

## Реферат

Бла-бла

## Содержание

Введение . . . . .	4
1 Аналитический раздел . . . . .	5
1.1 Понятие ситуации гонки . . . . .	5
1.2 Существующие методы поиска гонок . . . . .	6
1.3 Статические методы поиска гонок . . . . .	7
1.3.1 Locksmith . . . . .	12
1.3.2 CoBE . . . . .	12
1.3.3 Relay . . . . .	13
1.4 Выводы . . . . .	15
2 Конструкторский раздел . . . . .	16
2.1 Статический метод поиска гонок на основе относительного множе- ства блокировок . . . . .	16
2.2 Построение ГПУ функций . . . . .	16
2.3 Построение путей выполнения функций . . . . .	16
2.4 Построение таблицы защищенного доступа для потоков . . . . .	17
2.5 Определение состояния перекрёстных ссылок . . . . .	18
2.6 Вычисление относительного множества блокировок и его обобщения	19
2.7 Построение таблиц защищенного доступа . . . . .	20
2.8 Определение мест возможного возникновения гонок . . . . .	21
2.9 Выводы . . . . .	21
3 Технологический раздел . . . . .	25
3.1 Выбор формы представления программы . . . . .	25
3.1.1 Исходный код программы . . . . .	25
3.1.2 Промежуточное представление . . . . .	25
3.1.3 Код целевой машины . . . . .	25
3.2 Выбор языка промежуточного представления . . . . .	25
3.3 Выбор языка. Используемые библиотеки . . . . .	26
3.4 Про плагины для гсс (придумай как назвать . . . . .	26
3.5 Ограничения реализации . . . . .	26
3.6 Структура ПО . . . . .	26
3.7 Запуск программы. Формат выходных сообщений . . . . .	27
3.8 Выводы . . . . .	28
4 Исследовательский раздел . . . . .	30
4.1 Условия проведения экспериментов . . . . .	30
4.2 Исследование скоростных характеристик . . . . .	30
4.3 Исследование точности . . . . .	32
4.3.1 Программа 1 . . . . .	35

4.3.2	Программа 2 . . . . .	36
4.3.3	Программа 3 . . . . .	38
4.4	Программа 4 . . . . .	38
4.5	Выводы . . . . .	39
	Заключение . . . . .	41
	Список использованных источников . . . . .	42

## 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.

Существует три основные группы методов поиска гонок в программах: формальная верификация, динамические и статические методы. Каждая из групп методов имеет свою область применения. Программное обеспечение, реализующие статический поиск гонок, является удобным инструментом, позволяющим преждевременно еще на этапе разработки выявлять и устранять дефекты программ, связанные с возможными возникновения гонок при доступе к разделяемым ресурсам.

Целью работы является разработка статического метода поиска гонок в программах на языке Си. Для достижения поставленной цели необходимо решить следующие задачи:

- выполнить анализ методов поиска гонок в программах, выявить их достоинства и недостатки;
- разработать метод статического поиска гонок при доступе к разделяемой памяти;
- разработать алгоритмы, входящие в состав предложенного метода;
- разработать ПО, реализующее предложенный метод;
- провести исследование разработанного метода.

Данная пояснительная записка имеет следующую структуру:

- В главе 1 рассмотрены существующие методы поиска гонок при доступе к разделяемой памяти.
- В главе 2 приведено описание разработанного метода поиска гонок в программах на языке Си и используемых алгоритмов.
- В главе 3 рассмотрено проектирование программного обеспечения, реализующего разработанный метод.
- В главе 4 приведено описание экспериментов, проводимых с разработанным программным обеспечением, и их результаты.

## 1 Аналитический раздел

В данном разделе приводится понятие ситуации гонки, рассматривается классификация существующих методов поиска гонок в программах, выявляются достоинства и недостатки каждого из методов. Раздел содержит описание существующих методов статического поиска гонок в программах, описание основных идей, лежащих в их основе, рассматриваются современные проблемы статического поиска гонок.

### 1.1 Понятие ситуации гонки

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 1.1 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 взаимного исключения. Это может привести к возникновению гонок при одновременном доступе к ней из различных потоков. В листинге 1.2 показан пример исправленной программы из листинга 1.1. Гонки при доступе к разделяемой переменной `count` в нём устраняются посредством использования средства взаимного исключения — мьютекса.

Листинг 1.1 — Пример возникновения гонки при доступе к разделяемой переменной (`race.c`)

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 void *foo(void *arg) {
5     int *count = arg;
6     unsigned int thread_id = pthread_self();
7     while (*count < 10) {
8         printf("thread ID = %u ,count = %d\n", thread_id, ++(*count));
9     }
10    return NULL;
11 }
12
13 int main(int argc, char *argv[]) {
14     pthread_t thread1, thread2;
15     int count = 0;
16
17     pthread_create(&thread1, NULL, &foo, &count);
18     pthread_create(&thread2, NULL, &foo, &count);
19
20     pthread_join(thread1, NULL);
```

```

21     pthread_join(thread2, NULL);
22
23     return 0;
24 }

```

Листинг 1.2 — Пример защищенного доступа к разделяемой переменной (**protected.c**)

```

1  #include <stdio.h>
2  #include <pthread.h>
3
4  pthread_mutex_t lock;
5
6  void *foo(void *arg) {
7      int *count = arg;
8      unsigned int thread_id = pthread_self();
9      while (*count < 10) {
10         pthread_mutex_lock(&lock);
11         printf("thread ID = %u ,count = %d\n", thread_id, *count);
12         (*count)++;
13         pthread_mutex_unlock(&lock);
14     }
15     return NULL;
16 }
17
18 int main(int argc, char *argv[]) {
19     pthread_t thread1, thread2;
20     int count = 0;
21
22     pthread_mutex_init(&lock, NULL);
23
24     pthread_create(&thread1, NULL, &foo, &count);
25     pthread_create(&thread2, NULL, &foo, &count);
26
27     pthread_join(thread1, NULL);
28     pthread_join(thread2, NULL);
29
30     pthread_mutex_destroy(&lock);
31
32     return 0;
33 }

```

## 1.2 Существующие методы поиска гонок

Основными методами поиска гонок в программах являются:

— формальная верификация,

- динамические методы,
- статические методы.

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

Основная идея дедуктивного анализа состоит в том, что программа последовательными преобразованиями приводится 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 этих двух формул, что осуществляется с помощью методов, разработанных в логике. Данный метод хорошо разработан для последовательных программ, для параллельных – процесс сильно усложняется ]].

Метод проверки моделей основан на 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 ]]. 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 ]]. 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 ошибок второго рода, обнаружения 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 программа выполняется в рамках только одного процесса или потока.

### **1.3 Статические методы поиска гонок**

Основными методами статического поиска гонок являются:

- аннотирование кода,
- анализ потока выполнения программы.

Аннотирование кода выполняется за счёт добавления в исходный код программы специальных конструкций, содержащих информацию об объектах взаимного исключения, обеспечивающих защиту доступа к переменным и вызовам функций. В листинге 1.3 и листинге 1.4 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 его читаемости. Кроме того не исключена возможность допущения ошибок при самом аннотировании. Достоинством данного метода является гарантированная проверка захвата необходимых объектов взаимного исключения, указанных в аннотации, за исключением особых случаев, например, таких, как захват блокировки в условии (условная блокировка) и доступ к элементам массива. Примеры таких случаев продемонстрированы в листинге 1.5.

Листинг 1.3 — Пример аннотирования кода (**annotated1.java**)

```

1 class Account {
2     final Object lock = new Object();
3
4     /*# guarded_by lock */
5     int balance = 0;
6
7     /*# requires lock */
8     void update(int n) {
9         balance = n;
10    }
11
12    void deposit(int x) {
13        synchronized(lock) {
14            update(balance + x);
15        }
16    }
17 }
```

Листинг 1.4 — Пример аннотирования кода (**annotated2.cpp**)

```

1 Mutex CacheMutex;
2 Cache GlobalCache GUARDED_BY(CacheMutex);
3
4 class ScopedLookup {
5 public:
6     ScopedLookup(Key* K) EXCLUSIVE_LOCKS_REQUIRED(CacheMutex)
7         : Ky(K), Val(GlobalCache.lookup(K))
```



```

8   { }
9   ~ScopedLookup() EXCLUSIVE_LOCKS_REQUIRED(CacheMutex) {
10      GlobalCache.release(Ky);
11   }
12   ...
13 };

```

Листинг 1.5 — Пример аннотирования кода (**annotated3.cpp**)

