# УДК 004.415.53

# Метод статического поиска гонок в программах на языке Си на основе относительного множества блокировок

*Фроловский А.В., студент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,*

*кафедра «Программное обеспечение ЭВМ и информационные технологии»*

*Научный руководитель: Рудаков И.В., к.т.н,* *доцент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана*

*irudakov@bmstu.ru*

*Ключевые слова: Статический поиск гонок (static race detection), алгоритм Андерсена (Andersen algorithm), относительное множество блокировок (relative lockset)*

*Аннотация: Данная статья посвящена проблеме поиска гонок в программах на языке Си. Приводится определение состояния гонки при доступе к разделяемой памяти. Рассматривается метод статического поиска гонок на основе относительного множества блокировок. Даётся детальное описание каждого из этапов данного метода. Особое внимание уделяется нахождению перекрёстных ссылок. Выделяются проблемы данного метода.*

# Введение

Интенсивное развитие информационных технологий и расширение сферы их применения привело к значительному увеличению сложности используемого программного обеспечения, а также росту количества и критичности выполняемых им функций. С увеличением сложности возрастает количество ошибок. Ущерб от них несет существенные последствия. Одними из наиболее опасных являются ошибки, связанные с гонками при работе с данными. Они носят стохастический характер, что обуславливает сложность их выявления и исправления.

Под состоянием гонки при множественном доступе к разделяемой памяти будем понимать ситуацию, когда два или более потоков одновременно совершают доступ к разделяемой области памяти, и, по крайней мере, хотя бы один из них выполняет операцию записи в неё.

В листинге 1 показан пример программы, в которой возможно возникновение гонок при доступе к разделяемой переменной. Доступ к разделяемой переменной в функции является не защищенным ни одним из средств взаимоисключения. Это может привести к возникновению гонок при одновременном доступе к ней из различных потоков.

В листинге 2 показан пример исправленной программы из листинга 2, в которой проблема возникновения гонок при доступе к переменной устраняется посредством использования средства синхронизации — мьютекса.

Статические методы поиска гонок основаны на анализе исходного кода программы без его исполнения. Достоинством данных методов является теоретическая возможность анализа всех возможных путей выполнения программы. Недостатком является получение большого количества ложных предупреждений (обнаружение ситуаций гонок в тех местах программы, где их нет), что усложняет анализ и выявление тех мест программы, которые соответствуют действительным ситуациям гонок.

Рассмотрим метод статического поиска гонок, основанный на использовании относительного множества блокировок.

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  **void** \*foo(**void** \*arg) {  **int** \*count = arg;  **unsigned** **int** thread\_id = pthread\_self();  **while** (\*count < 10) {  printf("thread ID = %u ,count = %d\n", thread\_id, ++(\*count));  }  **return** NULL;  }  **int** main(**int** argc, **char** \*argv[]) {  pthread\_t thread1, thread2;  **int** count = 0;  pthread\_create(&thread1, NULL, &foo, &count);  pthread\_create(&thread2, NULL, &foo, &count);  pthread\_join(thread1, NULL);  pthread\_join(thread2, NULL);  **return** 0;  } |

Листинг . Пример гонки при доступе к разделяемой переменной

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  pthread\_mutex\_t lock;  **void** \*foo(**void** \*arg) {  **int** \*count = arg;  **unsigned** **int** thread\_id = pthread\_self();  **while** (\*count < 10) {  pthread\_mutex\_lock(&lock);  printf("thread ID = %u ,count = %d\n", thread\_id, \*count);  (\*count)++;  pthread\_mutex\_unlock(&lock);  }  **return** NULL;  }  **int** main(**int** argc, **char** \*argv[]) {  pthread\_t thread1, thread2;  **int** count = 0;  pthread\_mutex\_init(&lock, NULL);  pthread\_create(&thread1, NULL, &foo, &count);  pthread\_create(&thread2, NULL, &foo, &count);  pthread\_join(thread1, NULL);  pthread\_join(thread2, NULL);  pthread\_mutex\_destroy(&lock);  **return** 0;  } |

Листинг . Пример безопасного доступа к разделяемой переменной

# Метод поиска гонок на основе относительного множества блокировок

Метод основан на методе статического поиска гонок Relay [3]. В основе метода лежит понятие относительного множества блокировок. Метод состоит из четырех этапов:

1. Нахождение перекрёстных ссылок.
2. Формирование относительных множеств блокировок.
3. Формирование таблиц защищенного доступа.
4. Определение мест возможного возникновения гонок.

