Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Adicionando Informações Contextuais a Exceções de Deadlock

Maria Gabriela Toledo de Moraes Cardoso

Trabalho de Graduação

Recife, Julho de 2016

Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Adicionando Informações Contextuais a Exceções de Deadlock

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Aluna: Maria Gabriela Toledo de Moraes Cardoso
(mgtmc@cin.ufpe.br)

Orientador: Fernando José Castor de Lima Filho
(fjclf@cin.ufpe.br)

Recife, Julho de 2016

Agradecimentos

Primeiramente eu tenho que agradecer aos meus pais por todo o apoio que deram a mim durante a minha vida, desde o início da vida escolar até a conclusão do curso. É incrível o incentivo que eu recebo deles, desde as perguntas de como eu estava na faculdade, como eu tinha ido nas provas, bem como no início da minha vida profissional. Muitas vezes quando eu achava que ia falhar, meus pais estavam lá para dizer que eu conseguiria, que eu era capaz. Bastava eu batalhar muito por isso. Vocês me inspiram, levarei todos os ensinamentos de vocês pelo resto da minha vida.

Agradeço também ao meu orientador Fernando Castor por estar sempre disposto a me ajudar e tirar dúvidas. Sem ele esse trabalho não seria possível. Castor estava sempre paciente para tirar minhas dúvidas e debater soluções. Além da ajuda neste trabalho, Castor também me ajudou a conquistar o emprego dos meus sonhos, pois quando eu estava estudando para o processo seletivo Castor esclareceu diversas dúvidas que eu tinha acerca do conteúdo estudado.

Agradeço aos meus amigos de turma, pois sem eles, minha graduação teria sido muito mais complicado. Desde o primeiro período nós nos juntávamos para estudar e resolver as diversas listas e projetos. O companheiro de todos eles me fez ser uma pessoa mais feliz durante esses 4 anos e meio. Obrigada também pelos momentos de diversão e descontração, momentos estes essenciais para me manter sã durante os finais de período. Jamais esquecerei toda ajuda e apoio que vocês me deram.

Agradeço aos meus amigos do PET e também ao meu tutor Fernando Fonseca. Vocês fizeram minha graduação valer a pena, uma vez PETiana, sempre PETiana. Nossas reuniões semanais não eram suficientes para mim, eu amava passar a tarde no PET só para ter a companhia de vocês. Me esforçarei ao máximo para estar presente no grupo mesmo estando longe fisicamente. Juntamente aos meus amigos de turma, vocês me inspiraram a ser uma pessoa mais esforçada, me ajudaram a batalhar pelos meus sonhos, me fizeram acreditar que era possível conquista-los.

Por fim, mas não menos importante, agradeço aos meus amigos de escola e aos meus melhores amigos da faculdade (Leonardo, Larissa e Marina) por passarem por todos os problemas comigo. Por passarem horas e horas fazendo projetos, estudando até 5 da manhã para provas e fazendo o meu dia-a-dia mais feliz. Sem vocês eu não teria aguentado passar por tudo isso.

Resumo

Programas *multithreaded* são 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 gerar diversos problemas, como por exemplo o *deadlock*. Para resolver este problema, desenvolvedores criaram formas de detecção ou prevenção como: detecção estática, detecção dinâmica e verificação de modelos. Recentemente foi criada uma nova abordagem onde o *deadlock* é detectado durante a execução e sua ocorrência é sinalizada através de uma exceção. O objetivo principal deste trabalho é adicionar informações contextuais a exceções de *deadlock* [LOBO; CASTOR, 2015], com o intuito de melhorar a rastreabilidade dos problemas, informando mais detalhes sobre o *deadlock* encontrado. Levando-se sempre em consideração o overhead criado. Para alcançar este objetivo foram feitos estudos para determinar quais informações são úteis para o usuário entender o *deadlock* e como a mensagem de erro deve ser exibida ao usuário. As informações foram adicionadas e seu *overhead* na performance foi medido através de um *benchmark* bastante conservador. No qual foi medido o *overhead* das operações de trava quando *deadlocks* não são possíveis.

Palavras-chaves: *Deadlock*, Programação Concorrente, Lançamento de Exceções, Rastreabilidade.

Abstract

Multithreaded programs are hard to write when threads share resources. To deal with this resource sharing, programs typically use synchronization. However inadequate thread synchronization may generate several problems such as deadlock. To solve this problem, developers have created forms of detection or prevention as: static detection, dynamic detection and verification models. Recently a new approach has been created where the deadlock is detected in runtime, and its occurrence is signaled by an exception. The main goal of this work is to add contextual information to deadlock exceptions [LOBO; CASTOR, 2015], in order to improve the traceability of problems, informing more details about the deadlock found. Always taking into account the overhead created. To achieve this goal studies have been made to determine what information is useful for the user to understand the deadlock and how the error message should be displayed to the user. The new informations were added and the performance overhead was measured using a very conservative benchmark. Which measured the overhead of lock operations when deadlocks are not possible.

Keywords: Deadlock, Concurrent Programming, Throwing Exceptions, Traceability

Sumário

1. Introdução

Processadores multi-núcleo são processadores capazes de executar duas ou mais instruções em paralelo. Nos últimos 12 anos [JAGTAP, 2009], esses processadores tornaram-se pervagantes em dispositivos computacionais, como uma forma de melhorar o desempenho desses dispositivos. Como observado em [GU et al., 2015], 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*, como condições de corrida e violações de atomicidade, enquanto sincronização excessiva causa problemas de desempenho.

Em sistemas de software *multithreaded*, a possibilidade de acontecer *deadlock* deve ser considerada. *Deadlocks* podem ser 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 [TANENBAUM, 2009]. 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 [SINGHAL, 1989].

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 [WILLIAMS; THIES; ERNST, 2005][NAIK et al., 2009], detecção dinâmica [LI et al., 2005][JOSHI et al., 2009] e verificação de modelos [HAVELUND; PRESSBURGER., 2000]. 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 [LOBO; CASTOR, 2015] foi realizado um estudo acerca 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 sua ocorrência é sinalizada na forma de exceções da linguagem Java. As informações exibidas aos usuários são as disponíveis no *stacktrace*, ou seja, as classes e os métodos envolvidos.

Para avaliar o trabalho desenvolvido em [LOBO; CASTOR, 2015], foram realizados experimentos com dois nichos diferentes, alunos de graduação pouco experientes e alunos de mestrado e doutorado 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 se identificar, uma *thread* pode criar outra, uma *thread* pode adquirir vários *locks* em diversos momentos e várias *threads* 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.

O objetivo deste trabalho é ampliar o protocolo apresentado em [LOBO; CASTOR, 2015], 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.*

Foi realizado um estudo sobre quais informações são buscadas pelos desenvolvedores quando estes se deparam com um *deadlock.* Após estabelecer as necessidades dos usuários foi definido como as informações úteis podem ser coletados. Não é suficiente coletar todas as informações possíveis pois isso pode tornar o programa muito lento ou sobrecarregar o desenvolvedor com informações. O objetivo em questão é facilitar o entendimento do problema, sem prejudicar o desempenho.

Por fim foi 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 que 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.

Este documento está estruturado da seguinte maneira. No Capítulo 2 são apresentados os assuntos que o leitor precisa entender para ser capaz de compreender este trabalho. Tais como: fundamentos e alguns problemas inerentes à programação concorrente. O Capítulo 3 contém os trabalhos relacionados a esta pesquisa, os quais incluem informações sobre detecção de *deadlocks* e sobre a incorporação em linguagens de programação de detecção de erros em tempo de execução sobre problemas de concorrência. Além do trabalho base para esta pesquisa e trabalhos sobre mensagens de erros. O Capítulo 4 contém todas as informações relacionadas à nova abordagem, desde o estudo para identificar que novos dados seriam acrescentados, passando pela implementação e por fim uma análise de *overhead*. Por fim, o Capítulo 5 contém as principais conclusões desta pesquisa e possíveis trabalhos futuros.

2. Fundamentos

Será apresentado neste capítulo assuntos que o leitor precisa entender para ser capaz de compreender o trabalho desenvolvido por nós. Primeiramente será introduzido o que é a programação concorrente e alguns de seus princípios como: *threads* e *locks*. Em seguida serão apresentadas informações sobre *deadlocks* e por fim explicações acerca do que são os *deadlocks* apresentados em formas de exceções e algumas de suas implementações em linguagens atuais.

2.1 Programação concorrente

Em meados da década de 60 a computação deu os primeiros passos para uma compreensão mais profunda sobre o que é a programação concorrente. Em menos de quinze anos foram descobertos conceitos fundamentais, os quais foram incluídos em linguagens de programação, e estas linguagens foram usadas para escrever sistemas operacionais. Na década de 70 os novos conceitos de programação foram usados para escrever os primeiros livros sobre os princípios de sistemas operacionais e programação concorrente [HANSEN, 2001].

A programação concorrente e aquela onde as atividades envolvidas na resolução de um programa encontram-se decompostas em atividades independentes e as relações de trocas de dados encontram-se igualmente identificadas [CAVALHEIRO, 2004]. Programas que utilizam da programação concorrente na sua implementação são chamados de programas concorrentes.

Diferente dos programas sequenciais, nos quais existe somente um fluxo de execução, os programas concorrentes podem ser vistos como programas que apresentam vários fluxos de execução e estes podem ser executados ao mesmo tempo, se o hardware subjacente der suporte a execução paralela. Cada fluxo apresenta um conjunto de instruções que devem ser executadas de forma sequencial. [TOSCANI; CARISSIMI, 2003] A partir de agora iremos utilizar a palavra tarefa como sinônimo para fluxo de execução.

Durante a execução de um programa, duas tarefas são consideradas independentes quando as instruções executadas por uma, não influenciam na execução da outra, portanto, não há restrições temporais para a execução delas. Caso contrário é necessário que haja uma forma de comunicação para garantir que cada tarefa seja executada de forma correta e ao mesmo tempo.

Como visto em [SHARAN, 2014], para controlar e coordenar o acesso a uma seção crítica por várias *threads* é utilizada uma técnica conhecida como sincronização de *threads*. Este mecanismo garante que o acesso a dados gerados/modificados por uma tarefa seja obtido corretamente por outras tarefas. A sincronização pode ser feita tanto para garantir que uma tarefa que dependa da outra seja executada apenas após a execução da outra tarefa, bem como garantir acesso exclusivo ao recursos compartilhados entre as tarefas. [BUSTARD, 1990]

Contudo, esse paralelismo pode ocasionar diversos problemas se não for corretamente implementado. Dentre os alguns destes problemas têm-se: condições de corrida, *deadlocks*, *livelocks* e falhas de atomicidade. Condições de corrida ocorrem quando dois ou mais processos estão lendo ou escrevendo em alguns dados compartilhados e o resultado final depende da ordem de execução das ações. *Deadlocks* e *livelocks* são similares, deadlocks acontecem quando um conjunto de threads fica bloqueado e cada thread fica esperando a outra terminar suas ações. Já em *livelocks* as *threads* tentam ser educadas, ao tentarem adquirir um novo *lock* e não conseguirem, ambas liberam as travas já adquiridas e tentam novamente obtê-las de novo depois. Sendo assim, elas não fazem nenhum trabalho útil, mas também não ficam em estado de espera [TANENBAUM, 2009]. Atomicidade é a propriedade que diz que cada execução simultânea de um conjunto de transações é equivalente a alguma execução serial das mesmas operações.

2.2 *Threads*

Em um único processo, se queremos executar tarefas de forma concorrente, normalmente falamos de *threads.* Uma nova *thread* é criada quando deseja-se criar um novo fluxo de execução. Em diversas linguagens de programação, no momento da criação da *thread* é informado o conjunto de instruções que ela deve executar e os dados iniciais para o processamento destas instruções. Seja passando um objeto que contém as instruções como é em Java[1] (utilizando a classe Thread do pacote java.lang) ou passando uma função como é em C++ (pthread).

