Реферат

В данном документе, являющимся расчётно-пояснительной запиской к квалификационной работе магистра по направлению «Информатика и вычислительная техника» на тему «Метод статического поиска условий возникновения гонок», представлены сведения о программном продукте, предназначенном для проверки программного алгоритма на наличие гонок.

В документе представлена информация о существующих методах статического поиска условий возникновения гонок, а так же программных продуктах, реализующих тот или иной метод. Документ содержит описание разрабатываемого метода статического поиска в виде алгоритмов, описание программного обеспечения, реализующего разработанный метод и сведения о тестировании разработанного ПО и экспериментах. Так же в документе производится оценка времени анализа программ на основе модифицированной задачи обедающих философов и экспериментальная проверка полученной формулы.

Содержание

Глоссарий

Процесс — программа на стадии выполнения.

Поток — наименьшая единица диспетчеризации, которой может быть выделено процессорное время операционной системы; является частью процесса.

Гонка (состояние гонки, race condition) — состояние программы, при котором результат выполнения программы непредсказуемо зависит от последовательности действий, выполненных процессами или потоками.

Тупик (deadlock) — состояние программы, при котором два или более процессов ожидают действий друг друга по освобождению разделяемых ресурсов для своего дальнейшего выполнения.

Общий (разделяемый) ресурс — ресурс, доступный для одновременного использования несколькими процессами или потоками.

Объект синхронизации — объект, обеспечивающий взаимоисключение процессов или потоков при обращении к разделяемым ресурсам или выполнении частей кода программы.

Критическая секция — часть кода программы, которая может выполняться только одним процессом или потоком в один и тот же момент времени.

Захват объекта синхронизации — изменение состояния объекта синхронизации для получения процессом или потоком монопольного доступа к разделяемому ресурсу или критической секции.

Освобождение объекта синхронизации — изменение состояния объекта синхронизации процессом или потоком для предоставления доступа к разделяемому ресурсу или критической секции другим процессам или потокам.

Спинлок — объект синхронизации, использующий активное ожидание на процессоре.

Семафор — объект синхронизации, имеющий две атомарные операции: P и V (повышение значения и понижение)

Блокировка на чтение — тип блокировки для разделяемых ресурсов, при котором одновременно несколькими потоками или процессами может производиться только операция чтения.

Обозначения и сокращения

ОС — Операционная система.

JVM — Java Virtual Machine.

LLVM — Low Level Virtual Machine.

ГПУ — Граф потока управления.

ГЗП — Граф зависимости переменных.

ГЗД — Граф зависимости данных.

ПО — Программное обеспечение.

Введение

Возрастающая сложность программного обеспечения приводит к увеличению количества ошибок в нем, а одновременный рост количества и критичности выполняемых им функций влечет рост ущерба от этих ошибок. Для обеспечения корректности и надежности работы таких систем предназначены различные методы верификации алгоритмов, позволяющие выявлять дефекты на разных этапах разработки и сопровождения системы.

Наиболее сложными с точки зрения обнаружения и исправления являются ошибки, тесно связанные с параллельным характером выполнения программы. Возникновение такого рода ошибок, как правило, носит стохастический характер, что приводит к сложности повторения возникновения ошибки с целью её дальнейшей локализации и исправления.

В рамках решения задач по обнаружению ситуаций гонок в программах можно выделить два основных подхода [1]:

а) статический поиск условий возникновения гонок;
б) динамический поиск гонок.

В дальнейшем понятия методы статического поиска условий возникновения гонок и методы динамического поиска гонок для краткости будут обозначаться как статические и динамические методы соответственно.

Статический поиск основан на анализе исходных или оттранслированных кодов программы и поиске таких состояний программы, при которых может возникнуть гонка. Достоинством статических методов поиска условий возникновения гонок является возможность анализа всех возможных вариантов выполнения нескольких потоков приложения (всех возможных состояний программы) и возможность применения данных методов для анализа программ на стадии разработки. Однако число состояний программы может легко достигать значений порядка $10^8 - 10^{10}$, что может привести к неприемлемому времени анализа кода программы. Так же методы статического поиска имеют тенденцию к ложным срабатываниям, то есть к обнаружению гонки потоков, где в действительности её нет.

Динамический поиск основан на наблюдении за выполнением программы и непосредственным обнаружением состояний гонки. Методы динамического поиска при корректной реализации лишены недостатков статических методов, таких как неприемлемо большое время поиска и ложные срабатывания, так как подобные методы обнаруживают гонки потоков в тот момент, когда наступает такая ситуация. Однако, методы динамического поиска способны обнаружить только произошедшие ситуации гонок, в связи с чем, очевидно, часть возможных ситуаций гонок не будет обнаружена. С учетом того, что возникновение гонок имеет стохастический характер,

для обнаружения всех мест программного кода, в которых возникают гонки, может потребовать нескольких запусков программы с различными исходными данными, для обеспечения максимально полного «покрытия» возможных путей выполнения программы.

Актуальность данной работы заключается в том, что в настоящее время методы статического поиска условий возникновения гонок существуют в основном в виде идей и концепций, разработка программного обеспечения, реализующего данные методы, либо остановлена, либо ПО находится в «сыром» состоянии и малопригодно для использования.

Целью работы является разработка метода статического поиска возможных мест возникновения гонок. В рамках проведенной НИРС в качестве входных данных было принято решение использовать промежуточное представление кода в виде ассемблера виртуальной машины IR LLVM. Обоснование принятия такого решения приведено в соответствующих разделах данной пояснительной записки.

Для достижения поставленной цели необходимо решить следующие задачи:

а) провести обзор существующих методов поиска условий возникновения гонок;
б) выбрать формат промежуточного представления программы;
в) провести детальный обзор выбранного формата данных;
г) разработать метод статического поиска условий возникновения гонок;
д) разработать программное обеспечение, реализующее разработанный метод;
е) провести эксперименты с разработанным программным обеспечением.

Данная пояснительная записка имеет следующую структуру:

— В главе 1 приведен обзор статического подхода к поиску возможных ситуаций гонок в программах.

— В главе 2 приведен обзор возможных форм представления входных данных для анализа, а так же обоснование выбора представления в виде промежуточного кода на языке IR LLVM.

— В главе 3 приведено описание и основные алгоритмы разрабатываемого метода статического поиска возможных ситуаций гонок.

— В главе 4 приведено описание проектирования программного обеспечения, реализующего разработанный алгоритм.

— В главе 5 приведено описание реализации разработанного программного обеспечения, а так же описание тестирования ПО.

— В главе 6 приведено описание экспериментов, проводимых с разработанным программным обеспечением, а так же результаты этих экспериментов.

1 Статический поиск условий возникновения гонок

В данной главе приводится описание статического подхода к анализу программного кода при решении задачи поиска гонок потоков и условий их возникновения, приводятся основные термины и определения, связанные со статическим поиском гонок в программах. Глава содержит описание существующих методов статического поиска возможных ситуаций возникновения гонок, а так же существующих инструментов, реализующих те или иные методы.

1.1 Понятие ситуации гонки доступа к ОЗУ

Ситуацией гонки при доступе к памяти в программировании является ситуация, при которой два или более потоков выполнения программы получают доступ к одной и той же области памяти, при этом [2]:

а) хотя бы один поток осуществляет запись в область памяти;

б) для защиты доступа к переменной не используется каких либо средств взаимоисключения;

в) не существует внешних факторов, обеспечивающих упорядоченность доступа потоков к указанной области памяти[1].

Ситуации гонок в целом можно разделить на два класса:

а) одновременная запись — ситуация гонки возникает в случае, если два потока одновременно осуществляют запись в одну и ту же область памяти;

б) чтение-запись — ситуация гонки возникает в случае, когда один поток осуществляет чтение области памяти в тот момент, когда другой поток осуществляет запись в этот же участок памяти.

1.2 Основные сущности и определения

В рамках рассматриваемой предметной области можно выделить ряд важных для рассмотрения сущностей. На диаграмме 1.1 приведена ER-диаграмма, отражающая основные сущности рассматриваемой предметной области, а так же отношения между ними.

Граф потока управления

Граф потока управления — множество всех путей выполнения программы, представленное в виде графа. Граф потока управления является одним из ключевых объектов для некоторых методов анализа кода как с целью поиска гонок, так и для

[1]Данный пункт не присутствует в оригинальном определении. В качестве внешних факторов могут выступать особенности аппаратной или программной платформы, за счет которых обеспечивается упорядочивание доступа к памяти.

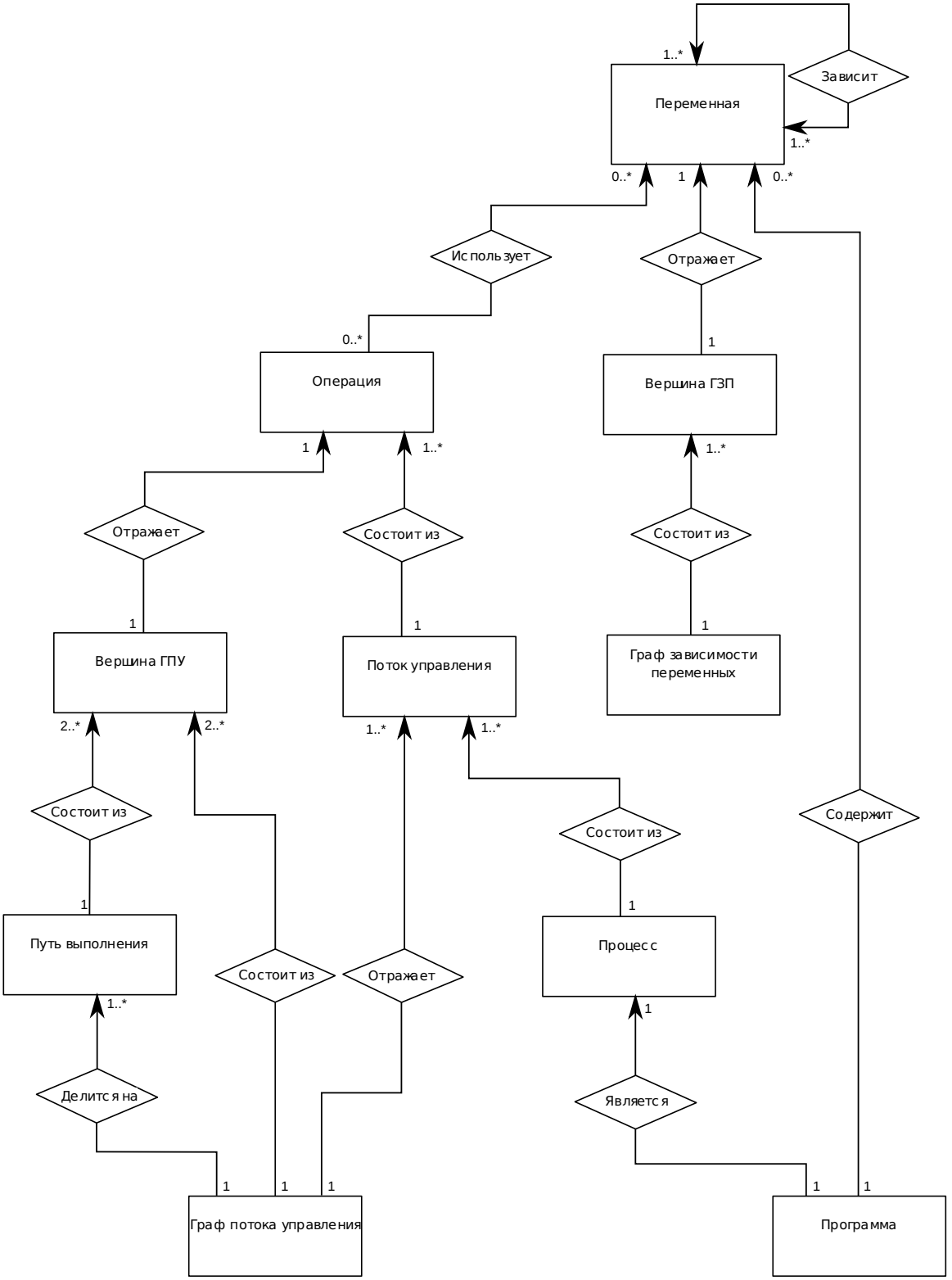

Рисунок 1.1 — ER-диаграмма основных сущностей рассматриваемой предметной области

других целей. Пример графа потока управления приведен на рисунке 1.2 для части кода программы на языке C, приведенной в листинге 1.1.

Листинг 1.1 — Пример части кода программы для построения ГПУ

```
1  int function(int a, int b) {
2      int rst;
3      rst = b - a;
4      if(rst < 0)
5          rst = -1;
6      else if(rst > 0)
7          rst = 1;
8      return rst;
9  }
```

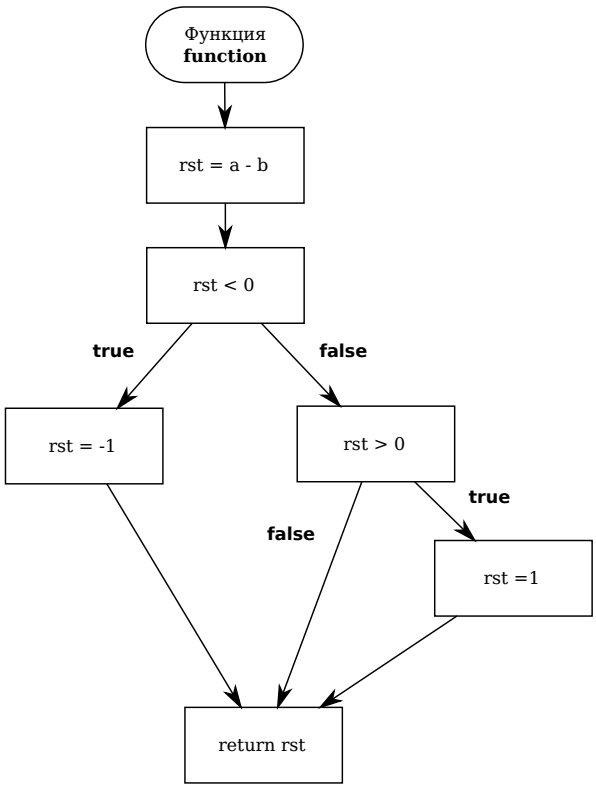

Рисунок 1.2 — Граф потока управления

Следует отметить, что граф потока управления может содержать различную информацию о выполняемых операциях, в том числе:

а) выполняемые операции;

б) переменные, к которым осуществляется доступ, с указанием вида доступа.

Видом доступа может быть либо чтение значения переменной, либо запись значения переменной.

В рамках данной дипломной работы используется вариант б, так как для выполнения анализа кода необходимы сведения о том, к каким переменным осуществляется доступ и какой вид доступа при этом используется.

Линейный граф потока управления (путь выполнения)

В рамках данной дипломной работы вводится такое понятие как линейный граф потока управления. Линейный граф потока управления — граф потока управления, не содержащий в себе ветвлений. Линейный граф потока управления представляет собой один из возможных путей обхода обычного графа потока управления, то есть один из возможных путей выполнения потока программы.

Линеаризация ГПУ

Под линеаризацией ГПУ понимается процесс поиска всех возможных путей обхода ГПУ и построение множества линейных ГПУ по найденным путям обхода.

Граф зависимости переменных

Граф зависимости переменных (ГЗП) — множество всех переменных программы и отношений между ними, представленными в виде графа. В контексте поиска условий возникновения гонок данный граф необходим для определения зависимых переменных, незащищенный доступ к которым может повлечь за собой возникновение ситуации гонки.

Пример графа зависимости переменных приведен на рисунке 1.3. В листинге 1.2 приведен пример части программы на языке C, соответствующей приведенному графу зависимости переменных.

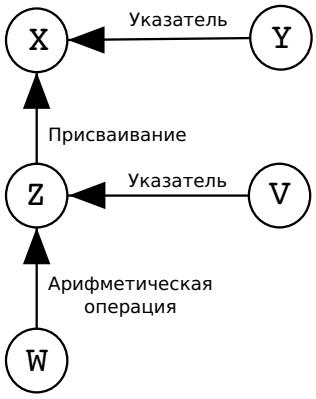

Рисунок 1.3 — Граф зависимости переменных для части кода программы

Листинг 1.2 — Код программы, соответствующий ГЗП

```
1 int x = 5;
2 int z, w;
```

```
3 int *y = &x;
4 int *v = &z;
5 z = x;
6 w = z * 10;
```

Определение зависимости переменных

В контексте данной дипломной работы под зависимыми данными понимаются переменные, значения которых зависят от значений других переменных. Переменная X зависит от переменной Y в том случае, если:

а) X является указателем на Y;

б) значение переменной Y присваивается значению переменной X;

в) значение переменной X получается посредством выполнения арифметических операций, в которых участвует переменная Y;

г) значение переменной X есть результат выполнения функции, одним из аргументов вызова которой является переменная Y.

Классификация зависимостей

Зависимости между переменными в коде программы можно разделить на несколько типов. Классификация данных зависимостей приведена в таблице 1.1. Данная классификация может использоваться как для отношениями между переменными, так и для отношений между переменными и константами.

Таблица 1.1 — Классификация зависимостей переменных

Класс	**Описание**
Указатель	Значение одной переменной является указателем на другую переменную.
Присваивание	Одной переменной присваивается значение другой переменной.
Разыменование	Переменной присваивается значение, адрес которого является значением второй переменной.
Операционная	Переменной присваивается значение, являющейся результатом выполнения функции или другой операции.

Следует заметить, что в общем случае отношение типа «Указатель» между переменными возникает в том случае, если адрес переменной задан любым возможным способом, таким как:

а) операцией получения адреса переменной;

б) заданием адреса переменной в виде константы;

в) вычисление адреса с использованием арифметических операций.

Операционная зависимость в общем случае возникает между переменной, в которую помещается результат выполнения операции, и переменными, являющимися аргументами производимой операции. Класс операционных зависимостей в общем случае можно разделить на два подкласса, описание которых приведено в таблице 1.2.

Таблица 1.2 — Классификация операционных зависимостей переменных

Подкласс	**Описание**
Арифметическая	Переменной присваивается результат выполнения арифметических операций.
Вызов функции	Переменной присваивается результат вызова функции.

Таблица указателей

Таблица указателей является табличным представлением подмножества графа зависимости переменных и предназначена для описания состояния указателей в определенный момент выполнения программы. Таблица состоит из двух столбцов, первый столбец содержит в себе список указателей, второй — текущее значение указателей.

1.3 Существующие методы статического поиска условий возникновения гонок

Достоинством статического поиска условий возникновения гонок является его теоретическая возможность анализа всех возможных путей выполнения программы, в том числе на уровне процессов или потоков.

Недостатком статического анализа является наличие ложных срабатываний анализаторов, то есть обнаружение ситуаций гонок в тех местах программы, где их нет, что усложняет анализ и выявление тех результатов, которые соответствуют действительным ситуациям гонок. Примером ложного срабатывания может являться инициализация переменных при старте программы, то есть в тот момент времени, когда программа выполняется в рамках одного процесса или потока. Так же в качестве примера можно привести инициализацию переменных в конструкторе класса, код которого, как правило, выполняется в рамках одного процесса или потока.

В статическом подходе к поиску гонок в программах можно выделить несколько основных методов анализа кода программы:

а) добавление аннотаций в код программы;
б) проверка модели программы (model-checking);
в) анализ потока управления (flow-based analysis);

г) комбинированный.

