УДК 004.415.53

Статический поиск гонок в программах на языке Си

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), состояние гонки (race condition), 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. Рассматриваются методы поиска гонок в программах. Даётся краткое описание каждого из них, выделяются их достоинства и недостатки. Особое внимание уделяется группе статических методов. Объясняется общий принцип статических методов поиска гонок. Кратко рассматривается статический метод поиска гонок на основе аннотирования кода. Подробно описываются статические методы поиска гонок на основе анализа потока выполнения программы: Locksmith, CoBE, Relay. Разбираются проблемы статического поиска гонок в программах. Предлагаются возможные пути их решения.

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.

В данной работе будут рассмотрены методы статического поиска гонок, которые возникают при одновременном доступе к разделяемой памяти из нескольких потоков.

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

В Листинг 2 показан пример исправленной программы из Листинг 1. Пример гонки при доступе к разделяемой переменной, в которой проблема возникновения гонок при доступе к переменной $count$ устраняется посредством использования средства синхронизации — мьютекса.

```c
#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 1. Пример гонки при доступе к разделяемой переменной

```c
#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 2. Пример безопасного доступа к разделяемой переменной

Методы поиска гонок

Основными методами поиска гонок в программах являются:

1. формальная верификация,
2. динамические методы,
3. статические методы.

Формальная верификация основана на установлении соответствия между программой и требованиями к программе, описывающими цель разработки [2]. Основными методами формальной верификации являются метод проверки моделей и дедуктивный анализ.

Основная идея дедуктивного анализа состоит в том, чтобы последовательными преобразованиями привести программу в формулу логики (требования к программе либо изначально формулируются на языке логики, либо переводятся с какого-либо языка на язык логики). После этого доказательство корректности программы сводится к доказательству эквивалентности двух формул, что осуществляется с помощью методов, разработанных в логике. Данный метод хорошо разработан для последовательных программ, для параллельных – процесс сильно усложняется [2].

Метод проверки моделей заключается в том, что желаемые свойства поведения реагирующей системы проверяются на заданной системе (модели) путём исчерпывающего перебора всех состояний, достижимых системой, и всех поведений (путей), проходящих через эти состояния [3]. Основным недостатком данного метода является «комбинаторный взрыв» в пространстве состояний, возникающий в случае, когда исследуемая система состоит из многих компонент, переходы в которых выполняются параллельно.

Динамические методы основаны на изучении потока событий, генерируемых программой во время выполнения [1]. Недостатком данных методов является то, что

состояние гонки может быть зафиксировано, только если оно возникло в проверяемом варианте исполнения программы, а, значит, нет гарантии, что оно не может возникнуть в каком-то ином случае. Другим существенным недостатком является то, что большинство средств динамического анализа зависит от оснащения приложения средствами мониторинга, что может менять поведение исполняющей среды.

Статические методы основаны на анализе исходного кода программы. Достоинством данных методов является теоретическая возможность анализа всех возможных путей выполнения программы. Недостатком является наличие большого количества ложных предупреждений (обнаружение ситуаций гонок в тех местах программы, где их нет), что усложняет анализ и выявление тех результатов, которые соответствуют действительным ситуациям гонок. Примером такой ситуации является инициализация переменных в момент, когда программы выполняется в рамках только одного процесса или потока.

Статические методы поиска гонок

Основными методами статического поиска гонок являются:

1. аннотирование кода,
2. анализ потока выполнения программы.

Аннотирование кода выполняется за счёт добавления в исходный код программы специальных конструкций, содержащих информацию об объектах взаимоисключения, обеспечивающих защиту доступа к переменным и вызовам функций. В Листинг 3 и Листинг 4 представлены примеры разных способов аннотирования кода для контроля захвата блокировок при доступе к переменным или функциям. Недостатком данного метода является необходимость добавления большого числа аннотаций, что влечет за собой увеличение объема кода, а, следовательно, и ухудшение читаемости кода, возможность допущения ошибок при самом аннотировании. Достоинством данного метода является гарантированная проверка захвата необходимых объектов взаимоисключения, указанных в аннотации, за исключением особых случаев, например, таких, как условная блокировка и доступ к элементам массива. Примеры таких случаев продемонстрированы в Листинг 5.

```
class Account {
        final Object lock = new Object();

        /*# guarded_by lock */
        int balance = 0;

        /*# requires lock */
        void update(int n) {
```

```
                balance = n;
        }

        void deposit(int x) {
                synchronized(lock) {
                        update(balance + x);
                }
        }
}
```

