Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Um estudo sobre a incidência de bugs relacionados a deadlocks em aplicações C# de código aberto

Rafael Acevedo de Aguiar

Trabalho de Graduação

Recife
Julho de 2016
Universidade Federal de Pernambuco
Centro de Informática

Rafael Acevedo de Aguiar

Um estudo sobre a incidência de bugs relacionados a deadlocks em aplicações C# de código aberto

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.

Orientador: *Prof. Dr. Fernando José Castor de Lima Filho*

Recife
Julho de 2016

Agradecimentos

Primeiramente, gostaria de agradecer à Universidade Federal de Pernambuco, principalmente, a todos que formam o Centro de Informática, centro de excelência e destaque internacional. Durante toda a minha graduação, puder ter contato com professores, funcionários e alunos que amam o que fazem e que, desde o início do curso me motivaram, inspiraram e contribuíram diretamente para que eu pudesse alcançar meus sonhos e objetivos.

Ao professor e orientador deste trabalho Dr. Fernando José Castor de Lima Filho, que tem feito um brilhante trabalho me orientando desde quando o conheci, em 2014, tendo a honra de ter sido seu aluno na disciplina de Paradigmas de Linguagens Computacionais. Foi graças a esta disciplina que me interessei pela área de programação concorrente, área considerada de bastante relevância para a computação e na qual pude me aprimorar com a ajuda do professor Fernando. Com essa base de conhecimento, pude conseguir meu primeiro emprego na área de engenharia de software, no CESAR, onde poderei exercitar e pôr em prática os conhecimentos aprendidos ao longo desses anos.

À minha turma, que esteve comigo desde o início da graduação, que me incentivaram e motivaram este tempo todo. Com certeza estar rodeado de pessoas tão inteligente fez com que eu pudesse extrair o máximo de conhecimento possível durante o período da graduação. Em especial, gostaria de agradecer a Eduardo, Lucas Netto, Guilherme, Mateus, Duhan, Bertha, Maria Gabriela, Leonardo, Lucas Lima, Marina, Larissa, Rafael, João Pedro e Vinícius, colegas de graduação e grandes amigos com que ao longo da graduação, dividi momentos ruins, mas também momentos muito bons que ficarão guardados na memória.

Por fim, à toda minha família. Especialmente aos meus pais, Rafael e Ana Cristina, que desde a escolha do curso me apoiaram e confiaram na minha decisão, me dando todo o suporte educacional e emocional para que essa graduação fosse possível.

Resumo

Linguagens de programação de alto nível são o padrão de fato para desenvolvimento de sistemas de software, o que torna cada vez mais abstrata a interação do programador com a máquina. Isso torna possível que sistemas cada vez mais complexos sejam construídos. Para fazer maior uso do potencial de processadores multinúcleo, processadores que são capazes de computar várias instruções ao mesmo tempo, são utilizadas threads para computar de forma paralela. Enquanto computar de forma paralela traz um ganho significativo de desempenho, também traz mais dificuldades para quem está programando, visto que o programador deve se preocupar com o acesso destas threads aos recursos do sistema Esta preocupação é real, pois caso a sincronização desse acesso esteja errada, pode levar a bugs como condições de corrida e deadlocks.Este trabalho tem como objetivo analisar bugs relacionados a deadlocks em aplicações C# open-source. Assim, o foco do estudo é levantar informações relevantes para a comunidade, como: o tipo do deadlock existente, quais threads estão envolvidas, se o bug é realmente relacionado a um deadlock, e abordar maneiras para solucionar o defeito analisado.

Palavras-chave: processadores multi-núcleo, threads, sincronização de acesso, deadlock, bugs, open-source.

Abstract

High-level programming languages are the true standard for software development, which makes the programmer-machine interaction even more abstract, allowing high complexity systems to be built. In order to make better use of multicore processors(the ones that can process several instructions simoutaneously), threads are used for parallel computing. While that improves performance significantly, it also brings more difficulty to the programmer, who has to worry about handling access of these threads to system resources. This is a big drawback, because if the access synchronization is not correct, bugs(such as deadlocks and race conditions) can happen very easily. This work aims toanalyze bugs related to deadlocks in open-source C# applications. Thus, the focus of the study is to gather relevant information for the community, such as the kind of deadlock, which threads are involved, whether the bug is really related to a deadlock and discuss ways to solve the analyzed defect.

Keywords: multicore processors, threads, access synchronization, deadlock, bugs, open-source.

Sumário

1. Introdução

Linguagens de programação de alto nível são o padrão de fato para desenvolvimento de sistemas de software[1], o que torna cada vez mais abstrata a interação do programador com a máquina. Isso torna possível que sistemas cada vez mais complexos sejam construídos. Para fazer maior uso do potencial de processadores multi-núcleo, processadores que são capazes de computar várias instruções ao mesmo tempo, são utilizadas threads para computar de forma paralela.

Enquanto computar de forma paralela traz um ganho significativo de desempenho[2], também traz mais dificuldades para quem está programando, visto que o programador deve se preocupar com o acesso destas threads aos recursos do sistema. Esta preocupação é real, pois caso a sincronização desse acesso esteja errada, pode levar a bugs como condições de corrida a deadlocks[3]. Se sincronização for sub-utilizada, duas ou mais threads podem acessar uma mesma região de memória ao mesmo tempo, o que pode levar a inconsitências. Por outro lado, excesso de sincronização pode causa problemas de desempenho[4]. Por fim, uma ordenação de operações de sincronização que torna possível que duas ou mais threads fiquem bloqueadas, cada uma esperando por um recurso que a outra não pode liberar, leva à ocorrência de um deadlock. Deadlocks podem ser extremamente difíceis de ser identificados. Por isso foram criadas varias abordagens de detecção e prevenção de deadlocks, como detecção estática[5,6] e dinâmica[7,8].

Uma das vantagens de projetos open-source é que toda a comunidade de desenvolvedores pode contribuir de alguma forma. Isto também se aplica a reportar bugs e discutir possíveis soluções. Como dito anteriormente, deadlocks são difíceis de detectar, assim, a ajuda de ferramentas e da comunidade é de extrema relevância. Um estudo que aborda isto é apresentado em [9], onde são apresentadas análises de bugs relacionados a deadlock em Java, além de uma implementação de travas[10] que trata deadlocks como exceções em tempo de execução. Este tratamento de deadlocks como exceções ajuda bastante na detecção da causa do deadlock, pois, em [9], a exceção levantada traz informações cruciais, como, por exemplo, as threads envolvidas no deadlock, facilitando a resolução de defeitos desse tipo.

1.1 Objetivos

O objetivo deste trabalho é analisar bugs relacionados a deadlocks em aplicações C# open-source. Assim, o foco do estudo é levantar informações relevantes para a comunidade, como: o tipo do deadlock existente, quais threads estão envolvidas, se o bug é realmente

relacionado a um deadlock, e, caso possível, abordar maneiras para solucionar o defeito analisado.

Será realizado um estudo sobre quais serão as aplicações usadas para obtenção de defeitos que se relacionam com deadlock. Após esse levantamento,os bugs serão divididos em categorias, de maneira similar ao que é feito em [9], categorizando-os para facilitar a análise que seguirá.

Em seguida, serão analisados profundamente os bugs selecionados a fim de obter as informações desejadas pela comunidade: tipo do deadlock(recurso, comunicação, etc), as threads envolvidas no deadlock(isto pode ser feito manualmente ou com ferramentas, como o jstack[11]).

1.2 Estrutura do Trabalho

Este trabalho está dividido em 5 capítulos, incluindo este capítulo introdutório. No Capítulo 2, será discutido com maior profundidade o conceito de projetos open-source, detalhando como são estruturados, focando nos papéis de cada integrante e no processo de *bug reporting*. Após isso, o Capítulo 3 discorrerá sobre concorrência e threads, abordando os problemas que surgem quando existem instruções rodando simultaneamente e entrando em detalhes na implementação desses conceitos em C#. O Capítulo 4 apresentará o estudo realizado pelo autor sobre *bugs* relacionados a deadlocks em C#, mostrando os resultados obtidos e discutindo o porquê deles. Ao final deste trabalho, o Capítulo 6 expõe as conclusões obtidas, mostrando os resultados do estudo e desafios encontrados na área, bem como as limitações e sugestões para trabalhos futuros.