Рассмотрим далее каждый из этапов метода подробнее.

# Нахождение перекрёстных ссылок

Основной целью данного этапа является определение переменных, которые могут в процессе выполнения программы ссылаться на одни и те же области памяти. В основе используемого алгоритма нахождения перекрёстных ссылок лежит нечувствительный к потоку выполнения алгоритм Андерсена [1].

Процесс нахождение перекрёстных ссылок выполняется для каждой функции независимо. Изначально множества областей, на которые может ссылаться каждая переменная, полагаются пустыми. Затем выполняется итеративный процесс формирования этих множеств областей для каждой переменной. Этот процесс продолжается до тех пор, пока множества не перестанут изменяться. Анализируемые инструкции присваивания, действия, выполняемые в процессе анализа, и примеры показаны в таблице 1.

Таблица . Анализируемые инструкции присваивания

|  |  |  |
| --- | --- | --- |
| **Инструкция** | **Действия** | **Пример** |
| p = q | Множество областей, соответствующее переменной p из левой части оператора присваивания, пополняется элементами из множества, соответствующего переменной q из правой части оператора присваивания. | *До выполнения*:  PT[“p”] = {“b”, “c”}, PT[“q”] = {“a”, “b”}  pq1.emf  *После выполнения:*  PT[“p”] = {“a”, “b”, “c”}, PT[“q”] = {“a”, “b”}  pq2.emf |
| p = &q | Во множество областей, соответствующее переменной p из левой части оператора присваивания, добавляется область, соответствующая переменной q из правой части оператора присваивания. | *До выполнения:*  PT[“p”] = {“a”, “b”}, PT[“q”] = {“c”}  pq3.emf  *После выполнения:*  PT[“p”] = {“a”, “b”, “q”}, PT[“q”] = {“c”}  pq4.emf |

Продолжение табл. 1.2

|  |  |  |
| --- | --- | --- |
| p = \*q | Пусть S – множество областей, соответствующее переменной q из правой части оператора присваивания. Тогда  множество областей, соответствующее переменной p из левой части присваивания, пополняется элементами из множеств областей, соответствующих переменным из S. | *До выполнения:*  PT[“p”] = {“a”, “b”}, PT[“q”] = {“d”, “e”}, PT[“d”] = {“a”}, PT[“e”] = {“c”}  pq5.emf  *После выполнения:*  PT[“p”] = {“a”, “b”, “c”}, PT[“q”] = {“d”, “e”}, PT[“d”] = {“a”}, PT[“e”] = {“c”}  pq6.emf |
| \*p = q | Пусть S – множество областей, соответствующее переменной q из левой части оператора присваивания, T – множество областей, соответствующее переменной q из правой части оператора присваивания. Тогда во все множества, соответствующие элементам из S добавляются элементы из T. | *До выполнения:*  PT[“p”] = {“a”, “b”}, PT[“q”] = {“c”, “d”}, PT[“a”] = {“c”}, PT[“b”] = {“e”}  pq7.emf  *После выполнения:*  PT[“p”] = {“a”, “b”}, PT[“q”] = {“c”, “d”}, PT[“a”] = {“c”, “d”}, PT[“b”] = {“c”, “d”, “e”}  pq8.emf |

Окончание табл. 1.2

|  |  |  |
| --- | --- | --- |
| \*p = &q | Пусть S – множество областей, соответствующее переменной p из правой части оператора присваивания. Тогда во все множества, соответствующие элементам из S, добавляется область, соответствующая переменной q из правой части присваивания. | *До выполнения:*  PT[“p”] = {“a”, “b”}  pq9.emf  *После выполнения:*  T[“p”] = {“a”, “b”}, PT[“a”] = {“q”}, PT[“b”] = {“q”}  pq10.emf |
| \*p = \*q | Пусть S – множество областей, соответствующее переменной p из правой части присваивания, T – множество областей, соответствующее переменной q из правой части присваивания. Тогда все множества, соответствующие элементам из S, пополняются элементами из множеств, соответствующих элементам из T. | *До выполнения:*  PT[“p”] = {“a”, “b”}, PT[“q”] = {“g”, “h”}, PT[“a”] = {“c”}, PT[“b”] = {“f”}, PT[“g”] = {“c”, “d”}, PT[“h”] = {“e”}  pq11.emf  *После выполнения:*  PT[“p”] = {“a”, “b”}, PT[“q”] = {“g”, “h”}, PT[“a”] = {“c”, “d”, “e”}, PT[“b”] = {“c”, “d”, “e”, “f”}, PT[“g”] = {“c”, “d”}, PT[“h”] = {“e”}  pq12.emf |