Uma *thread* é uma unidade básica de utilização da CPU, a qual compreende um ID da *thread*, um contador de programa, um conjunto de registradores e uma pilha. Ele compartilha com outras *threads* pertencentes ao mesmo processo sua seção de código, seção de dados e outros recursos do sistema operacional, como arquivos abertos e sinais [SILBERSCHATZ; GALVIN; GAGNE, 2008].

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Um processo ou programa tem seu próprio espaço de endereçamento e de controle de blocos [TANENBAUM, 2009]. Dentro de um processo ou programa, podemos executar várias threads (*multithreading)* simultaneamente para melhorar o rendimento do execução através da otimização da utilização dos recursos do sistema.

Quando um programa *multithreaded* entra em execução, o escalonador de *threads* escolhe um processo para executar, dando controle para seu quantum. O escalonador então decide qual *thread* executar, e coloca esta *thread* para rodar. O critério para determinar o acesso à CPU pode variar [TANENBAUM, 2009]. A ordem de execução das *threads* é não determinística.

Em alguns sistemas operacionais, processos (e *threads)* podem compartilhar memória e cada um pode ler e escrever. O armazenamento pode ser partilhado na memória principal (possivelmente numa estrutura de dados do *kernel),* ou pode ser um arquivo compartilhado; a localização da memória partilhada não altera a natureza da comunicação ou os problemas que surgem. Situações como esta, em que dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem rodou precisamente quando, são chamados de condições de corrida [TANENBAUM, 2009].

Como as *threads* compartilham dados, estas são obrigadas a se preocupar com o que as outras estão fazendo com esses dados compartilhados. Um exemplo disso são as aplicações do tipo "Produtor/Consumidor", onde um produtor produz dados que serão consumidos por um consumidor, os dois funcionam de maneira autônoma (normalmente cada um modelado como uma *thread*) e têm velocidades potencialmente diferentes, o que exige que exista um *buffer* entre eles. Por exemplo, uma *thread* (o produtor) poderia estar adicionando itens numa lista enquanto outra *thread* (o consumidor) remove itens desta mesma lista.

O fato das *threads* serem assíncronas pode ocasionar outros erros de execução. No exemplo citado acima, temos uma *thread* que produz e outra que consome, mas vamos supor agora que exista mais de um consumidor. Vamos supor também que toda *thread* consumidor checa se a lista está vazia antes de remover um elemento da lista. Na figura abaixo temos uma ilustração de um caso onde há erro de execução.

```
TAM_BUFFER = 100
n = 0;     /* itens na fila*/

Thread Produtor :
      loop {
          if (n == TAM_BUFFER) {
              /* buffer lotado */
              espera_sinal_de_nao_lotado();
          }
          else {
              item = produz_item();
              enfileira_item(item);
              /* atualiza variavel compartilhada */
              n++;
          }
      } /* endloop */

Thread Consumidor:
      loop {
          if (n > 0) {
              item = remove_item()
              n--;
          }
          else {
              espera_sinal_de_nao_vazio();
          }
      } /* endloop */
```

Figura 1: Exemplo de código do problema Produtor/Consumidor

A *Thread* 1 checa e vê que a lista não está vazia (n > 0), entretanto o escalonador do sistema decide parar a sua execução antes que ela remova o elemento. A *Thread* 2 inicia a sua execução e checa o tamanho da lista, vê que esta não está vazia (n > 0) e remove um elemento. Quando a *Thread* 1 retomar a execução, esta irá tentar remover um item da lista, entretanto a lista agora está vazia e o programa irá dar erro.

Quando múltiplas *threads* estão em execução, provavelmente, estas terão que se comunicar umas com as outras. A comunicação entre threads normalmente ocorre através da memória global, ou seja, uma memória compartilhada. Isso requer construções de sincronização [SHARAN, 2014] para garantir que mais de uma *thread* não está atualizando um mesmo objeto, como vimos no exemplo citado anteriormente, ou até que uma *thread* tenha terminado todas as atualizações e processamentos necessários.

Dentre os métodos mais comuns para realizar a sincronização de *threads* temos: *locks* (*mutexes)*, variáveis condicionais e semáforos [TANENBAUM, 2009]. Neste trabalho iremos nos focar no primeiro método, *locks,* o qual será detalhado na seção a seguir.

2.3 *Locks* (*Mutex*)

A palavra *mutex* é uma abreviação de "*mutual exclusion"*, a qual significa exclusão mútua. Uma variável do tipo *mutex* é uma trava que protege dados compartilhados pelas *threads* [SILBERSCHATZ; GALVIN; GAGNE, 2008]. Estas variáveis são a principal forma que as *threads* apresentam para a proteção de regiões críticas [TANENBAUM, 2009]. Uma região crítica é um conjunto de linhas de código que deve ser totalmente concluído sem interrupção. Tipicamente, a região crítica acessa algum recurso compartilhado, tal como uma estrutura de dados, ou um dispositivo, o qual não permite múltiplos acessos simultâneos.

O princípio básico da sincronização por travas é garantir o acesso à região crítica apenas para uma *thread.* Mesmo que diversas threads tentem efetuar o travamento, apenas uma delas será bem-sucedida. Além disso, nenhuma outra *thread* poderá ter posse do *mutex* antes que o mesmo seja liberado. Apenas a *thread* que possui o *lock* pode liberar a trava [FLANAGAN, 2005]. A sequência de uso de uma variável do tipo *mutex* normalmente é: „

1. O *mutex* é criado / inicializado.
2. Diversas *threads* tentam efetuar o *lock.*
3. Apenas uma *thread* consegue adquirir o *mutex.* Todas as outras que tentaram adquirir a trava e não conseguiram ficam bloqueadas até que a *thread* vencedora libere o *mutex.*
4. A *thread* vencedora realiza o processamento desejado, ou seja, entra na seção (região) crítica.
5. Após realizar os processamentos, a *thread* vencedora libera o *mutex.*
6. Uma das *threads* que estavam bloqueadas é selecionada para adquirir o *mutex.* As restantes continuam bloqueadas.

Este processo de seleção de uma nova *thread* para adquirir o *mutex* se repete indefinidamente, até que todas as threads bloqueadas consigam efetuar o *lock.* O *mutex* apresenta dois estados: *locked* (bloqueado) e *unlocked* (desbloqueado) [TANENBAUM, 2009], *lock* significa que alguma *thread* já adquiriu aquele *mutex* e *unlock* significa que trava está livre e pode ser adquirida por alguma *thread.* Inicialmente um *mutex* está *unlocked.*

É importante esclarecer que o exemplo de sequência citado anteriormente é comum, mas não é o único. Existem variações no *mutex* [ALESSANDRINI, 2015] que podem modificar *c*ertas características citadas anteriormente. Em algumas linguagens, ao tentar adquirir o *mutex* a *thread* apenas tenta efetuar o *lock* e, caso não consiga, a mesma não fica bloqueada; contudo é preciso que a *thread* tente novamente obter a trava no futuro. Outro exemplo é o caso do ReentrantLock[2] de Java, no qual se uma *thread* já adquiriu o *lock* e, antes de liberá-lo, a *thread* tenta adquirir o *mutex* já obtido anteriormente. Nesta execução não haverá nenhum problema, a *thread* não ficará bloqueada, pois ela já obtém o *mutex.*

A sincronização serve para limitar o acesso à memória compartilhada durante a execução de programas concorrentes. Devido a essa grande responsabilidade de travamento, é preciso que os desenvolvedores tenham bastante cuidado ao tentarem fazer a sincronização das *threads,* caso contrário diversos problemas podem surgir. Principalmente pelo fato de que as *threads* podem adquirir mais de uma trava ao mesmo tempo e acabar bloqueando o acesso a diversas seções críticas.

2.4 *Deadlocks*

Em um ambiente de multiprogramação, várias *threads* podem competir por um número finito de recursos (como por exemplo *mutexes*). Se uma *thread* A tenta obter um recurso e este não está disponível, a *thread* A fica em estado de espera até que o recurso seja liberado. Contudo, esta espera pode nunca acabar. Se uma *thread* B obtém este recurso desejado pela *thread* A, mas esta está em estado de espera por causa de um recurso que a *thread* A já adquiriu, ambas as threads ficarão esperando uma pela outra indefinidamente. Esta situação é chamada de *deadlock* [SILBERSCHATZ; GALVIN; GAGNE, 2008]. Formalmente, *deadlocks* podem ser definidos da seguinte maneira: 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 causar [TANENBAUM, 2009].

Para um *deadlock* ocorrer as seguintes quatro condições devem acontecer simultaneamente em um sistema:

1. Exclusão Mútua;
2. Posse e espera;
3. Não-preempção;
4. Espera circular.

[2] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html

Para a condição de exclusão mútua acontecer, pelo menos um recurso deve estar em um modo “não-compatilhável”, ou seja, apenas uma *thread* de cada vez pode obter o recurso. Se outro processo tenta adquirir o recurso, o processo solicitante deve esperar até que o recurso seja liberado. Já na condição de posse e espera um processo deve estar em posse de pelo menos um recurso e estar esperando para adquirir recursos adicionais, os quais estão em posse de outros processos [TANENBAUM, 2009].

A condição de não-preempção ocorre quando os recursos previamente adquiridos por uma *thread* não podem ser liberados à força [TANENBAUM, 2009]. Eles devem ser explicitamente liberados pela *thread* que os adquiriu. A condição de espera circular corresponde ao *deadlock* propriamente dito, para esta acontecer deve haver uma cadeia circular de dois ou mais processos, onde cada um dos quais está à espera de um recurso adquirido pelo próximo elemento da cadeia.

Até o momento, apenas falamos sobre *deadlocks* ocasionados pela disputa por recursos, os quais são chamados de *deadlock* de recursos [ROMERO, 2009]. Nesses *deadlocks*, dois ou mais processos querem algo que algum outro desses processos tem e devem esperar até esses recursos sejam liberados, o que nunca acontece. Às vezes, os recursos são de hardware ou objetos de software, tais como unidades de CD-ROM ou registros de banco de dados, mas às vezes eles são mais abstratos, como é o caso dos *mutexes.*

O *deadlock* de recursos não é o único tipo deadlock. Outro tipo de *deadlock* pode ocorrer em sistemas de comunicação (por exemplo, redes), em que dois ou mais processos se comunicam através do envio de mensagens. Este tipo de *deadlock* se chama *deadlock* de comunicação[ROMERO, 2009]. Apesar de não ser o foco deste trabalho, abaixo temos uma situação que ilustra este tipo de travamento.

"O processo A envia uma mensagem de solicitação para o processo B, e, em seguida, fica bloqueado até B enviar de volta uma mensagem de resposta. Suponha que a mensagem se perde. O processo A está bloqueado aguardando a resposta. Enquanto o processo B está bloqueado à espera de um pedido solicitando-o para fazer alguma coisa. Temos novamente um *deadlock”*

Quando se depara com um *deadlock,* o desenvolvedor normalmente tenta depurar o programa para entender o que está acontecendo. 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 *d*e 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 [WILLIAMS; THIES; ERNST, 2005][Naik et al., 2009], detecção dinâmica [LI et al., 2005][JOSHI et al., 2009] e verificação de modelos [HAVELUND; PRESSBURGER., 2000]. 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. Uma outra forma de detecção é a de *deadlock* em forma de exceções e é nela que este trabalho irá focar e propor melhorias. Na seção a seguir iremos dar mais detalhes sobre esta forma de detecção.

