I. V. Arzamartsev, G. I. Borzunov

A Method of Analysis of Multithreaded Applications Based on Symbolic Execution

Key words: symbolic execution, parallel data flow, race of the condition

This article is devoted to method of search for bugs in multithreaded applications based on the symbolic execution algorithm. This method can deterministically find race of the conditions and dead locks in multithreaded applications providing thread schedule and values of input data needed for reproducing a bug.

И. В. Арзамарцев, Г. И. Борзунов

АНАЛИЗ МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ НА ОСНОВЕ СИМВОЛЬНОГО ИСПОЛНЕНИЯ

В настоящее время различные программы используют алгоритмы параллельных вычислений, что позволяет существенно увеличить вычислительные возможности программы, а также логически разделить различные элементы функциональности программы. Однако использование параллельных вычислений часто привносит в реализацию различные ошибки:

- взаимоблокировки;
- гонки условий.

Взаимоблокировки возникают при некорректном использовании различных примитивов синхронизации, предоставляемых как самой операционной системой [1. С. 16–18], так и программными платформами [2. P. 1–2], а также при их некорректной реализации **непосредственно** в рамках программы и заключаются во взаимном удержании ресурсов, необходимых для выполнения нескольких единиц исполнения программы. Гонки условий заключаются в недостаточной синхронизации одновременного обращения к разделяемой области памяти из нескольких потоков, **что приводит** к появлению различных результатов выполнения в зависимости от планирования этих потоков. Обе ошибки имеют стохастический характер проявления в работающем приложении, поскольку на их появление влияют как входные данные, так и планирование потоков самой ОС. Существующие методы либо позволяют обнаружить данные ошибки эвристически [3, 4], либо не учитывают полноты всех функциональных возможностей реализации многопоточных программ, встречающихся на практике [5].

В данной статье предлагается новый метод анализа многопоточных приложений на наличие описанных выше ошибок параллельного программирования, основанный на алгоритме символьного исполнения [6]. Этот метод реализует анализ потоков данных, основанный на

построении алгебраических уравнений, и устанавливает взаимосвязь между входными и выходными параметрами кода исследуемой программы. При этом используется подстановка переменных величин вместо реальных значений входных данных и исполняется программа с использованием специальной переменной условия пути pc (от англ. path condition), отражающей ограничения на переменные в соответствии с выбранными условными переходами рассматриваемого пути. Для простоты изложения предположим, что при выполнении программы созданы лишь два потока, которые одновременно выполняют различные произвольные участки кода, имеющие:

- разделяемые области памяти, к которым возможно конкурентное обращение для чтения или записи информации;
- объекты синхронизации, осуществляющие синхронизацию параллельного исполнения программы посредством захвата и освобождения разделяемых областей памяти.

Предлагается в качестве модели исполнения нескольких потоков программы, представленной графом потока управления [7. P.401–403], использовать комбинированный граф параллельного исполнения программы. Этот граф включает в себя графы потоков управления G_i, $i \in N$, каждой единицы исполнения и предполагает возможность переходов между вершинами графов G_i в соответствии с планированием исполнения потоков. Рассматриваемую задачу поиска представленных выше ошибок можно свести к нахождению такого пути в комбинированном графе параллельного исполнения программы, который проходит через заданные вершины в подграфах G_i и определяет условия возможного появления ошибки:

- Конкурентный доступ к памяти с операцией записи: путь должен последовательно проходить через вершину $v \in G_j$, $j \in N$, с операцией записи и через вершину $u \in G_k$, $k \in N$ $(j \neq k)$, с произвольным действием с памятью.
- Захват необходимых для дальнейшего исполнения объектов синхронизации различными единицами исполнения; при этом путь должен проходить последовательно по вершинам, моделирующим захват объектов синхронизации и принадлежащим графам различных единиц исполнения, и далее путь не должен содержать вершины освобождения этих ресурсов.
- Захват одной единицей исполнения программы некоторого объекта синхронизации, необходимого для исполнения другой единицы исполнения, и последующее завершение работы единицы исполнения, захватившей данный ресурс. При этом путь последовательно проходит через вершину захвата объекта синхронизации одной единицей исполнения, через вершину захвата объекта синхронизации другой единицей исполнения и через вершину конца исполнения единицы исполнения, захватившей объект синхронизации.