На рис.3 представлена схема алгоритма нахождения перекрёстных ссылок для функции. На рис.4 и рис.5 представлены схемы алгоритмов, используемых в процессе нахождения перекрестных ссылок, функций eval-lhs и eval-rhs соответственно.

alias_analysis.emf

Рисунок . Схема алгоритма нахождения перекрёстных ссылок

eval-lhs.emf

Рисунок . Схема алгоритма функции eval-lhs

eval-rhs.emf

Рисунок . Схема алгоритма функции eval-rhs

Пример нахождения перекрёстных ссылок для последовательности инструкций, представленной в листинге 3, с использованием описанного алгоритма, представлен в таблице 2.

|  |
| --- |
| p1 = &a;  p2 = &b;  p1 = p2;  r = &p1;  \*r = &c;  p3 = \*r;  p2 = &d; |

Листинг . Пример демонстрации алгоритма нахождения перекрёстных ссылок

Таблица . Пример работы алгоритма нахождения перекрёстных ссылок

|  |  |  |
| --- | --- | --- |
| **Номер**  **итерации** | **Выполняемая инструкция** | **Состояние множеств областей** |
| 1 | p1 = &a | PT[“p1”] = {“a”}  alias_analysis_example_1.emf |
| 1 | p2 = &b | PT[“p1”] = {“a”}, PT[“p2”] = {“b”}  alias_analysis_example_2.emf |

Продолжение табл. 2.2

|  |  |  |
| --- | --- | --- |
| 1 | p1 = p2 | PT[“p1”] = {“a”, “b”}, PT[“p2”] = {“b”}  alias_analysis_example_3.emf |
| 1 | r = &p1 | PT[“r”] = {“p1”}, PT[“p1”] = {“a”, “b”}, PT[“p2”] = {“b”}  alias_analysis_example_4.emf |
| 1 | \*r = &c | PT[“r”] = {“p1”}, PT[“p1”] = {“a”, “b”, “c”}, PT[“p2”] = {“b”}  alias_analysis_example_5.emf |
| 1 | p3 = \*r | PT[“r”] = {“p1”}, PT[“p1”] = {“a”, “b”, “c”}, PT[“p2”] = {“b”}, PT[“p3”] = {“a”, “b”, “c”}  alias_analysis_example_6.emf |
| 1 | p2 = &d | PT[“r”] = {“p1”}, PT[“p1”] = {“a”, “b”, “c”}, PT[“p2”] = {“b”, “d”}, PT[“p3”] = {“a”, “b”, “c”}  alias_analysis_example_7.emf |

Окончание табл. 2.2

|  |  |  |
| --- | --- | --- |
| 2 | p1 = p2 | PT[“r”] = {“p1”}, PT[“p1”] = {“a”, “b”, “c”, “d”}, PT[“p2”] = {“b”, “d”}, PT[“p3”] = {“a”, “b”, “c”}  alias_analysis_example_8.emf |
| 2 | p3 = \*r | PT[“r”] = {“p1”}, PT[“p1”] = {“a”, “b”, “c”, “d”}, PT[“p2”] = {“b”, “d”}, PT[“p3”] = {“a”, “b”, “c”, “d”}  alias_analysis_example_9.emf |

# Формирование относительных множеств блокировок

На данном этапе выполняется формирование и сопоставление каждой инструкции относительного множества блокировок, которое уже сформировалось к началу её выполнения. Под относительным множеством блокировок понимается пара , где - множество обязательно захваченных блокировок, - множество освобождённых блокировок. Оно отражает изменение множества блокировок, производимое во время выполнения.

