MÉTODOS DE SINCRONIZAÇÃO DE THREADS

TEIXEIRA,Felipe Leivas
Universidade Federal de Pelotas
Curso de Bacharelado em Ciência da Computação
flteixeira@inf.ufpel.edu.br

PILLA,Maurício Lima e DU BOIS, André Rauber
Universidade Federal de Pelotas
CDTec - UFPEL
{pilla,dubois}@inf.ufpel.edu.br

1 INTRODUÇÃO

A programação concorrente está conquistando cada vez mais espaço e importância na computação, isso se deve à grande evolução dos computadores, que nos dias de hoje contam com múltiplos processadores. Isso favorece a programação concorrente visto que ela pode explorar estes múltiplos processadores de forma a melhorar o desempenho de um programa.

Um dos problemas da programação concorrente é a condição de corrida (SILBERSCHATZ, 2000), que consiste na situação em que vários *threads* ou processos acessam e manipulam os mesmos dados concorrentemente e na qual o resultado da execução depende da ordem específica em que o acesso ocorre.

Para evitar que ocorra a condição de corrida *(race condition)* existem vários métodos de sincronização de *threads* entre eles o mutex (locks) e memórias transacionais, que serão descritos e comparados nas seções seguintes.

As próximas seções serão apresentadas da seguinte forma: a Seção 2 descreve as duas técnicas de sincronização citadas acima, a Seção 3 compara as técnicas e a Seção 4 apresenta as conclusões e os trabalhos futuros.

2 METODOLOGIA (MATERIAL E MÉTODOS)

Um dos métodos de sincronização mais utilizados é o mutex (TANENBAUM, 2010). A palavra mutex vem da abreviação de *mutual exclusion*, que significa exclusão mutua, que é um método de sincronização que tem como principio que se um *thread* entrou em uma seção critica, nenhum outro poderá acessar essa seção enquanto esse *thread* estiver nela.

O mutex utiliza um valor binário que informa se a seção crítica está bloqueada ou não, ou seja, se um *thread* está acessando essa seção ou não. Esse valor, quando for zero, quer dizer que a seção está bloqueada, e se for um quer dizer que a seção está liberada para ser acessada.

A área de memória compartilhada por vários *threads* é chamada de **seção crítica**. Se um *thread* quer acessar a seção crítica, primeiramente ele verifica

se o mutex está ou não ativado, ou seja, verifica se não tem nenhum outro *thread* já acessando a seção. Se ninguém estiver acessando a seção, o *thread* então ativa o mutex com o mutex_lock, que bloqueia a seção crítica alterando o valor do mutex para 0, não deixando assim mais nenhuma *thread* acessar a seção.

Se a seção crítica estiver bloqueada, o *thread* pode utilizar o *thread_yield* que faz com que o *thread* bloqueado abra mão de seu processador para que outro *thread* possa utilizá-lo, assim não perdendo tempo parado e ocupando um processador. Depois de um *thread* ter feito acessos a seção crítica ele utiliza o mutex_unlock que à desbloqueia assim deixando que outro *thread* acesse a seção. Caso haja mais de um *thread* esperando pelo desbloqueio de uma seção crítica, e essa tenha sido desbloqueada então um *thread* é escolhido aleatoriamente para acessar a seção.

Um outro método de sincronização são as **memórias transacionais** (BANDEIRA, 2010) que se baseiam nas transações de bancos de dados. As características baseadas nas transações de banco de dados são as seguintes:

A primeira característica é a **atomicidade**, que significa que ou todas as ações de uma transação executam sem nenhum erro ocorrendo então um *commit,* ou em alguma ação da transação ocorreu um erro acontecendo um *abort.* Quando ocorre um *abort* as ações são abortadas e reiniciadas até que ocorra um *commit.*

A segunda característica é a **consistência**, que quer dizer que as modificações geradas a partir de uma transação não podem deixar que os dados da memória fiquem inconsistentes, ou seja, não podem deixar que os dados da memória estejam incorretos.

A terceira característica é a do **isolamento**, que significa que uma transação tem que produzir resultados corretos independentemente de outras transações, ou seja, uma transação não pode atrapalhar a outra.

A quarta e ultima característica é a de **durabilidade**, que significa que os dados gerados no final de uma transação terão que ser permanentes e devem estar disponíveis, enquanto não acabar a execução, para que as transações seguintes possam utilizar esses dados.

CMTJava é uma linguagem de domínio específico (LDE) para qual foi desenvolvido um compilador (BANDEIRA, 2010). A CMTJava foi desenvolvida para facilitar a programação em máquinas com múltiplos processadores (*multi-core*). Ela permite a abstração dos objetos transacionais, assim permitindo que seus atributos somente podem ser acessados através de métodos especiais de *get* e *set*.

3 RESULTADOS E DISCUSSÃO

A grande diferença entre as memórias transacionais e o método de exclusão mútua é que as memórias transacionais não utilizam *locks* explicitamente para fazer a sincronização, evitando a possibilidade de *deadlocks* devido a erros de programação.

Outra vantagem das memórias transacionais é a facilidade de se programar, pois em vez de sincronizar cada parte do código que acessa uma seção crítica como faz a exclusão mútua, as memórias transacionais colocam as seções críticas dentro de uma transação atômica.

Até o momento foi feito o estudo dos métodos de sincronização, onde foi estudado como funciona cada método que ferramentas utilizar para desenvolvê-los, como desenvolvê-los. Como até agora foi feito um estudo sobre os métodos os resultados não estão prontos, mas estarão assim que os testes forem desenvolvidos.

4 CONCLUSÃO

O trabalho desenvolvido até agora foi para o aprendizado dos métodos de sincronização, com eles funcionam e como implementá-los. Ficou claro que o uso de memórias transacionais permite programar em mais alto nível e com menos contenção e serialização de código, assim atingindo um potencial de desempenho maior para as arquiteturas *manycore* esperadas para o futuro.

Para o futuro pretende-se fazer uma comparação mais aprofundada entre os métodos, em relação ao tempo que cada um leva para executar um mesmo algoritmo. Também pretende-se medir a energia gasta e a contenção de memória em ambos os métodos.

5 REFERÊNCIAS

TANENBAUM, Andrew S. Processos e threads. In: TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Editora: Prentice Hall -Br, 3º Ed 2010. Cap. 2 50 – 106

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. Sistemas Operacionais: Conceitos e Aplicações. Tradução de Adriana Ceschin Rieche. Rio de Janeiro, 2000. Cap. 7 122 – 160

BANDEIRA, Rafael de Leão. **Compilador para a linguagem CMTJava.** Trabalho de Conclusão de Curso – Bacharelado em Ciência da Computação. Universidade Federal de Pelotas, Dezembro 2010.