2. Aplicações Open-Source

Antes de entrarmos em detalhes sobre *bug reports* em projetos open source(OS), precisamos discutir alguns conceitos relacionados a esse tipo de projeto, bem como sua evolução desde quando surgiu até os tempos atuais. Necessitamos também mostrar o cenário atual da área, abordando as ferramentas mais utilizadas e como é realizada a organização dos projetos. Após essas definições, este capítulo discutirá mais profundamente projetos OS em C#, mostrando alguns exemplos de projetos reais. Em seguida, discutiremos como os bugs são reportados em aplicações OS, exemplificando com defeitos reportados por contribuidores.

2.1 Definições e Histórico

Apesar de a tradução literal de OS ser algo como "código-fonte aberto", não basta apenas o código de um projeto estar aberto para ele ser considerado OS. Existe uma série de regras a serem seguidas para que um certa aplicação seja considerada OS(OPEN SOURCE INITIATIVE, 2007), como livre redistribuição do produto(incluindo venda), modificação do código-fonte(permitindo que aplicações derivadas sejam criadas livremente) e não haver nenhum tipo de discriminação com pessoas, grupos ou áreas de pesquisa(por exemplo, liberar uso apenas para pesquisa na área de biologia). Dessa maneira, é garantido que um projeto OS é livre para uso e redistribuição por qualquer um.

Todos os projetos OS devem ser livres para uso e redistribuição, porém, existe o que se chama de licença, que especifica questões como *copyrights* e termos para redistribuição, mas sempre respeitando os conceitos de OS. Cada projeto OS deve espeficicar a licença que usa. Existem licenças que dão maior flexibilidade, como a MIT[1], e outras mais restritas, que se preocupam com patentes, como a Apache License 2.0[2], usada em projetos como o Android[3].

Desenvolvimento baseado em colaboração distribuída existe desde quando os primeiros programas foram desenvolvidos(OPEN SOURCE INITIATIVE, 2012), porém, nessa época, OS não era tão popular devido a dificuldade de manter o código centralizado e gerenciar mudanças. Foi no final dos anos 90, com a abertura do código do browser

[1] https://opensource.org/licenses/MIT

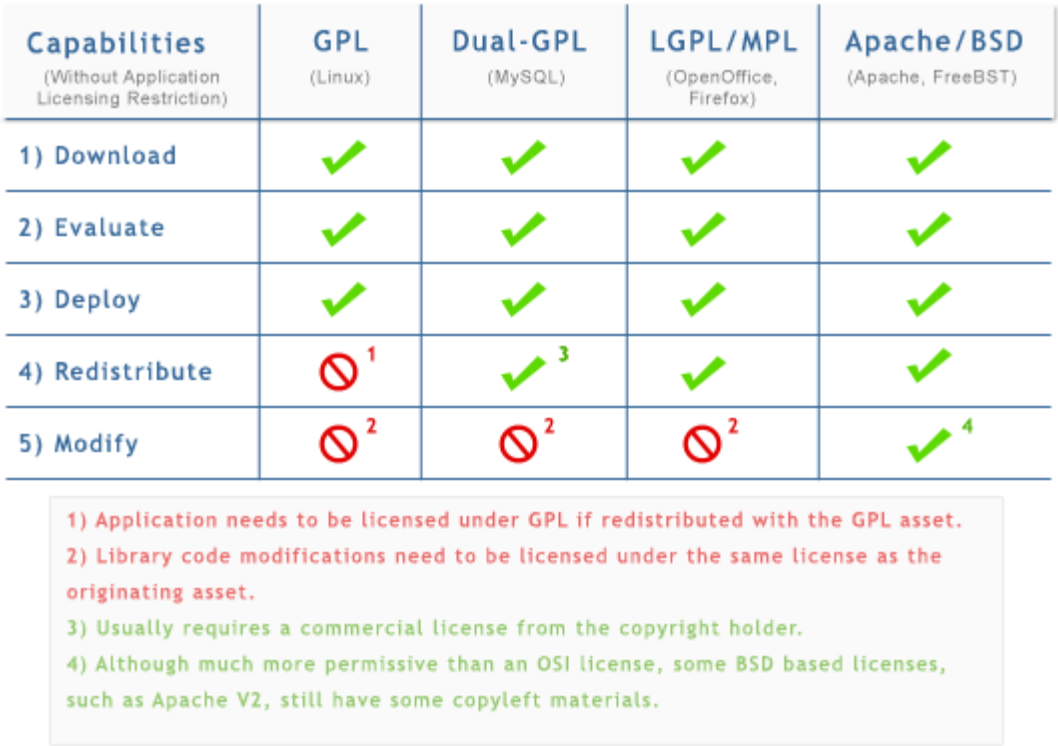
[2] https://opensource.org/licenses/Apache-2.0

[3] https://www.android.com

Netscape[4] e a crescente popularização do Linux que o movimento OS realmente foi alavancado, e, devido a essa demanda, foram criadas melhores ferramentas de controle de versão, como o SVN[5], que possibilitaram melhor colaboração distribuída.

Desde então, com a popularização da internet, ficou extremamente fácil para os colaboradores se comunicarem, dado que os contribuidores tem acesso à internet quase que o tempo todo, seja de um celular ou de um computador. Mais recentemente, em 2005, surgiu o Git, que foi uma espécie de evolução do SVN, que hoje é o sistema de controle de versão mais popular. Na Figura 1 são mostradas mais licenças e suas restrições.

Figura 1: Principais licenças OS e suas restrições



Fonte: stackexchange.com

2.2 Cenário atual

Desde sua popularização, projetos OS tem atraído cada vez mais usuários devido, entre outros fatores, ao reconhecimento que é dado a contribuidores de destaque nos grandes projetos. Financeiramente, não há nenhum grande ganho para quem contribui. Por outro lado, o colaborador que se destaca em um projeto é visto como um bom profissional, o que faz

[4] https://blog.lizardwrangler.com/2008/01/22/january-22-1998-the-beginning-of-mozilla/

[5] https://subversion.apache.org.

com que muitas empresas se interessem nele, assim, não é raro recrutadores entrarem em contato com desenvolvedores através de mensagens do tipo "Olhamos suas contribuições no GitHub e ficamos interessados em tê-lo conosco".

Hoje, existem várias plataformas que hospedam projetos open-source, como: GitHub[6] e BitBucket[7], e elas estão cada vez mais completas, de forma que tudo relacionado ao projeto pode ser gerenciado por elas: *bugs*, documentação(geralmente são utilizados arquivos de Markdown[8]) e projetos derivados(chamados popularmente de *forks*).

Alguns podem pensar que apenas projetos pequenos são open-source, mas há uma tendência atual nas grandes empresas de liberar o código de alguns de seus projetos, como é o caso do Google com o AngularJS[9] e o Android. Entre os principais motivos para isso estão: (BLEKH, 2014)

1) Economia: Ao abrir um projeto, a empresa ganha uma força de trabalho extra e sem custo
2) Aquisição de talentos: Como mencionado anteriormente, quando alguém se destaca num projeto, chama a atenção da empresa, facilitando o recrutamento.
3) Imagem da empresa: Ao liberar o código de algum produto, a empresa ganha uma maior visibilidade em relação à comunidade, o que promove sua imagem.

2.3 Open Source e C#

C# é uma linguagem de programação orientada a objeto criada em 2001 pela Microsoft como parte da plataforma .NET. Na época de sua criação, o movimento OS ainda não era tão forte como é hoje, e, por causa disso, a Microsoft escolheu manter o seu código fechado. Com o fortalecimento do movimento OS durante os anos, ficava cada vez mais claro que, nos projetos OS, a dominância era de linguagens que, ou possuíam seu codigo aberto, ou não eram ligadas a uma plataforma específica, como Ruby e Python(C#, até algum tempo atrás, só conseguia ser compilado no Windows, sistema operacional da Microsoft). Percebendo isso, a Microsoft vem, recentemente, abrindo parte do código da plataforma .NET(MICROSOFT, 2014) e criando compiladores compatíveis com outras plataformas[10], o que fez o uso de C# para projetos OS aumentar, ainda que lentamente. Podemos perceber isso através da Figura 2 abaixo.

