# 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:*

*Аннотация:*

# 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, соотвественно, 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 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 показан пример исправленной программы из листинга 1, в которой проблема возникновения гонок при доступе к переменной устраняется посредством использования средства синхронизации — мьютекса.

|  |
| --- |
| #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

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.

Нужен переход!

# Метод поиска гонок на основе относительного множества блокировок

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

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

IDEF-диаграммы метода представлены на рис. 1 и рис. 2.

idef0-black-box.emf

Рисунок Метод поиска гонок

idef0.emf

Рисунок Метод поиска гонок

Рассмотрим далее каждый из этапов подробнее.

# Нахождение перекрёстных ссылок

Основной целью данного этапа является определение переменных, которые могут в процессе выполнения программы ссылать на одни и те же области памяти.

Сказать, что бывает два типа flow-sensitive и flow-insensitive. Объясниь почему был выбран flow-insensitive… (если останется время!)

В основе используемого алгоритма нахождения перекрёстных ссылок лежит алгоритм Андерсена [].

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 меняться. Анализируемые инструкции присваивания, действия, выполняемые в процессе анализа, и примеры изменения искомых множеств 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 |
| 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 |
| \*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

Тут пример применения, если останется время.

# 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. Данное множество отражает изменение множества блокировок, производимое во время выполнения функции.

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 3.

|  |
| --- |
| **def** walk(v, p):  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

Описание самого алгоритма перед схемой, если останется время… Схема алгоритма формирования относительных множеств блокировок 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

# Формирование таблиц защищенного доступа

Бла-бла

# Определение мест возможного возникновения гонок

Бла-Бла

# Заключение

Бла-Бла