1.3.1 Метод аннотирования кода

Добавление аннотаций осуществляется за счет комментариев языка программирования, на котором ведется разработка программы. Комментарий должен быть оформлен определенным образом. Как правило, аннотации содержат информацию о том, какими объектами взаимоисключения обеспечивается защита переменной, какие объекты взаимоисключения требуются для вызова той или иной функции. Недостатком этого метода является необходимость большого числа аннотаций, что влечет за собой увеличение объема кода программы, ухудшения его читаемости и увеличение мест, в которых возможно допущение ошибок (в том числе возможность потери программистом необходимых в аннотации ссылок на объекты взаимоисключения). В качестве достоинства можно назвать гарантированность проверки захвата всех необходимых объектов взаимоисключения, указанных в соответствующей аннотации, при получении доступа к разделяемому ресурсу или критической секции.

Листинг 1.3 — Аннотированный код Java для инструмента rccjava

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

1.3.2 Метод проверки модели программы

Метод проверки модели программы заключается в анализе всех возможных путей выполнения программы для всех возможных значений переменных. Однако, в силу огромного количества различных комбинаций значений переменных и времени, требуемого для проведения подобного исследования программы, произвести подобный анализ для реальных программ не представляется допустимым. Таким образом, трудностью данного подхода является необходимость построения такой модели программы, которая может быть исследована за приемлемое время. Построенная модель программы представляет собой конечный автомат потока управления программы.

В рамках метода два способа анализа модели программы:

а) с сохранением состояний автомата;

б) без сохранения состояний.

Данные способы анализа различаются в поведении анализатора при переходе автомата в состояние, которое им уже было «посещено». В некоторых источниках [3] данный метод отнесен к динамическому подходу поиска гонок, однако такое утверждение является спорным в силу того, что в рамках данного метода выполнение программы лишь моделируется, но реального выполнения программы не происходит.

1.3.3 Метод анализа потока управления

Метод анализа потока управления основан на построении графа потока управления программы и его дальнейшем анализе с целью выявления частей программы, выполняющихся параллельно, и переменных, к которым возможен доступ нескольких потоков и блокировок, защищающих эти переменные.

Метод анализа потока управления программы является достаточно гибким методом, в основе которого лежит построение графа потока управления (ГПУ) программы. После построения граф потока управления может быть проанализирован сколь угодно большим числом методов, различное сочетание которых может дать приемлемые результаты в рамках решения поставленных задач.

Метод анализа графа потока управления позволяет строить достаточно сложные схемы анализа, а так же производить некоторые преобразования ГПУ с целью получения исходных данных для проведения анализа другими методами.

В рамках данного метода можно выделить два основных способа к поиску ситуаций гонок [4]:

а) анализ множеств захваченных объектов синхронизации (locksets);

б) анализ временных векторов.

Анализ временных векторов позволяет получить результаты проверки, содержащие небольшое количество ложных ситуаций гонок, однако при использовании данного способа достаточно трудно определить причину возникновения гонки.

Анализ множеств захваченных объектов синхронизации позволяет получить менее точные результаты анализа, однако позволяет достаточно легко определить причину возникновения гонки. Достоинством данного подхода является возможность отслеживания той дисциплины захвата объектов синхронизации, которая была заложена в программу разработчиком. Классический алгоритм анализа множеств захваченных объектов взаимоисключения ориентирован на то, что доступ к определенной переменной должен быть защищен одним и тем же объектом синхронизации в течение выполнения всей программы [4, 2], что, в общем случае, приводит к большому

количеству ложных сообщений о возможных ситуациях гонок. Несмотря на то, что этот метод был реализован в рамках динамического поиска в инструменте Eraser [2], он может быть применен и в рамках статического анализа, в связи с чем положение данного метода в общей иерархии достаточно неопределенное.

1.4 Существующие реализации методов статического поиска

В данном разделе приведено краткое описание существующих программ, реализующих методы статического поиска условий возникновения гонок.

Инструмент RacerX

На основе описания инструмента RacerX, приведенном в [1, 5], программа на момент написания указанных статей находилась в процессе разработки, но при этом инструмент был успешно применен для некоторых программных комплексов, в том числе существующих операционных систем Linux и FreeBSD, а так же некоторых коммерческих продуктов. По итогам использования в последних было выявлено некоторое количество ошибок, связанных с гонками.

Инструмент rcc-java

Race Condition Checker for Java является инструментом для поиска гонок в программах, реализованных на языке Java. Данный инструмент реализует метод аннотирования кода и предназначен для анализа программ размером до 20 тыс. строк[6]. Данный инструмент предоставляет пользователю графический интерфейс, который позволяет программисту определять сходные ситуации гонок, которые могут быть устранены путем внесения изменений в каком-либо одном месте программы.

1.5 Концепция работы анализаторов

На основе проведенного обзора подходов к статическому анализу программ и существующих инструментов для проведения статического анализа можно выделить концепцию работы анализаторов. На рисунке 1.4 приведена диаграмма стандарта IDEF0, отражающая основной бизнес-процесс анализа программ. Как видно из диаграммы, входными данными для анализа является программа, представленная каким-либо образом. Выходными данными является отчет о найденных возможных ситуациях гонок. В качестве управляющих воздействий выступают некоторые правила анализа, а в качестве механизма, осуществляющего анализ — программное обеспечение в виде анализатора.

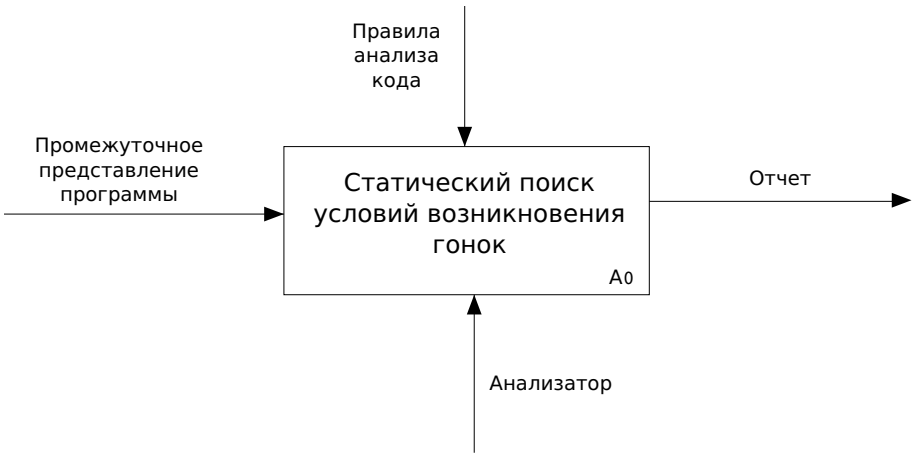

Рисунок 1.4 — Процесс поиска ситуаций гонок в программах

1.6 Выводы

Проведен обзор существующих методов статического поиска условий возникновения гонок в программах. Приведено описание существующих анализаторов программ для поиска гонок в программах. В результате проведения анализа существующих методов и программных реализаций статического поиска гонок выработана концепция работы анализаторов.

2 Выбор формы представления входных данных для анализа

В первой части данной главы описываются возможные формы представления входных данных для проведения анализа, приводится сравнение и выбор формы представления исходных данных в виде промежуточного представления программы на основе ассемблера какой-либо виртуальной машины для проведения анализа и проектирования программного обеспечения.

Во второй части данной главы приводится обзор виртуальных машин. Для определения используемой реализации промежуточного представления программы производится сравнение ассемблеров виртуальных машин Low Level Virtual Machine и Java Virtual Machine. В заключении главы приводится обоснование выбора ассемблера IR LLVM в качестве используемой реализации промежуточного представления программы.

2.1 Возможные формы представления программы

Для проведения статического поиска гонок программы в качестве возможных форм представления программы могут быть использованы следующие данные:

а) исходный код программы на каком-либо языке программирования;
б) промежуточное представление кода программы[1];
в) исполняемый код программы[2].

На рисунке 2.1 приведена классификация возможных форм представления программы.

[1]Здесь и далее под промежуточным представлением кода программы понимается текст программы на каком-либо ассемблерном языке, в том числе полученным путем дизассемблирования машинного кода программы

[2]Здесь и далее под исполняемым кодом программы понимается нетекстовое представление программы в виде машинного кода, в том числе кода виртуальных машин

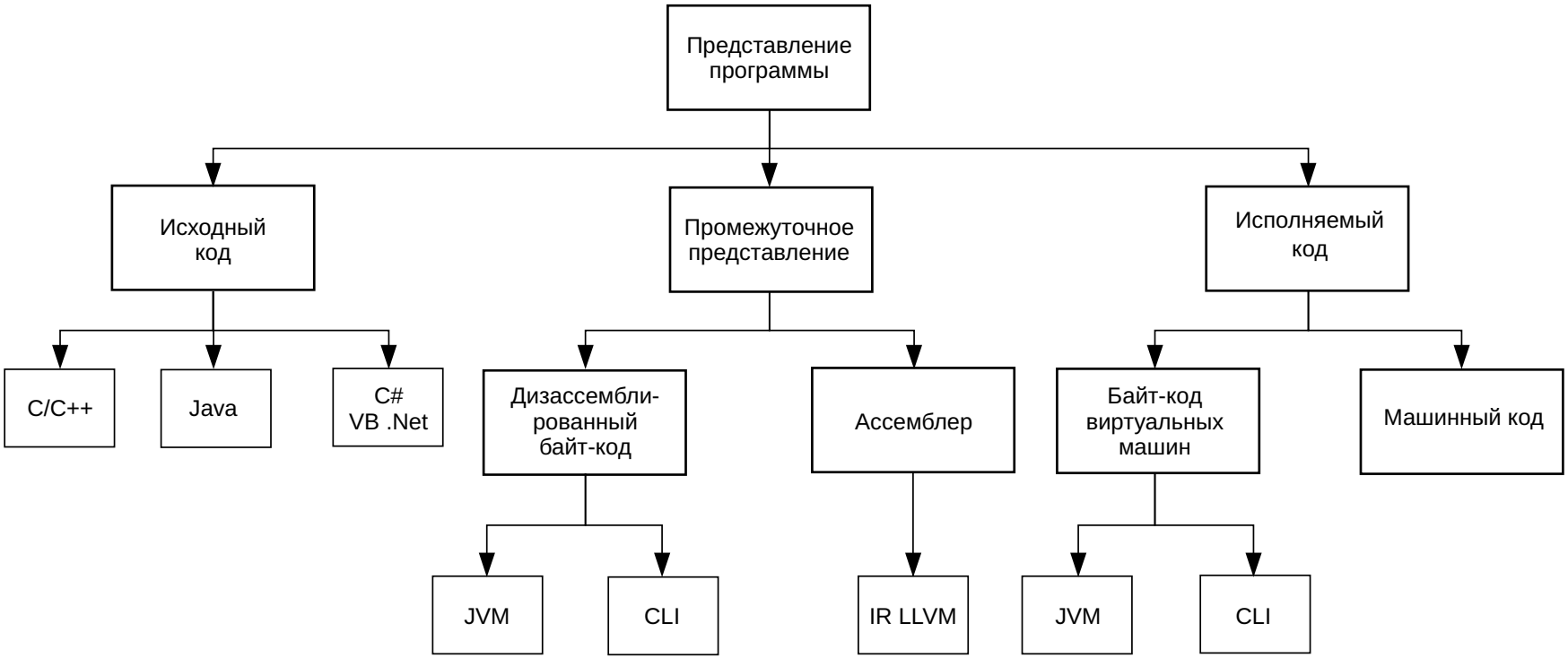

Рисунок 2.1 — Классификация возможных форм представления программы

Следует отметить, что промежуточное представление кода программы обычно используется различными виртуальными машинами, которые позволяют выполнять программы на различных программных и аппаратных платформах без внесения изменений в исходный код программы или, в некоторых случаях, без перекомпиляции.

Каждый из указанных вариантов имеет свои недостатки и достоинства. Сравнительная характеристика указанных форматов исходных данных приведена в таблице 2.1, обоснование указания оценок приведены в соответствующих разделах ниже.

Сравнение производилось по следующим критериям, важным с точки зрения достижения целей:

а) текстовое представление — является ли формат данных текстовым;

б) потенциальная информативность результатов — точность указания места программы, в котором возможно возникновение гонки относительно исходного кода программы;

в) сложность разбора кода в целом — сложность синтаксического анализа;

г) сложность выделения важных частей программы — сложность обнаружения обращений к переменным, создания, захвата и освобождения блокировок;

д) сложность построение ГПУ — сложность определения разветвлений потока управления, циклов, вызова функций;

е) сложность определения зависимости переменных — сложность определения связей между переменными и классификации этих связей.

Таблица 2.1 — Сравнение возможных форм представления исходных данных

Параметр	**Исходный код**	**Промежуточное представление**	**Исполняемый код**
Текстовое представление	Да	Да	Нет
Потенциальная информативность результатов	Высокая	Средняя	Низкая
Сложность разбора кода в целом	Высокая	Средняя	Низкая
Сложность выделения важных частей программы	Средняя	Средняя	Высокая
Сложность построения ГПУ	Средняя	Низкая	Низкая
Сложность определения зависимости переменных	Средняя	Низкая	Высокая

2.1.1 Исходный код программы

Исходный код программы является текстовым представлением программы на каком-либо высокоуровневом языке программирования. При разборе и анализе данного формата необходимо построение и использование синтаксических анализаторов класса LL, LR или LALR. Однако при использовании данного формата имеется возможность точного указания места возможного возникновения гонки, что позволяет программисту быстро найти и проанализировать указанную часть кода и внести необходимые правки.

С точки зрения построения ГПУ и графа зависимости данных сложность в общем случае сводится к сложности синтаксиса языка в целом. С учетом специфики высокоуровневых языков программирования, построение указанных графов не представляет особой сложности.

Пример исходного кода программы, содержащего гонку, приведен в листинге 2.1.