Графически случай гонки условий при конкурентном доступе к памяти, с операцией записи, представлен ниже на рис. 1.

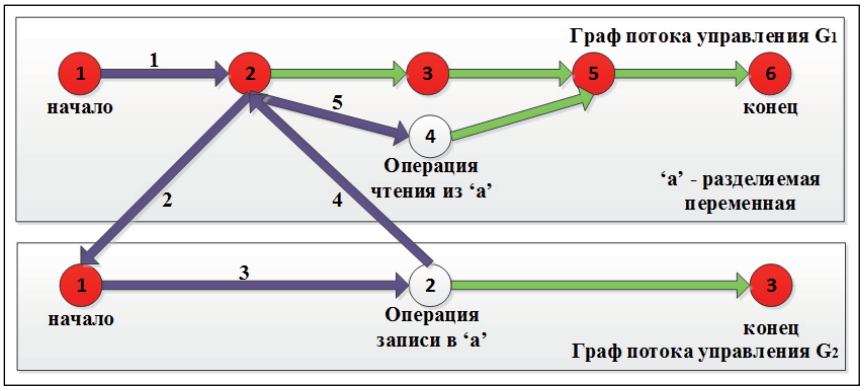

Рис. 1. Графическое представление задачи поиска пути, проходящего через вершину № 4 графа G1 и вершину № 2 графа G2, содержащие описанную ошибку гонки условий

Рис. 1 иллюстрирует возможное возникновение гонки условий в связи с наличием операции на чтение в вершине 4 графа G_1 и операции на запись в вершине 2 графа G_2. Согласно приведенным выше признакам ошибки, параллельное исполнение операций, соответствующих указанным вершинам, приведет к гонке условий. Иными словами, для выявления указанной ошибки мы должны найти такой путь в комбинированном графе $G_1 G_2$, который приведет нас из пары вершин (1,1) в пару вершин (4,2). Для использования существующих алгоритмов поиска путей в графе [8. С.284-291] введем в рассмотрение многопоточный граф управления программы MTG = (V,E) на основе параллельного исполнения потоков программы с графами потоков управления: $G_1 = (V_1,E_1)$ и $G_2 = (V_2,E_2)$. Здесь $V = V_1 Ч V_2$ – множество вершин графа MTG, а E – множество дуг этого графа. Для множества дуг E справедливо утверждение: пусть $v_{11},v_{21} \in V_1$ и $v_{12},v_{22} \in V_2$, $v_1 = (v_{11},v_{21})$, $v_2 = (v_{12},v_{22}) \in V$, e = (v_1, v_2). Тогда и только тогда $e \in E$, когда $(v_{11},v_{12}) \in E_1$ и $(v_{21},v_{22}) \in E_2$. Теперь выявление возможной ошибки сводится к задаче нахождения **в графе MTG = (V,E) пути, проходящего** через вершины $v_i \in V$, $i \in N_0$ ($N_0 = |V_1| Ч |V_2|$), которые определяются в соответствии с приведенными выше описаниями типов ошибок. Проанализируем некоторые оптимизации при работе с графом MTG. Можно сократить число рассматриваемых вершин графа MTG = (V,E) и получить граф $MTG_{opt} = (V_{opt}, E_{opt})$, оставив только те вершины $v_i \in V$, $i \in N_0$, которые соответствуют операциям с разделяемыми областями памяти или с разделяемыми объектами синхронизации. При этом множество дуг E_{opt} будет образовано следующим образом: существует дуга $e \in E_{opt}$ между вершинами v_1 и v_2 из V_{opt} тогда и только тогда, когда существует хотя бы один путь из v_1 в v_2 в исходном графе MTG. Полученный граф MTG_{opt} полностью отражает специфику совместного использования разделяемых ресурсов единиц исполнения программы, однако по сравнению с исходным графом MTG содержит меньшее число вершин, поэтому требует меньшего времени для соответствующих алгоритмов анализа. В некоторые вершины графа MTG переход невозможен в силу использования механизмов синхронизации, исключающих параллельное выполнение некоторых участков кода единиц исполнения программы, поэтому может быть выполнен следующий шаг оптимизации алгоритма:

- ввести специальную переменную условия ресурсов rc (от англ. resource condition), которая хранит все захваченные потоком ресурсы при прохождении заданного пути по аналогии с переменной path condition алгоритма символьного исполнения;
- посредством алгоритма анализа меток [9] определить переходы, условие которых зависит от разделяемой памяти;
- с помощью символьного исполнения определить условие пути pc и условие ресурсов rc при выполнении найденных на предыдущем шаге переходов в заданные вершины;
- исключить из графа MTG несовместимые вершины, у которых для всевозможных путей в них выполняется: pc1 AND pc2 = FALSE либо $rc1 \cap rc2 \neq \varnothing$;
- аналогичным образом исключить несовместимые пути.

С учетом описанных выше действий сформируем алгоритм анализа многопоточных приложений на наличие гонок условий и взаимоблокировок:

- построить графы потока управления G_1 и G_2 для рассматриваемых единиц исполнения программы, выделив операции с разделяемой памятью и объектами синхронизации в отдельные вершины;
- оптимизировать полученные графы в соответствии с рассмотренным алгоритмом для получения MTG_{opt};
- построить граф MTG_{opt}, убрав все несовместимые вершины;
- выделить множество неполных путей P_{err} в MTG_{opt}, в которых выполняется условие ошибки, если оно пустое, прекратить работу алгоритма с результатом: ошибка не обнаружена;

- найти все пути в графе MTG_{opt} из начальных вершин, проходящие через неполные пути найденного ранее множества P_{err};
- осуществить символьное исполнение, в соответствии с [9, 10, 11], найденного множества путей.

Результатом работы данного алгоритма являются входные данные и расписание планирования единиц исполнения для воспроизведения одного из вариантов ошибочной ситуации некорректного разделения ресурсов между потоками. Рассмотрим работу алгоритма на следующем примере.

Пусть есть два параллельно работающих потока, у которых имеются следующие объекты:
- param — входной параметр;
- g_criticalSection — разделяемая критическая секция, к которой применимы операции захвата EnterCriticalSection и освобождения LeaveCriticalSection соответственно;
- g_sharedInteger — разделяемая целочисленная переменная.

Пусть эти два параллельно работающих потока выполняют блок кода, представленный на рис. 2.

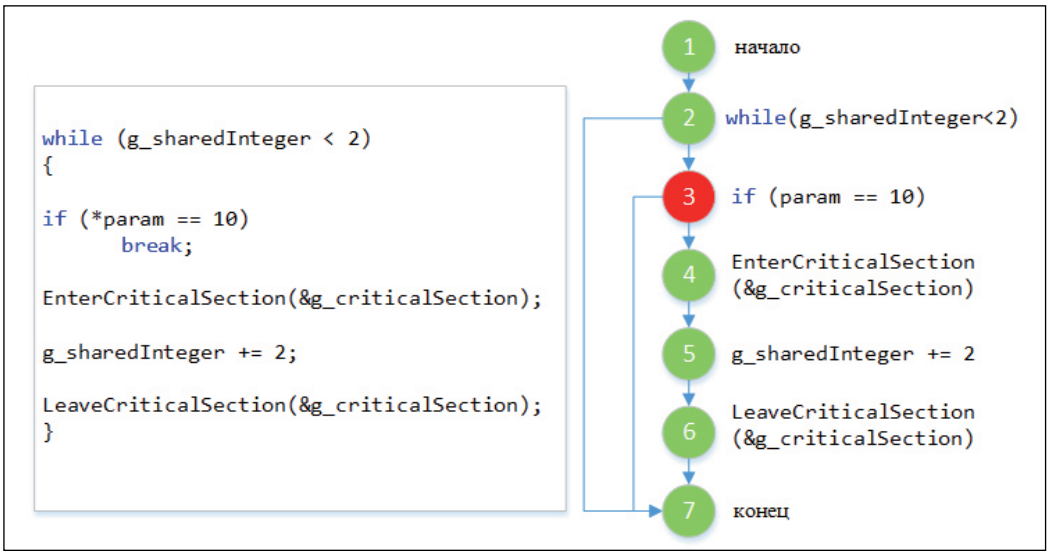

Рис. 2. Пример тела потока на языке C++, использующего API при работе с объектом синхронизации ОС Windows