---

[6] https://github.com

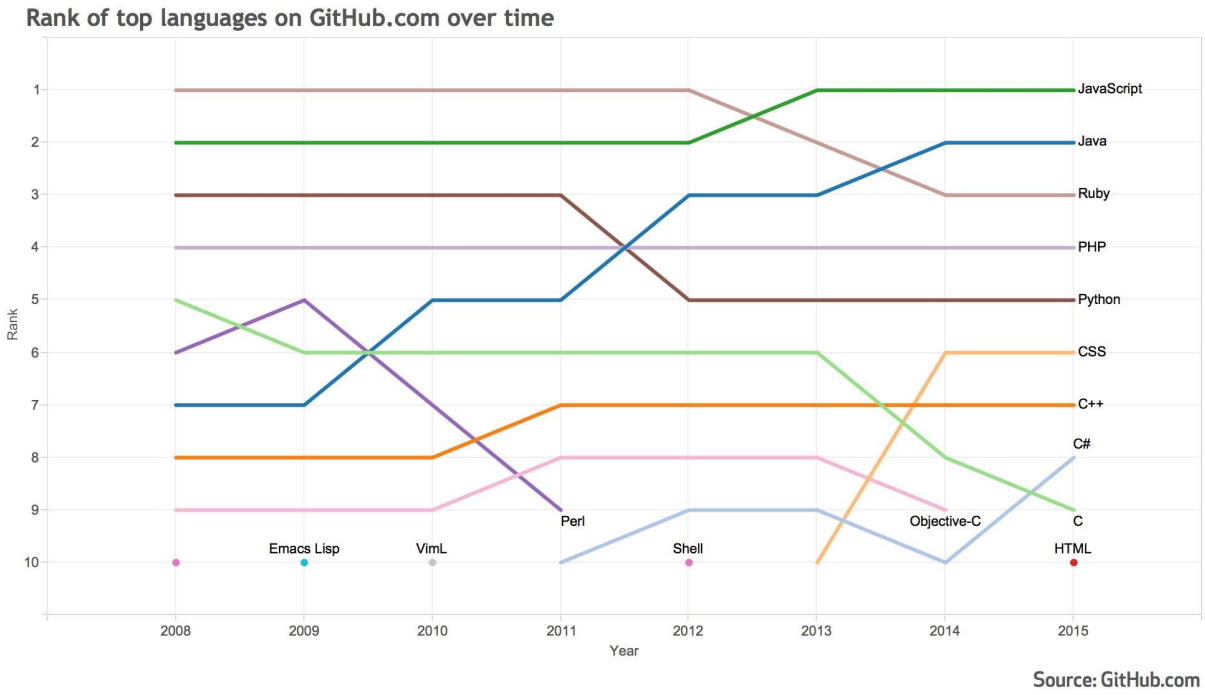
[7] https://bitbucket.org

[8] https://en.wikipedia.org/wiki/Markdown

[9] https://angularjs.org

[10] https://github.com/dotnet/roslyn

Figura 2: Linguagens mais utilizadas em projetos open source no GitHub



Fonte: github.com

Alguns projetos open-source em C# vêm ganhando destaque recentemente, e estes são discutidos abaixo:

2.3.1 Xamarin

Criada em 2011, a Xamarin veio com a proposta clássica de "Write once, run everywhere" para dispositivos móveis. O desenvolvedor escreve sua aplicação em C# e ela é portada automaticamente para as 3 plataformas mobile mais utilizadas atualmente: Android, iOS e Windows Phone.

A Xamarin sempre desenvolveu produtos OS, como sua IDE MonoDevelop[11], que mais tarde se tornou Xamarin Studio. O sucesso dos seus produtos e da sua proposta inicial foi tão grande que chamou a atenção da Microsoft, que adquiriu a empresa em fevereiro de

[11] http://www.monodevelop.com

2016. Em junho de 2016, a Microsoft anunciou que o código da SDK do Xamarin será aberto, o que definitivamente reforça a proposta de que a Microsoft está investindo cada vez mais em OS.

2.3.2 .NET Core

.NET é o framework da Microsoft sob o qual linguagens como C# e Visual Basic executam. Ele contém subdivisões, como o Common Language Runtime, que funciona de forma similar à JVM de Java, de forma que o código roda numa máquina virtual, a qual é responsável por gerenciar a memória e garantir a segurança da aplicação.

Como mencionado anteriormente, a Microsoft, em 2014, abriu parte do código do .NET, e o Core Common Language Runtime(CLR) foi incluído nessa abertura. Assim, constatamos que realmente a Microsoft almeja aumentar a popularidade de linguagens baseadas em .NET(sendo C# a principal) no mundo OS.

2.4 Bug Reports em Projetos Open Source

Uma das principais dificuldades em projetos OS é gerenciar os defeitos encontrados. Com a evolução das ferramentas de controle de versão, muitas delas começaram a suportar *bug reporting* nativamente, como é o caso do GitHub, e outras deixam essa tarefa para sistemas específicos de *bug reporting*, como o JIRA[12].

Contudo, se não gerenciados da forma correta, os relatos de defeitos podem se tornar bastante desorganizados e de difícil entendimento, como ilustra a Figura 3.

Figura 3: Exemplo de bug mal reportado

[12] https://www.atlassian.com/software/jira

Fonte: github.com

A fim de evitar esse problema, foram criadas diversas técnicas para organizar melhor os defeitos reportados, tais como:

- *Labels*
  Labels são marcadores que são atribuídos a um defeito para indicar o seu tipo, área do sistema afetada, prioridade e status. Por exemplo, é comum aplicar *labels* como "Backend", "UI" para indicar a parte do sistema que um *bug* afeta, ou "*Won't fix*", "*Fixing*" para indicar o estado atual do defeito.A Figura 4 mostra exemplos de marcadores usados no projeto *coreclr*[13] do .NET Estes marcadores facilitam bastante o gerenciamento dos *bugs*, uma vez que é possível listá-los, por exemplo, ordenados por prioridade.

Figura 4: Exemplo de labels aplicadas em um defeito

Fonte: github.com

- *Assignee*

[13] https://github.com/dotnet/coreclr

Assignee é um contribuinte do projeto que está responsável por corrigir um certo defeito. Isso é considerado uma boa prática, pois, caso um bug similar ocorra, ou o *fix* para um certo bug esteja errado, temos a informação de quem corrigiu o bug originalmente.

Considerando isso, um bom projeto OS deve possuir um sistema robusto de *bug reporting*, mantendo um bom *backlog* de defeitos e priorizando os bugs corretamente.

2.5 Considerações Finais

Analisando o crescimento da iniciativa OS nas últimas décadas, podemos dizer que grande parte deste crescimento se deve à melhora constante das ferramentas utilizadas para colaboração e à Internet, pois assim se consegue uma comunicação dentro do time mais rápida e fácil.

Com este crescimento, a linguagem C# foi ficando para trás principalmente por ser uma linguagem que é considerada "proprietária" da Microsoft. Recentemente, a Microsoft vem tentando reverter esse pensamento tornando uma parte do .NET open source, o que está causando um efeito positivo na comunidade. Com as grandes empresas abrindo o código de alguns dos seus produtos, eles ganham usuários tanto para colaborar com o desenvolvimento deles, quanto para reportar defeitos. Esses defeitos devem ser gerenciados de forma cautelosa, visto que, como podem ser reportados por qualquer um, tendem a ser mais desorganizados do que se fossem abertos por, por exemplo, engenheiros de qualidade de software.