Листинг 2.1 — Пример исходного кода, содержащего гонку, на языке C

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  void *thread_main(void *args);
7
8  int main(int argc, char *argv[]) {
9      pthread_t thread;
10     int ctr = 0;
11     pthread_create(&thread, NULL, thread_main, &ctr);
12     for(int i = 0; i < 10000000; i++)
13         ++ctr; // race condition
14     pthread_join(thread, NULL);
15     return ctr != 20000000;
16 }
17
18 void *thread_main(void *args) {
19     int *ctr = (int *) args;
20     for(int i = 0; i < 10000000; i++)
21         ++*ctr; // race condition
22     return NULL;
23 }
```

2.1.2 Промежуточное представление кода программы

Промежуточное представление кода программы как правило является текстовым и имеют достаточно простой ассемблероподобный синтаксис. Указанная особенность достаточно сильно упрощает разбор кода программы, но, однако, не делает его тривиальным. Для разбора такого кода обычно бывает достаточно использования регулярных выражений.

При этом следует отметить, что информативность результатов анализа несколько ниже, так как в общем случае возможно указание имени функции, в которой возможна гонка, а так же переменная, доступ к которой может стать причиной гонки.

Пример промежуточного кода на языке IR LLVM приведен в приложении Б.

2.1.3 Исполняемый код программы

Двоичный код программы, очевидно, имеет нетекстовый вид, что значительно усложняет определение зависимостей между переменными, а так же выделение мест обращения как к переменным, так и к блокировкам. В остальном двоичное представление программы в общем случае эквивалентно промежуточному представлению кода.

2.2 Обоснование выбора формы представления входных данных

Исходя из проведенного анализа в качестве исходных данных для анализа наиболее подходящим является промежуточное представление кода программы. Анализ промежуточного представления не требует использования сложных средств синтаксического анализа, давая при этом возможность достаточно точно указать место возможного возникновения гонки.

Для выбора конкретной реализации промежуточного представления наиболее подходящим является выбор промежуточного представления кода программы для какой-либо виртуальной машины. Такой подход позволит производить анализ уже существующих программ, а так же не потребует разработки дополнительных инструментальных средств для преобразования исходного кода программы в промежуточное представление.

2.3 Классификация виртуальных машин

Существующие в настоящее время виртуальные машины можно разделить на два класса:

а) виртуальная машина, которая эмулирует аппаратное обеспечение ЭВМ и позволяет работать различным операционным системам под управлением другой операционной системы;

б) виртуальная машина, которая позволяет выполняться программам на разных программных и аппаратных платформах без внесения изменений в исходных код программы или её перекомпиляции.

Наиболее известными платформами виртуализации, эмулирующими аппаратное обеспечение, являются Qemu, Virtual Box, VMWare ESX(i). В силу выполняемых функций данные платформы не представляют интереса в рамках данной дипломной работы.

Ко второму классу относятся виртуальные машины Java Virtual Machine (JVM), Low Level Virual Machine (LLVM) и Common Language Infrastructure Virtal Machine (CLI VM). Основной задачей данных виртуальных машин является исполнение кода программы на каком-либо промежуточном языке. В состав таких виртуальных машин как правило входит JIT-компилятор (just-in-time), позволяющий производить компиляцию промежуточного кода в код конечной машины во время исполнения программы, или появляющиеся в настоящее время AOT-компиляторы (ahead-of-time), которые производят компиляцию кода при первом его выполнении (или до его выполнения) и кэшируют на диске для дальнейшего использования другими программами или при следующем запуске программы.

На рисунке 2.2 приведено графическое изображение классификации виртуальных машин.

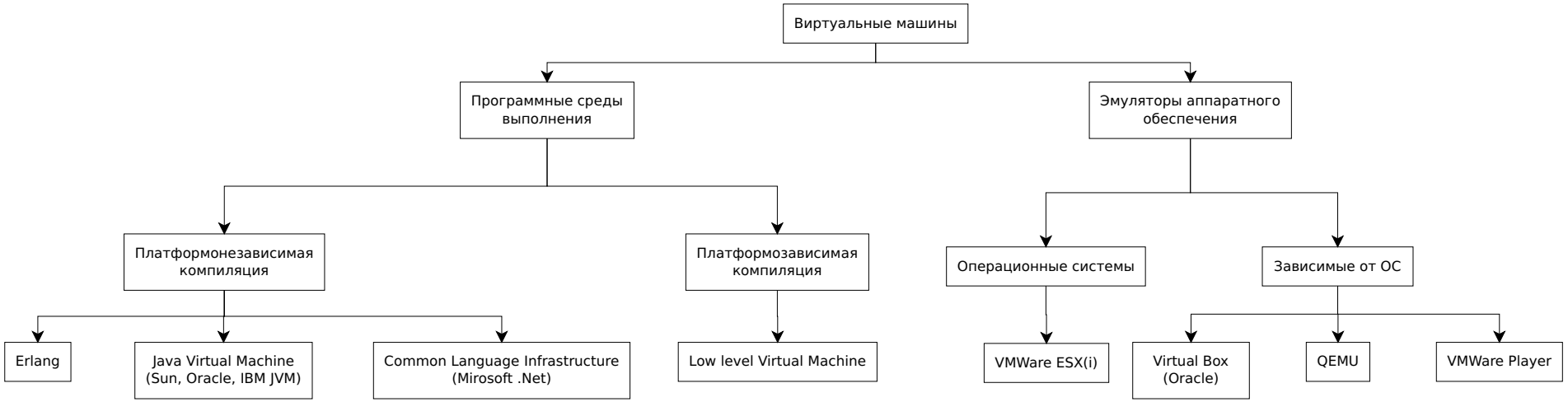

Рисунок 2.2 — Классификация виртуальных машин

В качестве виртуальных машин, возможных к использованию в рамках данной дипломной работы были выбраны Java Virtual Machine и Low Level Virtual Machine.

2.4 Сравнительный анализ промежуточных представлений программ

Исходный код программ приведен в приложении А, промежуточное представление кода на языке IR LLVM приведено в приложении Б, дизассемблированный байт-код Java — в приложении В.

В таблице 2.2 приведено сравнение промежуточного представления программы на языке IR LLVM и дизассемблированного байт-кода виртуальной машины Java.

Сравнение производилось по следующим параметрам:

а) сложность синтаксиса в целом;

б) сложность определения обращения к переменным;

в) сложность определения обращений к объектам блокировки;

г) потенциальная точность указания места программы, в котором возможна гонка;

д) размер промежуточного кода по отношению к исходному коду;

е) необходимость использования дополнительных средств для получения промежуточного представления.

Таблица 2.2 — Сравнение языка IR LLVM и байт-кода Java

Показатель	**LLVM IR**	**Байт-код Java**
Сложность синтаксиса в целом	Средняя	Низкая
Сложность определения обращения к переменным	Низкая	Высокая
Простота определения обращений к блокировкам	Низкая	Высокая
Потенциальная точность указания места программы, в котором возможна гонка	Средняя	Низкая
Размер промежуточного кода по отношению к исходному коду	увеличение в 3-4 раза	увеличение в 2 раза
Необходимость использования дополнительных средств для получения промежуточного представления	Возможно написания кода вручную, трансляция из исходного кода на других языках программирования	Необходимость компиляции и последующего дизассемблирования

Исходя их приведенного сравнения, байт-код Java имеет несколько преимуществ по сравнению с языком IR LLVM в виде меньших размеров кода и более простого синтаксиса кода. Однако в рамках решаемых задач IR LLVM имеет преимущества по ряду более значимых параметров, а именно:

а) простота определения обращений к переменным и блокировкам;
б) потенциальная точность указания мест возможных гонок.

Имеющиеся преимущества обеспечиваются в первую очередь за счет сохранения транслятором имен переменных и функций, что позволяет с легкостью определить момент захвата или освобождения блокировки, а так же указать программисту, на обращение к каким переменным в каком месте программы следует обратить внимание.

Следует отметить, что виртуальная машина Java является стековой, что влечет за собой необходимость отслеживания состояния стека в процессе анализа для определения зависимости между переменными.

Таким образом, в качестве реализации промежуточного представления кода программ наиболее подходящей реализацией с точки зрения простоты анализа является промежуточное представление программы на языке IR LLVM.

2.5 Обзор виртуальной машины Low Level Virtual Machine

Low Level Virtual Machine (LLVM) — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-подобными инструкциями. Байт-код может быть как статически скомпилирован для конкретных архитектур, так и использоваться для интерпретации или компиляции на целевой платформе.

LLVM является исследовательским проектом, начатым в университете Иллинойс (University of Illinois), основной целью которого является разработка современной стратегии компиляции на основе подхода SSA (Static Single Assigment), способной обеспечить статическую и динамическую компиляцию различных языков программирования.

LLVM позволяет компилировать программы написанные на языках C, C++, Objective-C, Fortran, Ada, Haskell, Java, Python, Ruby, JavaScript, GLSL или любом другом, для которого реализован front-end. В рамках проекта разработан front-end Clang для языков C и C++ и версия GCC, использующие LLVM в качестве back-end. В Glasgow Haskell Compiler также реализована компиляция посредством LLVM, существует ещё множество программ, использующих данную инфраструктуру.

2.6 Компиляция программ для виртуальной машины LLVM

В основе LLVM лежит идея трехфазной компиляции. Трехфазная компиляция представляет собой последовательность действий по преобразованию исходного кода программы в некоторое промежуточное представление, оптимизации промежуточного представления и генерация двоичного кода для целевой архитектуры процессора. Достоинством данного подхода является независимость процессов оптимизации и кодогенерации от исходного языка программирования, так как указанные действия производятся уже с промежуточным представлением. Вся последовательность действий схематично представлена на рисунке 2.3.

При таком подходе к процессу компиляции задача написания компилятора к новому языку программирования сводится к написанию соответствующего frontend'а, а не к написанию всего компилятора. Благодаря описанному подходу повышается переносимость программ между различными архитектурами процессов.

Следует отметить, что промежуточное представление программы по своей структуре заметно проще по сравнению с исходным кодом на высокоуровневом языке программирования, что в общем случае облегчает задачу анализа кода и не требует от программиста внесения какой-либо дополнительной информации в исходный код.

В настоящее время LLVM представляет собой множество проектов, основными из которых являются следующие:

а) ядро LLVM;

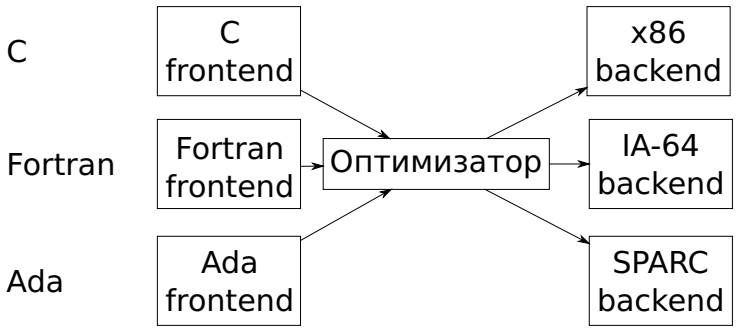

Рисунок 2.3 — Трехфазная компиляция с использованием промежуточного представления

б) компилятор Clang;

в) отладчик LLDB.

2.7 Ядро виртуальной машины LLVM

Ядро LLVM представляет собой набор библиотек и инструментов, предназначенных для оптимизации и кодогенерации для различных архитектур процессоров. Данные инструменты предназначены для работы с ассемблероподобным языком LLVM, известным так же как LLVM intermediate representation (LLVM IR). Краткий обзор этого языка приведен в разделе 2.8.

2.8 Язык IR LLVM

Язык LLVM IR является промежуточным представлением программы и используется ядром LLVM для оптимизации и последующей генерации двоичного кода для различных процессорных архитектур. В основе IR лежит RISC-подобный набор виртуальных команд. Рассмотрим основные конструкции языка, используемые при решении поставленных задач.

2.8.1 Классификация типов в языке IR LLVM

В IR LLVM используется классификация типов. приведенная в таблице 2.3.

Таблица 2.3 — Классификация типов IR LLVM

Класс	**Типы**
Целые числа	iN, где N – размер в битах
Числа с плавающей точкой	half, float, double, x86_fp80, fp128, ppc_fp128
Первый класс	integer, floating point, pointer, vector, structure, array, label, metadata
Примитивы	label, void, integer, floating point, x86mmx, metadata

Таблица 2.3 — Классификация типов IR LLVM

Класс	**Типы**
Унаследованные типы	array, function, pointer, structure, vector, opaque

Типы, принадлежащие к первому классу, являются наиболее важными, так как только они могут быть получены инструкциями IR [7].

В рамках данной работы рассматриваются только целочисленные типы и указатели. Это ограничение обусловлено тем, что использование прочих типов данных в общем случае можно смоделировать использованием целых чисел и указателей на целые числа.

2.8.2 Правила именования

В языке IR LLVM приняты следующие правила именования функций и переменных:

а) имя локальной переменной начинается с символа %, и следующих за ним букв латинского алфавита, цифр и нижнего подчеркивания _;

б) имя функции или глобальной переменной начинается с символа @, и следующих за ним букв латинского алфавита, цифр и нижнего подчеркивания _.

2.9 Основные инструкции языка IR LLVM

В данном разделе приведены инструкции языка IR LLVM, которые необходимо, полный перечень инструкций и более подробное описания синтаксиса можно найти в [7].

Арифметические операции в языке IR LLVM

В IR LLVM существует набор базовых арифметических операций и представлен следующими инструкциями:

а) add, sub, mul — инструкции сложения, вычитания, умножения целых чисел;

б) udiv, sdiv — инструкции знакового и беззнакового целочисленного деления;

в) urem, srem — инструкции знакового и беззнакового целочисленного получения остатка от деления;

г) fadd, fsub, fmul — инструкции сложения, вычитания, умножения чисел с плавающей точкой;

д) fdiv, frem — инструкции деления и получения остатка от деления чисел с плавающей точкой.

Данные инструкции имеют следующий формат:

<result> = <oper> [opt] <type> <op1>, <op2>, где:

<result> — переменная, в которую помещается результат операции;
<oper> — инструкция;
[opt] — nuw, nsw или отсутствует;
<type> — тип результата;
<op1>, <op2> — операнды.

Вызов функций

Вызов функций в LLVM IR производится с помощью инструкции **call** и выглядит следующим образом:

<result> = call <function> <args>, где:

<result> — переменная, в которую помещается результат выполнения функции;
<function> — имя функции;
<args> — параметры вызова функции.

В случае, если функция не возвращает результат или результат функции не используется, то часть инструкции « <result> =» может быть опущена.

Параметры вызова функции передаются в следующем виде:

<type> <param>, где <type> — тип параметра, <param> — переменная или константа.

Возврат из подпрограммы выполняется посредством инструкции **ret**:

ret <type> <value>, где <type> — возвращаемый тип, <value> возвращаемое значение.

В случае, если функция не возвращает результат, то инструкция **ret** используется следующим образом: **ret void**.

Инструкции для работы с памятью

Для работы с памятью используются следующие инструкции:

а) alloca — выделение памяти;
б) load — чтение памяти;
в) store — запись в память.

Инструкция **alloca** предназначена для выделения памяти для переменной и выглядит следующим образом:

<variable> = alloca <type>, где <variable> — имя переменной, <type> — тип переменной.

При выполнении инструкции в переменную сохраняется указатель на выделенный участок памяти. Для доступа к выделенному участку памяти используются инструкции **load** (чтение) и **store** (запись). Использование этих инструкций выглядит следующим образом:

store <ptr> <variable>

<result> = load <ptr> <variable>, где

<result> — переменная, в которую сохраняется результат;

<variable> — адрес, по которому производятся операции доступа к памяти;

<ptr> — тип <variable>, указатель.

Инструкции для преобразования типов

Для преобразования типов используются следующие инструкции:

а) bitcast;
б) [zs]ext;
в) fptrunc;
г) fpext;
д) fpto[us]i;
е) [us]itofp;
ж) ptrtoint;
з) inttoptr.

Синтаксис этих инструкций выглядит следующим образом:

<result> = <instr> <type> <variable>, где:

<result> — переменная, в которую сохраняется результат;

<instr> — инструкция;

<type> — тип, к которому производится преобразование;

<variable> — переменная, из которой берется значение.

Метки

В IR LLVM метка может стоять перед большинством инструкций; объявление метки производится следующим образом: **<label>:**, где <label> — имя метки, уникальное в рамках функции, в которой она объявлена.

Инструкции переходов

В IR LLVM существует два типа инструкций переходов — инструкции условного и безусловного перехода. Для переходов используется инструкция **br**, синтаксис инструкции различается для типа перехода.

Для безусловного перехода синтаксис инструкции выглядит следующим образом:

br <label>, где <label> — метка, на которую осуществляется переход.

Для условного перехода синтаксис команды следующий:

br <type> <cond> label <true lbl>, label <false lbl>, где

<type>	—	тип условной переменной, как правило **i1**;
<cond>	—	переменная, значение которой проверяется;
<true lbl>	—	метка, на которую осуществляется переход, если условие истинно;
<false lbl>	—	метка, на которую осуществляется переход, если условие ложно.

Для определения истинности или ложности условия используется логика, аналогичная языку C: если значение переменной больше 0, то условие истинно, иначе ложно.

Следует заметить, что в IR LLVM нет отдельных инструкций для циклов и организация циклов осуществляется с помощью меток и инструкций переходов.

2.10 Формирование входных данных

В силу того, что IR LLVM является ассемблерным языком, формирование исходных данных непосредственно в виде промежуточного кода является достаточно трудоемкой задачей. Однако LLVM предоставляет возможность трансляции программ, написанных на языках программирования высокого уровня, в код IR LLVM, в связи с чем для формирования исходных данных было решено использовать язык C с дальнейшей трансляцией текста программ в IR LLVM.

2.11 Выводы

В качестве формы входных данных выбрано промежуточное представление кода программы. В качестве реализации промежуточного представления принято решение использовать промежуточное представление кода программы виртуальной машины. В качестве используемого промежуточного кода программы выбран ассемблер IR LLVM. Так же проведен обзор виртуальной машины LLVM и выделены инструкции IR LLVM, необходимые для проведения анализа кода программ с целью

поиска возникновения условий гонок. Для формирования входных данных было решено использовать язык C с последующей трансляцией в IR LLVM.

3 Метод поиска условий возникновения гонок

В данной главе приведено описание ограничений, накладываемых на анализируемые программы, описание основных процессов поиска возможных мест возникновения гонок. Так же в данном разделе приводится описание алгоритмов, разработанных для достижения поставленной цели.

3.1 Ограничения, накладываемые на анализируемые программы

В силу существования большого числа различных API для работы с потоками, средствами взаимоисключения возникает необходимость определения средств, с помощью которых будут выполняться те или иные действия.

Использование Posix API

Ограничения по использованию Posix API в анализируемых программах накладываются в части:

а) создания и завершения потоков;
б) создания и взаимодействия с объектами синхронизации.

Создание и завершение потоков программы должно производиться посредством функций библиотеки **pthread**, в частности вызовом функции **pthread_create, pthread_cancel, pthread_exit**. Для создания и взаимодействия с объектами синхронизации необходимо использовать следующие функции:

а) **sem_init** — инициализация семафора;
б) **sem_wait** — ожидание освобождения и захват семафора;
в) **sem_post** — освобождение семафора;
г) **pthread_mutex_init** — инициализация мьютекса;
д) **pthread_mutex_lock** — захват мьютекса;
е) **pthread_mutex_unlock** — освобождения мьютекса.