Листинг 3. Пример аннотирования кода

```
Mutex CacheMutex;
Cache GlobalCache GUARDED_BY(CacheMutex);

class ScopedLookup {
public:
  ScopedLookup(Key* K) EXCLUSIVE_LOCKS_REQUIRED(CacheMutex)
    : Ky(K), Val(GlobalCache.lookup(K))
  { }
  ~ScopedLookup() EXCLUSIVE_LOCKS_REQUIRED(CacheMutex) {
    GlobalCache.release(Ky);
  }
  ...
};
```

Листинг 4. Пример аннотирования кода

```
void foo1() {
    if (threadsafe) Mu.lock();
    ...
    if (threadsafe) Mu.unlock();
}

void foo2() {
        for (int i = 0; i < 10; ++i) MutexArray[i].lock();
    ...
    for (int i = 0; i < 10; ++i) MutexArray[i].unlock();
}
```

Листинг 5. Пример аннотирования кода

Анализ потока выполнения программы производится на основе анализа последовательностей выполнения программы с целью выявления параллельно выполняющихся частей программы и переменных, одновременный доступ к которым возможен из нескольких потоков, и блокировок, защищающих доступ к ним.

Основная идея статического поиска гонок заключается в том, чтобы удостовериться, что для каждой общей области памяти существует, по крайней мере, одна блокировка, которая берётся во всех потоках при доступе к этой области. Тогда в случае, когда один поток удерживает блокировку, обеспечивается взаимоисключающий доступ к данным, что позволяет избежать гонок [6]. Для того, чтобы определить, существует ли общая блокировка, можно вычислить наборы блокировок, которые удерживаются в каждой точке программы. При этом должны отслеживаться только те блокировки, которые захватываются на всех путях, проходящих через рассматриваемую точку. Вычислив множества всегда удерживаемых блокировок для каждой точки программы,

достаточно проверить пересечение этих множеств в тех точках, где происходит доступ к исследуемой области памяти. Если пересечение не пусто, то можно сделать вывод о том, что гонки при доступе к данной области памяти отсутствуют, в противном случае существует потенциальная возможность возникновения гонок.

Чтобы применить эти базовые идеи к анализу реальных программ на C, нужно решить некоторые задачи. Прежде всего, нужно определить области памяти, к которым производится доступ, что само по себе является нетривиальной задачей. Даже без динамического выделения памяти нужно провести анализ указателей. Например, если два различных указателя, p и q, указывают на одну и ту же область памяти, то, например, доступ к полям структуры $p->data$ и $q->data$ может привести к гонкам. Операции захвата и освобождения блокировок на языке C лексически не ограничены, поэтому информацию об указателях необходимо отслеживать с учётом контекста. Рассмотрим пример, представленный в Листинг 6.

```
int x ; mutex m1 = MUTEX_INIT;
int y, z; mutex m2 = MUTEX_INIT;

void munge(int *v, mutex *m) {
    lock (m); (*v)++; unlock (m);
}

thread t1 () {
    munge (&x, &m1);
    munge (&y, &m2);
    munge (&z, &m2);
}

thread t2 () {
    munge (&x, &m1);
    munge (&y, &m1);
    munge (&z, &m2);
}
```

Листинг 6. Пример программы, содержащей гонки

Выполнение функции $munge$ зависит от контекста, в котором происходит её вызов. В функции последовательно выполняется захват блокировки, увеличение значения разделяемой переменной и освобождение блокировки. Переменная и блокировка передаются в функцию через указатели. Результат выполнения функции зависит от контекста, в котором она вызывается. К сожалению, получение информации об указателях с учётом контекста является вычислительно дорогостоящим. Анализ каждого набора значений параметров вызываемой функции на основе полного перебора приводит к комбинаторному взрыву.