2.5 Deadlocks em forma de exceções

Deadlocks são comumente tratados como erros de programação nas linguagens e sistemas de banco de dados atuais, contudo existem algumas situações nas quais os *deadlocks* são tratados como exceções. Exceções são úteis para controlar erros e tomar decisões baseadas nos mesmos, criar novos tipos de erros para melhorar o tratamento deles em sua aplicação e assegurar que um método funcionou como deveria.

Alguns sistemas de gerenciamento de banco de dados (SGBD) como o SQL Server[3] e o PostgreSQL[4] são capazes de detectar *deadlocks.* No caso do SQL Server, o gerenciador apresenta um sistema de monitoramento o qual tem um detector de *deadlocks* que verifica periodicamente os *locks* para ver se existe alguma espera circular entre *os* mesmos. Se encontrar alguma, ele seleciona uma das sessões associadas a uma thread suspensa, finaliza-a, desfaz as transações efetuadas e libera seus *locks.* Isso permite que a outra sessão continue a ser executada. Contudo o sistema não deixa de informar o usuário acerca do problema ocorrido e retorna uma mensagem de erro 1205 para o aplicação conforme abaixo:

```
Your transaction (process ID #52) was deadlocked on {lock | communication buffer | thread} resources with another process and has been chosen as the deadlock victim. Rerun your transaction.
```

[3] https://www.microsoft.com/en-us/cloud-platform/sql-server
[4] https://www.postgresql.org/

Vale salientar que se a linguagem faz requisições a um banco de dados sql server, como por exemplo c#, esta linguagem também dá suporte a essa exceção. Ou seja, se você faz uma conexão com o banco e este entra em *deadlock*, sua aplicação em C# também terá uma exceção de *deadlock.* No caso de C#, a aplicação lança uma SqlException[5] e dentro dela tem um atributo "number" que seria 1205, o qual indica que foi um *deadlock.* Detecção de deadlocks é fundamental em sistemas gerenciadores de bancos de dados (SGBD) porque esses sistemas fazem uso muito intenso de travas [LU et al., 2008]. Além disso, como todas as atividades realizadas por usuários no contexto de um SGBD são transações, cancelar uma dessas atividades não traz consequências mais graves. É possível inclusive executar novamente a transação cancelada logo em seguida. Por fim, SGBD gastam grande parte do seu tempo de execução em operações de sincronização e de entrada e saída. Consequentemente, o custo de monitoramento é baixo quando comparado ao que seria necessário para monitorar *deadlocks* em tempo de execução em uma linguagem de programação.

Outras linguagens de programação lançam exceções quando se deparam como um *deadlock* no seu próprio domínio, ou seja, o problema aconteceu na própria linguagem, diferentemente do caso citado acima. Haskell é um bom exemplo de linguagem, a exceção é lançada quando o garbage collector detecta uma thread como inalcançável e nenhuma outra thread pode acordá-la. A linguagem de programação Go também lança exceção, contudo esta só detecta o problema quando todo o programa entra em *deadlock*, e não quando um subconjunto de goroutines fica travado.

É importante salientar que a Java Virtual Machine (JVM), já provê a detecção de *deadlock* em tempo de execução desde o Java Development Kit (JDK) 1.4. Um *deadlock* pode ser detectado sempre que um thread stack dump é solicitado, ou seja, é necessário um esforço manual. O thread stack dump pode ser obtido através de ferramentas como JStack[6] e JConsole[7].

O jstack é capaz de imprimir as *stacktraces* (thread dump) de todas as *threads* que estão ligadas à máquina virtual, incluindo *threads* Java e *threads* internas da VM, e opcionalmente *stack frames* nativos. Thread dumps também podem ser obtidos por meio de programação usando o método Thread.getAllStackTraces(), ou no depurador usando a opção para imprimir as *stacktrace* de todas as *threads.* Segue abaixo um exemplo de saída do jstack:

[5] https://msdn.microsoft.com/pt-br/library/system.data.sqlclient.sqlexception
[6] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstack.html
[7] http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html

```
$ jstack -F 8321
Attaching to process ID 8321, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Deadlock Detection:

Found one Java-level deadlock:
=============================

"Thread2":
  waiting to lock Monitor@0x000af398 (Object@0xf819aa10, a
java/lang/String),
  which is held by "Thread1"
"Thread1":
  waiting to lock Monitor@0x000af400 (Object@0xf819aa48, a
java/lang/String),
  which is held by "Thread2"

Found a total of 1 deadlock.

Thread t@2: (state = BLOCKED)

Thread t@11: (state = BLOCKED)
 - Deadlock$DeadlockMakerThread.run() @bci=108, line=32 (Interpreted
frame)

Thread t@10: (state = BLOCKED)
 - Deadlock$DeadlockMakerThread.run() @bci=108, line=32 (Interpreted
frame)

Thread t@6: (state = BLOCKED)

Thread t@5: (state = BLOCKED)
 - java.lang.Object.wait(long) @bci=-1107318896 (Interpreted frame)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.ref.ReferenceQueue.remove(long) @bci=44, line=116
(Interpreted frame)
 - java.lang.ref.ReferenceQueue.remove() @bci=2, line=132 (Interpreted
frame)
```

```
 - java.lang.ref.Finalizer$FinalizerThread.run() @bci=3, line=159
(Interpreted frame)

Thread t@4: (state = BLOCKED)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.Object.wait() @bci=2, line=485 (Interpreted frame)
 - java.lang.ref.Reference$ReferenceHandler.run() @bci=46, line=116
(Interpreted frame)
```

Para utilizar a abordagem do jstack é necessário que haja um esforço manual por parte de desenvolvedor. O usuário nota que há um problema e depois executa o jstack para tentar entender o que aconteceu. Este trabalho acredita que é importante informar aos usuários os *deadlocks* que não são esperados, neste caso, lançando exceções em tempo de execução. Além disso, o resultado do jstack não informa a origem das travas, onde elas foram criadas. Na sua mensagem é incluído o monitor que a *thread* está esperando para adquirir e o id da trava. O que não estabelece um mapeamento para entre esse id e o elemento estático que o representa.

Apesar de fornecer informações importantes na detecção de *deadlock*, o modelo de detecção que iremos ampliar (o qual será detalhado mais adiante) possui vantagens que o modelo da JVM não oferece e por isso ele pode ser amplamente adotado.

1. Não necessita de esforço manual.
2. Permite que o usuário implemente soluções ou comportamentos, em caso de detecção de *deadlock.*
3. Indica claramente ao usuário onde os *locks* foram adquiridos pelas *threads.*
4. Indica onde as *threads* e *locks* foram criados.

3. Trabalhos Relacionados

Com o intuito de entender o problema da detecção de *deadlocks* nas linguagens de programação atuais, foram analisadas as abordagens já presentes na literatura para detecção de *deadlocks.* Os trabalhos relacionados foram divididos em quatro categorias: (i) trabalhos sobre detecção de *deadlocks* (ii) detecção de erros em tempo de execução sobre problemas de concorrência (iii) trabalhos base para o desenvolvimento desta pesquisa (iv) trabalhos relacionados à mensagem de erro.

Esta divisão foi feita para primeiro apresentar o estado da arte sobre detecção de *deadlocks* em geral, em seguida focar na detecção em tempo de execução e por fim apresentar os trabalhos que são a base para o trabalho atual e introduzir o protocolo escolhido a ser ampliado.

3.1 Detecção de ***deadlocks***

Nesta seção são apresentadas outras abordagens para detecção de *deadlocks.* A escolhida para este trabalho foi a detecção em tempo de execução e lançando exceções para o usuário. Existem outras formas utilizadas na literatura, abaixo temos duas delas: detecção estática e detecção dinâmica. Vale salientar quem existem abordagens que unem as duas formas [GRECHANIK et al., 2013][PYLA; VARADARAJAN, 2012] para aproveitar os pontos positivos de cada e suprimir os possíveis problemas

3.1.1 Detecção estática

As técnicas de análise estática verificam o código-fonte ou alguma representação intermediária de um programa e tentar identificar potenciais problemas sem sequer executar o programa. [WILLIAMS; THIES; ERNST, 2005] é um bom exemplo de método para detecção estática de *deadlocks* em bibliotecas Java. O método determina se é possível ocorrer um *deadlock* ao chamar um conjunto de métodos públicos. Se o *deadlock* for possível, ele fornece os nomes dos métodos e variáveis envolvidas.

Contudo, este método detecta *deadlocks* em todas as chamadas possíveis para uma biblioteca, o que é diferente de detecção de *deadlock* em um programa inteiro. Em um programa, muitas vezes o número de threads pode ser determinado, mas um cliente pode fazer chamadas a uma biblioteca a partir de qualquer número de *threads.* O detector de *deadlock*

faz uma análise de fluxo de dados para a construção de grafos representando a ordem em que os travamentos ocorrem nos *mutexes.* A análise é sensível ao fluxo e sensível ao contexto.

Existem diversos outros trabalhos [NAIK et al., 2009][BREUER; VALLS, 2004] na área entretanto apesar da detecção estática de *deadlock* ter feito progressos impressionantes nos últimos anos, os casos falso-positivos podem ser numerosos e o custo da reparação manual dos erros genuíno permanece alto. [WANG, 2009]

3.1.2 Detecção dinâmica

Algoritmos de detecção de *deadlocks* tradicionais buscam encontrar ciclos em grafos que representam os *locks* criados a partir do código da aplicação. Normalmente, estas abordagens sofrem de problemas de escalabilidade e desempenho; além de não poderem lidar com grandes aplicações de força industrial. O principal problema para a falta de escalabilidade e desempenho precário é causado pelo tamanho dos grafos das travas que têm de ser analisados.

Para exemplificar ferramentas de detecção dinâmicas escolhemos Multicore SDK [DA LUO; DAS; QI, 2011]. Este consiste de um algoritmo de detecção de *deadlocks* de duas fases, o qual almeja ser eficiente em termos de utilização de memória e tempo, mas também muito escalável.

Na primeira fase do algoritmo é criado um grafo de *locks* reduzido com base em locais do programa. Nessa fase são filtradas as travas que não podem entrar em *deadlock.* Na segunda fase do algoritmo é criado um grafo ainda menor, considerando apenas os *locks* que não foram filtrados na primeira fase. Por fim, o grafo gerado na segunda fase é analisado para encontrar possíveis *deadlocks* na aplicação. A detecção dinâmica pode identificar o problema tarde demais, quando a recuperação é complexa ou impossível; reversão automatizada e reexecução podem ajudar. Mas as ações irrevogáveis, como I/O podem impedir a reversão [WANG, 2009]. Outra desvantagem é que técnicas de detecção dinâmica são tão boas contra o conjunto de casos de teste empregado. Se os casos de teste não cobrem todas as situações que podem ocorrer em tempo de execução, não conseguirão detectar o *deadlock*.

3.2 Detecção de problemas de concorrência em tempo de execução

Nesta seção serão apresentados alguns problemas de concorrência que são detectados em linguagens de computação, além de exemplos de trabalhos e ferramentas que os detectam em tempo de execução.

3.2.1 Problemas de corrida

Uma condição de corrida ocorre em um programa *multi-threaded* quando duas *threads* acessam à mesma localização de memória sem restrições de ordenação impostas entre os acessos, de modo que pelo menos um dos acessos é uma escrita. Na maioria dos casos, uma condição de corrida é um erro de programação [TANENBAUM, 2009].

Além disso, os programas que contêm condição de corrida são notoriamente difíceis de depurar, dado que eles podem apresentar diferentes comportamentos funcionais, mesmo quando executados repetidamente com o mesmo conjunto de entradas e na mesma ordem de execução das operações de sincronização. Por causa dos efeitos prejudiciais da condição de corrida na confiabilidade e na compreensibilidade de softwares *multi-threaded,* é amplamente reconhecido que as ferramentas para detecção automática de condições de corrida podem ser extremamente valiosas.