```

1  void foo1() {
2      if (threadsafe) Mu.lock();
3      ...
4      if (threadsafe) Mu.unlock();
5  }
6
7  void foo2() {
8      for (int i = 0; i < 10; ++i) MutexArray[i].lock();
9      ...
10     for (int i = 0; i < 10; ++i) MutexArray[i].unlock();
11 }

```

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  $\parallel$ . 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 определение перекрестных

ссылок. Например, если два различных указателя,  $p$  и  $q$ , указывают на одну и ту же область памяти, то, например, доступ к полям структуры  $p \rightarrow data$  и  $q \rightarrow data$  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 листинге 1.6. Выполнение функции зависит от контекста, в котором происходит её вызов. В функции последовательно выполняется захват блокировки, увеличение значения переменной и освобождение блокировки. Переменная и блокировка передаются в функцию через указатели. 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.

Листинг 1.6 — Пример аннотирования кода (**munge.c**)

```
1 #include <pthread.h>
2
3 int x, y, z;
4 pthread_mutex_t m1, m2;
5
6 void munge(int *v, mutex *m) {
7     pthread_mutex_lock(m);
8     (*v)++;
9     pthread_mutex_unlock(m);
10 }
11
12 void* thread1(void* args) {
13     munge(&x, &m1);
14     munge(&y, &m2);
15     munge(&z, &m2);
16 }
17
18 void* thread2(void* args) {
19     munge(&x, &m1);
20     munge(&y, &m1);
21     munge(&z, &m2);
22 }
23
24 int main(int argc, char** argv) {
25     pthread_t threads[2];
26
27     pthread_mutex_init(&m1, NULL);
28     pthread_mutex_init(&m2, NULL);
29
30     pthread_create(&threads[0], NULL, thread1, NULL);
31     pthread_create(&threads[1], NULL, thread2, NULL);
```