Após detalharmos como funciona a iniciativa open source, faz-se necessário discutir mais profundamente sobre concorrência e deadlocks, conceitos fundamentais para entendermos corretamente *bug reports* de sistemas multi-thread. Assim, o próximo capítulo deste trabalho tratará dos principais conceitos de concorrência e deadlocks, focando nos tipos de deadlocks existentes e como tratar concorrência em C#.

3. Concorrência e deadlocks

Saber como funcionam programas multi-thread é essencial para qualquer programador, mas esta não é uma tarefa fácil. Assim, este capítulo abordará primeiramente uma visão geral sobre conceitos importantes de concorrência, focando sempre em deadlocks. Após essa fundamentação teórica, será discutido os problemas que são causados pelo uso de concorrência, como condições de corrida, deadlock e *starvation*. Por fim, discutiremos mais sobre os tipos de deadlock existentes, destacando quais error de programação podem causá-los.

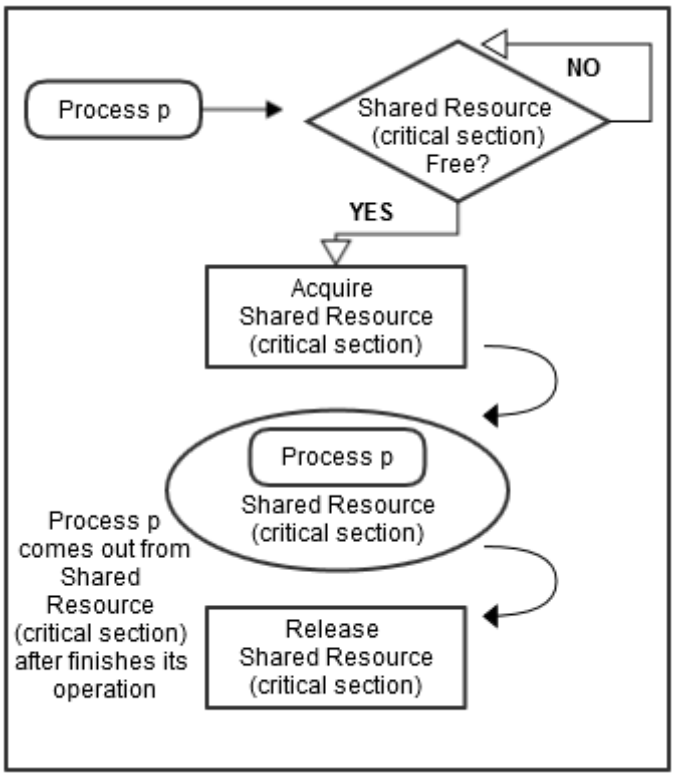
3.1 Concorrência

Com a popularização do uso de processadores multi-core, a necessidade da criação de softwares que pudessem fazer total uso desse potencial aumentou enormemente. Para suprir essa necessidade, criou-se o conceito de programação concorrente, que faz com que várias partes do programa rodem simultaneamente no processador.

Apesar de isso trazer um grande ganho de performance(ARGAWAL, 1992), também adiciona uma camada de complexidade a mais para o programador, uma vez que este deve gerenciar o acesso concorrente aos recursos do sistema. O mal gerenciamento desse acesso pode causar diversos problemas, que serão discutidos mais à frente neste capítulo.

Os programas geralmente fazem uso de *threads* para poder executar mais de uma instrução ao mesmo tempo no processador. *Threads* são "partes" de um processo que são executadas independentemente por um *scheduler*, o que quer dizer que o *scheduler* pode atribuir, por exemplo, uma *thread* para cada núcleo do processador, tornando-as completamente independentes e fazendo-as disputar por recursos do sistema(acesso a posições de memória, por exemplo). Essa disputa por recursos é controlada pelo programador, utilizando travas(ou abstrações equivalentes) que inibem acesso a um recurso enquanto ele é usado por uma *thread*. Esse processo de travamento é exemplificado na Figura 5, onde um processo "p"(contendo threads) está acessando um recurso compartilhado. Na figura, o recurso é chamado de seção crítica, justamente devido a esse controle que é feito no acesso. Podemos também perceber na figura que, para acessar o recurso, deve-se primeiro perguntar se ele está livre(nenhum outro processo/thread está acessando-o nesse momento), a fim de evitar inconsistências.

Figura 5: Diagrama ilustrando recurso compartilhado sendo acessado



Fonte: wikipedia.org

Outra parte importante do processo ilustrado na figura é a aquisição e a liberação do recurso(através da trava). Na aquisição, a *thread* adquire a trava, impedindo que outras *threads* tenham acesso ao recurso. Ao terminar de processar, a *thread* libera a trava, para que outras threads possam fazer uso do recurso.

3.2 Problemas de concorrência

Programar de maneira concorrente é comprovadamente difícil(LU et al., 2008), porque dá margem a vários tipos de problemas de sincronização. Tais problemas são de difícil detecção e reprodução, uma vez que geralmente dependem que instruções sejam executadas em uma ordem específica, e a chance de conseguir reproduzir o mesmo cenário novamente é muito pequena. Assim, podemos dividir os problemas causados pela concorrência em alguns principais, que serão discutidos a seguir.

3.2.1 Condição de corrida

Quando a exclusividade mútua do acesso a recursos compartilhados não é respeitada, um dos problemas que pode ocorrer é chamado condição de corrida. Condições de corrida ocorrem quando a corretude de um programa depende da sequência com que processos ou threads são executados. Por exemplo, consideremos a tabela abaixo.

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

A tabela representa um valor inteiro armazenado na memória que é acessado por duas threads diferentes, sendo incrementado pela Thread 1 e, em seguida, incrementado pela Thread 2. Dado que o valor inicial do inteiro é 0, ao final dos incrementos seu valor é 2. Considere agora a tabela a seguir.

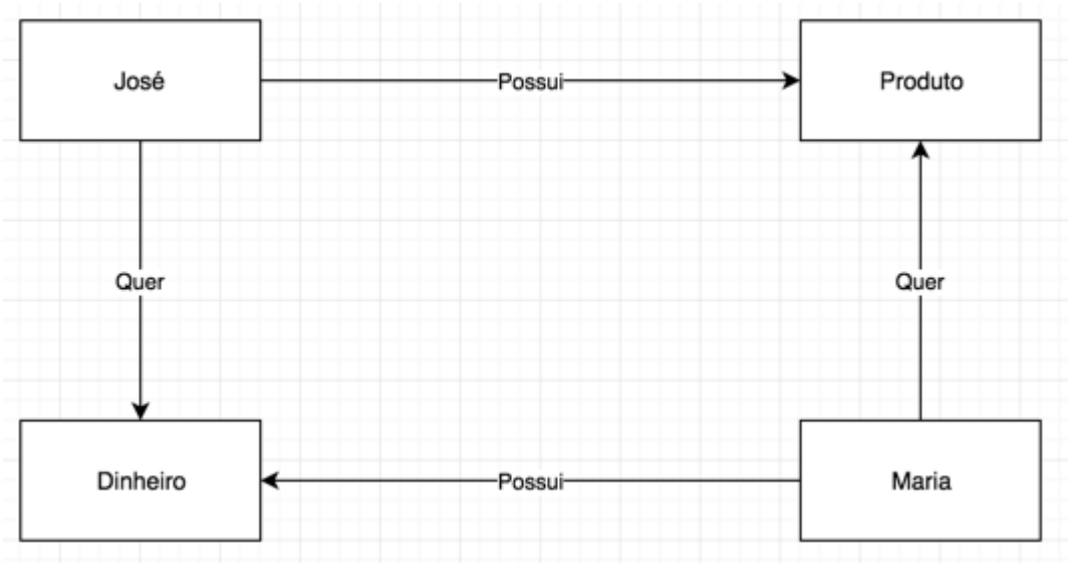
Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Esta tabela representa o que pode acontecer caso o programa dependa da sequência de execução das instruções. Neste caso, a leitura do valor é feita pela Thread 2 antes que a Thread 1 escreva o resultado do seu incremento na memória, o que causa uma inconsistência(o valor final esperado era 2, e não 1). Isso ilustra exatamente uma condição de corrida.