Em [CHOI et al., 2002] é apresentada uma abordagem para detecção dinâmica de condição de corrida para programas orientados a objeto com várias *threads.* Esta abordagem checa informa que há condição de corrida se pelo menos um dos acessos é de escrita. O algoritmo retorna pares de eventos que apresentam condições de corrida. Cada evento é representado como uma 5-tupla. A qual é formada por: uma memória que está sendo acessada, uma *thread,* um conjunto de *locks* obtidos no momento do acesso à memória, o tipo de acesso (escrita ou leitura) e o local de origem da instrução de acesso,o qual não é utilizado na detecção, apenas no resultado exibido ao usuário.

O algoritmo em pior caso roda em O (n^2), pois precisa analisar cada par de eventos possíveis. Para identificar se um par de eventos está em condição de corrida, é checado se:

- A posição de memória acessada pelos dois eventos é a mesma
- As *threads* de cada evento são diferentes
- A interseção dos conjuntos de *locks* obtidos nos dois eventos é vazia
- Pelo menos um dos eventos apresenta um acesso do tipo escrita

Esta abordagem é eficiente e precisa. A solução apresentada apresenta um overhead de tempo de execução que varia de 13% a 42%, o que está bem abaixo do tempo de execução de outras abordagens com precisão comparável.

Este desempenho é obtido através de uma combinação de técnicas de otimização estática e dinâmica que se complementam na redução do overhead do detector. Além disso, quase todos os casos relatados pelo sistema correspondem a erros reais, e a saída precisa da

ferramenta permite encontrar facilmente e compreender as linhas de código fonte problemáticas nos programas testados.

3.2.2 Problemas de atomicidade

Atomicidade é uma condição importante para a corretude de sistemas concorrentes. Informalmente, atomicidade é a propriedade que diz que cada execução simultânea de um conjunto de transações é equivalente a alguma execução serial das mesmas operações. Em programas *multi-threaded,* execuções de procedimentos (ou métodos) podem ser consideradas transações. Exatidão na presença de concorrência, muitas vezes requer atomicidade dessas operações. Ferramentas que detectam automaticamente violações na atomicidade podem descobrir erros sutis que são difíceis de encontrar com técnicas tradicionais de depuração e teste.

[WANG; STOLLER, 2006] apresenta um algoritmo para detecção em tempo de execução (dinâmica) das violações de atomicidade. Este algoritmo pode ser utilizado em programas Java. Quando o programa termina, o usuário pode obter uma informação gravada sobre a execução. A execução deste algoritmo é dividida em unidades. Uma unidade é uma sequência de eventos executados por uma única *thread.* Uma operação é uma unidade na qual se espera que esta se comporte atomicamente.

Por exemplo, a sequência de eventos executados durante uma chamada de método é muitas vezes considerada como uma operação. Os algoritmos verificam se as funções chamadas nestas unidades são equivalentes a uma chamada serial de todas as unidades, ou seja, se todos os eventos em cada operação destas unidades são consecutivos. Se assim for, pode-se dizer que as operações são atômicas. Caso contrário uma potencial violação de atomicidade é relatada.

As experiências relatadas no artigo mostram que estes algoritmos são mais eficientes na maioria dos experimentos e são mais precisos do que os algoritmos anteriores com complexidade assintótica comparáveis.

3.2.3 Corrida e atomicidade

jPredictor [CHEN; SERBANUTA; ROSU, 2008] é uma ferramenta para a detecção de erros de concorrência em programas Java. O conjunto de funções chamadas na execução de um programa é analisado pelo jPredictor. O modelo abstrato resultante é então exaustivamente analisado e os erros são relatados ao usuário. Assim, o jPredictor pode

"prever" erros que não aconteceram na execução observada, mas que poderiam ter acontecido se as *threads* fossem executadas em ordem diferente.

A técnica de análise empregada pelo jPredictor é totalmente automática, genérica (funcionam com qualquer *stacktrace*), segura (não produz falsos-positivos), mas é incompleta (pode perder erros). Dois tipos comuns de erros foram investigados em [CHEN; SERBANUTA; ROSU, 2008]: condições de corrida e violações de atomicidade. As experiências mostraram que o jPredictor é preciso nas suas previsões, eficaz e eficiente.

3.2.4 Problemas de *deadlock*

Programas concorrentes são notórios por conterem erros que são difíceis de reproduzir e diagnosticar. Um tipo comum de erro de simultaneidade é o *deadlock*, o que ocorre quando algumas threads são permanentemente bloqueadas.

Os trabalhos sobre a detecção de tempo de execução de potenciais *deadlocks* têm se concentrado em programas que usam travas. O algoritmo GoodLock [HAVELUND, 2000] proposto por Havelund detecta potenciais *deadlocks* envolvendo duas threads.

Agarwal, Rahul, e Stoller criaram uma extensão [AGARWAL; WANG; STOLLER, 2005] do GoodLock que é *multithreaded* e detecta potenciais *deadlocks* em programas que utilizam travamento em blocos estruturados, como os blocos *synchronized* em Java. Posteriormente [AGARWAL; STOLLER, 2006] desenvolveram um algoritmo para lidar com travamento utitilizando blocos não estruturados.

Travamento em blocos estruturados (ou seja, os *locks* são liberados na ordem oposta em que foram adquiridos, ou seja, o *lock* mais recentemente adquirido é o próximo a ser liberado) são construídos em Java. A biblioteca Java 5 (java.util.concurrent) e a biblioteca POSIX Threads para C fornecem travas que não são necessariamente estruturadas em blocos.

O algoritmo em [AGARWAL; STOLLER, 2006] constrói uma árvore de *locks* em tempo de execução para cada *thread,* como no algoritmo GoodLock [HAVELUND, 2000]. Cada árvore de travas representa o padrão aninhado em que os bloqueios são adquiridos e liberados pela *thread.* Cada nó da árvore de *locks* é rotulado com uma trava e a *thread* que adquiriu essa trava.

Ao final da execução, ele constrói um grafo de *locks,* que é um grafo direcionado G = (V, E), onde V contém todos os nós de todas as árvores de *locks,* e o conjunto E de arestas direcionadas. Para um grafo G, um caminho válido é um caminho que não contém arestas inter-consecutivas, tal que os nós de cada árvore de *locks* aparece como no máximo uma subsequência consecutiva no caminho. Do mesmo modo, um ciclo é um ciclo válido se não

contém interarestas consecutivas e nós a partir de cada *thread* aparecem como no máximo uma subsequência consecutiva no ciclo.

Apesar das melhorias, este algoritmo pode produzir falso-positivos a partir dos ciclos que contêm aquisições e liberações que não podem acontecer em paralelo.

3.3 Trabalhos base

O princípio deste trabalho é adicionar informações contextuais a exceções de *deadlock.* Desta forma será ampliado o protocolo apresentado em [LOBO; CASTOR, 2015]. Nesta seção será introduzido o protocolo citado e por fim serão discutidas algumas conclusões, as quais determinam o motivo de ter-se optado por fazer as ampliações neste trabalho.

Em resumo, o trabalho em [LOBO; CASTOR, 2015] defende que *deadlocks* não devem falhar silenciosamente, mas sim serem sinalizados como exceções em tempo de execução. Para tornar isto possível, os autores se basearam em duas perspectivas: (1) a grande maioria dos *deadlocks* existentes ocorre entre duas threads que tentam adquirir dois *locks* (como relatado por outros autores [LU et al., 2008] e confirmada pelo próprios autores do trabalho); (2) é possível introduzir de forma eficiente a detecção de *deadlocks* para este tipo de *deadlock* de duas *threads,* e dois *locks* (TTTL) dentro do próprio mecanismo do *lock,* apresentando um overhead que é baixo para aplicações cujo tempo de execução não é dominado por operações de travamento.

O protocolo criado apresenta um novo tipo de *lock* que verifica automaticamente se há *deadlocks* TTTL em tempo de execução; ou seja, se um *deadlock* for detectado, lança uma exceção indicando o problema. A implementação desta abordagem foi feita como uma extensão para a classe ReentrantLock de Java, parte do pacote java.util.concurrent desde o Java Development Kit (JDK) 1.5.

Antes de implementar o novo protocolo, os autores buscaram confirmar a perspectiva (1): a grande maioria dos *deadlocks* existentes ocorrem entre duas *threads* que tentam adquirir dois *locks.* Para tal foram analisados os repositórios de três projetos de código aberto que usam Java como sua principal linguagem de programação e fazem uso de programação concorrente: Lucene[8], Eclipse[9] e OpenJDK[10].

[8] https://lucene.apache.org/
[9] https://eclipse.org/
[10] http://openjdk.java.net/

Em cada repositório foi feita uma busca por problemas de *deadlocks*, coletando ao todo 541 problemas reportados ao total. O tamanho da amostra, a qual permitiria ter 95% de confiança e erro de amostragem de 5%, foi de 225 erros. A tabela abaixo representa os resultados encontrados:

Categoria	Número de erros	Erros estimados
Deadlock de recursos	101	146
Deadlock de recursos + TTTL	93	134
Deadlock de comunicação	32	46
Deadlock falso-positivo	23	33
Não é possível informar	69	0

Tabela 1: Resultado do estudo e sua estimativa.

No pior dos casos, 54,7% dos *deadlocks* de recursos são *deadlocks* TTTL, enquanto que no melhor dos casos, 95,29% são *deadlocks* TTTL. Na tabela acima (segunda coluna), podemos ver que, entre todos os *deadlocks* de recursos que foram identificados, 92,07% são de fato *deadlocks* TTTL. Contudo, nem o pior caso nem os melhores casos parecem ser realista. Acredita-se que um cenário mais realista seria assumir que os erros onde não foi possível informar a categoria são distribuídos aproximadamente da mesma forma que os das outras categorias. Se for esse o caso (terceira coluna da tabela acima), estima-se que 91,7% dos *deadlocks* de recursos também são *deadlocks* TTTL [LOBO; CASTOR, 2015].

Quando o *deadlock* é detectado, duas exceções são lançadas, uma para cada *thread* envolvida no *deadlock.* Em cada exceção é exibida a *stacktrace* de cada *thread.* A exceção é lançada no momento que as threads estão tentando adquirir o seu segundo *lock*, logo a *stacktrace* permite o acompanhamento da sequência de funções aninhadas chamadas até o momento de aquisição da segunda trava. Abaixo temos um exemplo das informações mostradas ao usuário:

```
Exception in thread "Thread-0" Exception in thread "Thread-2" locks.DeadlockException
        at locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:821)
        at locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:858)
        at locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1187)
        at locks.LockA$NonfairSync.lock(LockA.java:180)
        at locks.LockA.lock(LockA.java:256)
        at main.SWT.createSurface(SWT.java:20)
        at main.UIImage.<init>(UIImage.java:9)
        at main.BusyIndicator.updateAnimation(BusyIndicator.java:36)
        at main.BusyIndicator.run(BusyIndicator.java:47)
locks.DeadlockException
        at locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:821)
        at locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:858)
        at locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1187)
        at locks.LockA$NonfairSync.lock(LockA.java:180)
        at locks.LockA.lock(LockA.java:256)
        at main.BusyIndicator.getImage(BusyIndicator.java:17)
        at main.BusyIndicator.setBusy(BusyIndicator.java:31)
        at main.BusyIndicatorImage.paint(BusyIndicator.java:59)
        at main.UIWindow.paint(UIWindow.java:30)
        at main.UIThread.run(UIThread.java:17)
```

Figura 2: Exemplo de saída mostrada ao usuário pelo ReentrantLock modificado

Para avaliar o trabalho desenvolvido, 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 (*deadlock)* de forma precisa rapidamente, mesmo quando não utilizaram a abordagem proposta. Programas assim não são comuns no mundo real.