```

32
33     pthread_join(threads[1], NULL);
34     pthread_join(threads[2], NULL);
35
36     pthread_mutex_destroy(&m1);
37     pthread_mutex_destroy(&m2);
38     return 0;
39 }

```

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

- Locksmith,
- CoBE,
- RELAY.

Однако 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. Это связано с тем, что 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  $n$  рабочих потоков (англ. workers). Главный поток содержит массив  $A$  с  $n$  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 его не имеют; рабочие потоки следуют этому соглашению, что обеспечивает монопольный доступ. Поэтому для решения этой проблемы 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. Тради-

ционный 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.

### 1.3.1 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  $p \triangleright L$ . Пусть  $C$  — множество ограничений, тогда  $C \vdash p \triangleright L$  показывает, что ограничение  $p \triangleright L$  может быть получено из ограничений в  $C$ . Множество  $S(C, p) = \{L | C \vdash p \triangleright L\}$  обозначает множество всех множеств блокировок, которые захватываются при доступе к области  $p$ . Тогда область памяти  $p$  считается защищенной блокировкой при условии, что пересечение всех множеств блокировок является непустым:  $\cap S(C, p) \neq \emptyset$ . 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  $p \triangleright L$ , где  $L$  — текущее множество блокировок.

### 1.3.2 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 осу-

ществляться 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) [1]. Его ключевая идея состоит в итеративном проведении процесса 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 Стингарда. 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 [1]. 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. ссылаются на некоторую общую область  $a$ , в случае, когда существуют цепочки присваиваний  $\pi_1$  и  $\pi_2$ , которые семантически эквивалентны 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.

### 1.3.3 Relay

Метод основан на концепции относительного множества блокировок (англ. relative lockset) [1]. 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:

- символьное исполнение,
- анализ относительных множеств блокировок,
- 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  $\sum : O \rightarrow V$ , 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 Стингарда для анализа указателей. Тогда левая часть оператора присваивания  $o \in O$  может иметь вид  $x|x.f|p.f|(*o).f$ , а правая  $v \in V$  —  $\perp|T|i|init(o)|must(o)|may(os)$ , где  $\perp$  означает "значение еще не было присвоено",  $T$  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 Стингарда. Эти области памяти помечаются символом  $T$ , 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  $(L, \perp, \top, \sqsubseteq, \sqcup, \sqcap)$ , где:

- $\perp = (O, \emptyset)$ ,  $\top = (\emptyset, O)$
- $(L_+, L_-) \sqsubseteq (L'_+, L'_-) \iff L'_+ \subseteq L_+ \wedge L_- \subseteq L'_-$

- $(L_+, L_-) \sqcup (L'_+, L'_-) = (L_+ \cap L'_+, L_- \cup L'_-)$
- $(L_+, L_-) \sqcap (L'_+, L'_-) = (L_+ \cup L'_+, L_- \cap L'_-)$

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 функции  $unlock(l)$  –  $(, l)$ . Учитывая вызов функции  $e(a)$ , для каждой функции  $f$ , которую может представлять  $e$ , функция потока сначала получает обобщение  $LockSummary(f)$  и затем, используя функцию  $rebind$  ( $rebind(q, f, e) = q[formal(f) \rightarrow eval(e)]$ ), 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,  $L \in \mathbf{L}$  – 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 на 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.

#### 1.4 Выводы

Еще не придумал. Стоит сделать раздел с описанием того, что из себя представляет граф потока управления? В [1] указано, что... – для того, чтобы сборка не ломалась.

## 2 Конструкторский раздел

В данном разделе приводится описание разработанного метода и его ограничения (не знаю, где их разместить. может в выводах?), детально рассматриваются применяемые на каждом из этапов алгоритмы. ...

### 2.1 Статический метод поиска гонок на основе относительного множества блокировок

Разработанный метод статического поиска гонок в программах на языке Си основан на методе статического поиска гонок Relay[]. В основе метода лежит понятие относительного множества блокировок. Метод состоит из 4 этапов:

- построение ГПУ функций,
- построение путей выполнения функций,
- построение таблиц защищённого доступа для потоков,
- определение мест возможного возникновения гонок.

Схема метода представлена на рис. 2.1 и рис. 2.2. На вход метода подаётся исходный код программы. Выходом является список мест возможного возникновения гонок. Рассмотрим далее каждый из этапов метода подробнее.

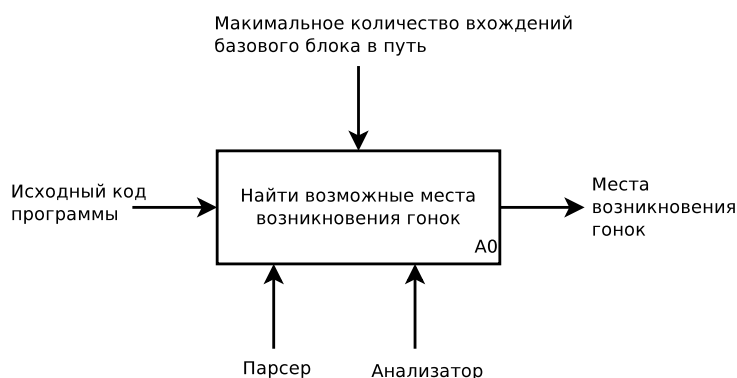


Рисунок 2.1 — Статический метод поиска гонок на основе относительного множества блокировок

### 2.2 Построение ГПУ функций

нужно?

### 2.3 Построение путей выполнения функций

При построении списка анализируемых путей выполнения функции используется алгоритм, основанный на обходе графа в глубину. Для того, чтобы избежать появления путей бесконечной длины, которые могут появляться в случае наличия



циклов в анализируемом графе потока управления, вводится ограничение  $K$  на максимальное количество вхождений каждого базового блока в анализируемый путь.

Схема рекурсивного алгоритма обхода графа для построения списка анализируемых путей представлена на рис. 2.3. При построении путей используется функция **walk**, которая принимает на вход текущий базовый блок  $v$ , пройденный путь  $p$  и граф потока управления анализируемой функции  $G$ . Вначале функции **walk** текущая вершина  $v$  добавляется в путь  $p$ , множество построенных путей  $S$  полагается пустым. Затем проверяется является ли вершина  $p$  заключительной. Если является, то функция возвращает множество построенных путей, включающее в себя только один путь  $p$ . Иначе - производится рекурсивный запуск функции **walk** для всех вершин  $w$  таких, что существует дуга, началом которой является вершина  $v$ , а концом - вершина  $w$ , и вершина  $w$  встречается в пути  $p$  не более  $K$  раз. Возвращаемые при этом множества включаются в множество  $S$ . Если вершина  $w$  встречается в пути более  $K$  раз, то к пути  $p$  добавляется заключительная вершина графа потока управления анализируемой функции, и полученный путь добавляется также к  $S$ . Таким образом для получения множества анализируемых путей функции необходимо подать на вход функции **walk** начальную вершину графа потока управления функции, пустой путь и непосредственно сам анализируемый граф потока управления функции.

## 2.4 Построение таблицы защищенного доступа для потоков

Построение таблиц защищённого доступа для потоков состоит из 5 этапов:

- определение состояния перекрестных ссылок для всех инструкций путей,
- определения относительных множеств блокировок для всех инструкций путей,
- определение обобщения относительного множества блокировок для функций,
- построение таблиц защищенного доступа для функций,
- конкретизация таблиц защищённого доступа для всех точек входа в потоки.

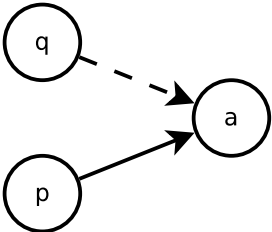
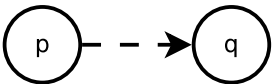
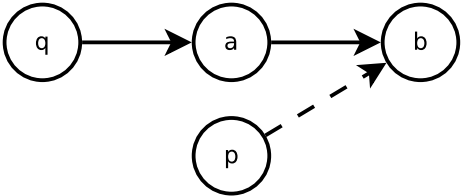
Сначала выполняется символьное исполнение каждого из путей для определения изменения состояний перекрестных ссылок во время выполнения каждого пути. Затем на основе этого выполняется вычисление относительных блокировок для каждой инструкции на каждом из путей. После анализа всех путей выполнения какой-либо функции для неё строится обобщение относительного множества блокировок, которое используется при анализе других функций, в которых производится её вызов. На основе путей, размеченных состояниями перекрёстных ссылок и относительными множествами блокировок, строятся таблицы защищенного доступа для всех функций. После чего ищутся все места создания потоков. Для каждого из них на основе имеющихся таблиц защищенного доступа для функций и известных состояний перекрестных ссылок на момент создания производится конкретизация уже

имеющихся таблиц для функций с учётом передаваемых при инициализации потоков значений. Схема построения таблиц защищённого доступа для потоков представлена на рис. 2.4. Рассмотрим далее каждый из этапов подробнее.

## 2.5 Определение состояния перекрёстных ссылок

Для определения состояния перекрёстных ссылок для каждой инструкции на каждом анализируемом пути используется символьное исполнение. Анализируемые инструкции присваивания, производимые действия и примеры приведены в таблице 2.1.

Таблица 2.1 — Анализируемые инструкции присваивания

Инструкция	Действия	Пример
$p = q$	Пусть переменная $q$ из правой части оператора присваивания ссылается на некоторую переменную $a$ . Тогда после выполнения оператора присваивания переменная $p$ из левой части оператора присваивания будет ссылаться на переменную $a$ .	
$p = \&q$	После выполнения оператора присваивания переменная $p$ из левой части оператора присваивания будет ссылаться на переменную $q$ из правой части оператора присваивания.	
$p = *q$	Пусть $a$ - переменная, на которую ссылается переменная $q$ из правой части оператора присваивания, $b$ - переменная, на которую ссылается переменная $a$ . Тогда после выполнения инструкции присваивания переменная $p$ из левой части оператора присваивания будет ссылаться на переменную $a$ .	