3.2.2 Deadlock

O deadlock é um dos principais tipos de problemas relacionados a programação concorrente. Um deadlock acontece quando existem duas ações (A e B, por exemplo) e a ação A só consegue terminar quando a ação B finalizar, e vice-versa, o que faz com que nenhuma consiga terminar. É importante notar que deadlocks não estão relacionados apenas com computação. Um exemplo do nosso cotidiano é mostrado na Figura 6 abaixo.

Figura 5: Diagrama ilustrando compra e venda de um produto



Fonte: Autor

Na figura, José quer vender um produto a Maria, e Maria quer comprar um produto de José. Porém, José só dará o produto a Maria quando ele receber o dinheiro, e Maria só efetuará o pagamento quando receber o produto de José. Dessa forma, a venda nunca acontecerá, pois eles estão em deadlock.

Existem algumas condições para que um deadlock ocorra:

- Exclusão mútua: Apenas uma thread pode acessar uma seção crítica por vez;
- Posse e espera: Cada thread pode solicitar um recurso, conseguí-lo, solicitar outro recurso e só soltar o que já possui quando conseguir o solicitado.
- Não-preempção: Recursos já alocados não podem ser tomados sem que a thread que o possui permita.
- Espera circular: Deve haver mais de uma thread, organizadas em forma de círculo, ou seja, uma thread precisa de um recurso que a próxima possui.

As três primeiras condições caracterizam um modelo de sistema necessário para ocorrência do deadlock, enquanto a última é o *deadlock* propriamente dito.

Existem diversos tipos de deadlock, porém o mais comum (LU et al., 2008)(LOBO; CASTOR, 2015) é o *two-thread, two-lock*(TTTL), que, como o nome sugere, ocorre quando há duas threads e dois locks envolvidos. Esse tipo de deadlock é geralmente causado por mau

uso de travas e pode ser detectado com certa facilidade, como feito em (LOBO; CASTOR, 2015) para a linguagem Java.

3.2.3 *Starvation*

Outro problema encontrado em concorrência é o *starvation*, que consiste em uma thread ou processo nunca ter os recursos suficientes para sua execução e, assim, nunca execute. O *starvation* pode ser causado por uma série de fatores:

- *Resource leak*

  Acontece quando alguma thread ou processo finalizou o uso de um recurso, mas este não foi liberado. Por exemplo, se dois programas tentam acessar o mesmo arquivo e um deles não libera o arquivo ao seu final, o outro software não consegue executar.

- Algoritmo de *scheduling*

  Este tipo de erro é causado quando o *scheduler* do sistema operacional não aloca tempo de processamento para a thread. Neste caso, o erro não é de programação, e sim do sistema operacional.

3.3 Considerações Finais

Após mostrar conceitos gerais sobre concorrência, este capítulo discutiu porque esse paradigma começou a ser utilizado(fazer melhor uso de processadores multi-núcleo) e as dificuldades enfrentadas pelos programadores ao lidar com diversas *threads* simultaneamente. Com essas dificuldades, alguns problemas podem acontecer, como deadlocks, condições de corrida e *starvation*, que são difíceis de ser identificados e reproduzidos, visto que eles geralmente dependem de como o *scheduler* do sistema operacional escalona as *threads* e processos. Com essa base teórica, no próximo capítulo discutiremos o estudo realizado sobre bugs relacionados a deadlock em aplicações C# open source.

4. Estudo de Bug Reports

Um dos pontos mais críticos para o sucesso de projetos OS é como os defeitos são reportados. A fim de poder identificar e corrigir os *bugs* encontrados por usuários, estes devem ser bem descritos, documentados e priorizados. Particularmente, bugs relacionados a deadlocks tendem a ser de difícil identificação e reprodução, conforme foi discutido no capítulo anterior.

Assim, este capítulo pretende apresentar o estudo feito pelo autor sobre bugs relacionados a deadlock em projetos C# open source, iniciando com uma introdução sobre bugs relacionados a deadlock, passando pela escolha das aplicações para estudo, detalhes do estudo realizado e, por fim, mostrando as conclusões obtidas.

4.1 Bugs relacionados a deadlocks

Como discutido no capítulo 3, ao programar de maneira concorrente, diversos problemas podem acontecer, sendo o deadlock um dos mais relevantes. Deadlocks podem ser facilmente confundidos com travamento ou lentidão em aplicações do mundo real (LOBO; CASTOR, 2015), o que dificulta detectar defeitos que realmente se relacionam com um deadlock de recurso.

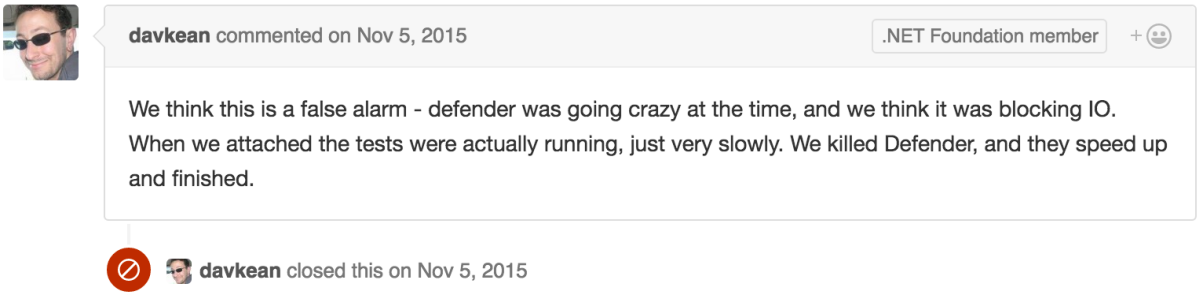
Figura 6: Exemplo de lentidão confundida com deadlock

Fonte: github.com/dotnet/roslyn/issues/6585

A Figura 6 ilustra um defeito aberto no projeto Roslyn[14](compilador open source para C#), que se deu por conta de outro processo estar bloqueando *input* e *output,* e não devido a um deadlock, como mostra o comentário da Figura 7, encontrado na discussão do defeito.

[14] https://github.com/dotnet/roslyn

Figura 7: Exemplo de lentidão confundida com deadlock



Fonte: github.com/dotnet/roslyn/issues/6585

Outro exemplo típico de *bug report* relacionado a deadlock é quando o usuário suspeita que o bug é relacionado a deadlock e, na descrição do defeito, coloca apenas o *thread dump*[15] ou a pilha de chamadas do programa. Neste caso, o usuário que tentará corrigir o bug deve possuir um conhecimento muito bom do sistema, visto que este tipo de descrição não indica especificamente que parte do código é afetada, dificultando a identificação do trecho de código falho e onde esta falha impacta.

4.2 Escolha de aplicações para estudo

Para realizar o estudo sobre bugs relacionados a deadlock, é necessário escolhermos algumas aplicações C# open source para a coleta de *bug reports*. As aplicações escolhidas são de uma gama diversificada de áreas, desde frameworks, a SGBD[16] e compiladores. A descrição de cada uma delas está a seguir.

- Firebird[17]

SGBD multiplataforma de alta performance. Escolhido devido a usar fortemente concorrência para aumentar a performance, abrindo possibilidade de muitos deadlocks.

- Robocode[18]

Uma espécie de jogo de programação em que robôs são programados para lutar entre si. Escolhido por utilizar *multi-threading* para controlar diversos robôs ao mesmo tempo.

- SharpDevelop[19]

[15] Contém informações sobre as threads que estão executando no programa.

[16] Abreviação para sistemas de gerenciamento de banco de dados

[17] https://sourceforge.net/projects/firebird/

[18] https://sourceforge.net/projects/robocode/

[19] https://github.com/icsharpcode/SharpDevelop

IDE[20] open source para programar aplicações C#. Utiliza várias threads para realizar análises sobre o código que está sendo escrito(erros de sintaxe, *warnings*, etc).

- ASP .NET MVC[21]

Framework MVC[22] para aplicações .NET. Usa várias threads para aumentar a performance.

