Улучшение алгоритма нахождения Deadlock в Google Thread Sanitizer

Константин Мазунин, Олег Доронин, и Андрей Дергачёв

[1] Санкт-Петербургский государственный университет информационных технологий, механики и оптики
[2] mazuninky@gmail.com

Абстракт. Данная работа посвящена исследованию алгоритмов нахождения deadlock в многопоточных приложения, а так же реализации улучшенного алгоритма Google Thread Sanitizer

Ключевые слова: Google Thread Sanitize · Multithreading · Deadlock · Multithreaded errors · Testing.

1 Введение

Взаимоблокировка или Deadlock [1] — это ситуация в многозадачной среде, при которой несколько процессов или потоков находятся в состоянии ожидания ресурсов, занятых друг другом, при этом ни один из них не может продолжать свое выполнение. Данная проблема встречается часто в многопоточных приложения и может не только снижать производительность но и приводить к полному “зависанию” всей системы в целом.

2 Существующие решения

Возможности нахождения потенциальных Deadlock существуют в таких инструментах, как Google thread sanitizer (GTSAN) [2] и Valgrind [3]. Алгоритм GTSAN базируется на построение графа занятия ресурсов потоками.

В процессе исполнения программы строится граф, вершинами которого являются мьютексы. В ходе исполнения потока строятся рёбра между последовательным захватом двух мьютексов. После построения графа производится поиск циклов, наличие которых говорит о deadlock. Однако алгоритм имеет ошибки первого и второго рода - ложное срабатывание и пропуск цели.

На Рис. 4 происходит ошибка первого рода - ложное срабатывание. Поток, который первым захватит mu1, сможет без взаимной блокировки выполнить захват и освобождение остальных мьютексов, пока второй будет ожидать освобождения mu1. В цикле присуствует граф хотя взаимной блокировки не происходит.

Рассмотрим примеры с ошибкой второго рода - пропуск цели. На рис 3 изображён один из возможных графов поиска взаимной блокировки. Код,

```
void *Thread1(void *x) {
    pthread_mutex_lock(&mu1);
    pthread_mutex_lock(&mu2);
    pthread_mutex_unlock(&mu2);
    pthread_mutex_unlock(&mu1);
}

void *Thread2(void *x) {
    pthread_mutex_lock(&mu2);
    pthread_mutex_lock(&mu1);
    pthread_mutex_unlock(&mu1);
    pthread_mutex_unlock(&mu2);
}
```

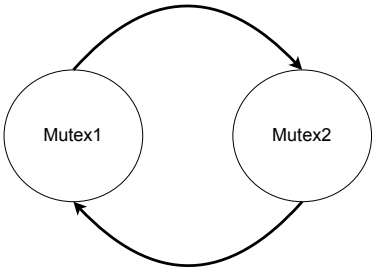

Рис. 1: Фрагмент с взаимоблокировкой на двух потоках

```
void *Thread1(void *x) {
    pthread_mutex_lock(&mu1);
    pthread_mutex_lock(&mu2);
    pthread_mutex_lock(&mu3);
    pthread_mutex_unlock(&mu3);
    pthread_mutex_unlock(&mu2);
    pthread_mutex_unlock(&mu1);
}

void *Thread2(void *x) {
    pthread_mutex_lock(&mu1);
    pthread_mutex_lock(&mu3);
    pthread_mutex_lock(&mu2);
    pthread_mutex_unlock(&mu2);
    pthread_mutex_unlock(&mu3);
    pthread_mutex_unlock(&mu1);
}
```

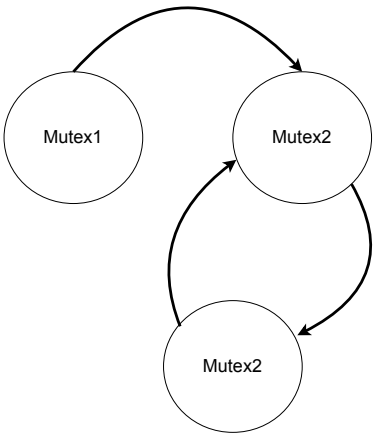

Рис. 2: Фрагмент программы с ложной взаимной блокировкой на трёх мьютексах

который может вызвать взаимную блокировку, исполняется по случайному условию. В зависимости от значения функции rand() программа попадёт в ситуацию взаимной блокировки. Данный алгоритм не может обнаружить случаи, которые происходят в возможных ветвлениях программы.

```
void *Thread1(void *x) {
    pthread_mutex_lock(&mu1);
    if(rand() > 15000) {
        pthread_mutex_lock(&mu2);
        pthread_mutex_unlock(&mu2);
    }
    pthread_mutex_unlock(&mu1);
}

void *Thread2(void *x) {
    pthread_mutex_lock(&mu2);
    pthread_mutex_lock(&mu1);
    pthread_mutex_unlock(&mu1);
    pthread_mutex_unlock(&mu2);
}
```