Продолжение на след. стр.

Продолжение таблицы 2.1

$*p = q$	<p>Пусть <math>a</math> - переменная, на которую ссылается переменная <math>q</math> из правой части оператора присваивания, <math>b</math> - переменная, на которую ссылается переменная <math>p</math> из левой части оператора присваивания. Тогда после выполнения инструкции присваивания переменная <math>b</math> будет ссылаться на переменную <math>a</math>.</p>	
$*p = \&q$	<p>Пусть переменная <math>p</math> из левой части оператора присваивания ссылается на переменную <math>a</math>. Тогда после выполнения инструкции присваивания переменная <math>a</math> будет ссылаться на переменную <math>q</math> из правой части оператора присваивания.</p>	
$*p = *q$	<p>Пусть переменная <math>q</math> из правой части оператора присваивания ссылается на переменную <math>a</math>, переменная <math>a</math> ссылается на переменную <math>b</math> и переменная <math>p</math> из левой части оператора присваивания ссылается на переменную <math>c</math>. Тогда после выполнения инструкции присваивания переменная <math>c</math> будет ссылаться на переменную <math>b</math>.</p>	

## 2.6 Вычисление относительного множества блокировок и его обобщения

Под относительным множеством блокировок  $L$  будем понимать пару  $(L_+, L_-)$ , состоящую из множества захваченных блокировок  $L_+$  и множества освобожденных блокировок  $L_-$ . Оно отражает изменение множества захваченных блокировок, производимое во время некоторого пути выполнения функции.

Перед началом анализа каждого пути выполнения функции относительное множество блокировок полагается пустым. Далее производится последовательный

анализ инструкций, встречаемых в пути. Изменение относительного множества блокировок может быть выполнено в момент вызова какой-либо функции. Для получения измененного относительного множества блокировок после вызова функции используется функция  $lock\_update((L_+, L_-), (L'_+, L'_-)) = ((L_+ \cup L'_+) - L'_-, (L_- \cup L'_-) - L'_+)$ . Её первым аргументом является текущее относительное множество блокировок, полученное к моменту вызова функции, а вторым - обобщение относительного множества блокировок вызываемой функции, в котором вхождения формальных параметров функции заменяются на соответствующие им передаваемые при вызове аргументы. Схема алгоритма вычисления относительного множества блокировок при анализе пути выполнения функции представлена на рис...

Под обобщением относительного множества блокировок  $lock\_summary(f)$  для некоторой функции  $f$  понимается относительное множество блокировок, отражающее изменение состояния множества захваченных блокировок во время выполнения функции вне зависимости от выбранного пути выполнения. Для его вычисления необходимо сначала определить относительные множества блокировок, получаемые в конце каждого из анализируемых путей выполнения функции. После чего множество освобожденных блокировок в обобщении будет вычислено как объединение всех множеств освобожденных блокировок, полученных в конце выполнения каждого из анализируемых путей функции, а множество захваченных - как пересечение полученных множеств захваченных блокировок для каждого из путей за вычетом уже подсчитанного множества освобожденных блокировок из обобщения. Таким образом обобщение относительного множества блокировок для функции будет иметь вид  $lock\_summary(f) = ((\bigcup_{i \in N} L_+[i]) / (\bigcap_{i \in N} L_-[i]), (\bigcup_{i \in N} L_-[i]))$ , где  $N$  - количество анализируемых путей исполнения функции, а  $L_+[i]$  и  $L_-[i]$  - множества захваченных и освобожденных блокировок, полученные на  $i$ -м анализируемом пути соответственно.

Для функций захвата и освобождения блокировок обобщения заранее известны, и их не требуется вычислять. Так для функции захвата некоторой блокировки  $l_a$  обобщение относительного множества блокировок будет равным  $(\{l_a\}, \{\})$ , а для функции освобождения -  $(\{\}, \{l_r\})$ , где  $l_r$  - освобождаемая блокировка.

## 2.7 Построение таблиц защищенного доступа

Под защищенным доступом  $A$  будем понимать тройку  $(o, L, k)$ , где  $o$  - lvalue  $||$ , к которому производится доступ,  $L$  - относительное множество блокировок на момент доступа,  $k$  - тип доступа («чтение» или «запись»). Таблицей защищенного доступа будем называть множество всех защищенных доступов, производимых во всех анализируемых путях выполнения функции.

Вначале анализа функции соответствующая ей таблица защищенного доступа полагается пустой. Затем по мере анализа различных путей исполнения функции в неё добавляются соответствующие защищенные доступы. Для построения данной таблицы производится последовательный анализ инструкций, выполняемых на каждом из путей. Если текущая анализируемая инструкция содержит доступ к разделяемой переменной (области памяти), то в таблицу добавляется соответствующая запись. Под разделяемой областью в контексте анализа функции понимаются глобальные переменные и формальные параметры функции, являющиеся указателями. В случае, когда анализируемая инструкция является вызовом функции, нужно выполнить конкретизацию соответствующей ей таблицы защищенного доступа, и добавить записи из полученной конкретизированной таблицы в таблицу защищенного доступа анализируемой функции. Схема алгоритма конкретизации представлена на рис... При конкретизации таблицы защищенного доступа сначала выполняется замена всех вхождений формальных параметров функции на соответствующие им передаваемые аргументы. После чего из полученной таблицы выбираются все записи, соответствующие разделяемым областям в контексте вызывающей функции. Затем выполняется модификация относительных множеств блокировок для каждой записи с использованием функции *lock\_update*. Первым аргументом в неё передается множество блокировок полученное до вызова функции, а вторым - относительное множество блокировок из соответствующей записи таблицы.

## 2.8 Определение мест возможного возникновения гонок

Для определения мест возможного возникновения гонок необходимо предварительно выявить все точки входа в потоки и выполнить конкретизацию соответствующих им таблиц защищенного доступа функций. Алгоритм конкретизации таблиц описан в 2.7. После конкретизации таблиц для каждой точки входа в поток 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 алгоритма представлена на рис...

## 2.9 Выводы

Не придумал( (пропущенные выше рисунки вставляю)

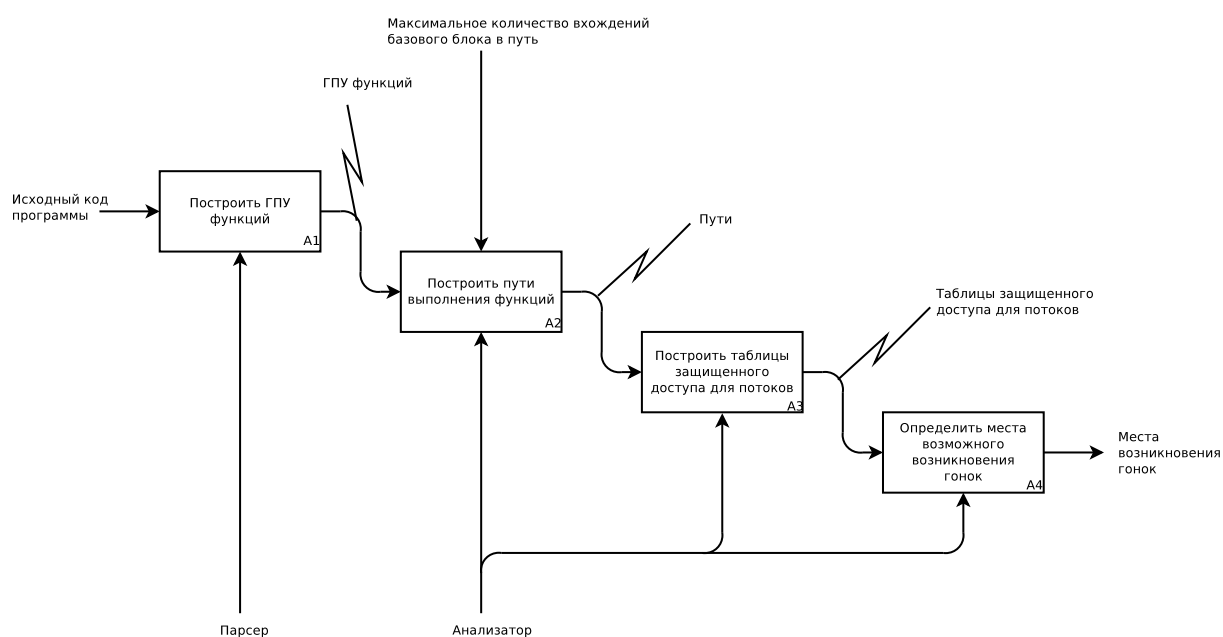


Рисунок 2.2 — Статический метод поиска гонок на основе относительного множества блокировок

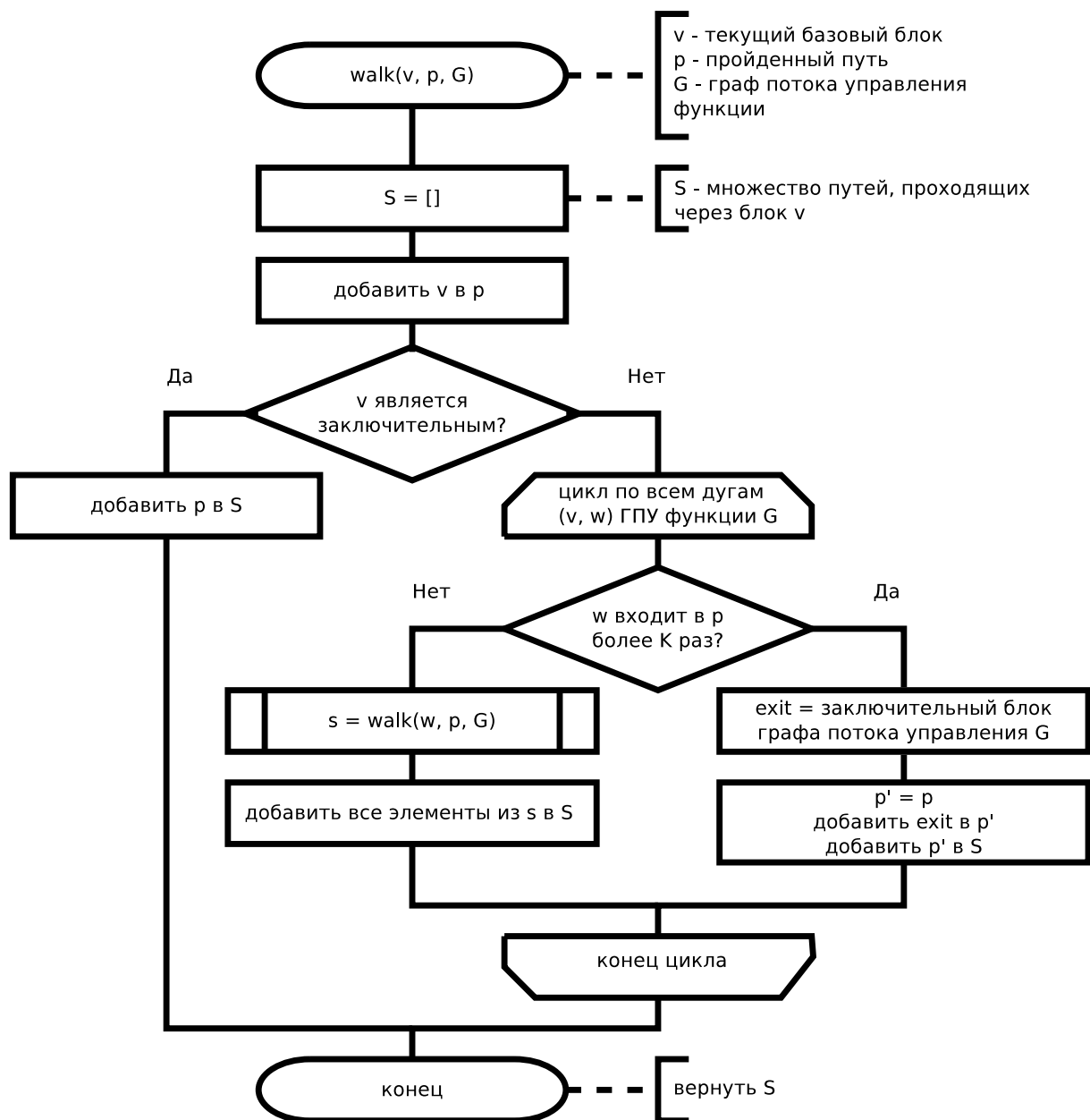


Рисунок 2.3 — Схема алгоритма построения множества анализируемых путей выполнения функции

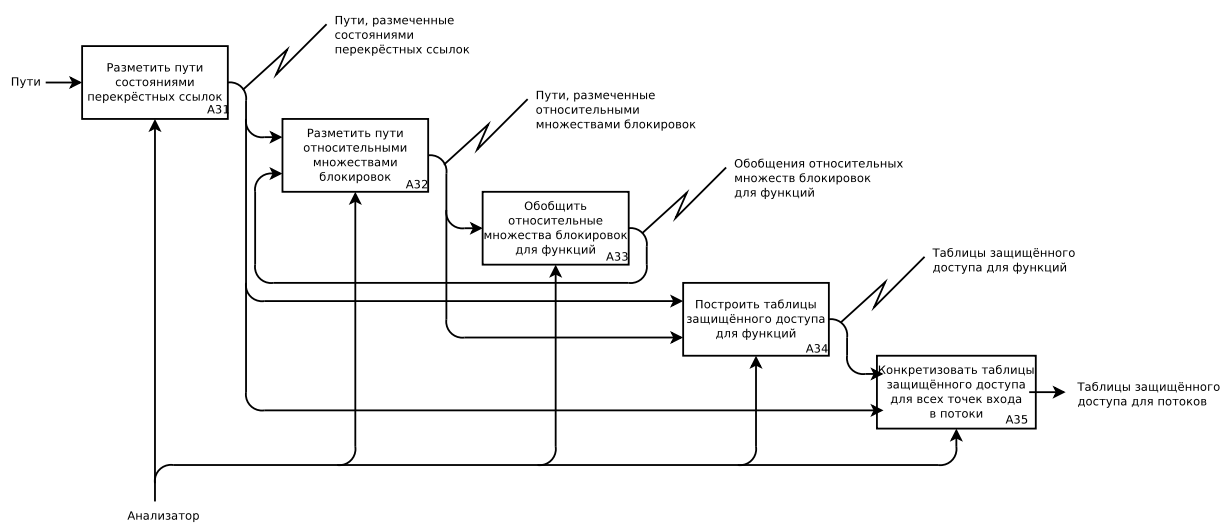


Рисунок 2.4 — Схема построения таблиц защищенного доступа для потоков



## 3 Технологический раздел

В данном разделе представлено проектирование анализатора поиска гонок и его реализация. Анализатор реализует метод статического поиска гонок на основе относительного множества блокировок.

### 3.1 Выбор формы представления программы

Большинство современных компиляторов поддерживает механизм трёхфазной компиляции, который состоит из:

- преобразования исходного кода в промежуточное представление,
- оптимизации промежуточного представления,
- получения кода целевой машины.

Каждому из этапов данного процесса компиляции соответствует своя форма представления программы: исходный код, промежуточное представление и код целевой машины. Отметим, что основным достоинством данного подхода является независимость процессов оптимизации и получения кода целевой машины от исходного языка программирования, на котором написана компилируемая программа. Рассмотрим подробнее каждую из форм представления программы.

#### 3.1.1 Исходный код программы

Исходный код программы является текстовым представлением программы на каком-либо высокоуровневом языке программирования.

#### 3.1.2 Промежуточное представление

Промежуточное представление имеет достаточно простой ассемблероподобный синтаксис.

#### 3.1.3 Код целевой машины

Код целевой машины представляет из себя двоичный код, имеющий нетекстовый вид.

### 3.2 Выбор языка промежуточного представления

llvm vs gimple vs java byte code.

### 3.3 Выбор языка. Используемые библиотеки

В качестве языка программирования был выбран язык Python. Он хорошо подходит для создания прототипов работающих программ и обладает следующими достоинствами:

- прост в изучении,
- позволяет решать сложные задачи,
- удобство сопровождения,
- превосходный синтаксис,
- поддержка объектно-ориентированного и структурного подходов программирования,
- широкий спектр стандартных библиотек,
- огромное количество сторонних свободно распространяемых библиотек с открытым исходным кодом,
- простой доступ к сторонним библиотекам через PyPI (Python Package Index),
- кроссплатформенность.

### 3.4 Про плагины для gcc (придумай как назвать

бла

### 3.5 Ограничения реализации

Для возможности реализации разработанного метода на анализируемый исходный код накладываются следующие ограничения:

- tra-ta-ta-ta

### 3.6 Структура ПО

Диаграмма компонентов + диаграмма последовательности

Структура разработанного программного обеспечения представлена на рис. 3.1.

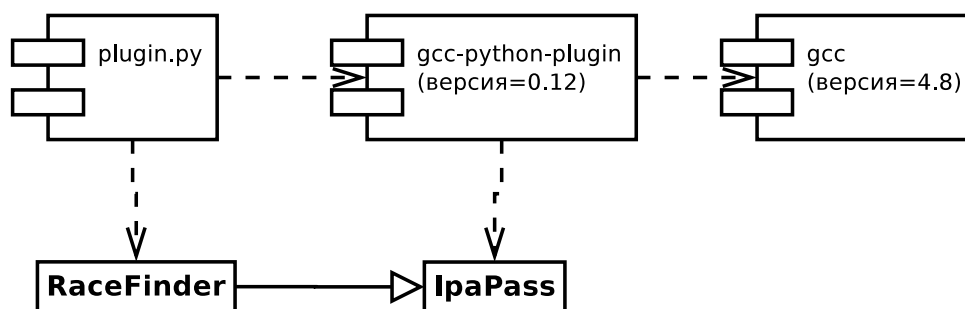


Рисунок 3.1 — Диаграмма копонентов анализатора

### 3.7 Запуск программы. Формат выходных сообщений

Для запуска анализатора необходимо выполнить следующую команду к командной строке:

```
gcc -fplugin=<path-to-gcc-python-plugin-lib> \
    -fplugin-arg-python=plugin.py \
    -fplugin-arg-python-max-level=<max-level> \
    -fplugin-arg-python-with-main=<with-main> \
    <others>
```

где:

- **<path-to-gcc-python-plugin-lib>** - путь до библиотеки **gcc-python-plugin**,
- **<max-level>** - максимальное количество раз, которое базовый блок может встретиться в анализируемом пути,
- **<with-main>** - флаг, разрешающий/запрещающий включение в анализ результатов работы главного потока программа, допустимыми значениями которого являются строки *'true'* и *'false'*,
- **<others>** - обычные параметры, задаваемые при компиляции программы с использованием компилятора **gcc**.

Пример запуска анализатора:

```
gcc -fplugin=/home/alex/gcc-python-plugin/python.so \
    -fplugin-arg-python=plugin.py \
    -fplugin-arg-python-max-level=1 \
    -fplugin-arg-python-with-main='true' \
    test.c -lpthread
```

Сообщение о найденном месте возникновения гонки имеет следующий вид:

```
WARNING: Race condition when accessing the variable <variable-name> (<visibility>)
```

где:

- **<variable-name>** - имя переменной, к которой осуществлялся доступ,
- **<visibility>** - область видимости переменной,
- **<line>** - строка, в которой производился доступ.

Пример сообщений, выдаваемых анализатором:

```
WARNING: Race condition when accessing the variable buffer (global) on line 37
WARNING: Race condition when accessing the variable buffer (global) on line 19
```

### 3.8 Выводы

бла-бла

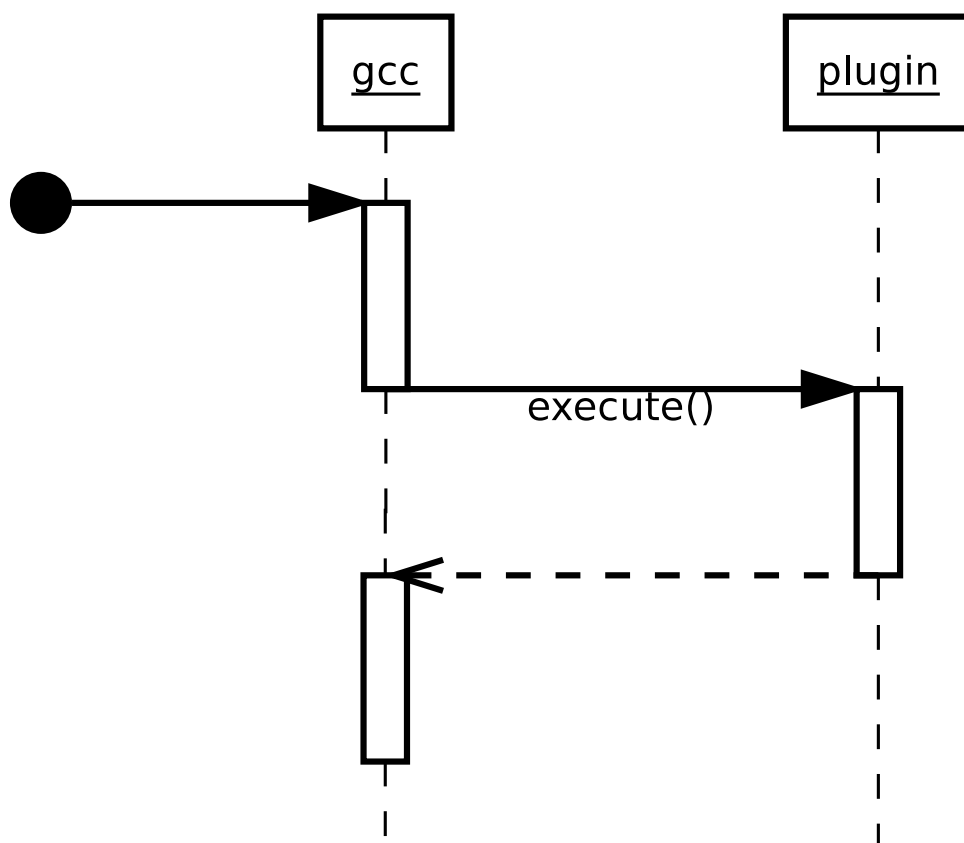


Рисунок 3.2 — Диаграмма последовательности анализатора

## 4 Исследовательский раздел

В данном разделе описаны эксперименты, проводимые с разработанным программным обеспечением. Эксперименты проводятся с целью определения скоростных характеристик разработанного программного обеспечения и метода в целом, производится оценка точности обнаружения ситуаций гонок в программах в зависимости от их структуры.

### 4.1 Условия проведения экспериментов

Проведение экспериментов производилось в следующих условиях:

а) аппаратное обеспечение:

- 1) AMD Turion(tm) X2 Ultra Dual-Core Mobile ZM-82 2,2GHz;
- 2) 3096 Мб ОЗУ;

б) программное обеспечение:

- 1) ОС Fedora GNU/Linux 20.0 3.11.10-301.fc20.i686;
- 2) GCC 4.8.2;
- 3) Python 2.7.5;
- 4) gcc-python-plugin 0.12.

### 4.2 Исследование скоростных характеристик

Для проведения экспериментов по определению скоростных характеристик за основу была взята задача «читатели-писатели». В ней предполагается, что есть некоторый общий для всех потоков ресурс. Часть потоков получает к нему доступ только для чтения, а часть - для записи. При этом чтение может осуществляться одновременно из нескольких потоков. Код анализируемой программы представлен в листинге 4.1.

Листинг 4.1 — Код решения задачи "читатели-писатели" (**readers-writers.c**)

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 int buffer = 0;
5 int reader_count = 0;
6 pthread_mutex_t reader_count_mutex;
7 pthread_mutex_t reader_mutex;
8 pthread_mutex_t writer_mutex;
9
10 void* reader(void* args) {
11     while (1) {
12         pthread_mutex_lock(&reader_count_mutex);
13         if (reader_count == 0) {
```

```

14         pthread_mutex_lock(&reader_mutex);
15     }
16     reader_count += 1;
17     pthread_mutex_unlock(&reader_count_mutex);
18
19     printf("reader: %d\n", buffer);
20
21     pthread_mutex_lock(&reader_count_mutex);
22     reader_count -= 1;
23     if (reader_count == 0) {
24         pthread_mutex_unlock(&reader_mutex);
25     }
26     pthread_mutex_unlock(&reader_count_mutex);
27     sleep(1);
28 }
29 return NULL;
30 }
31
32 void* writer(void* args) {
33     while (1) {
34         pthread_mutex_lock(&reader_mutex);
35         pthread_mutex_lock(&writer_mutex);
36
37         buffer += 1;
38         printf("writer: %d\n", buffer);
39
40         pthread_mutex_unlock(&writer_mutex);
41         pthread_mutex_unlock(&reader_mutex);
42         sleep(1);
43     }
44     return NULL;
45 }
46
47 int main(int argc, char** argv) {
48     pthread_t thread1, thread2, thread3, thread4, thread5, thread6;
49
50     pthread_mutex_init(&reader_count_mutex, NULL);
51     pthread_mutex_init(&reader_mutex, NULL);
52     pthread_mutex_init(&writer_mutex, NULL);
53
54     // create readers
55     pthread_create(&thread1, NULL, reader, NULL);
56     pthread_create(&thread2, NULL, reader, NULL);
57     pthread_create(&thread3, NULL, reader, NULL);
58     pthread_create(&thread4, NULL, reader, NULL);
59
60     // create writers

```

```

61 pthread_create(&thread5, NULL, writer, NULL);
62 pthread_create(&thread6, NULL, writer, NULL);
63
64 pthread_join(thread1, NULL);
65 pthread_join(thread2, NULL);
66 pthread_join(thread3, NULL);
67 pthread_join(thread4, NULL);
68 pthread_join(thread5, NULL);
69 pthread_join(thread6, NULL);
70
71 pthread_mutex_destroy(&reader_count_mutex);
72 pthread_mutex_destroy(&reader_mutex);
73 pthread_mutex_destroy(&writer_mutex);
74
75 return 0;
76 }

```

В процессе проведения экспериментов проводилось изменение числа потоков, создаваемых в функции **main**, и ограничения, накладываемого на максимальное количество вхождений базового блока в анализируемый путь. Графики зависимостей количества анализируемых путей и количества анализируемых инструкций от изменения максимального количества вхождений базового блока в путь представлены на рис. 4.1 и на рис. 4.2 соответственно. Зависимости времени анализа от количества анализируемых путей, количества анализируемых инструкций и количества потоков представлены на рис. 4.3, рис. 4.4 и рис. 4.5 соответственно. Видно, что с ростом ограничения, накладываемого на максимальное количество вхождений базового блока в путь, происходит экспоненциальный рост количества анализируемых путей и инструкций. Время анализа от количества анализируемых путей и инструкций зависит линейно. Повышение точности анализа с ростом значения ограничения, накладываемого на максимальное количество вхождений базового блока в путь, не была выявлена, следовательно, разумно в дальнейшем использовать при анализе значения данного ограничения, равным 1. Кроме того было выявлено, что количество созданных потоков слабо влияет на время анализа. Это объясняется тем, что анализ каждой функции программы выполняется только один раз, в местах, где производится её вызов, применяются результаты анализа, полученные ранее.

### 4.3 Исследование точности

Для исследования точности получаемых результатов использовался тестовый набор программ. Рассмотрим далее результаты полученные на каждом тестовом примере, определим количество правильно обнаруженных ситуаций гонок, количества ошибок первого и второго рода.



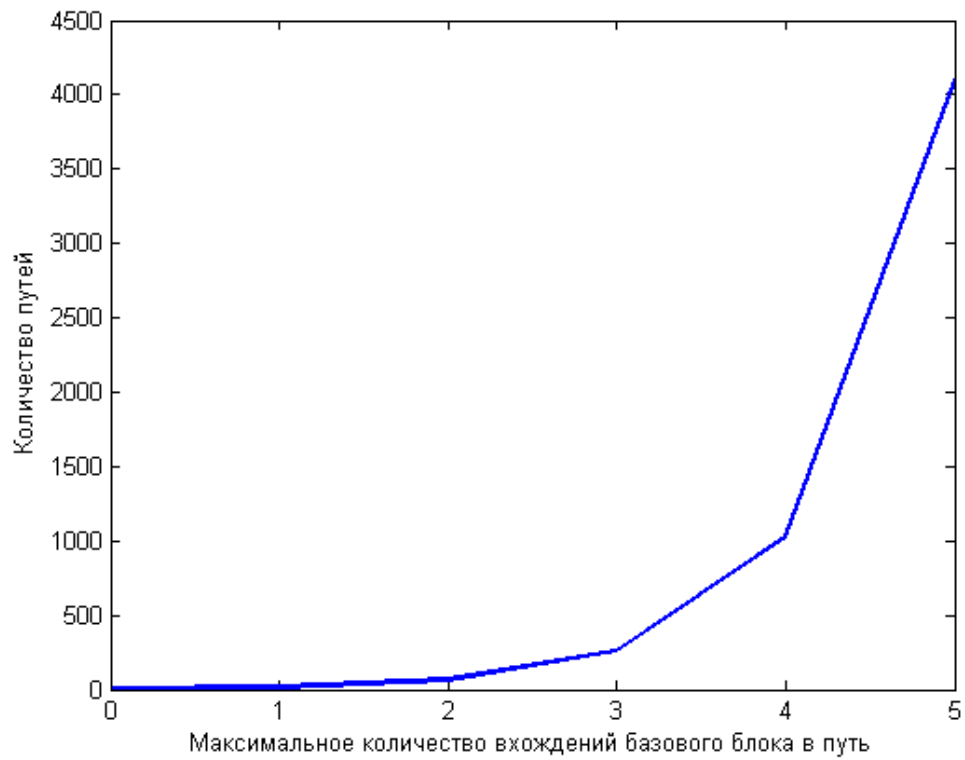


Рисунок 4.1 — Зависимость количества анализируемых путей от максимального количества вхождений базового блока в путь