- NuGet[23]

  Gerenciador de pacotes para .NET utilizado pelo Visual Studio[24]

- NHibernate[25]

  Biblioteca ORM[26] para C#

- Roslyn[27]

  Compilador open-source para C#

- MongoDB C# Driver[28]

  Driver do MongoDB[29] para C#

- Python Tools for Visual Studio[30]

  Extensão para Python no Visual Studio

- .NET CoreFX[31]

  Biblioteca *core* do .NET framework

- .NET CoreCLR[32]

  Runtime do .NET

---

[20] Sigla para *Integrated Development Environment*

[21] https://github.com/aspnet/Mvc

[22] Sigla para Model-View-Controller, um padrão para construção de aplicações

[23] https://github.com/NuGet/Home

[24] https://www.visualstudio.com/

[25] http://www.nhforge.org/

[26] Sigla para Object-Relational Mapping

[27] https://github.com/dotnet/roslyn

[28] https://github.com/mongodb/mongo-csharp-driver

[29] https://www.mongodb.com/

[30] https://github.com/Microsoft/PTVS

[31] https://github.com/dotnet/corefx/

[32] https://github.com/dotnet/coreclr

4.3 Análise de bugs

A fim de executar o estudo proposto neste trabalho, foram selecionados bugs relacionados a deadlock para análise, agrupando-os em categorias para análise dos resultados. Assim, dividiremos esta seção em 3 partes: critérios para escolha de bugs, onde será abordado como foi feita a seleção de bugs para análise nas diferentes ferramentas de *bug reporting,* categorização de bugs, que define as categorias nas quais os bugs serão classificados e, ao final, serão mostrados defeitos que se mostraram relevantes para as conclusões do trabalho.

4.3.1 Critérios para escolha de bugs

No início do estudo, foi necessário coletar os defeitos relacionados a deadlock nos projetos citados acima. Visto que os projetos usam diferentes ferramentas de bug reporting, foi necessário estabelecer critérios para cada uma das diferentes ferramentas. Para projetos que utilizam o GitHub para gerenciar bugs, foi utilizada a pesquisa “is:issue deadlock” que retorna todos os defeitos que possuem a palavra “deadlock" na sua descrição ou nos comentários. Já para projetos que utilizam o JIRA, foi realizada a busca por “deadlock” no campo “Contains text”, retornando apenas bugs que contém a palavra “deadlock" na sua descrição. Em projetos que utilizam o SourceForge, foi realizada a busca por “deadlock”, que, de maneira semelhante ao JIRA, retorna apenas defeitos que contém “deadlock" na sua descrição.

É importante notar que nem todos os defeitos que contém “deadlock” na descrição ou nos comentários estão diretamente relacionados com deadlock, como será discutido a seguir.

4.3.2 Categorização de bugs

Os bugs selecionados foram analisados e divididos em categorias, de maneira similar a (LOBO; CASTOR, 2015), que estão listadas a seguir.

A. Estamos confiantes de que este defeito é um deadlock. É possível identificar quantas threads estão envolvidas e como o bug ocorre.

B. O bug está indiretamente relacionado com deadlock. Há a preocupação com um possível deadlock nos comentários, ou o deadlock refere-se a outro bug. Em outras palavras, este bug não é causado por um deadlock.

C. Deadlock foi um termo usado para descrever uma lentidão ou travamento na aplicação. Isso é causado por mau entendimento dos usuários sobre o conceito de deadlock.

D. Não é um bug. Pode ser causado por mau uso do usuário ou ele foi resolvido por um *fix* para outro bug ou é apenas uma discussão, e não um bug.

E. Falta informação no *bug report* para que possamos posicioná-lo em outra categoria. Visto que não somos especialistas em nenhum dos projetos analisados, não temos essa confiança.

Para classificar bugs como categoria A eles devem conter explicitamente quais threads estão envolvidas no deadlock, como o deadlock acontece e que parte da aplicação é afetada por isso. Em alguns casos, a explicação na descrição do defeito não foi clara o suficiente, mas foi perceptível, com ajuda dos comentários do *bug*, que ele era, de fato, um deadlock, sendo este de recurso ou não(foram identificados 11 como sendo de recurso e 26 de outros tipos). Para a categoria B, foi percebido que muitos bugs estavam relacionados a deadlock, mas não eram deadlocks propriamente ditos. Neste caso, havia discussão sobre soluções que não causassem deadlock ou havia outro bug mencionado que era um deadlock, por isso o termo “deadlock" aparecia na discussão. Na categoria C, os usuários usaram o termo “deadlock” erroneamente, como sinônimo de travamento ou lentidão na aplicação e a causa do bug era outro fator que não era relacionado a multi-threading. Para o caso da categoria D, os resultados da pesquisa não eram bugs propriamente ditos, mas mencionavam deadlock. Se encaixam neste caso discussões sobre features novas e bugs que foram resolvidos com o *fix* para outro bug. Caso o bug não se encaixe nas categorias A, B, C e D, ele foi classificado como categoria E.

Os resultados da categorização estão discriminados no Apêndice A. A Tabela 4.1 contém a contagem de cada categoria.

Tabela 4.1: Quantidade de bugs por categoria

Categoria	Quantidade	Percentual
A	37(11 de recurso e 26 não)	34%
B	23	21%
C	6	5%
D	28	26%
E	15	14%
Total	109	100%

Fonte: Autor

4.3.3 Bugs relevantes

Ao longo do estudo, alguns bugs se destacaram, seja por possuir um ótimo report, ou pela discussão bem elaborada, ou pela falta de informação necessária para reprodução. Discutiremos alguns desses bugs aqui.

- Deadlock causado por usuário mudando funções da aplicação(https://github.com/NuGet/Home/issues/917)

Um bug no projeto NuGet chamou a atenção por possuir um deadlock causado pelo usuário. No bug, o usuário executa um *override*[33] numa função do sistema, causando deadlock. Isso é totalmente plausível para aplicações que desenvolvedores usam, como é o caso do NuGet, visto que o código do usuário não necessariamente respeita as regras do sistema.

- Deadlock causado por várias queries simultâneas num banco de dados relacional(https://nhibernate.jira.com/browse/NH-3375)

Um bug no NHibernate chamou a atenção por tratar de travas em banco de dados. O cenário é descrito na Figura 8.

Figura 8: Cenário de deadlock em banco de dados

1. 85 SELECT ... FROM tblOrders WITH (updlock,rowlock) INNER JOIN tblCustomerOrders ...
This gives 85 an "S" on tblCustomerOrders and an "U" on tblOrders
2. 85 UPDATE tblOrders SET Version=3 ...
This upgrade's 85's "U" lock on tblOrders to an "X" lock
3a. 89 SELECT ... FROM tblOrders WITH (updlock,rowlock) INNER JOIN tblCustomerOrders ...
This statement starts by obtaining an "S" lock on tblCustomerOrders, but then...
it's blocked trying to get a "U" lock on tblOrders (because of statement #1 above)
Meanwhile...
3b. 85 UPDATE tblCustomerOrders ...
This requires 85 to request an "X" lock on tblCustomerOrders where it previously only had a "S" lock. But it can't because statement #3 has an "S" lock on it.

Fonte: https://nhibernate.jira.com/browse/NH-3375

Na figura, o usuário que reportou o bug especifica os tipos de lock usados segundo (MICROSOFT, 2008). Quando uma thread(neste caso equivale a *query*) possui um lock do

[33] Ação de sobrescrever uma função de um sistema

tipo “S”, apenas threads que pedirem locks do tipo “S" podem ter acesso ao recurso, o que bloqueia as threads 3a e 3b, causando um deadlock.

4.4 Considerações Finais

Com o estudo realizado, pudemos chegar a algumas conclusões sobre deadlocks, principalmente sobre como são utilizados em C# e quais são os tipos de bugs relacionados a deadlocks. Conforme a categorização mostrada, podemos concluir que uma boa parcela das pessoas tem o conceito de deadlock consolidado, visto que em apenas 5% dos bugs analisados foi usado o termo deadlock para descrever um travamento ou lentidão na aplicação. Além disso, notamos uma quantidade grande de bugs indiretamente relacionados com deadlock, o que é um bom sinal, mostrando que os programadores estão preocupados com possíveis deadlocks, visto que este termo é mencionado na discussão do defeito.