Следует отметить, что использование функции **pthread_join** игнорируется, так как для определения ожидаемого потока требуется информация времени выполнения, которая недоступна в процессе статического анализа кода.

Требование к использованию Posix API и, в частности, указанных функций предъявляется в силу необходимости определения функций, приводящих к созданию новых потоков и объектов синхронизации. Из данного требования вытекает требование вызова указанных функций в явном виде.

Использование функций Posix API для захвата объектов блокировки (функции pthread_mutex_lock и pthread_mutex_unlock) так же приводит к неявному ис-

пользованию барьеров памяти [8], что позволяет избежать возникновение ситуаций гонок на уровне процессора в силу переупорядочивания операций доступа к памяти.

Использование семафоров для организации критических секций

Требование к использованию семафоров только для организации критических секций обусловлено тем, что семафоры в силу своих свойств могут быть использованы для организации достаточно сложных дисциплин взаимоисключения потоков, в том числе таких, которые не могут быть корректно обработаны при статическом подходе к анализу программ.

Использование указателей первого уровня

Требование использования указателей первого уровня выдвигается в силу того, что использование указателей более высокого уровня приводит к значительному усложнению определения зависимых переменных, а вместе с тем к усложнению определения мест программ с возможным возникновением гонок.

Использование скалярных типов данных

Требование использование атомарных типов данных обусловлено тем, что использование сложных структур данных в общем случае сводится к использованию членов этих структур, имеющих скалярный тип.

Использование косвенной адресации

Наложение ограничения на способы задания адресов в программе обусловлено тем, что в случае задания адресов в явном виде или при их расчете с использованием арифметических операций в рамках статического анализа недостаточно информации для определения зависимостей между переменными, что может повлечь за собой потерю части ситуаций гонок, так как информация о конкретных адресах различных переменных является информацией времени выполнения программы.

Явное определение указателей для аргументов создания потоков

Данное ограничение накладывается на программы с целью упрощения алгоритма разбора промежуточного представления программы и не влечет за собой уменьшение числа обрабатываемых ситуаций гонок.

Отсутствие рекурсивных функций

Данное ограничение накладывается в виду необходимости разворачивания кода функций и последующего встраивания в код функций потоков.

Полный перечень требований, предъявляемых к программам

Таким образом, к проверяемым программам предъявляются следующие требования:

а) использование Posix API:

б) использование только явных вызовов функций, т.е. без использования указателей на функции;

в) использование семафоров для организации критических секций;

г) использование только скалярных типов данных;

д) использование указателей первого уровня;

е) использование косвенной адресации для указателей;

ж) для передачи указателей в виде аргумента функции создания потока необходимо явное определение этого указателя в виде переменной;

з) отсутствие рекурсивных функций;

и) создание новых потоков внутри циклов должно производиться вне ветвлений.

Следует отметить, что часть из приведенных ограничений являются ограничениями реализации.

3.2 Описание разрабатываемого метода поиска условий возникновения гонок

За основу разрабатываемого метода статического поиска условий возникновения гонок был взят метод анализа графа потока управления программы как наиболее гибкий из всех рассмотренных методов.

Диаграмма основного бизнес-процесса приведена на рисунке 1.4. Как видно из диаграммы, в качестве входных данных используется промежуточное представление кода программы, в качестве выходных данных — отчет о найденных местах программы, в которых возможно возникновение гонок. В качестве управляющих воздействий используются правила анализа кода.

На рисунке 3.1 представлена диаграмма, отражающая основные этапы разрабатываемого метода поиска гонок. Поиск условий возникновения гонок в программе состоит из нескольких основных этапов:

а) построение базовых ГПУ;

б) построение и линеаризация ГПУ потоков программы;

в) анализ линейных ГПУ потоков программы;

г) анализ использования разделяемых переменных.

В качестве внутреннего представления кода программы используется множество линейных ГПУ потоков программы.

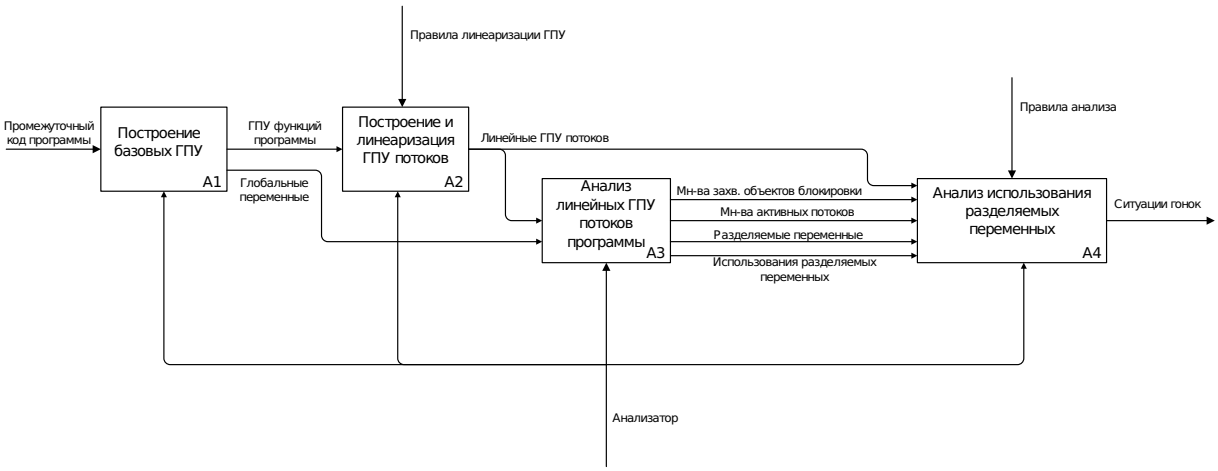

Рисунок 3.1 — Метод статического поиска условий возникновения гонок

Процесс поиска условий возникновения гонок в программе включает в себя несколько этапов:

а) построение базовых графов потока управления;
б) построение и линеаризация графов потока управления потоков программы;
в) анализ линейных ГПУ;
г) анализ использования разделяемых переменных.

При построении базовых ГПУ программы производится построение ГПУ функций программы, а так же определение глобальных переменных программы.

В силу того, что объявление глобальных переменных производится вне функций программы, возникает необходимость в дополнительном шаге, основной задачей которого будет определение глобальных переменных. Необходимость определения глобальных переменных вызвана в первую очередь тем, что глобальные переменные часто являются разделяемыми переменными. В силу того, что областью видимости глобальных переменных в общем случае является вся программа или отдельный модуль программы, ситуации гонок, возникающие при обращении к ним, являются достаточно частыми.

Построение графов потоков управления функций программы необходимо для подготовки исходных данных для построения как множества потоков программы в целом, так и для построения ГПУ потоков управления программы.

Функции программ состоят из линейных участков кода, то есть участков кода, не содержащих ветвления, и непосредственно ветвлений, то для построения ГПУ функции необходимо построить графы потоков управления всех её линейных

частей и затем объединить получившиеся подграфы на основе логики условных или безусловных переходов данной функции.

Таким образом, этап построения графов управления функций программы состоит из следующих шагов:

а) построение графов потоков управления линейных участков функций;

б) объединение множества линейных участков функций в графы потоков управления функций.

На рисунке 3.2 приведена диаграмма детализированного процесса обработки промежуточного представления программы.

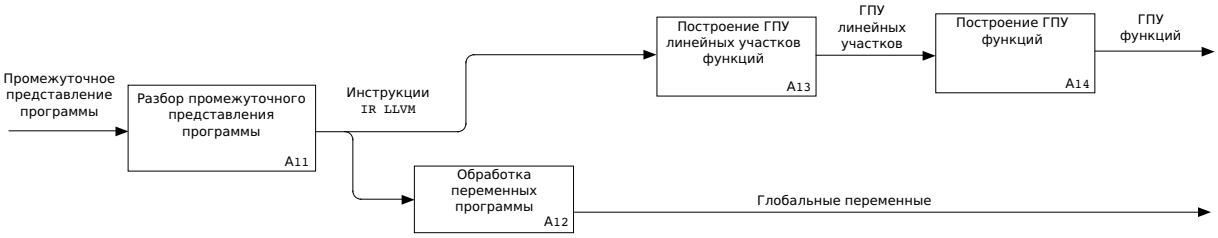

Рисунок 3.2 — Детализированный процесс обработки промежуточного представления программы

Построение и линеаризация ГПУ потоков программы состоит из трех основных этапов:

а) построение ГПУ потоков;

б) разворачивание вызовов функций;

в) линеаризация ГПУ.

На рисунке 3.3 представлена диаграмма, отражающая шаги построения и линеаризации ГПУ потоков программы.

Построение ГПУ потоков программы и их дальнейшая линеаризация необходима для более точного определения множеств захваченных объектов синхронизации в процессе анализа внутреннего представления программы.

Шаг построения ГПУ потоков программы необходим для определения всего множества потоков, которые могут быть созданы в программе, а так же для приведения ГПУ к необходимому для линеаризации виду. Для построения указанных ГПУ используется этап разворачивания вызовов функций, за счет которого производится встраивание ГПУ функций в основной ГПУ потока.

На шаге линеаризации производится построение возможных путей выполнения потока программы на основе соответствующего ГПУ.

На диаграмме 3.4 представлены основные шаги анализа линейных ГПУ потоков программы.

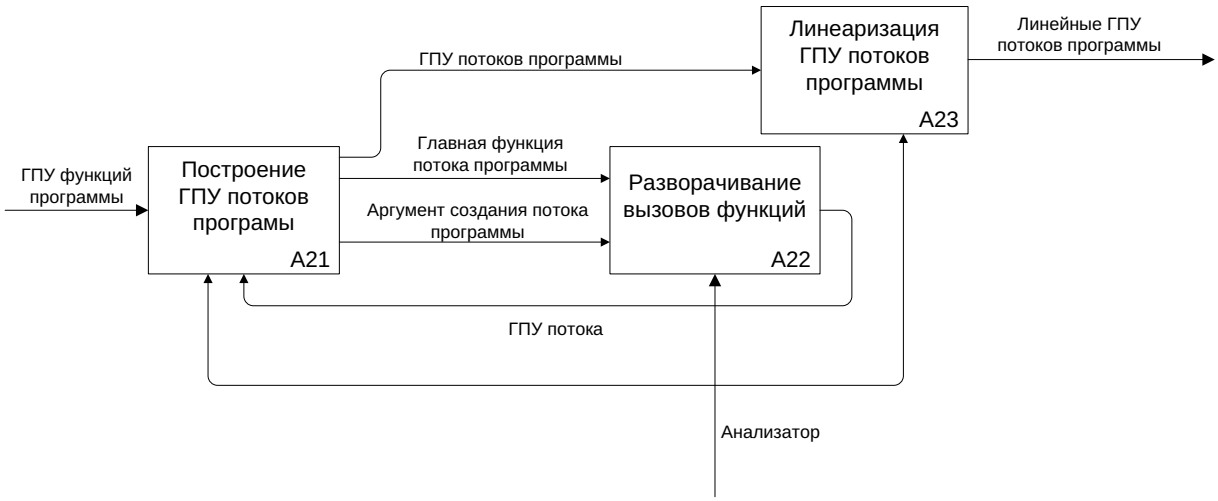

Рисунок 3.3 — Построение и линеаризация ГПУ потоков программы

Анализ линейных ГПУ программы состоит из следующих шагов:

а) построение множеств захваченных объектов блокировки;
б) построение частичных множеств активных потоков;
в) построение таблиц указателей;
г) поиск использований разделяемых переменных.

Построение множеств захваченных объектов блокировки является одним из главных этапов разрабатываемого метода. Построение этих множеств позволяет определить объекты блокировки, которыми защищен доступ к тем или иным переменным в том или ином потоке, и доступ к каким переменным в различных потоках может привести к возникновению ситуации гонки.

Построение множеств активных потоков необходимо для повышения точности определения ситуаций гонок. На основе данных множеств можно определить потоки, которые могут выполняться параллельно с определенным потоком в определенном месте его выполнения и могут получить доступ к каким либо разделяемым переменным.

Построение таблиц указателей необходимо для определения переменных, к которым производятся обращения через указатели в процессе выполнения программы. Эта информация так же необходима для определения переменной, указатель на которую передан в качестве аргумента функции создания потока, или же указателя, использование которого в итоге приведет к обращению к какой-либо разделяемой переменной.

Поиск мест использования разделяемых переменных необходимо для определения множеств ситуаций, в которых возможно возникновение гонок. На основе найденных множеств производится анализ, задачей которого является определение

Рисунок 3.4 — Анализ линейных ГПУ

во всем множестве возможных ситуаций гонок таких ситуаций, при которых возможно возникновение гонки в процессе выполнения программы.

На последнем этапе производится анализ использования разделяемых переменных, в рамках которого на основе собранной на предыдущих этапах информации производится определение ситуаций гонок.

3.3 Построение ГПУ функций программы

Построение графов потока управления функций программы, как отмечалось ранее, состоит из двух этапов:

а) построение линейных частей ГПУ функции;

б) связывание построенных графов потока управления линейных частей функции в единый ГПУ функции.

Построение ГПУ производится на основе обработки инструкций IR LLVM, для каждой инструкции применяются определенные действия, приведенные в таблице 3.1. При выполнении каждого действия так же производится связывание вновь созданного блока и последнего добавленного, если не указано иное.

При построении графов потоков управление используется понятие контекста. Под контекстом в данном случае понимается функция, в теле которой производится обработка инструкций IR LLVM.

Таблица 3.1 — Действия, выполняемые при обработке инструкций IR LLVM

Инструкция	**Действие**
x = load y x = <conv> y	Добавить в текущий граф блок операции чтения переменной y.
store x, y	Добавить в текущий граф блок операции записи переменной y.
Безусловный переход	Добавить блок безусловного перехода в текущий граф, установить этот блок точкой выхода графа.
Условный переход	Добавить блок условного перехода в текущий граф, установить этот блок точкой выхода графа.
Определение функции	Создать новый граф, добавить в него блок безусловного перехода на метку «entry». Связывания с последним блоком не производится.
Вызов функции	Действия указаны в таблице 3.2.
Метка	Создать блок объявления метки, создать новый граф, установить созданный блок его точкой входа. Установить новый контекст построения графа. Связывания с последним блоком не производится.
Инструкция арифметической операции	Добавить в текущий граф блок чтения операнды инструкции.
Инструкция сравнения	Добавить в текущий граф блок чтения операнды инструкции.
Инструкция возврата управления	Создать блок возврата управления. Установить созданный блок точкой выхода текущего графа и графа текущей функции.

Так как существует необходимость определения вызова функций Posix API, инструкции вызова функции обрабатываются особым образом. В зависимости от вызываемой функции производятся определенные действия, перечень которых приведен в таблице 3.2.

Таблица 3.2 — Действия, выполняемые при обработке инструкций вызова функции

Функция	**Действие**
pthread_create	Создать блок создания потока
sem_init pthread_mutex_init	Создать блок инициализации блокировки
sem_post pthread_mutex_unlock	Создать блок освобождения блокировки

Таблица 3.2 — Действия, выполняемые при обработке инструкций вызова функции

Функция	**Действие**
sem_wait pthread_mutex_lock	Создать блок захвата блокировки
Прочие функции	Создать блок вызова функции

После построения ГПУ линейных частей функций производится их связывание. Для связывания блока безусловного перехода производятся следующие действия:

а) поиск графа для метки перехода;

б) добавление связи между блоком перехода и блоком объявления метки;

в) копирование всех связей и блоков из графа для метки в граф блока перехода.

При связывании блоков условного перехода производятся аналогичные действия с той лишь разницей, что между блоком перехода и блоком метки добавляется связь со значением условия, при котором осуществляется переход.

Результатом работы данного алгоритма является множество графов потоков управления для всех функций программы.

3.4 Построение ГПУ потоков

Входными данными для построения ГПУ потоков является множество графов потоков управления функций программы. Блок-схема алгоритма построения ГПУ потоков программы приведена на рисунке 3.5.

Как видно из приведенной блок-схемы, построение ГПУ потоков управления программы производится в цикле, который выполняется до тех пор, пока в программе обнаруживаются новые блоки создания потоков. Построение начинается с главной функции программы, отмеченной на блок-схеме как функция **main**. Цикл состоит из следующих шагов:

а) из списка блоков создания потока извлекается очередной блок **CUR**;

б) блок **CUR** добавляется в множество обработанных блоков **TS_USED**;

в) для блока **CUR** делается копия ГПУ главной функции потока и помещается в **GR**;

г) производится рекурсивное разворачивание вызовов функций в **GR**;

д) построенный ГПУ **GR** помещается в результирующий список **RESULT**;

е) в **GR** находятся все блоки создания потоков и помещаются в список **TS_TEMP**;

ж) из множества блоков создания **TS_TEMP** в **TS_NEW** помещаются все блоки создания потоков, которые еще не были обработаны.

Проверка на последнем шаге в цикле необходима для избежания зацикливания алгоритма в случае рекурсивного создания потоков.

Таким образом, результатом данного алгоритма является множество ГПУ потоков управления программы.

3.5 Способ разворачивания вызовов функций

Входными данными для алгоритма разворачивания вызовов функций является граф потока управления, в котором необходимо развернуть вызовы функций, множество ГПУ функций программы и таблицы указателей для этих функций.

Алгоритм разворачивания вызовов функций состоит из нескольких шагов. На первом шаге в графе потока управления ищутся блоки вызова функции. Затем, для каждого найденного вызова функции производится рекурсивное разворачивание вызова функций. По окончании процесса рекурсивного разворачивания производятся следующие действия:

а) ребро ГПУ, входящее в блок вызова функции, заменяется ребром, входящим в точку входа в функцию;

б) ребро ГПУ, выходящее из блока вызова функции, заменяется ребром, выходящим из точки выхода функции.

На рисунке 3.6 представлена последовательность состояний ГПУ в процессе разворачивания вызовов функций в программе. На рисунке обозначены следующие состояния графов потоков выполнения:

— А — исходное состояние ГПУ;

— Б — состояние ГПУ после рекурсивного разворачивания вызовов функций в функции **F1**;

— В — состояние ГПУ после разворачивания вызовов функций в функции **F**.

В том случае, если для вызываемой функции отсутствует соответствующий граф потока управления, то блок вызова функции удаляется. Возникновение подобных ситуаций связано с вызовом библиотечных функций, код которых недоступен в момент проведения анализа.

При разворачивании вызовов функции так же осуществляется замена всех формальных параметров функции на фактические, то есть каждый параметр функции заменяется на соответствующую переменную функции, из которой происходит вызов. Так же производится слияние таблиц указателей родительской и разворачиваемой функции с заменой формальных параметров функций и порожденных ими временных значений.

Таким образом, результатом работы данного алгоритма является граф потока управления, не содержащий блоков вызова функции.

3.6 Линеаризация графа потока управления

Входными данными для алгоритма линеаризации является граф потока управления.

Линеаризация графа потока управления основана на алгоритме обхода графа в глубину. Обход ГПУ начинается с входной вершины и продолжается до блока условного перехода. При проходе каждой вершины в текущий путь выполнения потока добавляется пройденная вершина и соответствующее пройденное ребро графа.

При достижении блока условного перехода, который имеет два выходных ребра, создается копия пути выполнения. В данную копию добавляется блок безусловного перехода на одну из меток достигнутого условного перехода, после чего запускается рекурсивное выполнение данного алгоритма, но уже из блока объявления выбранной метки. После обработки первой ветви ГПУ, аналогичным образом обрабатывается вторая ветвь.

Полученные в результате обхода пути выполнения добавляются в результирующее множество путей выполнения ГПУ.

В процессе обхода дерева ГПУ формируется множество посещенных вершин. В том случае, если при обходе дерева была достигнута уже посещенная вершина, это означает, что в рассматриваемом ГПУ есть цикл.

При обнаружении цикла производится проверка достижимости выходной вершины ГПУ из текущей и построение пути, по которому она может быть достигнута. Проверка достижимости и построение пути производится на основе обхода графа в ширину.

Каждый найденный цикл может быть преобразован в один или два пути выполнения. При обнаружении цикла в результирующее множество путей выполнения добавляется путь выполнения с бесконечным циклом, то есть в текущий путь выполнения добавляется связь между текущей вершиной и посещенной. На рисунке 3.7 представлена блок-схема описанного выше алгоритма.

В том случае, если существует путь, по которому может быть достигнута выходная вершина ГПУ, то выполняется алгоритм, приведенный в листинге 3.1. Приведенный алгоритм предполагает, что вершины в найденном пути обхода упорядочены в соответствии с путем обхода ГПУ.

Листинг 3.1 — Составная часть алгоритма линеаризации ГПУ для обработки пути выхода из цикла

