Universidade Federal de Pernambuco

Bacharelado em Ciência da Computação

Proposta do Trabalho de Graduação

Adicionando Informações Contextuais a Exceções de Deadlock

Aluna: **Maria Gabriela Toledo de Moraes Cardoso**

Orientador: **Fernando José Castor de Lima Filho**

15 de Abril de 2016

Sumário

1. Contexto

Processadores multi-núcleo são processadores capazes de executar duas ou mais instruções em paralelo. Nos últimos 12 anos [1], esses processadores tornaram-se pervagantes em dispositivos computacionais, como uma forma de melhorar o desempenho desses dispositivos. Como mencionado em [2] programas *multithreaded* são pervasivos e difíceis de escrever quando as *threads* compartilham recursos. Para lidar com este compartilhamento de recursos é comum os programas utilizarem sincronização. Contudo uma sincronização inadequada de *threads* pode levar a *bugs* de corretude, como condições de corrida e violações de atomicidade, enquanto uma forte sincronização causa problemas de desempenho.

Em softwares *multithread*, a possibilidade de acontecer *deadlock* deve ser considerada. *Deadlocks* podem ser formalmente definidos da seguinte forma: Um conjunto de processos está em *deadlock,* se cada processo no conjunto está à espera de um evento que só outro processo no conjunto pode fazer acontecer [3]. Existem dois tipos bem documentados de *deadlocks, deadlocks* de recursos e *deadlocks* de comunicação. Em *deadlocks* de recursos a espera acontece porque alguma *thread* está esperando para obter um recurso de outra. Em *deadlocks* de comunicação uma ou mais *threads* estão esperando por uma mensagem [4].

Uma forma comum de se depurar *deadlocks* é parar a execução do programa e tentar analisar o status atual. Com o intuito de entender o momento em que o *deadlock* aconteceu, onde e quais recursos estão bloqueados. Entretanto esta abordagem pode ser falha, dado que em programas *multithreaded* nem sempre podemos garantir a ordem de execução das *threads* envolvidas. Sendo assim o programa pode apresentar respostas diferentes para uma mesma entrada.

Para lidar com esta dificuldade, desenvolvedores criaram formas de detectar ou prever *deadlocks* como: detecção estática [5, 6], detecção dinâmica [7, 8] e verificação de modelos [9]. Todavia, todas elas apresentam alguns problemas, podem: gerar de falso-positivos, ter um alto *overhead,* falta de informações necessárias para corrigir *o deadlock,* bem como escopos limitados.

No trabalho apresentado em [10] foi realizado um estudo a cerca dos *bug reports* de projetos *open-source* em Java e foi constatado que a maioria dos *deadlocks* que ocorrem em sistemas reais envolvem apenas duas *threads* adquirindo dois *locks.* Após este estudo, foi proposta uma abordagem diferente para detecção de *deadlocks*. Os autores criaram um novo protocolo onde o *deadlock* é detectado durante a execução e exibido na forma de exceções. As informações exibidas aos usuários são as disponíveis no *stacktrace*, ou seja, as classes e os métodos envolvidos.

Para avaliar o trabalho desenvolvido em [10], foram realizados experimentos com dois nichos diferentes, alunos de graduação pouco experientes e alunos de mestrado familiarizados com programação concorrente. Entretanto um dos programas dos experimentos era tão simples que os alunos do segundo grupo identificaram o problema de forma precisa rapidamente, mesmo quando não utilizaram a abordagem proposta. Programas assim não são comuns no mundo real.

Programas no mundo real são mais complexos, o problema pode ser mais difícil de identificar, uma *thread* pode criar outra, uma *thread* pode adquirir vários *locks* em diversos momentos e mais de uma *thread* da mesma classe podem ser criadas. Entretanto a exceção lançada não informa nenhum destes dados. Estas informações podem ser bastante úteis no momento da depuração do código, permitindo que o problema seja identificado de forma mais rápida e com uma maior precisão.