3.4 Mensagens de erro

As mensagens de erro são uma das ferramentas mais importantes que uma linguagem de programação oferece aos seus programadores. Para os novatos, este *feedback* é especialmente importante. Mensagens de erro normalmente contêm uma descrição textual do problema e uma indicação de onde o erro ocorreu no código [MARCEAU; FISLER; KRISHNAMURTHI, 2011].

[MARCEAU; FISLER; KRISHNAMURTHI, 2011] realizaram uma série de estudos que exploram interações dos alunos novatos com o vocabulário e os destaques da origem do problema, no ambiente de desenvolvimento DrRacket[11]. Dentre algumas conclusões, eles indicam que a mensagem de erro deve ajudar os alunos a relacionar termos da mensagem a partes do código.

Em [NIENALTOWSKI; PEDRONI; MEYER, 2008] foram realizaram testes para analisar o efeito de diferentes estilos de mensagens de erro (mensagem curta, representação visual e mensagem longa) em quão bem e quão rápido os alunos identificariam os erros em

[11] https://docs.racket-lang.org/drracket/

programas. Em cada questão que recebiam os estudantes deviam dizer o tipo de erro encontrado. Contudo os autores notaram que não podiam concluir qual tipo era melhor em questões de número de acertos. Porém vale salientar que eles puderam concluir que a uma mensagem de erro do tipo representação visual implica em respostas mais rápidas.

Outras pesquisas [BROWN, 1983][SHNEIDERMAN, 1982][TRAVER, 2010] na área relatam diversos padrões que as mensagens de erro deveriam ter, tais como: clareza, positivas, específicas, exibir linhas do código que originaram o problema, detalhes visuais, entre outros.

4. Abordagem

Neste capítulo é detalhada a ampliação feita no protocolo apresentado em [LOBO; CASTOR, 2015]. Nós expandimos a implementação feita no trabalho citado anteriormente para melhorar a rastreabilidade dos problemas. Com as novas informações contextuais adicionadas o desenvolvedor poderá saber explicitamente quais *locks* e *threads* ocasionaram o travamento e em que momento estes foram adquiridos, além dos locais onde as *threads* e os *locks* envolvidos no *deadlock* foram criados.

Primeiramente discutimos como decidimos quais novas informações seriam adicionadas a exceções de *deadlock* e em seguida como estas informações são coletadas em tempo de execução. Por fim, uma explicação de como as novas informações são exibidas aos usuários e a avaliação do *overhead.*

4.1 Informações contextuais adicionadas

Deadlocks são bem conhecidos por serem difíceis de depurar, especialmente quando se está lidando com uma grande quantidade de código concorrente. Provavelmente em sua vida todo desenvolvedor vai acabar lidando com *deadlocks* algumas vezes. Enquanto uma situação de *deadlock* não é tão difícil de se detectar, às vezes pode ser difícil detectar onde o dead*l*ock aconteceu. Isso geralmente leva a perguntas como "onde? E como faço para rastreá-lo?", se for um *deadlock* de recursos teremos perguntas como: “Quais recursos e threads participam do *deadlock*?”.

Como dito no capítulo anterior, as informações apresentadas no protocolo nem sempre são suficientes para detectar o *deadlock* corretamente, mesmo em programas simples. Isso decorre em parte de *deadlocks* serem fenômenos que só ocorrem em tempo de execução enquanto desenvolvedores trabalham com programas em tempo de desenvolvimento. Embora um programa possa definir apenas uma classe que represente *threads*, milhares de instâncias dessa classe podem ser criadas em tempo de execução e são estas que se envolvem no *deadlock*. Desenvolvedores não têm como lidar com esse grau de complexidade. Além disso, não-determinismo é inerente à concorrência, o que significa que cada entrelaçamento de instruções em execução em *threads* diferentes é potencialmente tão complexo quanto seria a execução do sistema como um todo se este fosse estritamente sequencial.

Para verificar que novas informações deveríamos prover ao desenvolvedor, foram analisadas quais informações já são mostradas aos usuários e quais informações são necessárias para entender como *deadlock* acontece.

A investigação se dividiu em duas partes: 1) Analisar os experimentos realizados pelos autores e ver quais informações estavam incompletas nas respostas dos alunos. 2) Buscar erros de *deadlocks* TTTL encontrados em repositórios de código aberto e analisar os comentários de cada erro, a fim de entender que informações são providas ao desenvolvedores para facilitar o entendimento do problema.

4.1.1 Análise dos resultados dos experimentos

Os autores realizaram dois experimentos, um com uma turma da graduação e o outro com alunos da pós-graduação. Em cada experimento os alunos resolveram dois problemas, um deles utilizando o protocolo e o outro sem. Aproximadamente metade da turma (de cada experimento) resolveu o primeiro programa (tipo A) com o protocolo e a outra metade resolveu o segundo problema (tipo B) com o protocolo. Foi pedido para cada aluno que ele descrevesse o problema encontrado no programa e as chamadas de código envolvidas, incluindo a explicação de como o problema acontecia. Resumidamente os resultados foram os seguintes:

I. Experimento com os alunos da graduação:
    A. 16 alunos fizeram o experimento do tipo A, dentre esses 16:
        1. Doze alunos identificaram todas as travas e métodos envolvidos, porém 2 deles apontaram também métodos que não estavam relacionados ao *deadlock.*
        2. Um aluno acertou as 2 travas envolvidas, mas não os métodos
        3. Um aluno acertou, 1 trava e os métodos envolvidos de uma das *threads.*
        4. Dois alunos acertaram, 1 das travas *e*nvolvidas, mas não os métodos. (1 deles apontou métodos não relacionados com o *deadlock)*

    B. 15 alunos fizeram o experimento do tipo B, dentre esses 15:
        1. Seis identificaram todas as travas e métodos envolvidos, porém 1 deles apontou também métodos que não estavam relacionados com o *deadlock).*

2. Um aluno acertou as 2 travas envolvidas, mas não os métodos.
3. Um aluno acertou as 2 travas envolvidas, e os métodos envolvidos de uma das *threads.*
4. Quatro alunos acertaram 1 das travas, e os métodos envolvidos de uma das *threads.*
5. Um aluno acertou só 1 das travas, mas nenhum dos métodos envolvidos (apontou métodos não relacionados)
6. Dois alunos não acertaram nada das travas, nem dos métodos.

II. Experimento com os alunos da Pós-graduação:

A. 8 alunos fizeram o experimento do tipo A, dentre esses 8:

1. Quatro alunos identificaram todas as travas e métodos envolvidos
2. Um aluno acertou as 2 travas envolvidas, mas não os métodos.
3. Um aluno acertou, 1 trava e os métodos envolvidos de uma das *threads.*
4. Um aluno acertou só 1 das travas, mas nenhum dos métodos envolvidos.
5. Um aluno acertou 1 das travas envolvidas, mas também todos os métodos envolvidos em ambas as *threads*.

B. 8 alunos fizeram o experimento do tipo B, dentre esses 8:

1. Cinco identificaram todas as travas e métodos envolvidos
2. Um acertou 1 das travas envolvidas, entretanto todos os métodos envolvidos.
3. Dois não acertaram nada das travas, nem dos métodos.

Como pode ser visto na Figura 2, os *stacktraces* que são produzidos não indicam os métodos responsáveis pelo travamento, nem os *locks* envolvidos no travamento. Através desses resultados podemos concluir que: saber quais foram as travas envolvidas e os métodos que as adquiriram/tentaram adquirir nem sempre é uma informação óbvia. Da forma como o protocolo se encontra, o usuário precisa percorrer os métodos citados na *stacktrace* e assim procurar os métodos e *locks* envolvidos diretamente no travamento. Vale salientar que a *stacktrace* contém os métodos relacionados à aquisição do segundo *lock,* o que não nos

garante que o usuário saberá identificar corretamente onde foi feita a aquisição da primeira trava.

4.1.2 Análise dos relatórios de erros

Nesta etapa, nosso objetivo era buscar erros de *deadlocks* TTTL encontrados em repositórios de código aberto e analisar os comentários de cada erro, a fim de entender que informações são providas aos desenvolvedores para facilitar o entendimento do problema. Dando continuidade ao trabalho feito anteriormente, decidimos analisar os 93 *deadlocks* do tipo TTTL classificados por [LOBO; CASTOR, 2015].

O intuito desta análise foi descobrir que informações são fornecidas aos desenvolvedores que estão tentando entender/resolver o travamento. Acreditamos que só a *stacktrace* de cada *thread* não é suficiente.

Primeiramente dentre esses 93 erros informados, quatro foram descartados, dois só tinham o *patch* e os outros dois eram só entre blocos *syncronized.* Dos 89 restantes, 68 deles continham informações a respeito das threads e *locks* envolvidos. Fora esses 68, outros 4 tinham as *threds* envolvidas, mas não tinham os *locks.* Isso dá um total de 76.40% contendo as duas informações. E 80,89% contendo as *threads* envolvidas.

Notamos também que 73 erros continham *stacktraces*, porém 57 deles (78%) também continham informações adicionais acerca do problema como: *lock* envolvidos, *threads* envolvidas e descrição detalhada de como ocorreu o *deadlock.* Fica evidente na nossa amostra de problemas no mundo real, que informar apenas as *stacktraces* das *threads* envolvidas não é suficiente.

Quanto às informações a serem adicionadas, como mencionado, 76.40% dos relatórios de erros analisados continham informações informando quais as *threads* e os *locks* envolvidos. Vale salientar que essa informação (*threads* e *locks)* era passada explicitamente na forma de linguagem falada como uma descrição de como o *deadlock* ocorre, ou apontando na *stacktrace* os métodos onde os travamentos eram feitos, ou apenas escrevendo um resumo como na imagem abaixo.

```
Thread 'main' calls PDOMManager.deleting() locking indexerJobMutex
Thread 'worker-0' calls PDOMIndexerQueue.fillQueue() locking taskMutex
Thread 'main' calls PDOMIndexerQueue.cancelJobs() waiting for taskMutex
Thread 'worker-0' calls PDOMManager.getNextTask() waiting for indexerJobMutex
```

Figura 3: *Threads*, métodos, *locks* envolvidos no *deadlock*

Conclui-se assim que é preciso mostrar quem são os *locks* e os métodos envolvidos no problema, salientando os momentos das tentativas de aquisições. Além de incluir uma representação visual para representar o fluxo de cada *thread,* como uma tentativa de remover a necessidade de uma descrição de como ocorre o problema.

4.1.3 Outras análises

Nesta abordagem, estão sendo usadas informações estáticas (código) para falar sobre um fenômeno que é dinâmico (*deadlock)*. Os elementos que fazem sentido em tempo de execução (*thread* e *object ids)* não têm um mapeamento para esses elementos estáticos. Por isso, é importante fornecer mais informações que são simples de analisar e podem ajudar o desenvolvedor a identificar quais são os elementos dinâmicos envolvidos no *deadlock* a partir desses elementos estáticos. Em outras palavras, esse mapeamento será feito através dos locais do código onde *threads* e *locks* são criados.