Существует три метода статического поиска гонок, позволяющие выполнить анализ кода с учётом контекста выполнения:

1. Locksmith,

2. CoBE,
3. RELAY.

Locksmith

Метод основан на аннотирование программы типами и эффектами. Основной идеей является составление ограничений корреляции между доступами к областям памяти и блокировками. Для каждого доступа к области памяти p с множеством блокировок L составляется ограничение корреляции $p \rhd L$. Пусть C — множество ограничений, тогда $C \vdash p \rhd L$ показывает, что ограничение $p \rhd L$ может быть получено из ограничений в C. Множество $S(C,p) = \{L | C \vdash p \rhd L\}$ обозначает множество всех множест блокировок, которые захватываются при доступе к p. Тогда область памяти p считается защищенной блокировкой при условии, что пересечение всех множеств блокировок является непустым: $\bigcap S(C,p) \neq \emptyset$. В таком случае говорят, что данные, к которым производится доступ, постоянно коррелируют с множеством блокировок.

В основе данного метода лежит распространение информации об указателях с учётом контекста. Множество обязательно захватываемых блокировок вычисляется с учётом контекста, т.е. с учётом потока выполнения программы. Информация об указателях при этом собирается без учёта контекста (анализируются все присваивания в теле функции независимо от порядка, в котором эти присваивания могут быть выполнены).

Получение множеств блокировок с учётом контекста выполняется на основе анализа потока выполнения программы через граф потока управления. Для этого используются переменные состояния. Они позволяют использовать ограничения реализации (англ. instantiation constraints) для анализа множеств блокировок с учётом контекста и добавляют дополнительную ясность при вызове функций.

Анализ указателей в функции без учёта контекста выполняется за счёт порождения подтипов (англ. sub-typing). Идея состоит в том, что каждая область памяти имеет тип, который ассоциируется меткой области памяти p. Вне зависимости от того выполняется ли операция чтения или записи в переменную типа $ref^p(\tau)$, порождается ограничение $p \rhd L$, где L — текущее множество блокировок.

CoBE

Метод состоит из двух этапов. Вначале определяются разделяемые переменные и места, в которых к ним производится доступ. Затем определяются множества удерживаемых блокировок. Если к некоторой разделяемой переменной может осуществляться одновременный доступ из двух различных потоков и множества захваченных блокировок различаются, то возможно возникновение гонок.

Основная идея определения разделяемых переменных состоит в том, что все глобальные переменные и указатели, передаваемые в функции, рассматриваются как разделяемые. Чтобы учесть локальные ссылки на глобальные ресурсы, те указатели, которые используются при непосредственном доступе к данным, а не только служат для передачи адреса, также считаются разделяемыми. Сам по себе анализ указателей основан на идее ускоренного анализа ссылок (англ. Bootstrapping alias analysis) [4]. Его ключевая идея состоит в итеративном проведении анализа таким образом, чтобы с увеличением точности анализа сужался размер анализируемой области (принцип “разделяй и властвуй, распараллеливай и обобщай функции”). Одним из подходящих алгоритмов анализа указателей для инициализации процесса ускоренного анализа является алгоритм Стинсгарда. Ключевая идея этого алгоритма заключается в том, что в случае, когда указатель p ссылается на две различные области памяти, эти области объединяются, т.е. рассматриваются как единая абстрактная область. В результате получается разбиение указателей на некоторые классы эквивалентности [5]. Получив классы эквивалентности, можно применить более точный и дорогостоящий с вычислительной точки зрения алгоритм анализа. Например. В качестве него может быть выбран алгоритм максимально полной последовательности обновлений (англ. Maximally Complete Sequence Update). Он решает задачу анализа указателей следующим образом: два указателя p и q являются эквивалентными, т.е. ссылаться на некоторую общую область a, в случае, когда существуют цепочки присваиваний π_1 и π_2, которые семантически эквиваленты присваиваниями $p = a$ и $q = a$. Для определения эффекта от вызова функции при анализе используется процедура обобщения, которая позволяет вычислить обобщение для функции только один раз и использовать его каждый раз, когда производится вызов этой функции.

Недостатком данного метода является то, что несмотря на то, что анализ указателей производится с учётом контекста в каждой точке программы. Различные контексты в которых происходит вызов, не различаются для тела вызываемой функции. Эта проблема решается в следующем описываемом методе — Relay.

Relay

Метод основан на концепции относительного множества блокировок (англ. relative lockset) [7]. Эти множества позволяют описать изменения множеств захваченных и освобожденных блокировок относительно точки входа в функцию. Относительные множества блокировок позволяют обобщить поведение функции независимо от контекста её вызова.