```
1 while(outbound_count(cv) == 1) {
2     put(cgpu, cv0);
3     cv = next(cv0, path);
4     if(is_conditional(cv)) {
5         if(is_visited(cv))
6             cv = to_branch(cv);
```

```
7           else {
8               remove(cv, visited);
9               tmp = recursive(cv);
10              put(tmp, result);
11              return result;
12          }
13      } else
14          cv0 = cv;
15  }
```

В листинге 3.1 использованы следующие условные обозначения:

а) переменные:
   1) cgpu — текущий путь выполнения программы;
   2) cv0 — исходная вершина;
   3) cv — текущая вершина;
   4) path — найденный путь;
   5) visited — множество посещенных вершин;
   6) result — множество результирующих путей выполнения программы;

б) функции:
   1) put — добавление вершины в список;
   2) next — получение следующей вершины;
   3) is_conditional — проверка, является ли вершина условным переходом;
   4) is_visited — проверка, находится ли вершина в списке посещенных;
   5) to_branch — преобразование блока условного перехода в блок безусловного перехода;
   6) remove — удаление вершины из списка;
   7) recursive — рекурсивный запуск алгоритма линеаризации.

Таким образом, результатом данного алгоритма является множество путей выполнения ГПУ, то есть множество ГПУ, не содержащих ветвлений.

Следует заметить, что данный алгоритм не обнаруживает все возможные пути выполнения, которые могут возникнуть в циклах. Так, могут быть не построены такие линейные ГПУ, охватывающие обе ветви условных переходов, в том случае, если путь, по которому может быть достигнута конечная вершина графа, не проходит через альтернативную ветвь перехода.

3.7 Построение таблицы указателей

Входным данным для построения таблицы указателей является линейный ГПУ потока программы и таблица указателей потока-родителя в момент создания обрабатываемого потока. Так же входная таблица должна быть дополнена строкой,

содержащей отображение аргумента создания потока, в случае если таковой передается потоку.

Построение таблицы указателей необходимо для поддержания актуальности зависимостей между переменными. Построение производится на основе обработки блоков ГПУ, соответствующих некоторым инструкциям IR LLVM и связанных с выполнением операций над указателями. При обработке в зависимости от инструкции выполняются различные действия, приведенные в таблице 3.3. В таблице с помощью x и y обозначены переменные.

Таблица 3.3 — Действия, выполняемые при обработке блоков ГПУ

Инструкция IR LLVM	**Действие**
$x = \text{load } y$	Если y указатель второго и более уровня, то добавить связь между x и значением указателя в y
store x, y	Если y указатель второго и более уровня, то добавить связь между y и x
$x = <\text{conv}^1> y$	Если x указатель, то добавить связь между x и текущим значением y

В том случае, если при разыменовании указателя (то есть при получении текущего значения по указателю) значение еще не определено, то производится дополнение таблицы временным значением, которое будет заменено на одном из следующих этапов метода.

3.8 Построение множества активных потоков

Входными данными для алгоритма является множество линейных ГПУ потоков программы.

Построение множеств активных потоков основано на обходе графа в глубину. Данный алгоритм производит построение множеств активных потоков только на основе потоков, создаваемых обрабатываемым ГПУ. Все прочие потоки, которые могут быть созданы родительскими потоками, будут рассмотрены уже в процессе поиска возможных мест возникновения гонок.

Построение множеств активных потоков только для рассматриваемого ГПУ обусловлено тем, что данный поток может быть создан в различных частях программы и множества потоков, созданных потоками-предками, может быть различным.

Алгоритм построения множеств активных потоков заключается в следующем: пока в процессе поиска активных потоков обнаруживаются новые активные потоки для каких либо блоков какого либо пути выполнения из множества входных

[1] Операция преобразования типов

ГПУ производится выполнение следующих действий для каждого блока создания потока в каждом ГПУ:

а) к текущему множеству активных потоков добавить создаваемый поток;

б) к текущему множеству активных потоков добавить все потоки, создаваемые добавленным потоком.

В процессе построения данных множеств производятся некоторые другие проверки:

а) если создаваемый поток уже находится в списке активных потоков, то есть был создан ранее, то данный поток помечается параллельным самому себе;

б) если в множестве потоков, создаваемых добавленным потоком, находится текущий поток, то все блоки, следующие за данным, помечаются параллельными самим себе.

В процессе построения множеств на каждом шаге к каждому блоку ГПУ добавляется информация о найденных активных потоках.

Таким образом, результатом выполнения данного алгоритма являются построенные множества активных потоков для всех ГПУ, а так же метки частичной или полной параллельности ГПУ самому себе.

3.9 Построение множеств захваченных объектов блокировки

Входными данными для алгоритма построения множеств блокировки является линейный ГПУ.

Построение множеств захваченных объектов блокировки основан на обходе графа в глубину. При выполнении обхода ГПУ в зависимости от типа блока выполняются различные действия, список которых приведен в таблице 3.4.

Таблица 3.4 — Действия, выполняемые при построении множеств захваченных блокировок

Тип блока	**Действие**
Блок создания объекта блокировки	Если объект блокировки создается в захваченном состоянии, то добавить его в текущее множество блокировок.
Блок захвата объекта блокировки	Добавить объект блокировки в текущий список блокировок.
Блок освобождения объекта блокировки	Удалить объект блокировки из текущего списка блокировок.
Прочие блоки	Добавить к блоку информацию о текущем множестве захваченных блокировок.

Результатом работы данного алгоритма является линейный ГПУ, содержащий информацию о множествах захваченных объектов блокировки для каждого блока.

3.10 Анализ графа потока управления

Входными данными для алгоритма является множество линейных ГПУ потоков приложения с метками активных потоков и множеств захваченных объектов блокировки, таблицы указателей, глобальные переменные и переменные, передаваемые в качестве аргументов потоков.

Анализ графа потока управления для поиска гонок при обращении к какой-либо переменной состоит из нескольких шагов:

а) поиск обращений к переменной во всех линейных ГПУ потоков;

б) попарный анализ найденных обращений к переменной с целью обнаружения возможности выявления гонки:

1) анализ множеств захваченных блокировок;

2) анализ возможности выполнения операции в параллельных потоках.

Анализ множеств блокировок заключается в поиске общих захваченных объектов блокировки для каждого из обращений, то есть производится пересечение этих множеств. Если полученное пересечение не пусто, то возникновение гонки в рассматриваемой ситуации невозможно, дальнейший анализ не требуется.

В том случае, если пересечение множеств пусто, то производится анализ возможности выполнения операций в параллельных потоках. Если обе операции выполняются в рамках одного и того же пути выполнения потока или одного и того же потока в целом, то анализируется состояние флагов параллельности и частичной параллельности. Если путь выполнения или операции помечены соответствующими флагами, то делается вывод о том, что в рассматриваемой ситуации возможно возникновение гонки.

Если операции производятся в рамках разных потоков, то осуществляется построение полного множества активных потоков для рассматриваемых обращений. Для построения полных множеств потоков для каждого обращения осуществляется поиск всех возможных путей выполнения программы, которые приводят к созданию рассматриваемых потоков. После определения множеств родительских потоков для рассматриваемой ситуации производится проверка, содержит ли множество активных потоков одного обращения поток другого обращения. В том случае, если условие выше выполняется для обоих обращений, то делается вывод о том, что данная ситуация является возможной ситуацией гонки в программе. Если же условие не выполняется, то делается вывод о том, что возникновение ситуации гонки маловероятно и описываемая ситуация не рассматривается.

Результатом работы данного алгоритма является множество ситуаций гонок, которые возможны в процессе выполнения программы.

3.11 Выводы

Описан и спроектирован метод статического поиска условий возникновения гонок. Описаны и спроектированы вспомогательные алгоритмы построения таблиц указателей, графов потоков управления программы, линеаризации ГПУ.

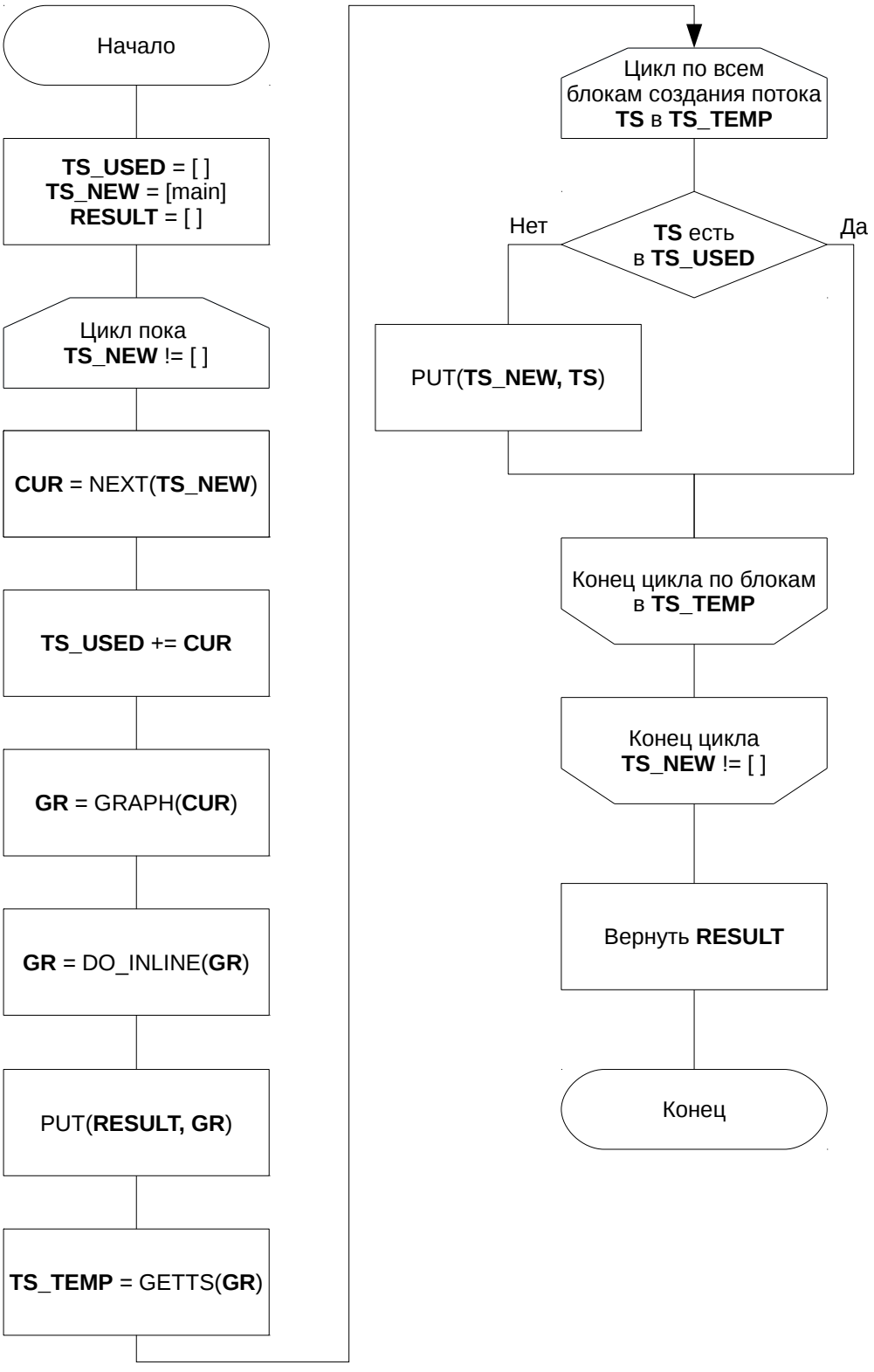

Рисунок 3.5 — Блок-схема алгоритма построения ГПУ потоков управления программы

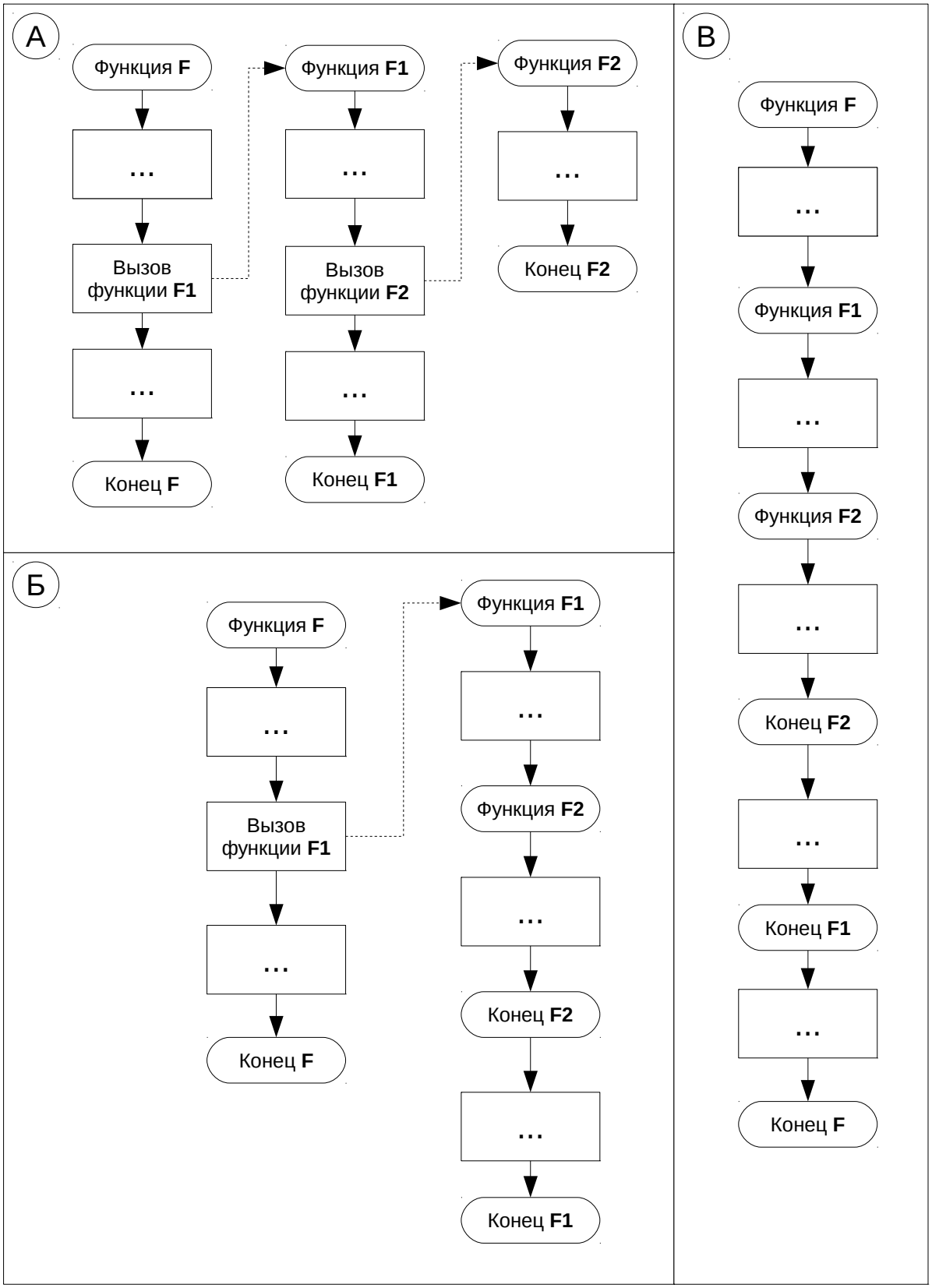

Рисунок 3.6 — Порядок разворачивания вызовов функций

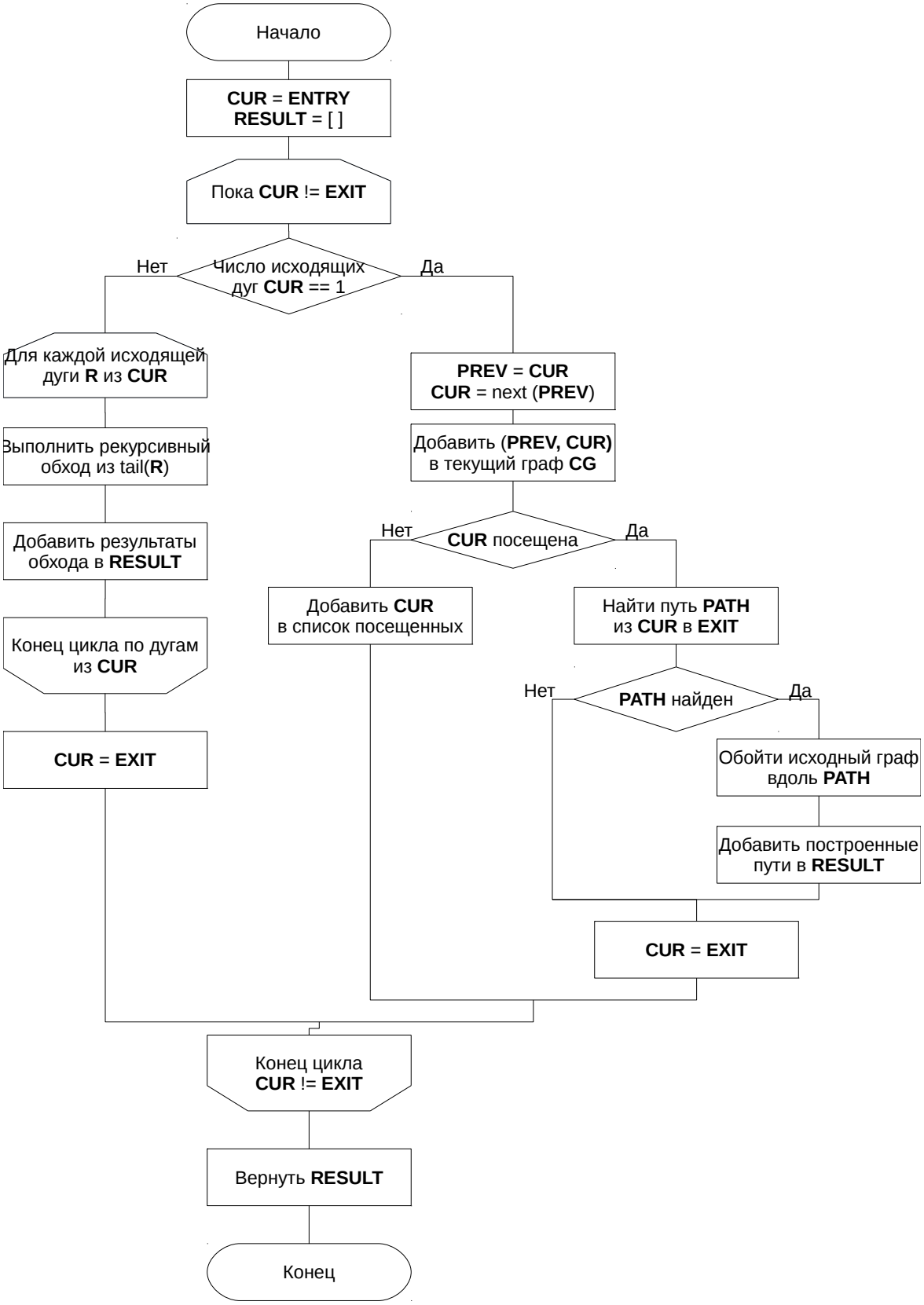

Рисунок 3.7 — Блок-схема алгоритма линеаризации ГПУ потока программы

4 Проектирование программной реализации метода

В данной главе приведено описание разрабатываемого программного обеспечения, реализующего метод статического поиска условий возникновения гонок. Глава содержит описание потоков данных между модулями программы, а так же описание особенности реализации обработки входных данных программы.

4.1 Структура программы

На рисунке 4.1 приведена схема структуры разрабатываемой системы.

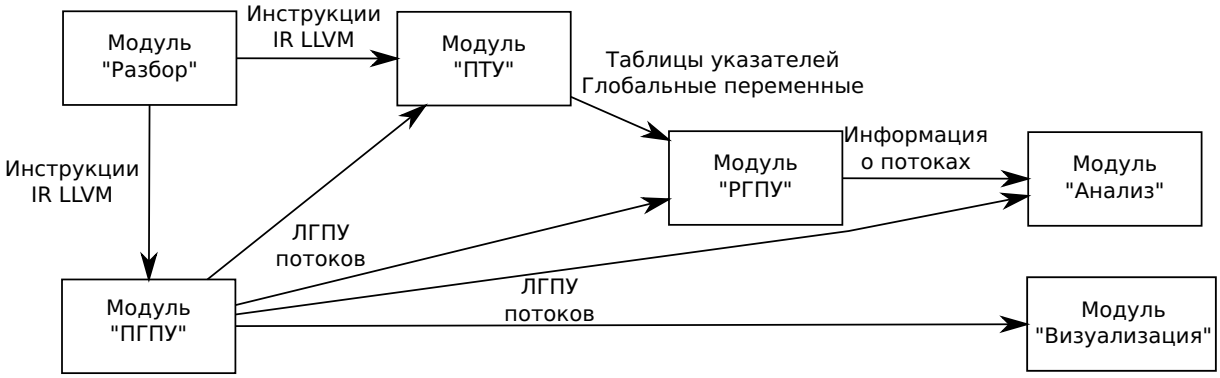

Рисунок 4.1 — Схема структуры программы

Как видно из рисунка, система состоит из следующих модулей:

а) «Разбор» — модуль проведения первичного разбора входных файлов и формирования данных для последующих этапов разбора кода;

б) «ПТУ» — модуль построения таблиц указателей;

в) «ПГПУ» — модуль построения ГПУ;

г) «РГПУ» — модуль проведения первичного анализа ГПУ и дополнения его информацией о множествах захваченных объектов синхронизации и множествах активных потоков;

д) «Анализ» — модуль проведения анализа ГПУ, является основным модулем программы;

е) «Визуализация» — вспомогательный модуль для подготовки ГПУ к визуализации.

4.2 Потоки данных

На рисунке 4.2 приведена DFD диаграмма потоков данных в системе.

Входным потоком данных для программы является промежуточное представление программы в виде текстового файла, содержащего инструкции языка IR LLVM. В модуле «Разбор» данный поток преобразуется в поток отдельных инструкций

IR LLVM путем проведения разбора файла, передаваемых в модули «ПГПУ» и «ПТУ».

В модуле «ПГПУ» из получаемых инструкции IR LLVM производится построение частичных ГПУ, которые затем преобразуются в ГПУ функций. На основе полученных ГПУ функций производится построение ГПУ потоков программы, а затем построение линейных ГПУ. Построенные линейные ГПУ передаются в модули «Анализ» и «Визуализация» для проведения анализа и подготовки графов к визуализации соответственно. Так же в модуль «Визуализация» передаются ГПУ потоков программы для подготовки к визуализации.

В модуле «РГПУ» на основе построенных линейных ГПУ производится построение множеств активных потоков и множеств захваченных объектов блокировки, которые также передаются в модуль «Анализ».

В модуле «ПТУ» на основе получаемых инструкций IR LLVM производится построение таблиц указателей и определение глобальных переменных. Полученные данные передаются в модуль «Анализ» для дальнейшей обработки.

В модуле «Анализ» на данных, получаемых от всех остальных модулей производится анализ графов потока управления входной программы с целью поиска возможных ситуаций возникновения гонок. По итогам анализа производится формирование отчета о найденных ситуациях гонок, который является одним из типов выходных данных программы.

В модуле «Визуализация» на основе полученных данных производится подготовка графов к визуализации. Результатом процесса подготовки является текстовое описание построенных графов потоков управления программы.

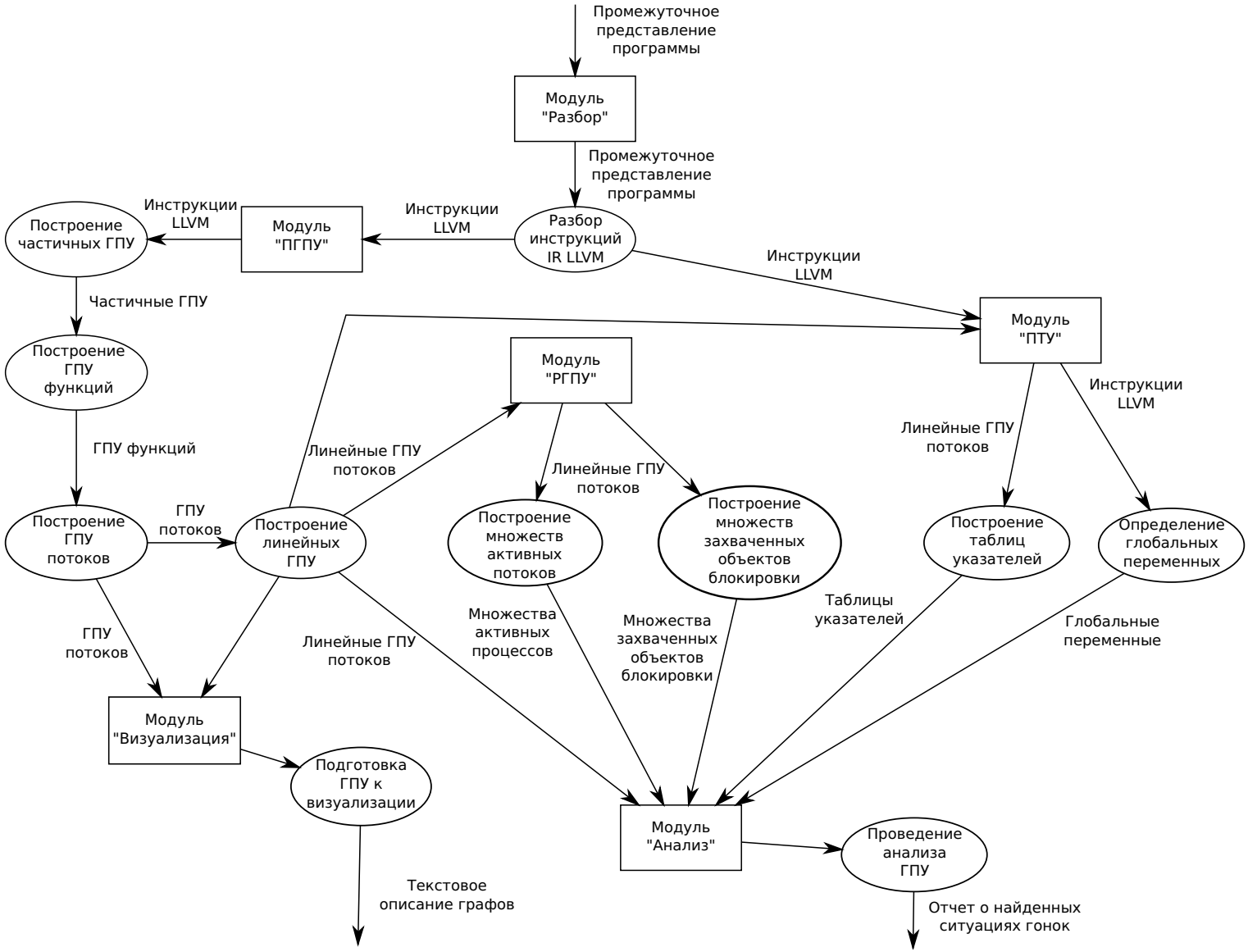

Рисунок 4.2 — Диаграмма потоков данных между модулями программы

4.3 Взаимодействие модулей программы

На рисунке 4.3 изображена диаграмма последовательности действий, отражающая взаимодействие модулей программы при проведении анализа входной программы с целью поиска возможных ситуаций возникновения гонок. На диаграмме использованы следующие сокращения:

а) МАП — множество активных процессов;
б) МЗОБ — множество захваченных объектов блокировки.

Как видно из диаграммы, процесс анализа инициируется модулем «Анализ», который является главным модулем программы. На первом этапе производится вызов модуля «Разбор» с целью разбора входного файла и дальнейшего построения ГПУ программы и таблиц указателей, а так же для определения глобальных переменных анализируемой программы.

На втором этапе модуль «Разбор» производит обработку входного файла и осуществляет вызовы модулей «ПГПУ» и «ПТУ» для построения ГПУ и таблиц указателей соответственно. Представленный на диаграмме процесс обмена данными является упрощенным, более подробное описание процесса взаимодействия указанных модулей приведено в разделе 4.4.

По окончании процесса разбора файла управление возвращается модулю «Анализ». На следующем этапе модуль «Анализ» забирает построенные ГПУ, таблицы указателей и глобальные переменные разбираемой программы из соответствующих модулей, после чего производится последующая обработка построенных ГПУ в модуле «РГПУ».

По окончании построения множеств активных потоков и множеств захваченных объектов синхронизации модуль «Анализ» производит анализ ГПУ с целью поиска возможных ситуаций гонок во входной программе. По завершении процесса поиска производится формирование отчета о найденных ситуациях гонок, после чего выполнение программы завершается.

4.4 Разбор промежуточного представления программы

На рисунке 4.4 представлена диаграмма деятельности, отображающая процесс разбора кода программы.

Разбор кода основан на событийной модели обработки данных с использованием шаблона проектирования «писатель-подписчик»[9]. Использование данного подхода позволяет при необходимости достаточно легко расширить функциональность модуля разбора входных данных, не внося при этом изменений в другие модули. Модуль первичного разбора файла, содержащего код программы на языке IR LLVM, производит построчное считывание файла до конца файла. После прочтения каж-

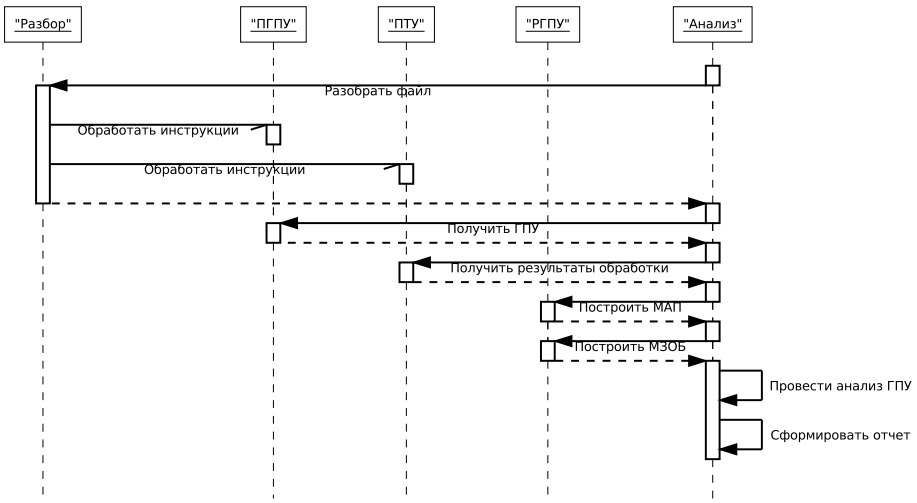

Рисунок 4.3 — Диаграмма последовательности действий программы при проведении анализа входной программы

дой строки производится проверка выполнения определенных условий и в случае успешного прохождения тех или иных условий создается соответствующее событие.

При обработке файла в качестве стороны «писателя» выступает модуль «Разбор», со стороны подписчиков — модули «ПГПУ» и «ПТУ».

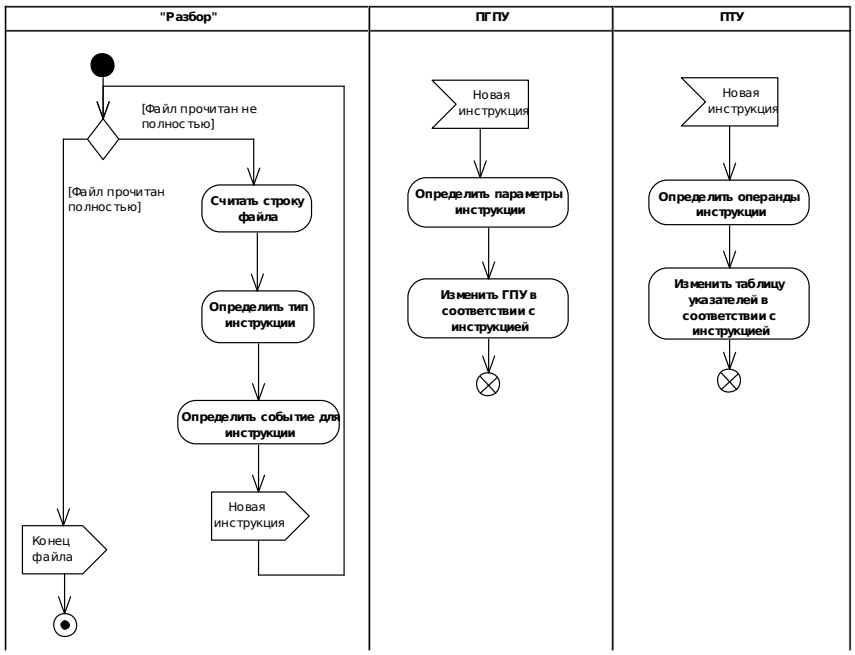

Рисунок 4.4 — Диаграмма деятельности при разборе промежуточного кода программы

4.5 Выводы

Спроектирована структура программного обеспечения, реализующего метод писка гонок, и описаны потоки данных между модулями. Для обработки входных данных принято решение использовать событийную модель взаимодействия.

5 Программная реализация метода

В данной главе приводится обоснование выбора языка программирования для реализации разработанного метода, а так же выбор среды разработки. В главе содержится описание запуска программы и аргументы командной строки, необходимые для указания. Во второй части данного раздела приводится описание системных тестов, используемых для определения работоспособности программы.

5.1 Выбор языка программирования

В качестве языка программирования был выбран язык Java. Язык Java является достаточно распространенным в настоящее время в силу того, что программы, написанные на данном языке программирования, могут выполняться под управлением различных операционных систем на различных аппаратных платформах без перекомпиляции.

В качестве платформы была выбрана платформа JDK 7. По сравнению с предыдущей версией платформы (JDK 6) в выбранной платформе были расширены возможности языка, что позволило несколько сократить объем исходного кода.

5.2 Выбор среды разработки

В качестве среды разработки была выбрана среда NetBeans. Данная среда разработки включает в себя богатый набор инструментов для разработки программного обеспечения на различных языках программирования, в том числе и язык Java. Среда разработки NetBeans распространяется по двойной лицензии Common Development and Distribution License (CDDL) и GNU General Public License (GPL) v2.

5.3 Запуск программы

Запуск программы осуществляется путем выполнения следующей команды в командной строке:

java -jar Analyser.jar

При запуске в качестве аргументов командной строки должен быть задан один или более путей к файлам, содержащим промежуточное представление кода программы на языке IR LLVM.

Так же могут быть указаны дополнительные аргументы командной строки, описание которых приведено в таблице 5.1.

Таблица 5.1 — Дополнительные аргументы командной строки

Аргумент	**Назначение**
-m <mode>	Режим работы программы.

Таблица 5.1 — Дополнительные аргументы командной строки

Аргумент	**Назначение**
-o <output>	Путь к файлу для вывода результатов. Если путь не указан или указан как «- -», то вывод осуществляется в **stdout**.
-do <output>	Путь к файлу для вывода ГПУ и путей выполнения в формате **dot**. Если путь не указан или указан как «- -», то вывод осуществляется в **stdout**.
-dg <dotmode>	Режим формирования графов.
-panic	Добавление в отчет ситуаций, которые не были определены как ситуации с возможными гонками.

Возможные режимы программы указаны в таблице 5.2.

Таблица 5.2 — Режимы работы программы

Режим	**Назначение**
s **std** **standard**	Анализ кода и формирование «dot» файла.
d **dot**	Формирование только «dot» файла.
rc **race**	Только анализ кода.

В таблице 5.3 приведены режимы формирования графов.

Таблица 5.3 — Режимы формирования графов

Режим	**Назначение**
a **all**	Построение графов обычных и линейных ГПУ потоков.
l **linear**	Построение только линейных ГПУ потоков.
p **primary**	Построение только обычных ГПУ потоков.

5.4 Формат выходного сообщения о найденной ситуации гонки

Сообщение о найденной ситуации гонки выглядит следующим образом:

```
Ситуация #1
[Тип = rtReadWrite, переменная = [flags = [G], ctx = global] deref @food]
```

```
Блок 1 [Поток = @philosopher(main__5)]:
  store i32 %2, i32* %3, align 4
Блок 2 [Поток = @philosopher(main__11)]:
  %1 = load i32* %0, align 4
```

В сообщении о найденной ситуации гонки используются следующие обозначения:

а) тип — тип найденной гонки:

1) rtReadWrite — гонка типа «чтение-запись»;

2) rtConcurWrite — гонка типа одновременная запись;

б) переменная — переменная или значение указателя, обращение к которому приводит к гонке;

в) блок 1, блок 2 — инструкции LLVM, при выполнении которых возникает ситуация гонки;

г) поток — имя главной функции потока, в рамках которого выполняется данная инструкция, а так же аргумент создания потока.

Для переменной, обращение к которой привело к гонке, так же имеет дополнительную информацию:

— контекст, в рамках которого объявлена переменная;

— а так же дополнительные флаги:

а) G — глобальная переменная;

б) TA — аргумент создания потока.

5.5 Тестирование разработанного программного обеспечения

В данном разделе приводятся тесты, на основе которых проводилось системное тестирование разработанного программного обеспечения.

Тест №1

Цель теста: убедиться в возможности обнаружения простейших ситуаций гонок при отсутствии защиты доступа к переменным.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.1.

Ожидаемые результаты: обнаружение трех[1] ситуаций гонок.

Результаты теста: обнаружено три ситуации гонок.

Листинг 5.1 — Исходные данные для теста 1

[1]Инкрементирование счетчика состоит из двух операций с памятью — чтение и последующая запись

```
#include <unistd.h>
#include <pthread.h>

void *thread_main(void *args);

int ctr = 0;

int main(int argc, char *argv[]) {
        pthread_t thread;
        pthread_create(&thread, NULL, thread_main, NULL);
        ++ctr; // race condition
}

void *thread_main(void *args) {
        ++ctr; // race condition
        return NULL;
}
```

Тест №2

Цель теста: убедиться в обнаружении защищенного доступа к переменной.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.2.

Ожидаемые результаты: отсутствие обнаруженных ситуаций гонок.

Результаты теста: ситуаций гонок не обнаружено.

Листинг 5.2 — Исходные данные для теста 2

```
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_main(void *args);
sem_t sem;

int ctr = 0;

int main(int argc, char *argv[]) {
        pthread_t thread;
        sem_init(&sem, 0, 1);
        pthread_create(&thread, NULL, thread_main, NULL);

        sem_wait(&sem);
        ++ctr; // no race condition
        sem_post(&sem);
}

```

```
20 void *thread_main(void *args) {
21         sem_wait(&sem);
22         ++ctr; // no race condition
23         sem_post(&sem);
24
25         return NULL;
26 }
```

Тест №3

Цель теста: убедиться в возможности обнаружения ситуаций гонок при использовании указателей и передаче разделяемой переменной в качестве аргумента потока.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.3.

Ожидаемые результаты: обнаружение трех ситуаций гонок.

Результаты теста: обнаружено три ситуации гонок.

Листинг 5.3 — Исходные данные для теста 3

```
1  #include <unistd.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4
5  void *thread_main(void *args);
6
7  int main(int argc, char *argv[]) {
8          pthread_t thread;
9          int ctr = 0;
10         int *pctr = &ctr;
11         pthread_create(&thread, NULL, thread_main, pctr);
12         ++ctr; // race condition
13 }
14
15 void *thread_main(void *args) {
16         int *ctr = (int *) args;
17         ++*ctr; // race condition
18         return NULL;
19 }
```

Тест №4

Цель теста: убедиться в возможности обнаружения ситуаций гонок при наличии защищенного доступа только в одном из потоков.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.4.

Ожидаемые результаты: обнаружение трех ситуаций гонок.

Результаты теста: обнаружено три ситуации гонок.

Листинг 5.4 — Исходные данные для теста 4

```
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_main(void *args);
sem_t sem;

int ctr = 0;

int main(int argc, char *argv[]) {
        pthread_t thread;
        sem_init(&sem, 0, 1);
        pthread_create(&thread, NULL, thread_main, NULL);
        sem_wait(&sem);
        ++ctr; // no race condition
        sem_post(&sem);
}

void *thread_main(void *args) {
        ++ctr; // race condition
        return NULL;
}
```

Тест №5

Цель теста: убедиться в возможности определения упорядоченности создания потоков и обращений к разделяемым переменным.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.5.

Ожидаемые результаты: обнаружено три ситуации гонок.

Результаты теста: обнаружено три ситуации гонок.

Листинг 5.5 — Исходные данные для теста 5

```
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_main(void *args);
void *thread_main2(void *args);
```

```
7
8  int main(int argc, char *argv[]) {
9          pthread_t thread;
10         int ctr = 0;
11         int *pctr = &ctr;
12         pthread_create(&thread, NULL, thread_main, pctr);
13 }
14
15 void *thread_main(void *args) {
16         int *ctr = (int *) args;
17         ++*ctr; // (1) no race condition
18         pthread_t thread;
19         pthread_create(&thread, NULL, thread_main2, args);
20         ++*ctr; // (2) race condition
21         return NULL;
22 }
23
24 void *thread_main2(void *args) {
25         int *ctr = (int *) args;
26         /*
27          * (1) no race condition
28          * (2) race condition
29          */
30         ++*ctr;
31         return NULL;
32 }
```

Тест №6

Цель теста: убедиться в возможности обнаружения гонок при создании дерева потоков.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.6.

Ожидаемые результаты: обнаружение трех ситуаций гонок.

Результаты теста: обнаружено три ситуации гонок.

Листинг 5.6 — Исходные данные для теста 6

```
1  #include <unistd.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4
5  void *thread_main(void *args);
6  void *thread_main2(void *args);
7
8  int main(int argc, char *argv[]) {
```

```
9           pthread_t thread;
10          int ctr = 0;
11          int *pctr = &ctr;
12          pthread_create(&thread, NULL, thread_main, pctr);
13          pthread_create(&thread, NULL, thread_main, pctr);
14  }
15
16  void *thread_main(void *args) {
17          pthread_t thread;
18          pthread_create(&thread, NULL, thread_main2, args);
19          return NULL;
20  }
21
22  void *thread_main2(void *args) {
23          int *ctr = (int *) args;
24          ++*ctr; // race condition
25          return NULL;
26  }
```

Тест №7

Цель теста: убедиться в возможности отслеживания аргументов создания потоков.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.6.

Ожидаемые результаты: ситуаций гонок не обнаружено.

Результаты теста: ситуаций гонок не обнаружено.

Листинг 5.7 — Исходные данные для теста 7

```
1   #include <unistd.h>
2   #include <pthread.h>
3   #include <semaphore.h>
4
5   void *thread_main(void *args);
6   void *thread_main2(void *args);
7
8   int main(int argc, char *argv[]) {
9           pthread_t thread;
10          int ctr1 = 0;
11          int ctr2 = 0;
12          int *pctr = &ctr1;
13          pthread_create(&thread, NULL, thread_main, pctr);
14          pctr = &ctr2;
15          pthread_create(&thread, NULL, thread_main, pctr);
16  }
```

```
17
18 void *thread_main(void *args) {
19         pthread_t thread;
20         pthread_create(&thread, NULL, thread_main2, args);
21         return NULL;
22 }
23
24 void *thread_main2(void *args) {
25         int *ctr = (int *) args;
26         ++*ctr; // no race condition
27         return NULL;
28 }
```

Тест №8

Цель теста: убедиться в наличии ложного срабатывания при неявном упорядочивании доступа к переменной.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.8.

Ожидаемые результаты: обнаружение трех ситуаций гонок.

Результаты теста: обнаружено три ситуации гонки.

Листинг 5.8 — Исходные данные для теста 8

```
1  #include <unistd.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4
5  void *thread_main(void *args);
6  sem_t sem;
7
8  int ctr = 0;
9
10 int main(int argc, char *argv[]) {
11         pthread_t thread;
12         sem_init(&sem, 0, 0);
13         pthread_create(&thread, NULL, thread_main, NULL);
14         sem_wait(&sem);
15         ++ctr; // no race condition
16 }
17
18 void *thread_main(void *args) {
19         ++ctr; // no race condition. main locked by sem
20         sem_post(&sem);
21
22         return NULL;
```

```
23 }
```

Тест №9

Цель теста: убедиться в определении потоков, которые не могут выполняться одновременно.

Исходные данные: в качестве исходных данных используется программа, листинг которой приведен в листинге 5.9.

Ожидаемые результаты: отсутствие обнаруженных гонок.

Результаты теста: ситуаций гонок не обнаружено.

Листинг 5.9 — Исходные данные для теста 9

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5
6  void *thread_main(void *args);
7  sem_t sem;
8
9  int ctr = 0;
10
11 int main(int argc, char *argv[]) {
12         pthread_t thread;
13         rand();
14         int r = rand();
15         int *r_ptr = &r;
16         pthread_create(&thread, NULL, thread_main, r_ptr);
17 }
18
19 void *thread_main(void *args) {
20     int *prob = (int *) args;
21     if(*prob > 50)
22         pthread_create(&thread, NULL, thread_high, args);
23     else
24         pthread_create(&thread, NULL, thread_low, args);
25 }
26
27 void *thread_high(void *args) {
28     int *i = (int *) args;
29     ctr -= *i;
30 }
31
32 void *thread_low(void *args) {
33     int *i = (int *) args;
```

```
34      ctr += *i;
35  }
```

Тест №10

Цель теста: убедиться в возможности подготовки ГПУ к визуализации.