Vale salientar que *threads* e *l*ocks podem ser criados dinamicamente, por necessidade, desta forma se faz mais necessário ainda informar os locais de criação das travas e *threads.*

4.2 Coleta de informações

Nesta seção iremos informar como salvamos as novas informações que serão apresentadas ao usuário. Detalhes de como exibimos estes dados aos usuários serão detalhados na próxima seção. As informações a serem coletadas são: momento de criação dos *locks* e os métodos envolvidos no *deadlock.*

4.2.1 Visão geral do funcionamento do ReentrantLock modificado

A classe ReentrantLock tem um atributo Sync, o qual é a base de sincronização da trava e provê os mecanismos de implementação. O Sync pode ser FairSync ou NonfairSync, ambos estendem de Sync. A classe Sync estende da classe AbstractQueuedSynchronizer, que é onde as exceções são lançadas.

O construtor pode ser chamado de dois jeitos, o primeiro o qual não recebe nenhum parâmetro e o segundo recebendo um booleano. O construtor vazio cria uma instância padrão do ReentrantLock (NonfairSync), já o segundo construtor cria uma instância do ReentrantLock com uma política justiça; em outras palavras se essa trava deve usar uma política de ordenamento justa (FairSync) ou não (NonfairSync).

Para melhorar o entendimento da abordagem feita neste trabalho, além das informações citadas anteriormente, é preciso ter em mente esses dois fatos:

1. Quando o *deadlock* acontece, cada *thread* já adquiriu um *lock* e está tentando adquirir o segundo.
2. Apenas uma *thread* pode ter posse da trava por vez.
3. Quando uma *thread* chama o método lock() ou tryLock(), ela pode adquirir o *lock* em três métodos diferentes. Isso acontece porque em alguns casos a *thread* obtém a trava assim que chama o método lock(), e em outros casos ela precisa esperar a trava ser liberada.

Vale salientar que em cada um dos métodos onde as *threads* adquirem a trava, a quantidade de funções aninhadas chamadas entre o momento do lock() e eles mesmo são diferentes. A quantidade de chamadas varia com o tipo do Sync.

4.2.2 Criação de locks

Primeiramente precisamos reforçar que o momento (classe, método e linha) em que o *lock* foi criado precisa estar disponível em todas em *threads*, independentemente da *thread* que criou a trava. Caso contrário, não teremos acesso a essa informação quando a mensagem da exceção de *deadlock* for ser lançada.

Quando o *lock* vai ser inicializado seu construtor é chamado. Assim que o construtor é chamado, a trava é criada, obtemos a *stacktrace* atual e salvamos a informação desejada. Abaixo temos o pseudo-código que representa esse funcionamento:

```
Sync sync;
ReentrantLock(boolean fair) {
     sync = fair ? new FairSync() : new NonfairSync();
     Thread current = Thread.currentThread();
     sync.setIniatilizationStackElement(current.getStackTrace()[2]);
}
```

Figura 4: Código que representa o armazenamento do momento onde a trava foi criada.

O método da classe Thread do pacote java.lang, currentThread() retorna a *thread* atual. *C*ada *thread* tem a sua pilha de chamadas (*stacktrace)*, ao chamarmos o método getStackTrace() para a *thread* atual obtemos um *array* de StackTraceElement[12], o qual

[12] https://docs.oracle.com/javase/7/docs/api/java/lang/StackTraceElement.html

também pertence ao pacote java.lang. Desta forma, pode-se obter o conteúdo da *stacktrace* atual em qualquer método do código. Cada item do *array c*orresponde a elemento da pilha.

O *array* retornado pelo método getStackTrace() retorna os elementos na ordem em que aparecem na pilha, ou seja, o primeiro elemento foi o último a ser chamado pela *thread.* Nós salvamos o terceiro elemento (índice 2) pois este contém a linha de código que inicializou a trava atual, ou seja, que chamou o construtor. O elemento do índice 0 corresponde a uma chamada dentro do método getStackTrace() e o elemento no índice 1 é a chamada ao método getStackTrace() no próprio construtor do *lock.*

O método setIniatilizationStackElement é apenas um método *set* como é recomendado nas boas práticas de Java. Vale salientar que não precisamos garantir nenhum acesso exclusivo a esta chamada, pois mesmo que duas *threads* chamem o construtor do ReentrantLock ao mesmo, serão criadas duas instâncias diferentes, ou seja, uma não interfere na outra. Também não teremos problemas no momento da leitura (ao lançar as exceções de *deadlock),* pois a *thread* que detectou o problema cria a sua mensagem de erro antes de avisar a outra que ela está em *deadlock* com a *thread* atual.

4.2.3 Métodos envolvidos no *deadlock*

Inicialmente nosso objetivo era informar apenas a linha de código que efetuou o travamento, ou seja, a que chamou o método lock() ou tryLock() na classe ReentrantLock. Essa informação seria mostrada para cada trava de cada *thread*, em outras palavras, em cada exceção teríamos uma mensagem explicando onde a *thread* adquiriu o primeiro e o segundo *lock.*

Dado que as exceções são lançadas no momento que as *threads* estão tentando adquirir a sua segunda trava, e podemos obter a *stacktrace* atual a qualquer momento no código, só seria preciso salvar/acessar onde o primeiro *lock* foi adquirido; uma vez que para a segunda trava poderemos saber essa informação através da *stacktrace* atual.

Inicialmente a ideia era salvar esse dado no mesmo local onde é salvo a *thread* a qual o *lock* pertence. Para guardar essa informação, faremos como na seção anterior, obtendo a *stacktrace* atual e salvando em uma variável o StackTraceElement que contém a informação desejada. Contudo a *thread* torna-se dona de uma trava em três métodos diferentes. Dependendo de onde o *lock* é adquirido, a *stacktrace* vai apresentar um tamanho diferente, consequentemente o índice do *array* de StackTraceElements que deveremos obter vai mudar.

A princípio isso não é problema, pois quando a chamada parasse em algum dos métodos, poderíamos saber o índice exato da *stacktrace* onde foi chamado o lock()

externamente. Porém essa quantidade de chamadas, consequentemente o índice, varia com o tipo do Sync, ou seja, não é possível definir um índice fixo a ser acessado para cada um dos 3 métodos. Para contornar esse problema tem-se as seguintes opções:

1. Checar em cada um dos três métodos onde a *thread* pode adquirir a trava, qual dos dois tipos de Sync está sendo utilizado e assim saberemos o valor do índice desejado.
    a. Pontos positivos: Continuaríamos acessando diretamente o índice correto no *array*
    b. Pontos negativos: Se um novo tipo de sincronizador for criado, seria preciso mudar a implementação da detecção de *deadlock.* Além disso, dependendo de onde esse código ficar, podemos ter uma superclasse que depende de uma de suas subclasses, algo que deve ser evitado sempre.
2. Percorrer o *array* da *stacktrace* inteiro e procurar pela ocorrência de "ReentrantLock.lock", essa é a primeira chamada feita após o método lock() ser chamado externamente. Sabendo o índice desta chamada, saberemos o índice desejado.
    a. Pontos positivos: Simples
    b. Pontos negativos: Tem-se que percorrer o *array* inteiro. Além disso se o nome da classe for mudado, seria preciso mudar a implementação, pois esta abordagem procura diretamente por ReentrantLock.lock.
3. Apesar da aquisição ser feita em três métodos diferentes, o método lock() da classe ReentrantLock é chamado antes deles. Devido a isso é possível obter o *array* de elementos da *stacktrace* neste método, acessar o índice 2 (assim como no momento de salvar onde a trava foi inicializada) e ir repassando essa informação para os métodos seguintes (adicionando um novo parâmetro nos métodos) até chegar nos três métodos finais.
    a. Pontos positivos: Independe do nome da classe
    b. Pontos negativos: Seria necessário mudar a implementação de todos os métodos nos caminhos para poder passar esse dado até os três últimos. O que não é uma boa prática.
4. Para cada *thread* que adquiriu/tentou adquirir o *lock,* salvar o par Thread e StackTraceElement assim que for chamado o método lock() ou tryLock() na classe ReentrantLock. Da mesma forma citada acima, seria acessado o índice 2 no *array*

de elementos da pilha. No momento que cada *thread* for lançar a exceção de *deadlock*, basta buscar o StackTraceElement referente a ela mesma.

a. Pontos positivos: Organizado, limpo e direto.
b. Pontos negativos: Aumento da memória necessária. Ao invés de armazenar apenas uma variável, teríamos que salvar um StackTraceElement para todas as *threads* que tentaram adquirir aquela trava, independentemente se ela teve sucesso ou não.

A opção escolhida foi a número 4, optamos por essa opção pois acreditamos que a memória a ser consumida não é um problema tão grande, os pontos positivos sobrepõem esse ponto negativo.

Para a implementação, foi considerado a princípio se utilizar um Map[13] onde a chave seria a Thread e o valor o StackTraceElement, contudo seria preciso lidar com a sincronização. Mesmo que as *threads* acessem chaves diferentes, se mais de uma *thread* modifica a estrutura do Map, nesse caso adicionando novas chaves, é possível se deparar com problemas de concorrência.

Sincronização é muito custoso, o que aumentaria consideravelmente o *overhead*. Devido a isso, decidimos utilizar uma variável ThreadLocal[14] do tipo StackTraceElement, pois não precisamos lidar com o fator sincronização. Além disso, esse tipo de variável permite que seu valor esteja disponível à *thread* que a criou. Em resumo, cada *thread* só acessa a "mesma" variável, porém essa variável apresenta um valor diferente para cada *thread*.

Para facilitar o entendimento de como ocorreu o *deadlock*, é exibida a *stacktrace* da *thread* e onde a *thread* tentou adquirir as duas travas conforme a Figura 5. Como mencionado, a *stacktrace* mostrada só informa as funções aninhadas referente ao caminho que a *thread* fez para adquirir o segundo *lock*. Devido a isso, a chamada de código na qual a *thread* adquiriu a primeira trava não estará visível na *stacktrace*, e assim o usuário teria que ir analisando chamada por chamada na pilha para procurar onde foi feita a aquisição. Em alguns casos isto é simples, como no caso abaixo:

[13] https://docs.oracle.com/javase/7/docs/api/java/util/Map.html
[14] https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html

```
Second lock acquisition:
main.SWT.createSurface(SWT.java:21)

First lock acquisition:
main.BusyIndicator.updateAnimation(BusyIndicator.java:35)

Full StackTrace:
      at locks.ReentrantLock.lock(ReentrantLock.java:290)
      at main.SWT.createSurface(SWT.java:21)
      at main.UIImage.<init>(UIImage.java:9)
      at main.BusyIndicator.updateAnimation(BusyIndicator.java:37)
      at main.BusyIndicator.run(BusyIndicator.java:48)
```

Figura 5: Saída exibida ao usuário contendo a *stacktrace* e onde cada *lock* foi adquirido.

Nesse caso a chamada "main.BusyIndicator.updateAnimation(BusyIndicator.java:35)" não está na pilha, contudo o usuário pode perceber a chamada de "main.BusyIndicator.updateAnimation(BusyIndicator.java:37)" na pilha e entender o fluxo da *thread.* Porém isso pode ser mais complicado, o método onde a *thread* adquiriu a primeira trava pode não estar sendo mostrado na *stacktrace* e assim o usuário terá que procurar de método em método até encontrar o momento da primeira aquisição.

Para resolver o problema citado acima, decidiu-se salvar a *stacktrace* inteira (um *array* de StackTraceElement), ao invés de salvar apenas o momento de aquisição do *lock* (StackTraceElement). Assim é possível informar ao usuário os métodos que estão omitidos na pilha e assim não será preciso procurar no código os caminhos feitos pela *thread* até chegar nas duas travas.