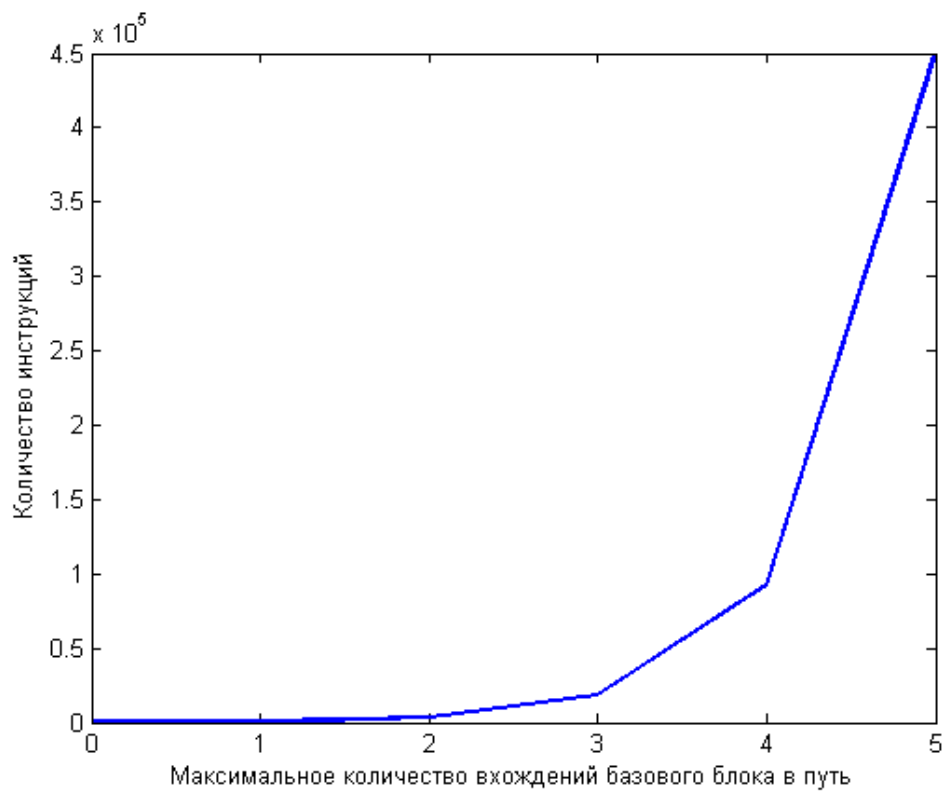


Рисунок 4.2 — Зависимость количества анализируемых инструкций от максимального количества вхождений базового блока в путь

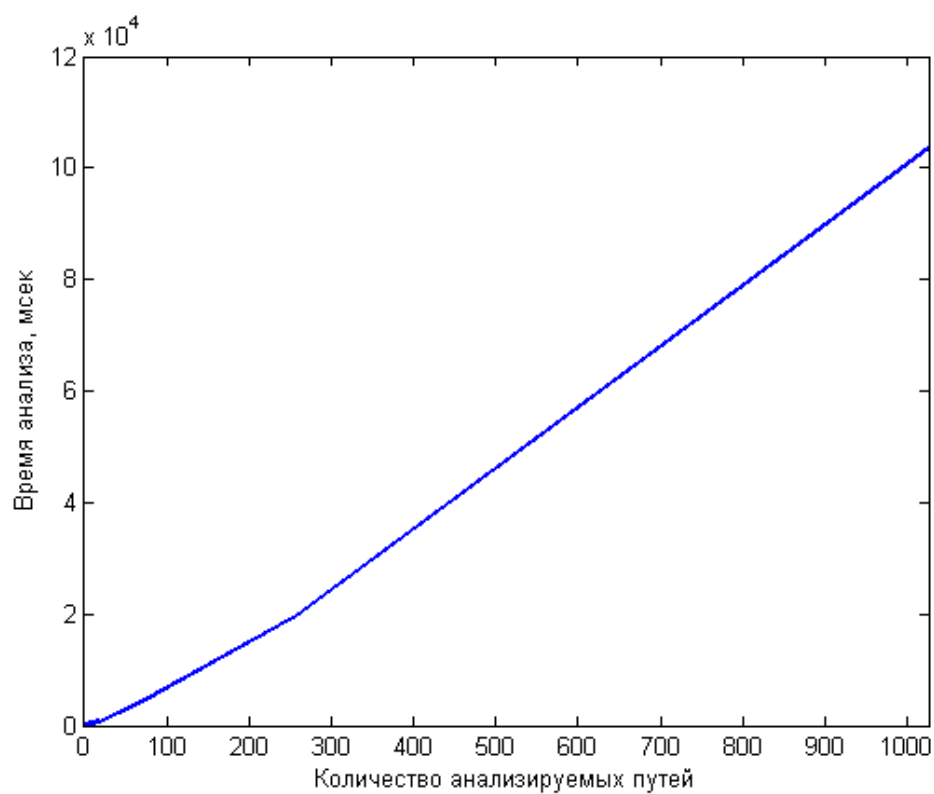


Рисунок 4.3 — Зависимость времени анализа от количества анализируемых путей

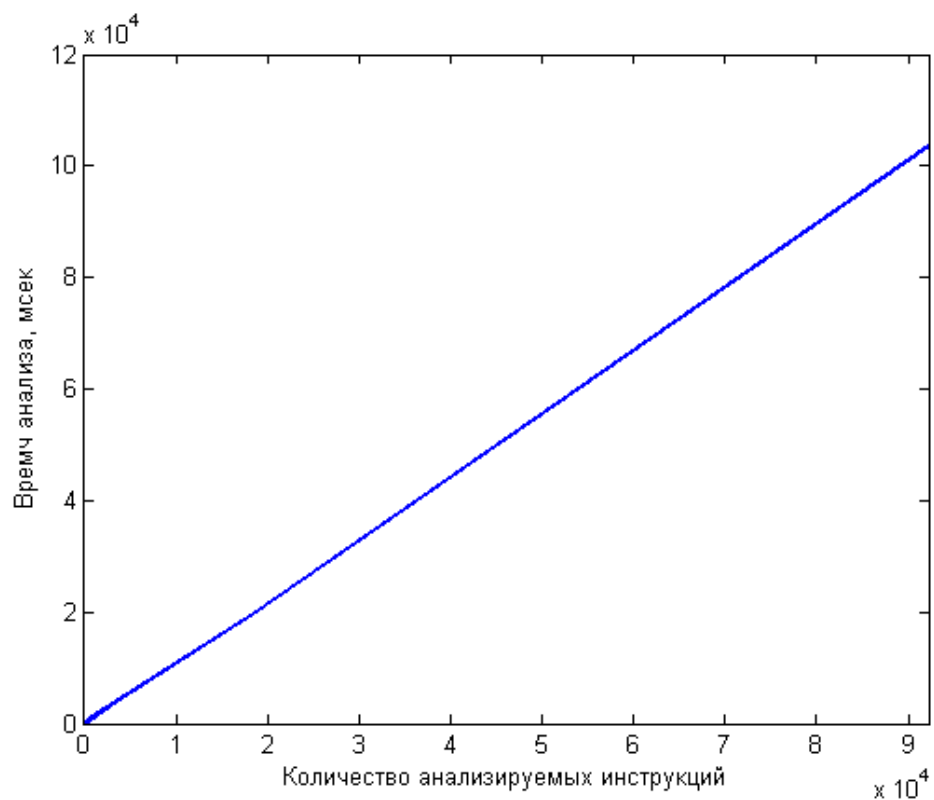


Рисунок 4.4 — Зависимость времени анализа от количества анализируемых инструкций

### 4.3.1 Программа 1

Текст программы представлен в листинге 4.2. В представленной программе в функции **main** создается 3 потока, в каждом из которых выполняется вызов функции **munge** с различными параметрами. В тексте программы отсутствуют циклы и ветвления. Анализатор выявил следующие места возникновения гонок:

```
WARNING: Race condition when accessing the variable y (global) on line 34
WARNING: Race condition when accessing the variable x (global) on line 34
WARNING: Race condition when accessing the variable z (global) on line 34
WARNING: Race condition when accessing the variable y (global) on line 8
WARNING: Race condition when accessing the variable x (global) on line 8
WARNING: Race condition when accessing the variable z (global) on line 8
```

Среди найденных анализатором возможных мест возникновения гонок таковыми являются только два:

```
WARNING: Race condition when accessing the variable y (global) on line 8
WARNING: Race condition when accessing the variable z (global) on line 8
```

Остальные обнаруженные места являются ошибочными и относятся к ошибкам второго рода. Их появление связано с тем, что при разработке метода было сделано предположение о параллельном выполнении всех потоков программы, поэтому, хотя инициализация разделяемых переменных *x*, *y* и *z* выполняется в функции **main** до создания остальных потоков, анализатор берет во внимание место их инициализации и отмечает его, как потенциальное место возникновения гонок, т.к. при доступе ним не были захвачены никакие блокировки.