Вначале осуществляется построение графа потока управления для рассмотренного блока кода, который представлен на рис. 2 справа. Далее выполняется оптимизация полученного графа, в результате которой остаются только вершины, соответствующие операциям с разделяемой памятью и с объектами синхронизации. В данном случае результирующий граф будет аналогичен графу, представленному на рис. 2, за исключением вершины 3, помеченной красным. При этом дуга между вершинами 2 и 3 преобразуется в дугу между вершинами 2 и 4. Затем необходимо осуществить построение графа MTG_{opt}. При этом из полученного графа будут отброшены вершины (5,5), (5,6), (6,5), (6,6) за счет использования механизмов синхронизации в соответствии с описанным выше алгоритмом контролирования ресурсов. В результате получим граф, схематически представленный ниже на рис. 3.

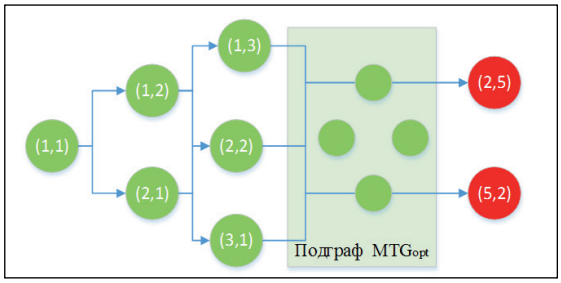

Рис. 3. Схематическое изображение графа MTG_{opt}

В соответствии с предложенным алгоритмом отыскиваются вершины, в которых возможно появление ошибки. В данном случае такими вершинами являются (2,5) и (5,2), поскольку в них присутствует конкурентный доступ к разделяемой переменной g_sharedInteger при выполнении операций чтения и записи соответственно. Затем для проверки условия появления ошибки в рамках программы мы должны осуществить поиск всевозможных путей из вершины (1,1) в (2,5) и (5,2). Примером такого пути является путь P = {(1,1) (2,1) (2,2) (2,3) (2,4) (2,5)}, который далее мы должны проверить на выполнимость алгоритмом символьного исполнения. При этом получим, что для перехода в вершину (2,5) должно быть выполнено условие: param = 10. Далее определим необходимое расписание планирования для прохождения анализируемого пути P. В целом получим, что для реализации ошибки требуется выполнение следующего условия:

- param = 10;
- планирование: (1,1,2,2,2,2,2), где i-ая координата вектора определяет номер потока, выполняющегося в i-й момент времени.

Действительно, получим гонку условий, в которой результирующее значение g_sharedInteger будет равняться либо 2, либо 4 при значении входных данных param = 10 в зависимости от выбранного алгоритма планирования, реализуемого операционной системой.

Разработанный алгоритм может быть легко обобщен для любого числа единиц исполнения программы при расширении множества вершин графа MTG посредством декартова умножения множеств вершин графов потока управления всех рассматриваемых единиц исполнения программы и применении аналогичного алгоритма нахождения множества всевозможных дуг. Однако это увеличит временную сложность алгоритма в |V`| раз, где V` — множество вершин, соответствующих операциям над разделяемой памятью в добавляемой единице исполнения программы. Поэтому при рассмотрении дополнительных единиц исполнения программы, имеющих одинаковое строение графов потока управления, что весьма часто встречается на практике, предложенный вариант расширения алгоритма может быть улучшен: если добавление единицы исполнения не приводит к увеличению числа путей с различными иными значениями разделяемых переменных и объектов синхронизации, то такие единицы исполнения можно исключить из рассмотрения.