4.2.4 Criação de *threads*

Inicialmente a ideia era fazer o mesmo que está sendo feito com o *lock*: modificar a implementação da classe atual. Contudo isto não foi possível. Foi deito o download da classe Thread[15] do pacote java.lang, ao tentar modificá-la notou-se que ela utiliza na sua implementação classes do mesmo pacote. Por questões de visibilidade não é possível modificar a Thread sem ter acesso a essas classes. Para resolver esse problema, tentou-se baixar as classes necessárias para remover esse problema de visibilidade. À medida que uma nova classe era incluída (baixada), novas classes eram necessárias. Em um momento mais de

[15] http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/lang/Thread.java

20 classes já tinham sito baixadas para tentar contornar esse problema. Devido a isso acredita-se que esta opção não é muito viável.

A segunda abordagem tentada foi utilizar API de instrumentação[16]. Esta API fornece serviços que permitem que agentes de linguagem de programação Java instrumentem programas em execução no JVM. Contudo também foram encontrados alguns problemas. Os dois problemas principais são:

1) Se o usuário utiliza versões diferentes do JDK e do Java Runtime Environment (JRE) é lançada uma exceção. Conforme a imagem abaixo. Para resolver esse problema é preciso copiar o arquivo attach.dll do JDK utilizado e colocar no JRE.

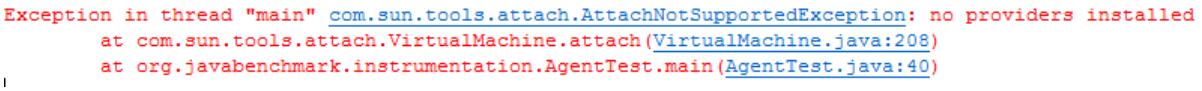

Figura 6: Erro obtido ao tentar utilizar Java Instrumentation com versões diferentes de JDK e JRE.

2) O ClassFileTransform[17] não aplica as transformações na classe Thread do pacote java.lang. Não há uma certeza na explicação do porquê que isso ocorre, mas acredita-se que o instrumentador já carregou a classe Thread antes do método premain ser chamado no agente.

O ClassFileTransform é uma interface que provê a implementação responsável por transformar as classes. Depois que a Java Virtual Machine (JVM) foi inicializado, cada método premain será chamado na ordem os agentes foram especificados, então o método main da aplicação real será chamado. O método premain adiciona os ClassFileTransform no agente.

Em decorrência desses impedimentos, optou-se por não adicionar onde as *threads* começaram a ser executadas no momento. É necessário um estudo mais profundo acerca das possíveis soluções.

4.3 Exibição das novas informações

Nesta seção serão detalhadas as mudanças/adições feitas à mensagem exibida nas exceções para melhorar o entendimento do usuário acerca do *deadlock.* Primeiramente foram removidas informações desnecessárias da *stacktrace*, depois foi adicionado um fluxograma

[16] https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html
[17] https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/ClassFileTransformer.html

parte representar todos os métodos chamados pela *thread* que influenciam no *deadlock*, além das *stacktraces* para cada trava. Por fim adicionou-se a chamada de código que representa onde as travas envolvidas no problema foram criadas.

4.3.1 Remoção de informações desnecessárias

A pilha de chamadas de Java mostra por padrão todas as funções aninhadas chamadas pela *thread* até o momento do lançamento da exceção. A imagem abaixo exemplifica como eram as *stacktraces* anteriormente:

```
Exception in thread "Thread-0" Exception in thread "Thread-2" locks.DeadlockException
        at locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:821)
        at locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:858)
        at locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1187)
        at locks.LockA$NonfairSync.lock(LockA.java:180)
        at locks.LockA.lock(LockA.java:256)
        at main.SWT.createSurface(SWT.java:20)
        at main.UIImage.<init>(UIImage.java:9)
        at main.BusyIndicator.updateAnimation(BusyIndicator.java:36)
        at main.BusyIndicator.run(BusyIndicator.java:47)
```

Figura 7: Mensagem de erro exibida pelo ReentrantLock modificado ao encontrar uma situação de *deadlock*.

As linhas que vão de "locks.LockA.lock()" até "locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt()" são chamadas internas da implementação do *lock.* Optou-se por removê-las pois estas não influenciam positivamente no entendimento do problema. Além disso, novas informações estão sendo adicionadas, como visto na seção de mensagens de erros, é importante exibir apenas dados que irão ajudar o usuário a entender o problema. Foi mantida apenas a primeira chamada interna (o método lock()) para o usuário saber exatamente o que aconteceu em cada pilha. O resultado final de como a mensagem de erro é exibida atualmente será detalhado na próxima seção.

4.3.2 Fluxograma

Como dito anteriormente, a pilha de chamadas está incompleta. Tendo acesso as duas *stacktraces* para cada trava é possível encontrar os métodos que são omitidos na *stacktrace* já exibida. Baseado nos estudos acerca das mensagens de erros, acredita-se que unir as duas pilhas de alguma forma visual fica mais claro para o usuário, ao invés de mostrar apenas as duas *stacktraces.* Optou-se então por representá-las através de um grafo, ou melhor, um fluxograma, com o ancestral comum e o caminho até ele.

Cada *thread* terá seu fluxograma, consequentemente serão exibidos dois fluxogramas e em cada um é possível entender o que acontece em cada *thread* até acontecer o *deadlock.* A imagem abaixo mostra o resultado final para uma *thread:*

Figura 8: Exceção lançada pela primeira *thread*, chamada de Thread-0.

Figura 9: Exceção lançada pela segunda *thread*, chamada de Thread-2.

Decidiu-se por mostrar também as duas *stacktraces* caso o usuário deseje ter uma visão separada para cada trava. A sequência de chamadas no lado esquerdo representa a

stacktrace que seria mostrada ao usuário, exceto pelo método encapsulado por uma caixa com pontos e hifens. Essa caixa contém o primeiro método da *stacktrace* para a primeira trava que não é comum com os métodos da *stacktrace* do segundo *lock.* Todos os métodos apresentados antes da caixa pontilhada são comuns a ambas as travas.

A caixa com pontos e barras retas representam o caminho que levou a *thread* a adquirir cada trava. A caixa onde tem escrito "FIRST LOCK" representa a primeira *stacktrace* sem os ancestrais comuns entre as pilhas.

Como visto na Figura 8, o primeiro método chamado pela thread-0 é "main.UIThread.run(UIThread.java:17)", depois a *thread* chama o método "main.UIWindow.paint(UIWindow.java:25)", dentro deste método são chamados todas as funções apresentadas na caixa da direita até o momento do *lock.* Logo após efetuar o travamento, a execução volta para o método paint e continua sua execução até parar em " main.BusyIndicator.getImage(BusyIndicator.java:18)" e a segunda tentativa de *lock* ser efetuada. Neste momento o *deadlock* é detectado e a exceção é lançada.

4.3.3 Criação das travas

O objetivo de mostrar onde as travas foram criadas é mapear os entre os ids de objeto delas para seu elemento estático, ou seja, sua instância. Essa informação é crucial para o usuário, uma vez que saber quem são as travas envolvidas é uma das primeiras coisas que os desenvolvedores buscam saber. Optou-se por mostrar onde ambas as travas foram criadas no início da mensagem. Desta forma o usuário primeiro entende quais são as travas envolvidas e logo após, através das *stacktraces* e fluxograma, entende como o problema acontece. Abaixo temos uma visão simplificada das mensagens que o usuário irá visualizar.

```
Exception in thread "Thread-0" locks.DeadlockException:

The first lock was created at:
       main.SWT.<clinit>(SWT.java:13)

The second lock was created at:
       main.BusyIndicator.<init>(BusyIndicator.java:13)

First lock Stacktrace:
       ...

Second lock Stacktrace:
       ...

THREAD FLOWCHART:
       ...

Exception in thread "Thread-2" locks.DeadlockException:

The first lock was created at:
       main.BusyIndicator.<init>(BusyIndicator.java:13)

The second lock was created at:
       main.SWT.<clinit>(SWT.java:13)

First lock Stacktrace:
       ...

Second lock Stacktrace:
       ...

THREAD FLOWCHART:
       ...
```

Figura 10: Nova mensagem de erro contendo o local no código onde cada *lock* foi criado.

4.4 Avaliação de overhead

Foi realizado um conjunto preliminar de experimentos para analisar o overhead da abordagem deste trabalho. Comparou-se a atual implementação de ReentrantLock, com o ReentrantLock original e o ReentrantLock modificado em [LOBO; CASTOR, 2015], sem as novas informações.

Assim como no trabalho base [LOBO; CASTOR, 2015], foi desenvolvido um *benchmark* sintético que cria N *threads* que executam adições a dez contadores de inteiros onde cada incremento em um contador é protegido por uma trava explícita. Cada *thread* tinha que incrementar seu respectivo contador 1000 vezes antes de terminar a sua execução, os contadores foram distribuídos uniformemente entre as *threads.* Portanto, cada contador tem exatamente (N / 10) *threads* fazendo incrementos sobre ele e os valores mais elevados de N resultam em uma maior contenção, isto é, mais *threads* irão competir umas contra as outras

para um contador particular. Dado que nenhuma *thread* no *benchmark* adquire mais de uma *t*rava ao mesmo tempo, *deadlocks* não podem ocorrer. Ressalta-se que esta configuração é muito conservadora, uma vez que cada operação que cada *thread* executa requer travamento. Assim, a sobrecarga obtida será uma estimativa do pior caso e, portanto, muito maior do que se poderia encontrar em uma aplicação do mundo real [LOZI et al., 2012].

Nesta avaliação preliminar, foram realizadas medições para valores de N igual a 10, 50, 100 e 200. As medições foram feitas em um processador Intel Core™ i7 4510U (4 Mb de cache, 2.0 GHz) rodando Windows 8.1, usando a versão 1.7.0_79 do JDK, com Java (TM) SE Runtime Environment (versão 1.8.0_91-b15) e Java HotSpot (TM) 64-Bit VM Server (build 25.91-b15, de modo misto). Cada célula da Tabela 2 é a média de 50 execuções (precedidas por 20 execuções que serviram como um *warm-up).*

# *Threads*	ReentrantLock	ReentrantLock modificado	ReentrantLock modificado + novas informações
10	5.94	7.80	35.22
50	5.62	6.26	98.00
100	10.94	11.86	197.30
200	21.26	22.20	393.10

Tabela 2: Média das 50 execuções para os três tipos de ReentrantLock. O original, o previamente modificado e o atual com as novas informações.

A diferença entre os tempos do ReentrantLock original e a nova versão com as modificações nele são de aproximadamente 7 a 19 vezes mais. Isso ocorre porque é necessário obter a *stacktrace* sempre que ocorre um travamento. Obter a *stacktrace* é bastante custoso até o momento. A chamada Thread.currentThread() não é tão custosa, o problema está em obter a *stacktrace* no formato usado internamente e depois convertê-la para o formato StackTraceElement[].

Devido a isso, buscou-se outras formas de se obter a *stacktrace* e foi visto que obter a *stacktrace* a partir de um Throwable[18] é um pouco mais rápido, porém o tempo ainda não é bom. Era criado um novo objeto Throwable e logo em seguida era obtida a *stacktrace* do

[18] https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

mesmo. Contudo, foi visto que ao criar um novo objeto Throwable é chamado o método fillInStackTrace() e é este método que constrói a *stacktrace* no formato usado internamente, sendo assim, resolveu-se salvar apenas a instância do novo objeto Throwable e no momento de detecção do *deadlock* é feita a transformação da *stacktrace* interna para o formato StackTraceElement[] ao chamar o método getStackTrace(). Esta mudança diminuiu bastante o tempo em relação à primeira alternativa, mas ainda não é a forma ideal. Todavia, não foi encontrada nenhuma outra forma de se obter a *stacktrace*. Os tempos para cada uma das opções pode ser visto na tabela abaixo:

# *Threads*	*Stacktrace* da *thread* atual	*Stacktrace* de um Throwable	Salvar o Throwable e obter a *stacktrace* depois
10	24.04	31.82	18.12
50	97.26	82.32	24.52
100	192.72	163.46	48.20
200	387.22	326.56	96.44

Tabela 3: Média das 50 execuções para as três formas de coleta das *stacktraces*.