Для формирования множеств обязательно захваченных блокировок необходимо знать какие базовые блоки графа потока управления функции входят в ядро функции, т.е. встречаются на всех путях выполнения функции из графа потока управления. Ввёдем ограничение на количество раз, которое базовый блок может встретиться в пути, равное . Тогда алгоритм определения ядра функции, основанный на поиске в глубину может быть описан следующим псевдокодом, показанным на листинге 4.

|  |
| --- |
| **def** walk(v, p):  # v – текущий базовый блок  # p – базовый блок  p = p + v # добавляем блок v в путь p  # v.next() – получение списка базовых блоков, на которые  # может быть совершён переход из блока v  **if** v.next() **is** empty:  **return** set(p)  # получаем множества блоков,  # встречающихся в путях, проходящих через v  C = []  N = 0  **for** w **in** v.next():  # count(w in p) – получение количества раз,  # которое блок w встречается в пути p  **if** count(w **in** p) <= K:  C[N] = walk(w, p)  N = N + 1  # строим пересечение множеств блоков, которые  # встречаются в путях, проходящих через v  core = C[0]  **for** I = 0 to N:  **if** core **is** empty:  **break**  core = intersect(core, S[I])  **return** core  entry = <начальный базовый блок функции>  core = walk(entry, []) |

Листинг . Алгоритм определения ядра функции

Перед началом анализа функции текущее относительное множество блокировок полагается пустым. Далее производится последовательный анализ инструкций из каждого базового блока. Относительное множество блокировок может измениться в процессе анализа только после вызова какой-либо функции. Относительное множество блокировок, отражающее изменение состояния множества блокировок в процессе выполнения некоторой функции , называется обобщением относительного множества блокировок функции . Для функции захвата блокировки обобщение равно , где – захватываемая блокировка, а для функции освобождения – , где – освобождаемая блокировка. Таким образом, относительное множество блокировок во время анализа после вызова функции может быть получено с использованием функции , где – текущее относительное множество блокировок, – обобщение относительного множества блокировок вызываемой функции, в котором формальные параметры функции заменяются передаваемыми в функцию аргументами. Схема алгоритма формирования относительных множеств блокировок для функции представлена на рис.6, рис.7 и рис.8.

relative_lockset.emf

Рисунок . Алгоритм формирования относительных множеств блокировок

relative_lockset_analysis_walkBlock.emf

Рисунок . Алгоритм формирования относительных множеств блокировок

relative_lockset_analysis_analyzeBlock.emf

Рисунок . Алгоритм формирования относительных множеств блокировок

Рассмотрим пример из листинга 5. Функции и являются точками входа в потоки. В строке 4 производится операция захвата блокировки , поэтому состояние относительного множества блокировок после её исполнения становится равным . В строке 6 производится её освобождение, поэтому состояние относительного множества блокировок становится равным , и, следовательно, обобщение относительного множества блокировок для функции будет равно . В силу этого вызов функции не изменит состояний относительных множеств блокировок ни в функции , ни в функции .

|  |
| --- |
| 1. **static** **int** x =0, y = 0; 2. pthread\_mutex\_t m1, m2; 3. **void** incr(**int** \* value, pthread\_mutex\_t\* mutex) { 4. pthread\_mutex\_lock(mutex); 5. (\*value)++; 6. pthread\_mutex\_unlock(mutex); 7. } 8. **void**\* thread1(**int** \*arg) { 9. **int**\* z = &x; 10. incr(z, &m1); 11. incr(&y, &m2); 12. } 13. **void**\* thread2(**int** \*arg) { 14. incr(&x, &m1); 15. **int**\* z = &y; 16. incr(z, &m1); 17. } |

Листинг . Пример программы

# Формирование таблиц защищенного доступа

Целью данного этапа является формирование таблиц защищенного доступа для каждой функции. В каждой строке таблицы содержится структура, называемая защищенным доступом. Под защищённым доступом понимается тройка , где – lvalue [2], к которому производится доступ, – относительное множество блокировок в момент доступа к , - тип доступа (“чтение” или “запись”).