Исходные данные: в качестве исходных данных используется программа, текст которой приведен в листинге 5.9.

Ожидаемые результаты: файл в формате **dot**, содержащий описание 6 графов:

а) 4 графа для поточных ГПУ;

б) 2 графа для путей выполнения потока **thread_main**.

Результаты теста: файл содержит 6 описаний графов.

5.6 Выводы

Разработано программное обеспечение, реализующее метод статического поиска возможных ситуаций возникновения гонок. Проведено системное тестирование, установлена работоспособность разработанного программного обеспечения, реализующего статический метод поиска условий возникновения гонок в программе.

6 Проведение экспериментов

В данном разделе описаны эксперименты, проводимые с разработанным программным обеспечением. Эксперименты производятся с целью определения скоростных характеристик разработанного ПО и метода в целом, производится выявление зависимости времени работы анализатора от числа обращений к разделяемым переменным.

6.1 Исходные данные для проведения экспериментов

В данном разделе приведено описание программ, используемых для проведения экспериментов.

6.1.1 Задача обедающих философов

Для проведения экспериментов за основу была взята задача «Обедающие философы». В классической постановке данная задача может быть использована для тестирования средств обнаружения тупиков в программах. Формальная постановка задачи обедающих философов звучит следующим образом: в системе существует некоторое количество ресурсов, каждый из которых является общим для двух потоков. Для выполнения каких-либо операций поток должен захватить два ресурса, являющихся общими между ним и «соседними» потоками. В листинге 6.1 приведен алгоритм работы потока системы, соответствующий классической задаче обедающих философов. В данном листинге **left_fork** и **right_fork** — разделяемые ресурсы, **lock** и **unlock** — захват и освобождение соответствующего ресурса.

Листинг 6.1 — Пример алгоритма работы потока в задаче обедающих философов

```
1 while(my_food > 0) {
2     lock(left_fork);
3     lock(right_fork);
4     my_food -= 1;
5     unlock(right_fork);
6     unlock(left_fork);
7 }
```

Для адаптации задачи обедающих философов для поиска ситуаций гонок были сделаны следующие изменения в постановке задачи:

а) ресурсы, разделяемые между парами потоков, объединены в общий пул ресурсов;

б) введены дополнительные общие ресурсы, к которым могут получать доступ потоки.

В листинге 6.2 приведен пример алгоритма работы одного из потоков системы для адаптированной задачи обдающих философов. В приведенном листинге **forks** — общий для всех потоков ресурс, **food*N*** — общий ресурс для некоторых потоков, **lock** и **unlock** — блокировка и освобождение доступа к соответствующему ресурсу.

Листинг 6.2 — Пример алгоритма работы одного из потоков в адаптированной системе

```
1  do {
2      got_forks = 0;
3      while(got_forks == 0) {
4          lock(forks);
5          if(forks >= 2) {
6              forks -= 2;
7              got_forks = 1;
8          }
9          unlock(forks);
10     }
11     lock(food1);
12     if(food1 > 0)
13         food1 -= 1;
14     unlock(food1);
15     lock(food2);
16     if(food2 > 0)
17         food2 -=1;
18     unlock(food2);
19     lock(forks);
20     forks += 2;
21     unlock(forks);
22     exit = food1 > 0 || food2 > 0;
23 } while(!exit);
```

6.1.2 Задача «Читатели-Писатели»

Суть данной задачи заключается в следующем: в системе есть ресурс, общий для всех потоков. Часть потоков получают доступ к этому ресурсу только для чтения, часть — для записи. При этом потоки, которые получают доступ к ресурсу только для чтения, могут производить чтение одновременно. Алгоритмы работ потоков читателя и писателя приведены в листинге 6.3. В приведенном листинге **readers** и **writers** — семафоры, означающие нахождение потока-читателя и потока-писателя в критической секции.

Листинг 6.3 — Пример алгоритма работы одного из потоков в адаптированной системе

```
1  reader_count = 0;
2
3  proc reader() {
4      lock(reader_count);
5      if(reader_count == 0)
6          lock(readers);
7      reader_count += 1;
8      unlock(reader_count);
9
10     read X;
11
12     lock(reader_count);
13     reader_count -= 1;
14     if(reader_count == 0)
15         unlock(readers);
16     unlock(reader_count);
17 }
18
19 proc writer() {
20     lock(readers);
21     lock(writers);
22     write X:
23     unlock(writers);
24     unlock(readers);
25 }
```

6.2 Условия проведения экспериментов

Проведение экспериментов производилось в следующих условиях:

а) аппаратное обеспечение:
   1) Intel(R) Core(TM)2 CPU T5500 1.66GHz;
   2) 768 Мб ОЗУ;

б) программное обеспечение:
   1) ОС Debian GNU/Linux 6.0 2.6.32-5-686;
   2) Java SE Runtime Environment 1.7.0.

6.3 Определение скоростных характеристик

Данный эксперимент направлен на выявление зависимости скорости работы программы от числа обращений к разделяемым переменным. Для проведения данного эксперимента используется программа, реализующая адаптированный вариант задачи обедающих философов. Исходных код программы приведен в приложении Г. В программе используется постоянное количество разделяемых переменных, измене-

ние числа обращений к переменным достигается за счет изменение числа потоков, обращающихся к разделяемым переменным.

В процессе проведения эксперимента в указанной выше программе производилось изменение числа потоков, создаваемых в функции **main**. На рисунке 6.1 представлен график зависимости времени выполнения анализа от количества потоков в программе. В таблице 6.1 приведена соответствующая информация о числе потоков, обращений к разделяемым переменным и числа анализируемых ситуаций, а так же соответствующее время анализа кода программы.

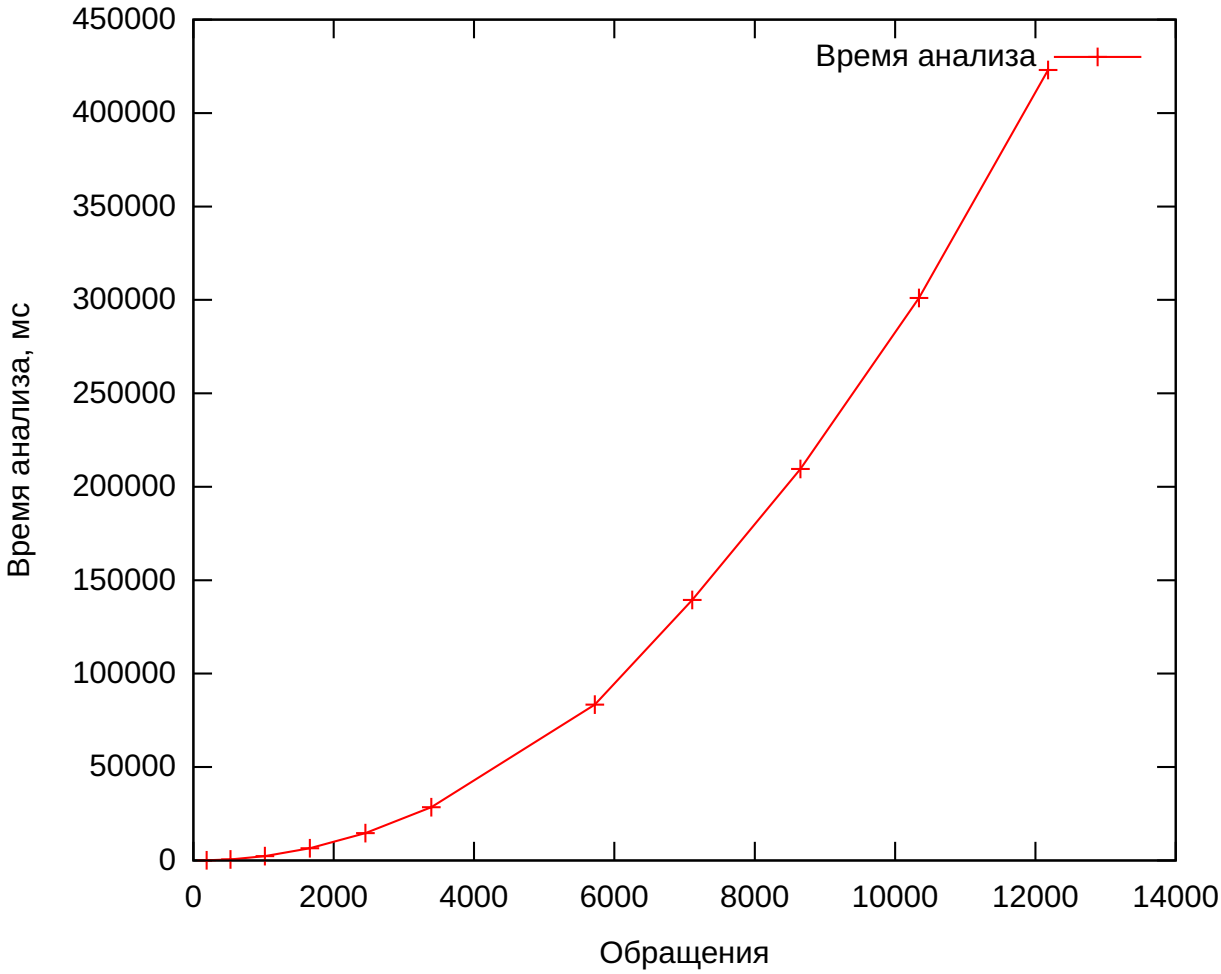

Рисунок 6.1 — График зависимости времени выполнения анализа от количества обращений к разделяемым переменным

Таблица 6.1 — Число обращений к переменным и число проанализированных ситуаций

Потоки	**Обращения**	**Анализируемые ситуации**	**Время анализа, мс**
5	190	3040	90
10	530	16380	580

Таблица 6.1 — Число обращений к переменным и число проанализированных ситуаций

Потоки	**Обращения**	**Анализируемые ситуации**	**Время анализа, мс**
15	1020	44520	2308
20	1660	91960	6500
25	2450	163200	14600
30	3390	262740	28500
40	5720	564720	83400
45	7110	776160	139400
50	8650	1033900	209500
55	10340	1342440	301000
60	12180	1706280	423000

Множество обращений к разделяемым переменным порождает собой множество ситуаций, которое необходимо проанализировать с целью обнаружения ситуации гонки или определения, что рассматриваемая ситуация не содержит гонки. Так как необходимо рассмотреть все множество пар обращений к разделяемым переменным, кроме обращений в рамках одно и того же потока, то общее число анализируемых ситуаций можно вычислить по формуле ниже:

$$V = \sum_{i=1}^{N}(C_{S_i}^2 - \sum_{j=1}^{M} C_{u_{ij}}^2) \tag{6.1}$$

$$S_i = \sum_{j=1}^{M} u_{ij} \tag{6.2}$$

В приведенных выше формулах V — число анализируемых ситуаций, N — число разделяемых переменных, M — число потоков, u_{ij} —число обращений к i переменной в j потоке. На рисунке 6.2 представлен график зависимости времени выполнения от числа анализируемых ситуаций.

Из приведенного графика видно, что зависимость времени от числа анализируемых ситуаций так же является нелинейной. Нелинейность связана с тем, что при анализе каждой ситуации производится построение множества активных потоков. Время построения данного множества в общем случае эквивалентно времени обхода дерева создания потоков в ширину. Время обхода дерева в ширину линейно зависит от числа вершин в этом дереве [10], из чего можно сделать вывод о том, что время построения полного множества активных потоков линейно зависит от числа потоков, или, в более общем случае, от числа путей выполнения потоков в программе.

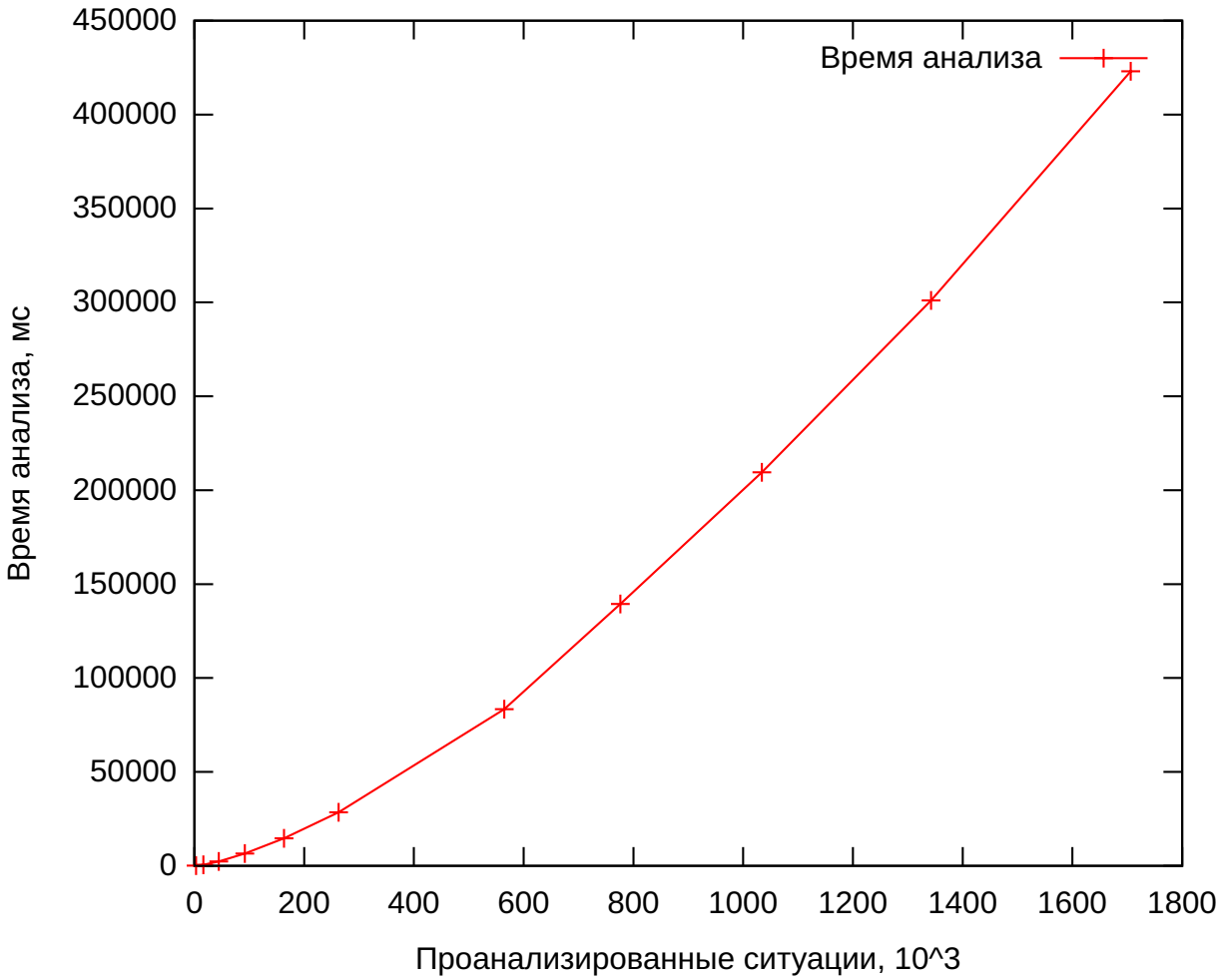

Рисунок 6.2 — Зависимость времени анализа от числа анализируемых ситуаций

Таким образом, домножив число обращений, рассчитанное по формуле (6.1), на число потоков, получим следующую зависимость времени анализа:

$$T = c * P \sum_{i=1}^{N} (C_{S_i}^2 - \sum_{j=1}^{M} C_{u_{ij}}^2) \tag{6.3}$$

В формуле (6.3) c — константа, T — время анализа, P — число путей выполнения потоков программы.

На рисунке 6.3 представлены графики зависимости экспериментального и расчетного времени анализа программы от числа обращений к разделяемым переменным. Как видно из графиков, полученная функция зависимости времени анализа достаточно хорошо аппроксимирует результаты, полученные экспериментальным путем.

Для проверки формулы (6.3) ниже приведен расчет времени анализа для рассматриваемой задачи обедающих философов для 65 потоков.

Рассматриваемая программа содержит 14170 обращений к разделяемым переменным и 2129920 ситуаций для анализа. На основе полученных эксперимен-

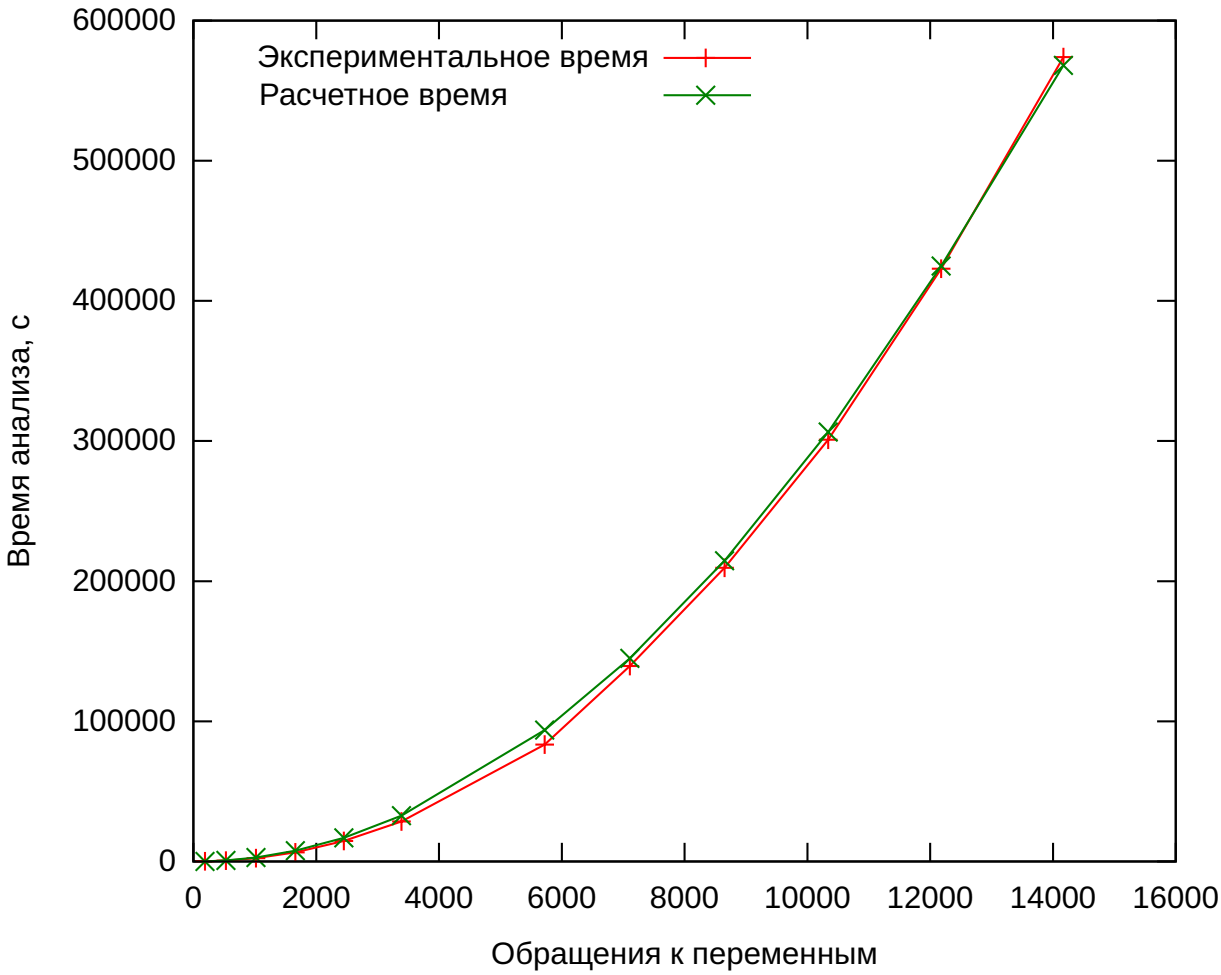

Рисунок 6.3 — Сравнение экспериментального и расчетного времени анализа

тальных данных для рассматриваемой задачи обедающих философов коэффициент $c = 0.00083$. Таким образом, расчетное время анализа 574 секунды. Время анализа, полученное экспериментальным путем — 568 секунд. Учитывая погрешность проводимых измерений, расхождение в 6 секунд можно считать приемлемой ошибкой.