Листинг 4.2 — Программа 1(**test1.c**)

```
1 #include <pthread.h>
2
3 pthread_mutex_t m1, m2, m3;
4 int x, y, z;
5
6 void munge(int* value, pthread_mutex_t* mutex) {
7     pthread_mutex_lock(mutex);
8     (*value) += 1;
9     pthread_mutex_unlock(mutex);
10 }
11
12 void* run_thread1(void* args) {
13     munge(&x, &m1);
14     munge(&y, &m2);
15     munge(&z, &m3);
16     return NULL;
```

```

17 }
18
19 void* run_thread2(void* args) {
20     munge(&x, &m1);
21     munge(&y, &m1);
22     return NULL;
23 }
24
25 void* run_thread3(void* args) {
26     munge(&x, &m1);
27     munge(&y, &m2);
28     munge(&z, &m1);
29     return NULL;
30 }
31
32 int main(int argc, char** argv) {
33     pthread_t thread1, thread2, thread3;
34     x = 0; y = 0; z = 0;
35
36     pthread_mutex_init(&m1, NULL);
37     pthread_mutex_init(&m2, NULL);
38     pthread_mutex_init(&m3, NULL);
39
40     pthread_create(&thread1, NULL, run_thread1, NULL);
41     pthread_create(&thread2, NULL, run_thread2, NULL);
42     pthread_create(&thread3, NULL, run_thread3, NULL);
43
44     pthread_join(thread1, NULL);
45     pthread_join(thread2, NULL);
46     pthread_join(thread3, NULL);
47
48     pthread_mutex_destroy(&m1);
49     pthread_mutex_destroy(&m2);
50     pthread_mutex_destroy(&m3);
51
52     return 0;
53 }

```

#### 4.3.2 Программа 2

Текст программы представлен в листинге 4.3. В представленной программе в функции **main** выполняется создание двух потоков, в каждом из которых производится модификация глобальной переменной *x*. В одном из потоков захват блокировки, доступ к глобальной переменной и освобождение блокировки происходит в случае,

когда выполняется определенное условие. Анализатор не выявил мест возникновения гонок, т.к. доступ к переменной *x* является защищенным во всех потоках.

Листинг 4.3 — Программа 2(**test2.c**)

```
1 #include <pthread.h>
2
3 pthread_mutex_t m1;
4 int x = 0;
5
6 void* run_thread1(void* args) {
7     int i = 0;
8     while (1) {
9         if (i % 10 == 5) {
10             pthread_mutex_lock(&m1);
11             x = x - 1;
12             pthread_mutex_unlock(&m1);
13         }
14         i++;
15     }
16     return NULL;
17 }
18
19 void* run_thread2(void* args) {
20     while (1) {
21         pthread_mutex_lock(&m1);
22         x = x + 1;
23         pthread_mutex_unlock(&m1);
24     }
25     return NULL;
26 }
27
28 int main(int argc, char** argv) {
29     pthread_t thread1, thread2;
30
31     pthread_mutex_init(&m1, NULL);
32
33     pthread_create(&thread1, NULL, run_thread1, NULL);
34     pthread_create(&thread2, NULL, run_thread2, NULL);
35
36     pthread_join(thread1, NULL);
37     pthread_join(thread2, NULL);
38
39     pthread_mutex_destroy(&m1);
40
41     return 0;
42 }
```

### 4.3.3 Программа 3

Текст программы представлен в листинге 4.4. В программе создается 2 потока, в каждом из которых происходит чтение и печать на экран значения глобальной *x*. Анализатор не выявил мест возникновения гонок, т.к. к разделяемой переменной *x* производился доступ только для чтения.

Листинг 4.4 — Программа 3(**test3.c**)

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 int x = 123;
5
6 void* run_thread(void* args) {
7     printf("x = %x\n", x);
8     return NULL;
9 }
10
11 int main(int argc, char** argv) {
12     pthread_t thread1, thread2;
13
14     pthread_create(&thread1, NULL, run_thread, NULL);
15     pthread_create(&thread2, NULL, run_thread, NULL);
16
17     pthread_join(thread1, NULL);
18     pthread_join(thread2, NULL);
19
20     return 0;
21 }
```

### 4.4 Программа 4

Текст программы представлен в листинге 4.1. Программа является решением задачи «читатели-писатели». Анализатор выявил следующие места возникновения гонок:

WARNING: Race condition when accessing the variable buffer (global) on line 37

WARNING: Race condition when accessing the variable buffer (global) on line 19

Оба найденные места возникновения гонок при доступе к разделяемой переменной *buffer* являются ошибочными и относятся к ошибкам второго рода. Их появление вызвано тем, что блокировка *reader\_mutex* захватывается не на всех анализируемых путях при доступе к переменной *buffer*. Это связано с тем, что все пути анализируются независимо друг от друга.

## 4.5 Выводы

Проведены эксперименты с разработанным ПО. По итогам проведенных экспериментов определено, что разработанный метод допускает обнаружение ложных ситуаций гонок в ситуациях, когда не производится явного входа в критическую секцию путем захвата какого-либо объекта блокировки, например, при условной блокировке. Кроме того в силу предположения о параллельном выполнении всех потоков, метод также допускает обнаружение ложных ситуаций гонок в ситуациях, когда выполняется только один поток, например, при инициализации переменных в главном потоке до порождения остальных потоков. На основе экспериментальных данных получена зависимость времени анализа от количества анализируемых путей, количества анализируемых инструкций и количества анализируемых потоков для задачи «читатели-писатели».

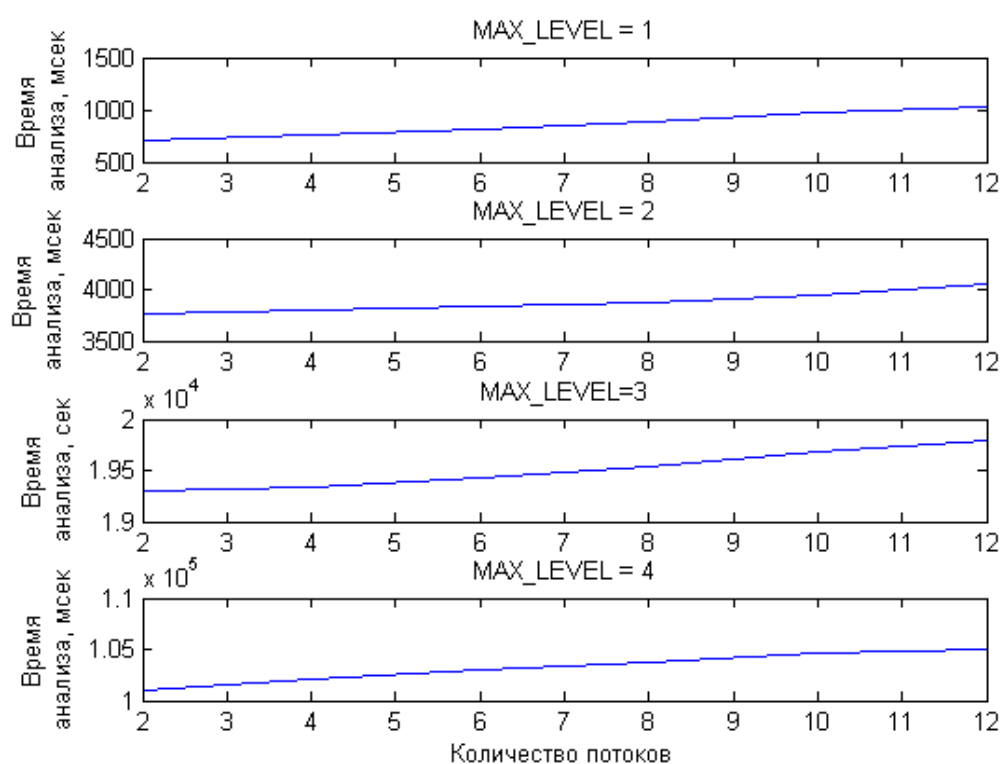


Рисунок 4.5 — Зависимость времени анализа от количества потоков



## **Заключение**

В результате проделанной работы стало ясно, что ничего не ясно...

## Список использованных источников

1. *Пупкин, Василий*. L<sup>A</sup>T<sub>E</sub>X для «чайников» / Василий Пупкин, А. Эйнштейн.  
— М., 2009.