2. Objetivos

O objetivo deste trabalho é ampliar o protocolo apresentado em [10], adicionando mais informação às *threads* e consequentemente à exceção lançada após a detecção. Com isto o foco deste estudo é melhorar a rastreabilidade dos problemas, informando mais detalhes sobre o *deadlock* encontrado. Entretanto almeja-se fazer essas alterações sem aumentar muito o *overhead* de criação e execução de *threads.*

Será realizado um estudo sobre quais informações são buscadas pelos desenvolvedores quando estes se deparam com um *deadlock.* Após estabelecer as necessidades dos usuários será definido como estes dados serão coletados. Não basta coletar todos os dados, tem-se que avaliar se estas informações adicionais não tornam o programa muito lento. O objetivo em questão é facilitar o entendimento do problema, sem prejudicar a performance.

Por fim será implementada uma abordagem amigável para exibir estas informações adicionais aos usuários. Esta atividade apresenta grande importância, pois como um dos intuitos deste projeto é fazer com os usuários entendam o *deadlock* de forma mais precisa, a abordagem escolhida terá que mostrar de forma clara os motivos e os envolvidos no problema encontrado.

3. Cronograma

Atividade	Março		Abril		Maio		Junho		Julho	
Estudo literário sobre programação concorrente										
Elaborar proposta										
Analisar informações adicionais para coletar										
Estabelecer como coletar novas informações										
Avaliar o overhead										
Implementar abordagem mais amigável para exibir as informações adicionais										
Escrever relatório final										
Apresentação										

4. Lista de Possíveis Avaliadores

Henrique Rebelo
Kiev Gama
Sérgio Soares

5. Referências Bibliográficas

[1] Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software." Dr. Dobb's journal 30.3 (2005): 202-210. <http://www.gotw.ca/publications/concurrency-ddj.htm> Acessado em: 14/04/2016

[2] Gu, Rui, et al. "What change history tells us about thread synchronization."Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015. <http://dl.acm.org/citation.cfm?id=2786815> Acessado em: 11/04/2016

[3] Tanenbaum, Andrew. "Modern operating systems." (2009).

[4] Singhal, Mukesh. "Deadlock detection in distributed systems." *Computer*22.11 (1989): 37-48. <http://www.kiv.zcu.cz/~ledvina/ds/~singhaldeadsurvey.pdf> Acessado em: 13/04/2016

[5] Williams, Amy, William Thies, and Michael D. Ernst. "Static deadlock detection for Java libraries." *ECOOP 2005-Object-Oriented Programming*. Springer Berlin Heidelberg, 2005. 602-629. <http://people.csail.mit.edu/amy/papers/deadlock-ecoop05.pdf> Acessado em: 12/04/2016

[6] Naik, Mayur, et al. "Effective static deadlock detection." *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009. <http://www.cc.gatech.edu/~naik/pubs/icse09.pdf> Acessado em: 12/04/2016

[7] Li, Tong, et al. "Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution." *USENIX Annual Technical Conference, General Track*. Vol. 44. 2005. <http://static.usenix.org/event/usenix05/tech/general/full_papers/li/li_html/> Acessado em: 12/04/2016

[8] Joshi, Pallavi, et al. "A randomized dynamic program analysis technique for detecting real deadlocks." *ACM Sigplan Notices* 44.6 (2009): 110-120. <http://www.cc.gatech.edu/~naik/pubs/pldi09b.pdf> Acessado em: 12/04/2016

[9] Havelund, Klaus, and Thomas Pressburger. "Model checking java programs using java pathfinder." *International Journal on Software Tools for Technology Transfer* 2.4 (2000): 366-381.
<https://www.researchgate.net/publication/2823469_Model_Checking_Java_Programs_Using_Java_PathFinder> Acessado em: 13/04/2016

[10] Lobo, Rafael, and Fernando Castor. "Deadlocks as Runtime Exceptions."*Programming Languages*. Springer International Publishing, 2015. 96-111. <http://link.springer.com/chapter/10.1007/978-3-319-24012-1_8> Acessado em: 13/04/2016

6. Assinaturas

_____

Aluna: Maria Gabriela Toledo de Moraes Cardoso

_____

Orientador: Fernando José Castor de Lima Filho