Java resolve e implementa seu sistema de monitores, que neste caso adiciona um custo menor de tempo para gerenciar recursos compartilhados que os *mutex* e semáforos implementados em C++ pelo programador.

5. Conclusão

Os principais desafios da implementação em C++ foram: encontrar um modo de gerenciar os trabalhadores ociosos e notificar ao despachante quando um trabalhador finalizasse sua execução, que foi resolvido utilizando o padrão observador e uma fila de trabalhadores ociosos, e, a implementação da lógica de acesso concorrente usando os *mutex* e semáforos.

Em Java, o principal desafio foi implementar o acesso concorrente usando monitores, de forma que a lógica de execução fosse a mesma do programa em C++, para que os dois pudessem ser comparados. Esta implementação foi tão difícil quanto à dos *mutex* e semáforos em C++, o que é de certa forma inesperado, visto que o uso de monitores pelo Java tem, em grande parte, o objetivo de retirar a complexidade do controle de acesso concorrente das mãos do programador.

Em vista das dificuldades encontradas e, principalmente, dos tempos de execução obtidos, conclui-se que há vantagens em implementar esta solução em Java, quando comparado à C++.