СПИСОК ЛИТЕРАТУРЫ:

1. *Галатенко В. А.* Программирование в стандарте POSIX. Курс лекций. Учебное пособие. М.: Интернет-университет информ. Технологий, 2012.
2. *Lindholm T., Yellin F., Bracha G., Buckley A.* The Java Virtual Machine Specification. Java SE 7 Edition. URL: http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf (дата обращения: 12.10.2014).
3. *Sen K., Agha G.* Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols. Technical Report UIUCDS-R-2006-2676. UIUC. 2006.
4. *Bergan T., Grossman D., Ceze L.* Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts // ACM International Conference on Object Oriented Programming Systems Languages & Applications 2014. P. 491–506.
5. *Fonseca P., Li C., Rodrigues R.* "Finding complex concurrency bugs in large multi-threaded applications" // Proceedings of the European Conference on Computer Systems. 2011. P. 215–228.
6. *King J. C.* Symbolic execution and program testing // Magazine Communications of the ACM. CACM Homepage archive. Vol. 19. Issue 7. July 1976. P. 385–394.
7. *Aho A. V., Sethi R., Ullman J. D.* Compilers: Principles, Techniques, and Tools. B.: Pearson Education Limited, 2006.
8. *Новиков Ф. А.* Дискретная математика для программистов. СПб.: Питер, 2009.
9. *Schwartz E. J., Avgerinos Th., Brumley D.* All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) // 2010 IEEE Symposium on Security and Privacy 2014. P. 317–331.
10. *Арзамарцев И. В., Борзунов Г. И.* Детерминированный алгоритм анализа кода программного обеспечения // Безопасность информационных технологий. 2013. № 4. С. 5–9.

11. *Арзамарцев И. В., Борзунов Г. И.* Обобщенный алгоритм символьного исполнения / Материалы XXI Всероссийской научно-практической конференции «Проблемы информационной безопасности в системе высшей школы» // Безопасность информационных технологий. 2014. № 1. С. 47–48.

REFERENCES:

1. *Galatenko V. A.* Programmirovanie v standarte POSIX. Kurs lekcii. Uchebnoe posobie. M.:Internet-Universitet inform. Technologii, 2012.
2. *Lindholm T., Yellin F., Bracha G., Buckley A.* The Java Virtual Machine Specification. Java SE 7 Edition. URL: http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf
3. *Sen K., Agha G.* Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols. Technical Report UIUCDS-R-2006-2676. UIUC. 2006.
4. *Bergan T., Grossman D., Ceze L.* Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts // ACM International Conference on Object Oriented Programming Systems Languages & Applications 2014. P. 491–506.
5. *Fonseca P., Li C., Rodrigues R.* "Finding complex concurrency bugs in large multi-threaded applications" // Proceedings of the European Conference on Computer Systems. 2011. P. 215–228.
6. *King J. C.* Symbolic execution and program testing // Magazine Communications of the ACM. CACM Homepage archive. Vol. 19. Issue 7. July 1976. P. 385–394.
7. *Aho A. V., Sethi R., Ullman J. D.* Compilers: Principles, Techniques, and Tools. B.: Pearson Education Limited, 2006.
8. *Novikov F. A.* Discretnaya matematika dlya programmistov. SPb.: Piter, 2009.
9. *Schwartz E. J., Avgerinos Th., Brumley D.* All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) // 2010 IEEE Symposium on Security and Privacy 2014. P. 317–331.
10. *Arzamartsev I. V., Borzunov G. I.* Determenirovanyi algoritm analiza koda programmnogo obespechenia // Bezopasnost' informacionnyh technologii. 2013. № 4. P. 5–9.
11. *Arzamartsev I. V., Borzunov G. I.* Obobshennyi algoritm simvolnogo ispolnenia / Materialy XXI Vserossiiskoi nauchno-practicheskoi konferencii "Problemy informacionnoi bezopasnosti v sisteme vishei shkoly" // Bezopasnost' informacionnyh technologii. 2014. № 1. P. 47–48.