При анализе функции обрабатываются изолированно друг от друга снизу вверх в графе вызовов функций. Анализ каждой функции выполняется в 3 этапа:

1. символьное исполнение,
2. анализ относительных множеств блокировок,
3. анализ защищенного доступа.

Основа для анализа относительных множеств блокировок и для анализа защищенного доступа закладывается на этапе символьного исполнения программы. Основной задачей этого этапа является выражение значений переменных функции через её формальные параметры и глобальные переменные программы. В ходе анализа для каждой инструкции в программе строится отображение вида $\Sigma: O \to V$, где O и V множества левых и правых частей операторов присваивания (англ. lvalue и rvalue соответственно), которые используются при символьном исполнении, соответственно. Обозначим через $os \in 2^O$ — множество левых частей оператора присваивания, через $x \in X$ — формальные параметры функции и глобальные переменные программы и через $p \in P$ — узлы представителей классов эквивалентности, полученные с использованием алгоритма Стинсгарда для анализа указателей. Тогда левая часть оператора присваивания $o \in O$ может иметь вид — $x|x.f|p.f|(*o).f$, а правая $v \in V$ — $\bot \mid \top \mid i \mid init(o)|must(o)|may(os)$, где $\bot$ означает "значение еще не было присвоено", $\top$ означает "любое возможное значение", i означает целочисленную константу, $init(o)$ представляет присваиваемое значение, $must(o)$ представляет значение, которое должно указывать на левую часть оператора присваивания o, $may(os)$ представляет значение, которое может указывать на любую левую часть операторов присваивания из os.

При символьном исполнении функции также следует учитывать влияние, которое могут оказывать другие потоки на состояние переменных. Для определения областей памяти, которые могут быть доступны вне потока, используется алгоритм Стинсгарда. Эти области памяти помечаются символом $\top$, означающим, что они могут иметь "любое возможное значение" после каждого вызова функции.

После завершения этапа символьного исполнения, начинается анализ относительных множеств блокировок. Относительным множеством блокировок называется пара (L_+, L_-), состоящая из $L_+ \in O$ — множества безусловно захватываемых при выполнении блокировок и $L_- \in O$ — множества блокировок, которые могут быть освобождены при выполнении. Обозначим множество всех относительных множеств блокировок как $L = 2^O \times 2^O$. Анализ множества блокировок — это анализ потока данных на решётке $(L, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$, где:

- $\perp = (O, \emptyset)$, $\top = (\emptyset, O)$;
- $\left(L_+, L_-\right) \sqsubseteq \left(L'_+, L'_-\right) \Leftrightarrow L'_+ \subseteq L_+ \wedge L_- \subseteq L'_-$;
- $\left(L_+, L_-\right) \sqcup \left(L'_+, L'_-\right) = (L_+ \cap L'_+, L_- \cup L'_-)$;
- $\left(L_+, L_-\right) \sqcap \left(L'_+, L'_-\right) = (L_+ \cup L'_+, L_- \cap L'_-)$.

Анализ выполняется снизу вверх в графе вызовов функций. После того, как функция f проанализирована, её влияние на множества блокировок сохраняется как обобщение $LockSummary(f) \in L$, представляющее относительное множество блокировок в конце функции. Обобщение функции $lock(l)$ моделируется относительным множеством блокировок $(\{l\}, \{\})$, а фукнции $unlock(l)$ – $(\{\}, \{l\})$. Учитывая вызов функции $e(a)$, для каждой функции f, которую может представлять e, функция потока сначала получает обобщение $LockSummary(f)$ и затем, используя функцию $(rebind(q, f, e) = q[formal(f) \rightarrow eval(e)]$, заменяет все вхождения формальных аргументов функции f на те, которые были переданы фактически. Результирующее обобщение представляет изменения множества блокировок, которое происходит с момента начала выполнения f и до выхода из неё. Чтобы найти относительное множество блокировок после вызова f (в каком-либо месте программы) к нему применяются изменения, задаваемые обобщением f.