Assim, neste capítulo foi dada uma introdução a bugs relacionados a deadlock, mostrando exemplos de bugs reais, além de mostrar o estudo realizado, desde a escolha dos sistemas avaliados, passando pelo processo de categorização dos defeitos. Em seguida, foram destacados bugs considerados interessantes para maior discussão, mostrando em detalhes como ocorre um bug relacionado a deadlock na prática. No próximo capítulo, iremos detalhar melhor as conclusões obtidas por este trabalho, apontando as contribuições deste para a área e dando opções de trabalhos futuros para complementá-lo.

5. Conclusão

O objetivo deste trabalho foi realizar um estudo sobre bugs relacionados a deadlocks em aplicações C#, usando projetos de médio ou grande porte como fonte de defeitos para análise e categorização.

Inicialmente, foi discutido com profundidade a área de open source, mostrando desde o surgimento da ideia até o seu estado atual. Assim, foram mostrados projetos open source de grande porte escritos em C#, como o Xamarin e o .NET, ambos da Microsoft. Foi também discutido a tendência crescente em grandes empresas de abrir o código de alguns de seus produtos, juntamente com as vantagens e desvantagens que isso pode trazer. A seguir, foi abordado como é feita a identificação e gerenciamento dos defeitos, mostrando exemplos reais de como os usuários tornam projetos open source organizados e de fácil gerenciamento, característica fundamental para seu sucesso, uma vez que o gerenciamento é totalmente distribuído entre os contribuintes.

Em seguida, foi tratado o tema de concorrência, com foco em deadlocks. Inicialmente, foi dada uma visão geral sobre concorrência, mostrando para que ela foi criada e quais os problemas que vieram com essa criação. Após essa visão geral, foram discutidos em detalhe os principais problemas causados por error de programação concorrente, como condições de corrida, deadlock e *starvation*. Focando na parte de deadlock, foi mostrado quais são as condições necessárias e suficientes para um deadlock ocorrer, ilustrando com exemplos do nosso cotidiano e depois trazendo para o contexto de computação.

5.1. Contribuições

A principal contribuição deste trabalho foi mostrar quão bem os usuários conhecem deadlock, a ponto de, num *bug report*, identificar um deadlock. Para isso, foi utilizado uma área que vem ganhando cada vez mais reconhecimento tanto por parte da comunidade de desenvolvedores como por parte de grandes empresas, que é open source.

Assim, foi realizada uma coleta de bugs relacionados a deadlock, esses bugs foram analisados um a um e categorizados em 5 tipos, representando o seu nível de relação com deadlock.

A seguir, foram tiradas conclusões sobre o estudo realizado, analisando a quantidade de bugs em cada categoria, podendo assim obter informações sobre tanto sobre os bugs de projetos open source que utilizam concorrência, quanto sobre os usuários que reportam bugs desse tipo.

Para a categoria A, concluímos que o número de bugs relacionados a deadlock que são, de fato, causados por deadlock é significativo(34%), ou seja, uma boa parte dos contribuintes sabe do que está falando quando diz que um bug é relacionado a deadlock.

Ainda nesta categoria, os defeitos foram divididos em deadlock de recurso ou não, resultando em 11 de recurso e 26 não. Isso nos indica que, em sua maioria, os deadlocks ocorrem por falta de sinalização ou mal uso das estruturas de sincronização da linguagem. Em relação à categoria B, chegamos à conclusão de que bugs indiretamente relacionados a deadlock(caso em que o termo “deadlock" foi mencionado apenas para discutir sua possibilidade) também são bastante comuns, representando 21% do total de bugs analisados. Isso indica que os desenvolvedores estão bastante preocupados com deadlocks quando desenvolvem sistemas *multi-thread*. Quanto aos defeitos em que travamentos e lentidões foram confundidos com deadlock(categoria C), eles representam um percentual muito baixo do total(5%), o que é mais um sinal de que os contribuintes estão familiarizados com os conceitos de concorrência, mais especificamente deadlock. Ao longo do estudo, apareceram muitos “bugs" que foram marcados como “*Won't fix*”, indicando que ele, na realidade, não era um defeito, e, assim, não necessitaria de correção. Defeitos desse tipo foram marcados com a categoria D, e representaram 26% do total analisado, indicando que uma boa parte dos bugs analisados não era bug, mas foram selecionados com as buscas realizadas. Isso indica que a busca por defeitos nas ferramentas de *bug report* pode ser melhorada para suportar esse tipo de *query*. Bugs que não foram categorizados em nenhuma das categorias acima(visto que não somos especialistas em nenhuma das aplicações) foram marcados com a categoria E, que representou 14% do total de defeitos analisados.

5.2. Trabalhos Futuros

Os resultados obtidos foram satisfatórios e suficientes para aplicações C# open source que utilizam fortemente *multi-threading.* Porém, algumas extensões deste trabalho podem ser realizadas para a obtenção de conclusões mais abrangentes.

Primeiramente, pode-se expandir o estudo para outras linguagens de programação(principalmente as mais populares em projetos de código aberto), como feito em (LOBO; CASTOR, 2015) para a linguagem Java. Neste caso, será necessário selecionar uma boa quantidade de projetos que possuam bons sistemas de *bug report*, para que a fonte dos dados seja confiável.

No estudo apresentado, foi tratado apenas bugs relacionados a deadlock, porém isso pode ser facilmente extendido para outros problemas que se sabe que causam muitos defeitos, como, por exemplo, condições de corrida.

Por fim, ao longo do desenvolvimento do estudo, uma dificuldade encontrada foi realizar uma busca mais precisa dos bugs, visto que as ferramentas disponibilizadas pelas ferramentas de gerenciamento de defeitos são um tanto limitadas quanto à causa do *bug*. Assim, será possível evitar que defeitos que não estejam diretamente relacionados com um certo tema sejam analisados.

Referências Bibliográficas

AGARWAL, A.. Performance tradeoffs in multithreaded processors. **Ieee Transactions On Parallel And Distributed Systems.** [s. L.], p. 525-539. set. 1992. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=159037&url=http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=159037>. Acesso em: 14 abr. 2016.

BLEKH, Aleksandr. **Why do huge profit-oriented software companies contribute to open-source software when it is akin to doing charity work which runs against their commercial instinct?** 2014. Disponível em: <http://qr.ae/14BIXo>. Acesso em: 6 jul. 2016.

OPEN SOURCE INITIATIVE. **The Open Source Definition.** Disponível em: <https://opensource.org/osd>. Acesso em: 06 jul. 2016.

OPEN SOURCE INITIATIVE. **History of the OSI.** Disponível em: <https://opensource.org/history>. Acesso em: 06 jul. 2016.

MICROSOFT. **Lock Modes** 2008. Disponível em: <https://technet.microsoft.com/en-us/library/ms175519(v=sql.105).aspx>. Acesso em: 12 jul. 2016.

MICROSOFT. **.NET Core is Open Source** 2014. Disponível em: <https://blogs.msdn.microsoft.com/dotnet/2014/11/12/net-core-is-open-source/>. Acesso em: 6 jul. 2016.

LOBO, R.; CASTOR, F. "**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 abr. 2016.