A tabela abaixo representa a nova comparação entre a abordagem atual e o ReentrantLock original e o já modificado para detectar *deadlocks*.

# Threads	ReentrantLock	ReentrantLock modificado	ReentrantLock modificado + novas informações
10	4.30	4.70	15.36
50	12.22	10.96	22.72
100	10.24	11.16	45.78
200	21.22	22.38	97.54

Tabela 4: Nova média das 50 execuções para os três tipos de ReentrantLock. O original, o previamente modificado e o atual com as novas informações após alterar o modo de obtenção de *stacktraces*.

A diferença entre os tempos do ReentrantLock original e a nova versão com as modificações são de 2 a 4 vezes mais. O que ainda não é o ideal, mas é o possível a ser feito se for desejado apontar os locais no código onde foram feitas chamadas ao construtor e ao momento do lock() e tryLock().

5. Conclusão

Neste trabalho, foi inicialmente discutido o desafio que é lidar com *deadlocks* e resolvê-los em *softwares* do mundo real. Primeiramente foram introduzidos os fundamentos da programação concorrente. Logo em seguida apresentou-se algumas abordagens anteriores que tentaram identificar ou solucionar *deadlocks* com análise estática, análise dinâmica ou uma mistura das duas abordagens. Além de informar alguns trabalhos que buscaram solucionar problemas da programação concorrente como condição de corrida e problemas de atomicidade em tempo de execução.

Visto que a saída exibida ao usuário que utiliza o protocolo [LOBO; CASTOR, 2015] nem sempre é suficiente para um usuário detectar corretamente o *deadlock*, decidiu-se ampliá-lo. Foram investigadas quais novas informações seriam adicionadas ao protocolo, além de terem sido realizados estudos acerca da resposta dos alunos em experimentos, bem como analisando as informações informadas pelos desenvolvedores em fóruns de *softwares* de código aberto. Além disso foi feito um estudo acerca de quais informações devem ser mostradas na mensagem de erro e como elas devem ser mostradas.

Ampliou-se o protocolo [LOBO; CASTOR, 2015] informando ao usuário onde os *locks* envolvidos foram criados, as *stacktraces* no momento de aquisição de cada trava e um fluxograma representando o início da execução da *thread* até o momento do *deadlock*. Vale salientar que o fluxograma inclui apenas os métodos relevantes para o entendimento do *deadlock.*

A avaliação de *overhead* permitiu concluir que mostrar esse nível de detalhes ao usuário pode ser bastante caro em termos de tempo computacional. Obter a *stacktrace* não é uma tarefa rápida, dado que esta é obtida diversas vezes durante a execução do programa de teste. Entretanto, vale salientar que os testes de *overhead* foram feitos através de um programa com uma configuração muito conservadora, uma vez que cada operação que cada *thread* executa requer acesso ao único *lock* da classe. Assim, o *overhead* obtido será uma estimativa do pior caso e, portanto, muito maior do que se poderia encontrar em uma aplicação no mundo real.

5.1 Contribuições

De forma resumida, pode-se dizer que as principais contribuições deste trabalho foram:

1. Investigou-se as informações que são importantes para o usuário entender como o *deadlock* ocorre. Dentre elas, têm-se: métodos envolvidos no *deadlock*, onde foram feitos os travamentos, travas e *threads* envolvidas.
2. Estudo acerca das mensagens de erro e como elas devem ser mostradas.
3. Ampliação do protocolo feito em [LOBO; CASTOR, 2015].
4. Avaliação do *overhead*. Chegou-se à conclusão que é bastante custoso obter diversas vezes a *stacktrace* durante a execução.

5.2 Trabalhos futuros

Baseado nos problemas encontrados neste trabalho, as seguintes propostas de trabalho são oportunidades de melhoria à implementação atual:

1. Realizar um estudo mais aprofundado acerca de como exibir as informações na mensagem de erro da exceção. Acredita-se que há pouco trabalho feito nesta área, uma alteração na mensagem exibida pode facilitar o entendimento do *deadlock* por parte do usuário.
2. Investigar se é preciso adicionar novas informações. Como por exemplo onde foram criadas as *threads*.
3. Investigar outras formas menos custosas para apontar onde foram feitos os travamentos nos *locks* envolvidos por cada *thread*. É importante diminuir o overhead de execução ao adquirir essa informação.
4. Realizar experimentos e averiguar se os usuários identificam o problema corretamente e mais rápido.

Referências Bibliográficas

AGARWAL, RAHUL; WANG, LIQIANG; STOLLER, SCOTT D. "Detecting potential deadlocks with static analysis and run-time monitoring." Haifa Verification Conference. Springer Berlin Heidelberg, 2005. <https://www.research.ibm.com/haifa/Workshops/PADTAD2005/papers/article.pdf> Acessado em: 06/07/2016

AGARWAL, RAHUL; STOLLER, SCOTT D. "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables."Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging. ACM, 2006. <http://dl.acm.org/citation.cfm?id=1147413> Acessado em: 06/07/2016

ALESSANDRINI, VICTOR. Shared Memory Application Programming: Concepts and Strategies in Multicore Application Programming. Morgan Kaufmann, 2015.

BREUER, PETER T.; VALLS, MARISOL GARCIÁ. "Static deadlock detection in the Linux kernel." International Conference on Reliable Software Technologies. Springer Berlin Heidelberg, 2004. <http://link.springer.com/chapter/10.1007/978-3-540-24841-5_4> Acessado em: 16/07/2016

BROWN, PHILIP J. "Error messages: the neglected area of the man/machine interface." Communications of the ACM 26.4 (1983): 246-249. <http://dl.acm.org/citation.cfm?id=358083> Acessado em: 15/07/2016

BUSTARD, DAVID W. Concepts of Concurrent Programming. No. SEI-CM-24. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1990. <ftp://ftp.sei.cmu.edu/pub/education/cm24.pdf> Acessado em: 04/06/2016

CAVALHEIRO, GERSON GH. "Princípios da programação concorrente." ERAD, 2004. <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2004/002.pdf> Acessado em: 06/06/2016

CHEN, FENG; SERBANUTA, TRAIAN FLORIN; ROSU, GRIGORE. "jPredictor: a

predictive runtime analysis tool for Java." Proceedings of the 30th international conference on Software engineering. ACM, 2008. <http://dl.acm.org/citation.cfm?id=1368119> Acessado em: 06/07/2016

CHOI, JONG-DEOK, et al. "Efficient and precise datarace detection for multithreaded object-oriented programs." ACM SIGPLAN Notices 37.5 (2002): 258-269. <http://dl.acm.org/citation.cfm?id=512560> Acessado em: 05/07/2016

DA LUO, ZHI; DAS, RAJA; QI, YAO. "Multicore sdk: a practical and efficient deadlock detector for real-world applications." 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. IEEE, 2011. <http://dl.acm.org/citation.cfm?id=1990097> Acessado em: 04/07/2016

FLANAGAN, DAVID. Java in a Nutshell. " O'Reilly Media, Inc.", 2005.
Grechanik, Mark, et al. "Preventing database deadlocks in applications."Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013. <http://dl.acm.org/citation.cfm?id=2491412> Acessado em: 16/07/2016

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

HANSEN, PER BRINCH. "The invention of concurrent programming." The origin of concurrent programming. Springer New York, 2001. 3-61. <http://oberon2005.oberoncore.ru/paper/bh2002.pdf> Acessado em: 04/06/2016

HAVELUND, KLAUS; PRESSBURGER, THOMAS. "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

HAVELUND, KLAUS. "Using runtime analysis to guide model checking of Java programs." International SPIN Workshop on Model Checking of Software. Springer Berlin Heidelberg,

2000. <https://ti.arc.nasa.gov/m/pub-archive/archive/0177.pdf> Acessado em: 06/07/2016

JAGTAP, MAHESHKUMAR P. "Era of Multi-Core Processors." Power 2 (2009): 2.<http://www.drdo.res.in:8080/alpha/drdo/pub/dss/2009/main/16-ANURAG.pdf> Acessado em: 11/04/2016

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

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

LOBO, RAFAEL; CASTOR, FERNANDO. "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

LOZI, JEAN-PIERRE, et al. "Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications."Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12). 2012. <https://www.usenix.org/system/files/conference/atc12/atc12-final237.pdf> Acessado em: 16/07/2016

LU, SHAN, et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." ACM Sigplan Notices. Vol. 43. No. 3. ACM, 2008. <http://dl.acm.org/citation.cfm?id=1346323> Acessado em 06/07/2016

MARCEAU, GUILLAUME; FISLER, KATHI; KRISHNAMURTHI, SHRIRAM. "Mind your language: on novices' interactions with error messages." Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software. ACM, 2011. <https://cs.brown.edu/~sk/Publications/Papers/Published/mfk-mind-lang-novice-inter-error-msg/paper.pdf> Acessado em 16/07/2016

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

NIENALTOWSKI, MARIE-HÉLÈNE; PEDRONI, MICHELA; MEYER, BERTRAND. "Compiler error messages: What can help novices?" ACM SIGCSE Bulletin. Vol. 40. No. 1. ACM, 2008. <http://se.ethz.ch/~meyer/publications/teaching/compiler-errors.pdf> Acessado em: 15/07/2016

PYLA, HARI K.; VARADARAJAN, SRINIDHI. "Transparent runtime deadlock elimination." Proceedings of the 21st international conference on Parallel architectures and compilation techniques. ACM, 2012. <http://dl.acm.org/citation.cfm?id=2370905> Acessado em: 16/07/2016

ROMERO, FERNANDO. "Operating Systems. A concept-based approach."Journal of Computer Science & Technology 9 (2009).

SHARAN, KISHORI. Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams. Apress, 2014.

SHNEIDERMAN, BEN. "Designing computer system messages."Communications of the ACM 25.9 (1982): 610-611. <https://babu.cs.umd.edu/~ben/papers/Shneiderman1982Designing.pdf> Acessado em 15/07/2016

SILBERSCHATZ, ABRAHAM; GALVIN, PETER BAER; GAGNE, GREG. Operating System Principles. John Wiley & Sons, 2008.

SINGHAL, MUKESH. "Deadlock detection in distributed systems." Computer22.11 (1989): 37-48. <http://www.kiv.zcu.cz/~ledvina/ds/~singhaldeadsurvey.pdf> Acessado em: 13/04/2016

TANENBAUM, ANDREW. "Modern operating systems." (2009).

TOSCANI, SIMÃO SIRINEO; CARISSIMI, ALEXANDRE DA SILVA. Sistemas operacionais e programação concorrente. Sagra Luzzatto, 2003.

TRAVER, V. JAVIER. "On compiler error messages: what they say and what they mean." Advances in Human-Computer Interaction 2010 (2010). <http://www.hindawi.com/journals/ahci/2010/602570/abs/> Acessado em: 15/07/2016

WANG, LIQIANG; STOLLER, SCOTT D. "Accurate and efficient runtime detection of atomicity errors in concurrent programs." Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 2006. <http://dl.acm.org/citation.cfm?id=1122993> Acessado em: 06/07/2016

WANG, YIN. Software failure avoidance using discrete control theory. ProQuest, 2009. <https://deepblue.lib.umich.edu/bitstream/handle/2027.42/62364/yinw_1.pdf?sequence=4> Acessado em: 04/07/2016

WILLIAMS, AMY; THIES, WILLIAM; ERNST, MICHAEL D. "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