Вначале анализа таблица защищенного доступа для функции полагается пустой. Затем анализируется каждая инструкция на предмет доступа к разделяемой переменной. Если текущая анализируемая инструкция содержит доступ к разделяемой области памяти, то в таблицу добавляется соответствующая запись. В случае, когда анализируемая инструкция является вызовом функции, нужно все записи из её таблицы защищенного доступа изменить в соответствии с текущим состоянием множеств блокировок и добавить в таблицу анализируемой функции. Схема алгоритма формирования таблицы защищенного доступа для функции показана на рис. 9 и на рис.10.

guarded_access_analysis_analyzeFunction.emf

Рисунок . Схема алгоритма формирования таблиц защищенного доступа

guarded_access_analysis_updateAccessSet.emf

Рисунок . Схема алгоритма формирования таблицы защищенного доступа

Рассмотрим пример из листинга 5. В строке 5 производится доступ и на чтение, и на запись к разделяемой области, адрес которой передается в функцию через формальный параметр . Относительное множество на момент доступа равно , поэтому добавляем в таблицу доступов для функции incr 2 строки: и . Защищенный доступ для функции представлен в таблице 3. Поскольку в строке 10 в функции осуществляется вызов функции , необходимо выполнить конкретизацию таблицы 3 и добавить её строки в таблицу для функции . Аналогичные действия выполняются в строках 11, 14 и 16. Результирующие защищенные доступы для функций и показаны в таблице 4 и таблице 5 соответственно.

Таблица . Таблица защищенного доступа функции incr

|  |  |  |
| --- | --- | --- |
| **Доступ** | **Относительное множество блокировок** | **Тип доступа** |
|  |  | Чтение |
|  |  | Запись |

Таблица . Таблица защищенного доступа функции thread1

|  |  |  |
| --- | --- | --- |
| **Доступ** | **Относительное множество блокировок** | **Тип доступа** |
|  |  | чтение |
|  |  | запись |
|  |  | чтение |
|  |  | запись |

Таблица . Таблица защищённого доступа функции thread2

|  |  |  |
| --- | --- | --- |
| **Доступ** | **Относительное множество блокировок** | **Тип доступа** |
|  |  | чтение |
|  |  | запись |
|  |  | чтение |
|  |  | запись |

# Определение мест возможного возникновения гонок

На данном этапе на основе полученных на предыдущем этапе таблиц защищённого доступа производится определение мет в программе, в которых возможно возникновение гонок при доступе к разделяемым между несколькими потоками областям. Вначале производится определение точек входа в потоки. Затем осуществляется конкретизация таблиц защищённого доступа для каждой точки входа: замена формальных параметров, присутствующих в таблице, на передаваемые в поток аргументы. После этого производится перебор всех пар точек входа в потоки и сравнение соответствующих им таблиц защищённого доступа. В случае, когда в таблицах, соответствующих разным точкам входа, присутствуют доступы к одной и той же области, и при этом хотя бы один из них является доступом на запись, и пересечение множеств захваченных блокировок пусто, то данная область помечается как потенциально опасное место возникновения гонок. Схема описанного алгоритма показана на рис. 11.

generate_warnings.emf

Рисунок . Схема алгоритма поиска мест возможного возникновения гонок

Рассмотрим пример из листинга 5. Видно, что в защищенных доступах функций и , представленных в таблице 4 и таблице 5, при доступе к глобальной переменной захватываемые множества блокировок не пересекаются, и производится доступ на запись, следовательно, возможно возникновение гонки.

# Заключение

В данной работе представлен метод статического поиска гонок в программах на языке Си, основанный на концепции использования относительного множества блокировок. Рассмотрены детали каждого из этапов и приведены примеры, поясняющие выполняемые при анализе действия.

Основным достоинством метода является частичная независимость по данным анализа одной функции от анализа другой. Это даёт возможность эффективно выполнить распараллеливание анализа программы, что может существенно снизить время его выполнения.

Одним из недостатков метода является предположение о параллельном выполнении всех потоков, хотя некоторые исполняются заведомо последовательно. Также в представленном методе поиск перекрёстных ссылок внутри функции выполняется вне зависимости от пути исполнения программы, в результате чего могут появиться дополнительные ошибки 2 рода (ложные предупреждения, англ. false alarms).

# Список литературы

1. Fischer C.N. CS 701 Lecture Notes. Lecture 26. Режим доступа: http://pages.cs.wisc.edu/~fischer/cs701.f08/lectures/Lecture26.pdf (дата обращения 20.02.2014).
2. Kernighan B.W., Ritchie D.M. The C Programming Language, 2nd Edition. Prentice Hall Professional Technical Reference, 1988
3. Voung J.W., Jhala R., Lerner S. RELAY: static race detection on millions of lines of code. // Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007, p.205-214. DOI:10.1145/1287624.1287654