Apêndice A: Anexos do Estudo de Bug Reports

Tabela A.1: Categorização de bugs

ID	Link	Categoria
FIREBIRD-01	http://tracker.firebirdsql.org/browse/DNET-382	A(recurso)
FIREBIRD-02	http://tracker.firebirdsql.org/browse/DNET-524	A
FIREBIRD-04	http://tracker.firebirdsql.org/browse/DNET-170	A
ROBOCODE-01	https://sourceforge.net/p/robocode/bugs/1/	A(recurso)
NUGET-02	https://github.com/NuGet/Home/issues/2361	A
NUGET-03	https://github.com/NuGet/Home/issues/917	A
NUGET-05	https://github.com/NuGet/Home/issues/236	A(recurso)
NHIBER-02	https://nhibernate.jira.com/browse/NH-3375	A(recurso)
ROSLYN-03	https://github.com/dotnet/roslyn/issues/11019	A
ROSLYN-06	https://github.com/dotnet/roslyn/issues/9692	A
ROSLYN-09	https://github.com/dotnet/roslyn/issues/7825	A
ROSLYN-10	https://github.com/dotnet/roslyn/issues/7670	A
ROSLYN-11	https://github.com/dotnet/roslyn/issues/7610	A
ROSLYN-20	https://github.com/dotnet/roslyn/issues/4879	A
ROSLYN-21	https://github.com/dotnet/roslyn/issues/4858	A(recurso)
MONGO-01	https://jira.mongodb.org/browse/CSHARP-1558	A
MONGO-02	https://jira.mongodb.org/browse/CSHARP-294	A(recurso)
MONGO-03	https://jira.mongodb.org/browse/CSHARP-188	A
MONGO-04	https://jira.mongodb.org/browse/CSHARP-1133	A
MONGO-05	https://jira.mongodb.org/browse/CSHARP-406	A
MONGO-07	https://jira.mongodb.org/browse/CSHARP-454	A(recurso)
MONGO-10	https://jira.mongodb.org/browse/CSHARP-268	A(recurso)
PTVS-03	https://github.com/Microsoft/PTVS/issues/427	A
PTVS-05	https://github.com/Microsoft/PTVS/issues/49	A
PTVS-06	https://github.com/Microsoft/PTVS/issues/19	A
PTVS-08	https://github.com/Microsoft/PTVS/issues/1	A

ID	Link	Categoria
COREFX-01	https://github.com/dotnet/corefx/issues/7198	A(recurso)
COREFX-08	https://github.com/dotnet/corefx/issues/4738	A
COREFX-14	https://github.com/dotnet/corefx/issues/3137	A
COREFX-16	https://github.com/dotnet/corefx/issues/3089	A
COREFX-23	https://github.com/dotnet/corefx/issues/1734	A(recurso)
COREFX-27	https://github.com/dotnet/corefx/issues/387	A
COREFX-28	https://github.com/dotnet/corefx/issues/283	A
COREFX-30	https://github.com/dotnet/corefx/issues/202	A
CORECLR-03	https://github.com/dotnet/coreclr/issues/3912	A
CORECLR-07	https://github.com/dotnet/coreclr/issues/2093	A
CORECLR-09	https://github.com/dotnet/coreclr/issues/2021	A(recurso)
SHARP-01	https://github.com/icsharpcode/SharpDevelop/issues/30	B
NETMVC-01	https://github.com/aspnet/Mvc/issues/1884	B
NETMVC-02	https://github.com/aspnet/Mvc/issues/782	B
NHIBER-04	https://nhibernate.jira.com/browse/NH-3710	B
NHIBER-05	https://nhibernate.jira.com/browse/NH-3568	B
NHIBER-06	https://nhibernate.jira.com/browse/NH-3227	B
ROSLYN-13	https://github.com/dotnet/roslyn/issues/6394	B
ROSLYN-22	https://github.com/dotnet/roslyn/issues/3927	B
ROSLYN-25	https://github.com/dotnet/roslyn/issues/570	B
MONGO-08	https://jira.mongodb.org/browse/CSHARP-204	B
MONGO-09	https://jira.mongodb.org/browse/CSHARP-183	B
PTVS-04	https://github.com/Microsoft/PTVS/issues/97	B
COREFX-05	https://github.com/dotnet/corefx/issues/5868	B
COREFX-10	https://github.com/dotnet/corefx/issues/4429	B
COREFX-11	https://github.com/dotnet/corefx/issues/4068	B
COREFX-12	https://github.com/dotnet/corefx/issues/3562	B
COREFX-13	https://github.com/dotnet/corefx/issues/3480	B
COREFX-15	https://github.com/dotnet/corefx/issues/3114	B
COREFX-19	https://github.com/dotnet/corefx/issues/2569	B
COREFX-25	https://github.com/dotnet/corefx/issues/702	B

ID	Link	Categoria
COREFX-26	https://github.com/dotnet/corefx/issues/394	B
CORECLR-01	https://github.com/dotnet/coreclr/issues/4908	B
CORECLR-06	https://github.com/dotnet/coreclr/issues/2184	B
FIREBIRD-03	http://tracker.firebirdsql.org/browse/DNET-52	C
ROSLYN-12	https://github.com/dotnet/roslyn/issues/6585	C
ROSLYN-24	https://github.com/dotnet/roslyn/issues/900	C
PTVS-02	https://github.com/Microsoft/PTVS/issues/668	C
COREFX-03	https://github.com/dotnet/corefx/issues/6036	C
COREFX-09	https://github.com/dotnet/corefx/issues/4467	C
NHIBER-03	https://nhibernate.jira.com/browse/NH-2330	D
NHIBER-07	https://nhibernate.jira.com/browse/NH-2088	D
ROSLYN-02	https://github.com/dotnet/roslyn/issues/11116	D
ROSLYN-04	https://github.com/dotnet/roslyn/issues/10879	D
ROSLYN-05	https://github.com/dotnet/roslyn/issues/10503	D
ROSLYN-07	https://github.com/dotnet/roslyn/issues/9655	D
ROSLYN-08	https://github.com/dotnet/roslyn/issues/8458	D
ROSLYN-14	https://github.com/dotnet/roslyn/issues/6320	D
ROSLYN-15	https://github.com/dotnet/roslyn/issues/6257	D
ROSLYN-16	https://github.com/dotnet/roslyn/issues/6210	D
ROSLYN-17	https://github.com/dotnet/roslyn/issues/6166	D
ROSLYN-18	https://github.com/dotnet/roslyn/issues/6098	D
ROSLYN-19	https://github.com/dotnet/roslyn/issues/5781	D
ROSLYN-23	https://github.com/dotnet/roslyn/issues/2796	D
ROSLYN-27	https://github.com/dotnet/roslyn/issues/119	D
ROSLYN-28	https://github.com/dotnet/roslyn/issues/98	D
MONGO-06	https://jira.mongodb.org/browse/CSHARP-1457	D
PTVS-07	https://github.com/Microsoft/PTVS/issues/9	D
COREFX-04	https://github.com/dotnet/corefx/issues/6026	D
COREFX-06	https://github.com/dotnet/corefx/issues/5205	D
COREFX-07	https://github.com/dotnet/corefx/issues/5044	D
COREFX-18	https://github.com/dotnet/corefx/issues/2576	D

ID	Link	Categoria
COREFX-20	https://github.com/dotnet/corefx/issues/2454	D
COREFX-21	https://github.com/dotnet/corefx/issues/2335	D
COREFX-22	https://github.com/dotnet/corefx/issues/2329	D
CORECLR-04	https://github.com/dotnet/coreclr/issues/2952	D
CORECLR-11	https://github.com/dotnet/coreclr/issues/206	D
CORECLR-12	https://github.com/dotnet/coreclr/issues/193	D
SHARP-02	https://github.com/icsharpcode/SharpDevelop/issues/131	E
NUGET-01	https://github.com/NuGet/Home/issues/2560	E
NUGET-04	https://github.com/NuGet/Home/issues/540	E
NHIBER-01	https://nhibernate.jira.com/browse/NH-3023	E
ROSLYN-01	https://github.com/dotnet/roslyn/issues/11368	E
ROSLYN-26	https://github.com/dotnet/roslyn/issues/204	E
PTVS-01	https://github.com/Microsoft/PTVS/issues/1266	E
COREFX-02	https://github.com/dotnet/corefx/issues/6118	E
COREFX-17	https://github.com/dotnet/corefx/issues/2607	E
COREFX-24	https://github.com/dotnet/corefx/issues/1712	E
COREFX-29	https://github.com/dotnet/corefx/issues/240	E
CORECLR-02	https://github.com/dotnet/coreclr/issues/4267	E
CORECLR-05	https://github.com/dotnet/coreclr/issues/2744	E
CORECLR-08	https://github.com/dotnet/coreclr/issues/2038	E
CORECLR-10	https://github.com/dotnet/coreclr/issues/801	E