Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Coordenação de Pós-Graduação em Informática - COPIN

PROJETO DE SOFTWARE ORIENTADO A OBJETO

RESUMO - THREADS

Maio de 2008

João Arthur Brunet Monteiro
Mestrado em Ciências da Computação - CEEI/UFCG
Campina Grande, Maio de 2008

1 Resumo

Este documento traz uma sumarização dos artigos Software and the Concurrency Revolution [2] e The problem with Threads [1], além de minhas opiniões a respeito do assunto abordado pelos autores.

Os autores discutem as implicações de concorrência em *software* e suas consequências para programadores e linguagens de programação. Inicialmente, os autores lembram que com a crescente fabricação de processadores de vários núcleos, o software precisa usar concorrência em sua execução para acompanhar o desempenho do hardware. Isto implica em construir cada vez mais *software* concorrente, fato este que requer mudanças nas linguagens de programação e na maneira como os programadores pensam ao construir um software. Para os autores, esta mudança é difícil de ser implantada, pois não existem ferramentas adequadas e a forma de se programar uma aplicação paralela não é natural para o programador.

Para os autores, concorrência em *software* causa uma ruptura ainda maior do que a causada pelo paradigma OO quando foi apresentado. Como exemplo é citada a linguagem C, que ainda hoje permance sendo usada em larga escala apesar de ser imperativa. Isto porque concorrência pode ser adicionada a uma liguagem através da adição de módulos (e. g., pthread). O paradigma OO, por sua vez, requer mudanças profundas na linguagem. De fato, para os autores, linguagens que não abordarem concorrência tenderão a ficarem obsoletas, visto que o *hardware* tem sido fabricado cada vez mais com vários núcleos. Além disso, construir software concorrente é uma atividade extremamente mais difícil do que o modelo sequencial.

Três modelos de paralelismos que diferem entre si pelo grau de acoplamento entre as instruções e a granularidade das operações paralelas são apresentados:

- Paralelismo independente: É o tipo mais simples de paralelismo. Aquele em que as tarefas são completamente independentes e podem ser executadas seguindo qualquer ordem, além de não compartilharem dados de entrada ou saída.

- Paralelismo regular: É o tipo de paralelismo em que as tarefas são mutuamentes dependentes.

- Paralelismo não-estruturado: Tipo de paralelismo mais geral, aquele em que é preciso coordenar o acesso aos dados através de sincronização, uma vez que não é possível preve-lo. Este tipo de paralelismo é muito comum em programas que usam *threads* e sincronização explícita.

Um problema com o paralelismo não-estruturado surge quando duas ou mais *threads* acessam e modificam o estado de uma variável ao mesmo tempo, causando inconsistências. A solução mais usada para este problema é o uso de *locks*, estruturas garantem acesso exclusivo a um recurso. *Locks* são bastante utilizados e evitam condições de corrida. No entanto, eles não são agrupáveis, ou seja, dois blocos que usam *locks* separadamente, quando juntos podem não formar um *lock*. Além disso, a união destes dois blocos podem causar *deadlock* na execução do programa.

São apresentadas duas alternativas para *locks*. A primeira baseia-se na construção de programas sem locks, usando conhecimentos profundos sobre o modelo de memória da linguagem. Acredito que essa alternativa seja totalmente inviável pelo fato de requerer conhecimento sobre o modelo de memória da linguagem. Esses conhecimentos são comumente abstraídos pelos programadores, além de serem demasiadamente complicados. A segunda alternativa, *transactional memory*, que me parece mais viável, é a construção de programas com instruções essencialmente atômicas.

Os autores ressaltam alguns pontos que são necessários para lidar melhor com programação concorrente, tais como, melhores abstrações, uso de linguagens funcionais, paralelização automática e ferramentas apropriadas.

Em minha opinião, acredito que o uso de boas práticas de programação podem amenizar os problemas que surgem de programação concorrente. Por exemplo, o uso de objetos imutáveis em linguagens de programação podem ajudar no contexto de evitar condições de corrida.

Outros autores [1] advogam que o uso de *threads* é precário e causa problemas sérios de sincronização como *deadlocks*. Estes problemas surgem da escolha errada das abstrações para concorrência, que tornam os programas incompreensíveis para humanos. Para os autores, é preciso descartar *threads* como modelo de programação, partindo da construção de um modelo simples baseado no princípio de que o determinismo deve ser explícito e cuidadosamente utilizado.

Alternativas a *threads* e são apresentadas pelos autores. Além disso, desafios futuros, como a introdução de uma linguagem de coordenação são apontados como uma das técnicas para amenizar o problema.

Acredito ser de extrema importância mudar a forma como programamos concorrentemente. Ainda, acho que o problema não reside em *threads*, e sim na forma como a usamos. As alternativas apresentadas pelos autores me parecem interessantes no que diz respeito a criação de linguagens para coordenar a execução dos programas. No entanto, reafirmo que *threads* não constituem a maior parte do problema. O problema está na forma como abstraímos concorrência e na forma como a implementamos.

Referências

[1] E. A. Lee. The problem with threads, 2006.

[2] H. Sutter and J. Larus. Software and the concurrency revolution, 2005.