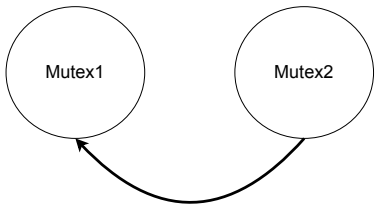

Рис. 3: Фрагмент программы с возможным пропуском возможной взаимной блокировки

На Рис. 4 происходит блокировка на одном потоке, что не обнаруживает Google thread sanitizer.

```
void *Thread1(void *x) {
    pthread_mutex_lock(&mutex);
    pthread_mutex_lock(&mutex);
}
```

Рис. 4: Фрагмент с взаимоблокировкой на одном потоке

При этом Google TSAN и Valgrind не способны обнаружить Deadlock, который произошел уже во время исполнения программы, поэтому в случае возникновения блокировки может быть неясно - это программа долго выполняется или произошёл Deadlock.

2.1 PostgreSQL

В PostgreSQL реализован алгоритм автоматического обнаружения Deadlock[4]. Рассмотрим пример выполнения двух процессов, приведенный на Рис. 5.

```
process A: BEGIN;
process B: BEGIN;
process A: UPDATE users SET name = "Peter"WHERE id = 1;
process B: UPDATE users SET name = "Marko"WHERE id = 2;
process A: UPDATE users SET name = "John"WHERE id = 2;
process B: UPDATE users SET name = "John"WHERE id = 1;
```

Рис. 5: PostgreSQL flow

В данном случае процесс A блокирует запись с идентификатором id = 1, а процесс B блокирует запись с идентификатором id = 2. После чего процесс B пытается изменить запись, которую заблокировал процесс A с id = 1, и ждёт пока процесс A завершиться. Аналогичная ситуация происходит и с процессом A, после чего процессы находятся во взаимной блокировке.

PostgreSQL распознает ситуацию, когда два процесса блокируют другу друга, ожидает в течении некоторого интервала времени, после чего выводит ошибку.

3 Цель работы

Целью данной работы является улучшение алгоритма обнаружения deadlock, используемого в Google thread sanitizer, дополнением его возможностью обнаружения взаимоблокировок в процессе выполнения программного кода, подобно алгоритму PostgreSQL. Для достижения данной цели предполагается:

1. Исследовать существующие реализации алгоритмов обнаружения Deadlock в многопоточных программах;
2. Создать концепцию алгоритма обнаружения Deadlock в режиме исполнения программы.

4 Реализация

GTSAN [5] является инструментом компилятора. Он поставляется вместе GCC и Clang. Механизм работы GTSAN состоит в том, что в ходе компиляции перед операциями с памятью вставляется вызов функции библиотеки GTSAN. В ходе реализации алгоритма было решено использовать данный подход для обнаружения взаимоблокировок в процессе выполнения программного кода.

4.1 Алгоритм детектирования

В ходе исследования был разработан алгоритма, который базируется на построение графа, пример которого приведен на Рис. 6 для двух потоков.

```
void *Thread1(void *x) {
    pthread_mutex_lock(&mu1);
    pthread_mutex_lock(&mu2);
    pthread_mutex_unlock(&mu2);
    pthread_mutex_unlock(&mu1);
}

void *Thread2(void *x) {
    pthread_mutex_lock(&mu2);
    pthread_mutex_lock(&mu1);
    pthread_mutex_unlock(&mu1);
    pthread_mutex_unlock(&mu2);
}
```

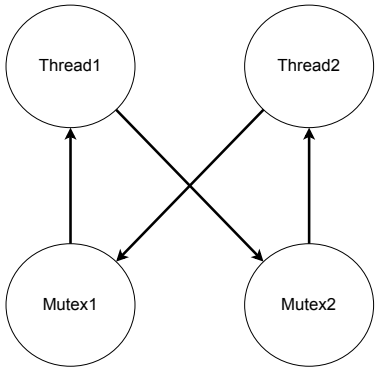

Рис. 6: Фрагмент программы с графом детектирования deadlock

T1: mu1, mu2
T2: mu2, mu1

Рис. 7: История захвата мьютекса в потоках

Разработанный алгоритм учитывает, в каком потоке захвачен мьютекс, а также какие еще потоки пытаются захватить этот же мьютекс. История захвата мьютекса представлена на Рис 7.

На Рис. 7 перед двоеточием указано имя потока, а после двоеточия - история захвата мьютексов потоком. Вершинами графа являются потоки и мьютексы. Захваченный потоком мьютекс отображается на графе как направленное ребро от вершины мьютекса к вершине потока. Попытка захвата мьютекса потоком отображается как ребро из вершины потока к вершине мьютекса. Обнаружение цикла в данном графе (как в нашем случае) и будет считаться ситуацией взаимной блокировки, о чем необходимо сообщить пользователю. Из остающихся недостатков данного алгоритма можно отметить, что данный подход применим пока лишь только для обычного mutex.