6.4 Определение ситуации гонки в задаче «Читатели-Писатели»

Для проведения эксперимента использовались две реализации задачи читателей-писателей. Одна из реализаций не содержит ситуаций гонок, вторая же содержит незащищенный доступ к переменной-счетчику числа активных читателей, что при реальном выполнении программы может привести к возникновению ситуации гонки при доступе к общей для всех потоков переменной.

Листинги программ, реализующих задачу читателей-писателей, приведены в приложении Д.

При обработке корректной реализации задачи анализатором были обнаружены ложные ситуации гонок при доступе к переменной **ctr**. Обнаружение этих ситуаций связано с тем, что захват объекта блокировки **no_readers** происходит не во

всех возможных путях выполнения потока читателей и в силу того, что анализатор рассматривает пути выполнения независимо друг от друга, происходит обнаружение гонки. После изменения алгоритма, обеспечивающего захват семафора **no_readers** во всех путях выполнения потока читателей, обнаружение ложной ситуации гонки не происходит.

При обработке некорректной реализации кроме ситуаций гонок, описанных выше, так же обнаруживаются ситуации гонок при доступе к переменной **nreaders**. Обнаружение этих ситуаций гонки связано с тем, что проверка значения данной переменной производится вне критических секций и возможна ситуация, в которой один из потоков изменит значение указанной переменной в момент проверки, что в конечном счете приведет к ситуациям гонок, связанных с доступам к переменной **ctr**.

6.5 Выводы

Проведены эксперименты с разработанным ПО. По итогам проведенных экспериментов определено, что разработанный метод допускает обнаружение ситуаций гонок, однако в некоторых ситуациях, где не производится явного входа в критическую секцию путем захвата какого-либо объекта блокировки, возможно обнаружение ложных ситуаций гонок. На основе экспериментальных данных и описаниях алгоритмов получена и проверена формула для расчета времени проведения анализа для модифицированной задачи обедающих философов.

Заключение

В результате проделанной работы была достигнута цель работы в виде разработки и реализации метода статического поиска условий возникновения гонок, а так же решены все поставленные задачи:

а) проведен обзор существующих методов поиска условий возникновения гонок;

б) выбран формат входных данных в виде промежуточного представления кода программы на языке IR LLVM;

в) разработан метод статического поиска условий возникновения гонок;

г) разработано программное обеспечение, реализующее метод статического поиска;

д) проведены эксперименты с разработанным программным обеспечением.

Список использованных источников

1. *Beckman, Nels E.* A Survey of Methods for Preventing Race Conditions / Nels E. Beckman. — 2006.

2. Eraser: A dynamic data race detector for multi-threaded programs / Stefan Savage, Michael Burrows, Greg Nelson et al.

3. Dynamic Model Checking with Property Driven Pruning to Detect Race Conditions / Chao Wang, Yu Yang, Aarti Gupta, Ganesh Gopalakrishnan. — Princeton, New Jersey, USA.

4. *Elmas, Tayfun.* Precise Race Detection and Efficient Model Checking Using Locksets / Tayfun Elmas, Shaz Qadeer, Serdar Tasiran. — One Microsoft Way, Redmond, WA 98052.

5. *Engler, Dawson.* RacerX: Effective, Static Detection of Race Conditions and Deadlocks / Dawson Engler, Ken Ashcraft.

6. *Flanagan, Cormac.* Detecting Race Conditions in Large Programs / Cormac Flanagan, Stephen N. Freund.

7. Официальная документация IR LLVM. `http://llvm.org/docs/LangRef.html`.

8. *Butenhof, David R.* Programming With Posix Threads / David R Butenhof.

9. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. — 2001.

10. *Knuth, Donald E.* The Art Of Computer Programming Vol 1. 3rd ed / Donald E Knuth. — 1997.

Приложение А Примеры исходного кода демонстрационной программы на различных языках программирования

Листинг А.1 — Пример программы на языке Java

```
public class Main {
    private static int ctr = 0;
    public static void main(String[] args) {
        Thread thr = new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i = 0; i < 100000000; i++)
                    ++ctr; // race condition
            }
        });

        thr.start();

        for(int i = 0; i < 100000000; i++)
            ++ctr;

        try {
            thr.join();
        }
        catch(InterruptedException ex) {
            /* do nothing */
        }
    }
}
```

Листинг А.2 — Пример программы на языке C++

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *thread_main(void *args);

int main(int argc, char *argv[]) {
    pthread_t thread;
    int ctr = 0;
    pthread_create(&thread, NULL, thread_main, &ctr);
    for(int i = 0; i < 10000000; i++)
        ++ctr; // race condition

    pthread_join(thread, NULL);

```

```
17      return ctr != 20000000;
18  }
19
20  void *thread_main(void *args) {
21      int *ctr = (int *) args;
22      for(int i = 0; i < 10000000; i++)
23          ++*ctr; // race condition
24
25      return NULL;
26  }
```

Приложение Б Пример промежуточного кода демонстрационной программы на языке IR LLVM

Листинг Б.1 — Пример исходного кода на языке IR LLVM

```
1  ; ModuleID = 'main.cpp'
2  target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
       i64:32:64-f32:32:32-f64:32:64-v64:64:64-v128:128:128-a0:0:64-
       f80:32:32-n8:16:32"
3  target triple = "i386-linux-gnu"
4
5  %union.pthread_attr_t = type { i32, [8 x i32] }
6
7  define i32 @main(i32 %argc, i8** %argv) {
8  entry:
9    %argc_addr = alloca i32                         ; <i32*> [#uses=1]
10   %argv_addr = alloca i8**                        ; <i8***> [#uses=1]
11   %retval = alloca i32                            ; <i32*> [#uses=2]
12   %0 = alloca i32                                 ; <i32*> [#uses=2]
13   %thread = alloca i32                            ; <i32*> [#uses=2]
14   %ctr = alloca i32                               ; <i32*> [#uses=5]
15   %i = alloca i32                                 ; <i32*> [#uses=4]
16   %"alloca point" = bitcast i32 0 to i32          ; <i32> [#uses=0]
17   store i32 %argc, i32* %argc_addr
18   store i8** %argv, i8*** %argv_addr
19   store i32 0, i32* %ctr, align 4
20   %ctr1 = bitcast i32* %ctr to i8*                ; <i8*> [#uses=1]
21   %1 = call i32 @pthread_create(i32* noalias %thread,
       %union.pthread_attr_t* noalias null, i8* (i8*)* @_Z11thread_mainPv,
       i8* noalias %ctr1) nounwind ; <i32> [#uses=0]
22   store i32 0, i32* %i, align 4
23   br label %bb2
24
25 bb:                                               ; preds = %bb2
26   %2 = load i32* %ctr, align 4                    ; <i32> [#uses=1]
27   %3 = add nsw i32 %2, 1                          ; <i32> [#uses=1]
28   store i32 %3, i32* %ctr, align 4
29   %4 = load i32* %i, align 4                      ; <i32> [#uses=1]
30   %5 = add nsw i32 %4, 1                          ; <i32> [#uses=1]
31   store i32 %5, i32* %i, align 4
32   br label %bb2
33
34 bb2:                                              ; preds = %bb, %entry
35   %6 = load i32* %i, align 4                      ; <i32> [#uses=1]
36   %7 = icmp sle i32 %6, 9999999                   ; <i1> [#uses=1]
37   br i1 %7, label %bb, label %bb3
38
```

```
39 bb3:                                           ; preds = %bb2
40   %8 = load i32* %thread, align 4              ; <i32> [#uses=1]
41   %9 = call i32 @pthread_join(i32 %8, i8** null)  ; <i32> [#uses=0]
42   %10 = load i32* %ctr, align 4                ; <i32> [#uses=1]
43   %11 = icmp ne i32 %10, 20000000              ; <i1> [#uses=1]
44   %12 = zext i1 %11 to i32                     ; <i32> [#uses=1]
45   store i32 %12, i32* %0, align 4
46   %13 = load i32* %0, align 4                  ; <i32> [#uses=1]
47   store i32 %13, i32* %retval, align 4
48   br label %return
49
50 return:                                          ; preds = %bb3
51   %retval4 = load i32* %retval                 ; <i32> [#uses=1]
52   ret i32 %retval4
53 }
54
55 define i8* @_Z11thread_mainPv(i8* %args) nounwind {
56 entry:
57   %args_addr = alloca i8*                      ; <i8**> [#uses=2]
58   %retval = alloca i8*                         ; <i8**> [#uses=2]
59   %0 = alloca i8*                              ; <i8**> [#uses=2]
60   %ctr = alloca i32*                           ; <i32**> [#uses=3]
61   %i = alloca i32                              ; <i32*> [#uses=4]
62   %"alloca point" = bitcast i32 0 to i32       ; <i32> [#uses=0]
63   store i8* %args, i8** %args_addr
64   %1 = load i8** %args_addr, align 4           ; <i8*> [#uses=1]
65   %2 = bitcast i8* %1 to i32*                  ; <i32*> [#uses=1]
66   store i32* %2, i32** %ctr, align 4
67   store i32 0, i32* %i, align 4
68   br label %bb1
69
70 bb:                                              ; preds = %bb1
71   %3 = load i32** %ctr, align 4                ; <i32*> [#uses=1]
72   %4 = load i32* %3, align 4                   ; <i32> [#uses=1]
73   %5 = add nsw i32 %4, 1                       ; <i32> [#uses=1]
74   %6 = load i32** %ctr, align 4                ; <i32*> [#uses=1]
75   store i32 %5, i32* %6, align 4
76   %7 = load i32* %i, align 4                   ; <i32> [#uses=1]
77   %8 = add nsw i32 %7, 1                       ; <i32> [#uses=1]
78   store i32 %8, i32* %i, align 4
79   br label %bb1
80
81 bb1:                                             ; preds = %bb, %entry
82   %9 = load i32* %i, align 4                   ; <i32> [#uses=1]
83   %10 = icmp sle i32 %9, 9999999               ; <i1> [#uses=1]
84   br i1 %10, label %bb, label %bb2
85
```

```
86  bb2:                                              ; preds = %bb1
87    store i8* null, i8** %0, align 4
88    %11 = load i8** %0, align 4                     ; <i8*> [#uses=1]
89    store i8* %11, i8** %retval, align 4
90    br label %return
91
92  return:                                           ; preds = %bb2
93    %retval3 = load i8** %retval                    ; <i8*> [#uses=1]
94    ret i8* %retval3
95  }
96
97  declare i32 @pthread_create(i32* noalias, %union.pthread_attr_t* noalias,
        i8* (i8*)*, i8* noalias) nounwind
98
99  declare i32 @pthread_join(i32, i8**)
```

Приложение В Пример промежуточного кода демонстрационной программы в виде дизассемблированного байт-кода Java

Листинг В.1 — Пример дизассемблированного байт-кода Java

```
Compiled from "Main.java"
public class Main extends java.lang.Object{
public Main();
  Code:
    0:    aload_0
    1:    invokespecial    #2; //Method java/lang/Object."<init>":()V
    4:    return

public static void main(java.lang.String[]);
  Code:
    0:    new #3; //class java/lang/Thread
    3:    dup
    4:    new #4; //class Main$1
    7:    dup
    8:    invokespecial    #5; //Method Main$1."<init>":()V
    11:   invokespecial    #6; //Method
       java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
    14:   astore_1
    15:   aload_1
    16:   invokevirtual    #7; //Method java/lang/Thread.start:()V
    19:   iconst_0
    20:   istore_2
    21:   iload_2
    22:   ldc #8; //int 100000000
    24:   if_icmpge    41
    27:   getstatic    #1; //Field ctr:I
    30:   iconst_1
    31:   iadd
    32:   putstatic    #1; //Field ctr:I
    35:   iinc     2, 1
    38:   goto     21
    41:   aload_1
    42:   invokevirtual    #9; //Method java/lang/Thread.join:()V
    45:   goto     49
    48:   astore_2
    49:   return
  Exception table:
   from    to  target type
     41    45    48   Class java/lang/InterruptedException

```

```
40
41 static int access$004();
42   Code:
43     0:   getstatic   #1; //Field ctr:I
44     3:   iconst_1
45     4:   iadd
46     5:   dup
47     6:   putstatic   #1; //Field ctr:I
48     9:   ireturn
49
50 static {};
51   Code:
52     0:   iconst_0
53     1:   putstatic   #1; //Field ctr:I
54     4:   return
55
56 }
```

Приложение Г Исходный код программы, реализующей задачу «Обедающие философы»

```
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

void *philosopher(void *args);

sem_t fork_sem;
int forks = 5;

sem_t food_sem;
int food = 10;

int main(int argc, char *argv[]) {
    pthread_t thread;
    int *food_ptr = &food;
    pthread_create(&thread, NULL, philosopher, food_ptr);
    /* N times */
    pthread_create(&thread, NULL, philosopher, food_ptr);
}

void eat_food(int *food) {
    sem_wait(&food_sem);
    --*food;
    sem_post(&food_sem);
}

void *philosopher(void *args) {
    int *food = (int *)args;
    int got_forks = 0;
    while(!got_forks) {
        sem_wait(&fork_sem);
        if(forks >= 2) {
            forks -= 2;
            got_forks = 1;
        }
        sem_post(&fork_sem);
    }
    eat_food(food);
    sem_wait(&fork_sem);
    forks += 2;
    sem_post(&fork_sem);
}
```

Приложение Д Исходные коды программ, реализующих задачу «Читатели-Писатели»

Д.1 Корректная реализация алгоритма

```
1  #include <unistd.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4
5  void *reader(void *args);
6  void *writer(void *args);
7
8  sem_t no_writers;
9  sem_t no_readers;
10 sem_t ctr_lock;
11
12 int nreaders = 0;
13
14 int main(int argc, char *argv[]) {
15     pthread_t thread;
16     int ctr = 0;
17     int *arg = &ctr;
18     sem_init(&no_writers, 0, 1);
19     sem_init(&no_readers, 0, 1);
20     sem_init(&ctr_lock, 0, 1);
21     pthread_create(&thread, NULL, writer, arg);
22     pthread_create(&thread, NULL, writer, arg);
23     pthread_create(&thread, NULL, reader, arg);
24     pthread_create(&thread, NULL, reader, arg);
25 }
26
27 void *writer(void *arg) {
28     int *ctr = (int *)arg;
29     int prev, cur;
30     sem_wait(&no_writers);
31     sem_wait(&no_readers);
32     ++*ctr;
33     sem_post(&no_readers);
34     sem_post(&no_writers);
35 }
36
37 void *reader(void *arg) {
38     int *ctr = (int *)arg;
39     sem_wait(&no_writers);
40     sem_wait(&ctr_lock);
41     if(nreaders == 0)
42         sem_wait(&no_readers);
```

```
43      ++nreaders;
44      sem_post(&ctr_lock);
45      sem_post(&no_writers);
46      int x = *ctr; // imitating read
47      sem_wait(&ctr_lock);
48      --nreaders;
49      if(nreaders == 0)
50          sem_post(&no_readers);
51      sem_post(&ctr_lock);
52  }
```

Д.2 Реализация, не вызывающая ложных срабатываний

```
1   #include <unistd.h>
2   #include <pthread.h>
3   #include <semaphore.h>
4
5   void *reader(void *args);
6   void *writer(void *args);
7
8   sem_t no_writers;
9   sem_t no_readers;
10  sem_t ctr_lock;
11
12  int nreaders = 0;
13
14  int main(int argc, char *argv[]) {
15      pthread_t thread;
16      int ctr = 0;
17      int *arg = &ctr;
18      sem_init(&no_writers, 0, 1);
19      sem_init(&no_readers, 0, 1);
20      sem_init(&ctr_lock, 0, 1);
21      pthread_create(&thread, NULL, writer, arg);
22      pthread_create(&thread, NULL, writer, arg);
23      pthread_create(&thread, NULL, reader, arg);
24      pthread_create(&thread, NULL, reader, arg);
25  }
26
27  void *writer(void *arg) {
28      int *ctr = (int *)arg;
29      int prev, cur;
30      sem_wait(&no_writers);
31      sem_wait(&no_readers);
32      ++*ctr;
33      sem_post(&no_readers);
```

```
34      sem_post(&no_writers);
35  }
36
37  void *reader(void *arg) {
38      int *ctr = (int *)arg;
39      sem_wait(&no_writers);
40      sem_wait(&ctr_lock);
41      sem_wait(&no_readers);
42      ++nreaders;
43      sem_post(&ctr_lock);
44      sem_post(&no_writers);
45      int x = *ctr; // imitating read
46      sem_wait(&ctr_lock);
47      --nreaders;
48      sem_post(&no_readers);
49      sem_post(&ctr_lock);
50  }
```

Д.3 Некорректная реализация алгоритма

```
1   #include <unistd.h>
2   #include <pthread.h>
3   #include <semaphore.h>
4
5   void *reader(void *args);
6   void *writer(void *args);
7
8   sem_t no_writers;
9   sem_t no_readers;
10  sem_t ctr_lock;
11
12  int nreaders = 0;
13
14  int main(int argc, char *argv[]) {
15      pthread_t thread;
16      int ctr = 0;
17      int *arg = &ctr;
18      sem_init(&no_writers, 0, 1);
19      sem_init(&no_readers, 0, 1);
20      sem_init(&ctr_lock, 0, 1);
21
22      pthread_create(&thread, NULL, writer, arg);
23      pthread_create(&thread, NULL, writer, arg);
24      pthread_create(&thread, NULL, reader, arg);
25      pthread_create(&thread, NULL, reader, arg);
26  }
```

```
27
28 void *writer(void *arg) {
29     int *ctr = (int *)arg;
30     int prev, cur;
31     sem_wait(&no_writers);
32     sem_wait(&no_readers);
33     sem_post(&no_writers);
34     ++*ctr;
35     sem_post(&no_readers);
36 }
37
38 void *reader(void *arg) {
39     int *ctr = (int *)arg;
40     sem_wait(&no_writers);
41     sem_wait(&ctr_lock);
42     ++nreaders;
43     sem_post(&ctr_lock);
44     if(nreaders - 1 == 0)
45         sem_wait(&no_readers);
46
47     sem_post(&no_writers);
48     int x = *ctr; // imitating read
49     sem_wait(&ctr_lock);
50     --nreaders;
51     sem_post(&ctr_lock);
52     if(nreaders == 0)
53         sem_post(&no_readers);
54 }
```