После окончания анализа множества блокировок для функции выполняется анализ защищенного доступа (англ. guarded access analysis). Под защищенным доступом понимается тройка $a = (o, L, k)$, где $o \in O$ - левые части операторов присваивания, к которым производится доступ, $l \in L$ – относительное множество блокировок, $k \in K = \{Read, Write\}$ – вид доступа (чтение или запись) в точке доступа. Построение обобщения защищенных доступов для функции аналогично построению обобщения для относительных множеств блокировок за исключением того, что обходить присваивания можно в любом порядке.

На основе данных, полученных после анализа защищённого доступа, производится непосредственное определение мест возможного возникновения гонок. Последовательно просматриваются все возможные пары точек входа в потоки и анализируются их таблицы защищенного доступа. Если в таблицах для двух различных потоков найдется пара строк с одинаковыми значениями левых частей операторов присваивания, и хотя бы в одной из строк указан доступ “запись”, такая ситуация свидетельствует о возможном наличии гонки. В результате анализа таблиц появляется большое количество ложных угроз, поэтому необходимо среди всех потенциально опасных операций доступа к памяти выявить те, которые представляют реальную угрозу.

Заключение

Контекстная зависимость является не единственной проблемой, возникающей при статическом поиске гонок в программах на языке C. Во-первых, помимо системных мьютексов существуют и другие способы обеспечения монопольного доступа к данным, например, семафоры. Несмотря на то, что поведение семафоров может быть проэмулировано с использованием мьютексов, это может привести к возникновению новых ложных предупреждений. Также при анализе указателей возникают проблемы, связанные с динамическим выделением памяти. И наконец, разрешение проблем, связанных с условными блокировками, также является весьма нетривиальной задачей анализа.

Самым надёжным способом обеспечения отсутствия гонок является исполнение только одного потока. Даже в многопоточной программе поток может не всегда выполняться параллельно с другими потоками. Существует много механизмов обеспечения синхронизации без блокировок (англ. lock-free). Например, предположим, что есть главный поток и k рабочих потоков (англ. workers). Главный поток содержит массив A с k элементами (по одному для каждого из потоков) такой, что $A[i]$ влияет на i поток. Более того, предположим, что основной поток инициализирует этот массив перед порождением рабочих потоков (англ. worker threads) и обрабатывает этот массив после завершения всех потоков. Хотя нет блокировок, программа не содержит гонок, т.к. главный поток осуществляет доступ к массиву только тогда, когда рабочие потоки не имеют, и рабочие потоки следуют этому соглашению, что обеспечивает монопольный доступ.

Обычно выделяют временные фазы работы программы такие, как инициализация, обработка и постобработка. Анализатор, имея информацию о том, к каким ресурсам в какие моменты времени производится доступ, должен уметь определять, какие из выполняющихся потоков действительно конфликтуют. Традиционный подход заключается в попытке частично упорядочить инструкции по последовательности исполнения, когда это возможно. Гонка может произойти при одновременном доступе только в том случае, когда нет ограничений последовательности выполнения.

Список литературы

1. Ковега Д.Н., Крищенко В.А. Использование системы LLVM при динамическом поиске состояний гонок в программах. // Инженерный журнал: наука и инновации. Электрон. журн. 2013. №2(14). Режим доступа: http://engjournal.ru/catalog/it/hidden/549.html (дата обращения 20.02.2014).

2. Кропачева М.С. Формальная верификация параллельных программ. // Исследования наукограда. 2012. №2(2). С.35-38.
3. Пелед Д., Грамберг О., Кларк Э.М. Верификация моделей программ: Пер. с англ. / под ред. Смелянского Р. М.: Московский центр непрерывного математического образования, 2002. 416 с. [Edmund M. Clarke, Orna Grumberg, Doron Peled. Model Checking. Cambridge: The MIT Press, 1999.].
4. Kahlon V. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. // Proceeding of the Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation. Tucson 2008. P.249-259. DOI:10.1145/1375581.1375613
5. Steensgaard B. Points-to analysis in almost linear time. // Proceeding of the POPL '96 Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. St. Petersburg Beach 1996. P.32-41. DOI:10.1145/237721.237727
6. Vojdani V. Static Data Race Analysis of Heap-Manipulating C Programs: PhD. Tartu: University of Tartu, 2010. 131 p.
7. 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. Dubrovnik 2007. P.205-214. DOI:10.1145/1287624.1287654