4.2 Алгоритм построения графа

В ходе работы программы необходимо отслеживать действия:

- Попытку захвата мьютекса
- Захват мьютекса
- Освобождение мьютекса

Для этих событий необходимо знать поток, который осуществляет это действие, и мьютекс, над которым происходит действие.

Алгоритму необходимо хранить список захваченных мьютексов для каждого потока captured[T,mutlist], где T - поток, а mutlist - список захваченных

мьютекс в потоке T, а так же хранить для каждого потока попытку захвата мьютекса try[T, M], где T - поток, а M - захваченный мьютекс.

Далее предстален алгоритм для обработки каждого из действий:

Попытка захвата мьютекса

function TryMutexLock(thread, mutex)
  $try[thread] = mutex$
end function

Захват мьютекса

function MutexLock(thread, mutex)
  $try[thread] = null$
  if $thread \notin captured$ **then**
    $captured[thread] = []$
  end if
  $captured[thread] = captured[thread] \cup mutex$
end function

Освобождение мьютекса

function MutexFree(thread, mutex)
  $captured[thread] = captured[thread] \backslash mutex$
  if $captured[thread] = \{\}$ **then**
    $captured = captured \backslash thread$
  end if
end function

Матрица смежности для графа строиться на основе потоков и мьютексов, которые содержаться в captured и try. Для дальнейшего использования использования DFS вершины нумеруются начиная от потоков и заканчивая мьютексами.

4.3 Алгоритм поиска пути в графе

Условием взаимной блокировки в программе является наличие пути в графе. Для алгоритма поиска пути в графе предъявляется условия:

1. Минимальна возможная асимптотика
2. Возможность получить путь, который образует цикл в графе, для дальнейшей интрепритации программой

Для реализации был выбран алгоритм DFS[6], так как он удолетворял условию возможного получения пути для получения цикла, так же его асимптотик для графа G = (V,E), где V — множество вершин графа, E — множество ребер граф, равняется O(|V|+|E|).

4.4 Оптимальный алгоритм поиска пути

В ходе работы программы приходиться перестраивать граф и производить поиск после каждого из событий: попытка захвата мьютекса, захват мьютекса, освобождение мьютекса. Чтобы не приходилось каждый раз производить поиск цикла в графе, необходимо использовать структуру, которая позволяет хранить предыдущий результат поиска циклов в графе и обновляеть её при каждой операции добавления или удаления вершины из графа.

Данная проблема была решена в работе "Real-time Constrained Cycle Detection in Large DynamicGraphs"[7], в которой представлен алгоритм поиска цикла в динамических графах, что позволяет за меньшее время находить цикл в графе, но увеличивает потребление памяти программой.

4.5 Интерпретация цикла в графе

Для конечного пользователя необходимо интепретировать путь в графе как ресурсы из-за которых произошла взаимная блокировка, то есть вывести данных о потоках и мьютекса, которые привели к данной ситуации.

Рассмотрим ситуацию с Рис. 6. Для алгоритма DFS пронумеруем сначала потоки, а потом мьюксы. Соотвественно Thread1 - вершина номер 1, Thread2 - вершина номер 2, Mutex 1 - вершина номер 3, Mutex 2 - вершина номер 4. Алгоритм поиск цикла вернёт вершины в таком порядке: [1, 4, 2, 3]. Пронумеруем элементы списка вершин в цикле от 1 до 4, то есть элементом номер 1 будет 1 вершина, а элементом под номером 4 будет 3 вершина.

Результат работы алгоритма можно интерпретировать как: элемент 1 захватил элемент 3 и пытается захватить элемент 2, пока элемент 3 захватил элемент 4 и пытается захватить элемент 3.

5 Заключение

В ходе данной работы были рассмотрены существующие алгоритмы обнаружения Deadlock, а также предложена и реализована концепция улучшенного алгоритма Google Thread Sanitizer, которая использует принцип, похожий на метод обнаружения deadlock в PostgreSQL.

Список литературы

1. Bil LewisDaniel, J. Berg. PThreads Primer - 2th Edition. BergSunSoft Press, 1996, p. 42
2. Github, https://github.com/google/sanitizers/wiki/ThreadSanitizerDeadlockDetector. Last accessed 10 Nov 2019
3. Valgrind, http://valgrind.org/docs/manual/hg-manual.html. Last accessed 10 Nov 2019
4. Shiroyasha, http://shiroyasha.io/deadlocks-in-postgresql.html. Last accessed 10 Nov 2019

5. LLVM, https://clang.llvm.org/docs/ThreadSanitizer.html. Last accessed 10 Nov 2019
6. Lecture Notes forData Structures and Algorithms,https://www.cs.bham.ac.uk/ jxb/DSA/dsa.pdf. Last accessed 10 Nov 2019
7. Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang,Xuemin Lin, and Jingren Zhou. Real-time Constrained CycleDetection in Large Dynamic Graphs.PVLDB, 11 (12): 1876-1888, 2018.DOI: https://doi.org/10.14778/3229863.3229874