Содержание

Глоссарий

IR-код — промежуточное представление кода

LLVM — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-подобными инструкциями

Введение

При разработке параллельных приложений часто возникают трудные в локализации ошибки, вызванные нарушениями в синхронизации доступа потоков в общей памяти. Подобные ошибки называются состоянием гонки между потоками. Состояния гонки затруднительны в выявлении и устранении, поскольку выполнение многопоточного приложения является недетерминированным и от запуска к запуску могут возникать различные проявления данной ошибки. В работе рассмотрены гонки возникающие в результате несинхронизированного доступа к памяти. Гонки, вызванные взаимодействием с внешними, по отношению к программе, источниками данных не рассматриваются в рамках данной работы.

При реализации систем поиска гонок используется три подхода: статический анализ, динамический анализ, формальная верификация. Каждый из подходов имеет свою область применения. Для анализа исходного кода программ, как правило используются динамические анализаторы, производящие поиск гонок во время выполнения программы.

Динамические анализаторы как правило представляют собой виртуальную машину, выполняющую исполняемый файл анализируемой программы. Также динамические анализаторы реализуются в виде расширения к компиляторам и производят модификацию исходного кода во время компиляции программы для вставки инструкций для анализа гонок. Последнее решение является эффективным по процессорному времени и по памяти, но не позволяет производить анализ библиотек без их перекомпиляции. Основной проблемой динамических анализаторов гонок являются большие накладные расходы при выполнении анализируемой программы — производительность программы падает в 10–200 раз [1].

В настоящее время параллельные серверные и пользовательские приложения, например, веб-сервера (Nginx, Apache, Lighttpd), системы управления базами данных (MySQL, PostgreSQL), веб-браузеры, реализуются как правило на таких языках как Си/Си++ с использованием POSIX API.

Целью работы является разработка метода динамического поиска гонок в программах, реализованных на языка Си. Для достижения поставленной цели необходимо решить следующие задачи:

— проанализировать подходы к поиску гонок в ПО;

— проанализировать динамические системы поиска гонок и рассмотреть применяемые в них алгоритмы;

— разработать алгоритм динамического поиска гонок, применимый для анализа программ, реализованных на языке Си;

— реализовать систему динамического поиска гонок в программах, разработанных на языке Си, использующих POSIX API;

— провести и спланировать эксперимент с целью определения область применения метода.

В главе 1 рассмотрена проблема поиска гонок и основные подходы к обнаружению состояний гонок в ПО. В главе 2 описан разработанный метод поиска гонок в программах на языке Си, включающий в себя алгоритм поиска гонок и описание подхода построения динамического анализатора. В главе 3 рассмотрено проектирование анализатора поиска гонок и его реализация. В главе 4 проведены эксперименты с целью определения области применения разработанного метода.

1 Поиск состояний гонок

В данной главе рассмотрены основные подходы к поиску гонок в многопоточной среде выполнения. Проанализированы алгоритмы динамического поиска гонок и особенности их применения. Рассмотрены подходы построения динамических анализаторов.

1.1 Гонки в многопоточной среде

Поток выполнения является наименьшей единицей выполнения, которая может быть связана с одним логическим процессором. Несколько потоков выполнения в POSIX среде существуют в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов (рисунок 1.1). В частности, потоки выполнения выполнения разделяют инструкции процесса и его контекст — значения переменных, которые они имеют в любой момент времени [2].

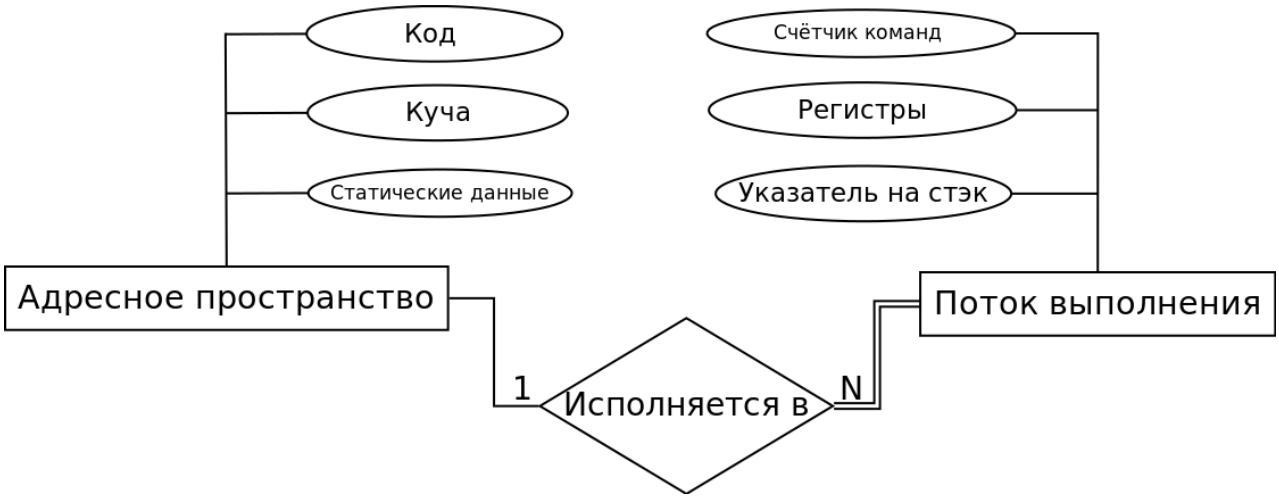

Рисунок 1.1 — Общее адресное пространство потоков выполнения

В современных ОС потоки в многопроцессорном среде могут выполняться параллельно. Обычно программа при запуске имеет один основной поток, который во время своего выполнения создает дополнительные потоки, которые в свою очередь могут так же создавать новые потоки или завершаться.

Если два параллельных потока одновременно используют одну и туже область памяти и хотя бы один из них изменяет ее, то такое состояние называется **состоянием гонки**. При существовании гонки выполнение про-

граммы непредсказуемое и зависит от работы планировщика задач, причем следствием гонок могут быть ошибки, проявляющиеся в различных местах выполняемой программы. Гонки являются распространенной проблемой при разработке сложных параллельных систем, их выявление и устранение затруднительно ввиду нелокальности ошибок порождаемых ими.

1.2 Существующие подходы поиска гонок

Для поиска гонок в приложениях существуют инструменты следующих классов:

— верификация моделей программ [3];
— статические анализаторы кода [4, 5, 6];
— динамические анализаторы кода [1];
— нагрузочное тестирование.

Верификация моделей в первую очередь применяется для проверки моделей программ и алгоритмов. Проверяется некоторый параллельный алгоритм или протокол взаимодействия — например, новый транспортный протокол SCTP. Данное решение как правило применяется в системах, где необходимо полностью исключить состояния гонок, взаимные блокировки. Для поиска гонок в конкретной реализации ПО методом формальной верификации существуют средства автоматического построения модели программы по исходному коду, но сгенерированные модели требуют ручной корректировки [7]. Количество состояний и переходов в полученной модели экспотенциально зависит от размера исходного кода программы, что ведет к комбинаторному «взрыву» при верификации сложных программных комплексов. По этим причинам метод проверки моделей практически не применим при поиске гонок в конечном ПО [8].

Статические анализаторы работают с исходным кодом программы и обрабатывают все возможные пути выполнения программы. Данный подход покрывает весь доступный код. Статический анализ программ на языках, имеющих средства управления памятью, позволяет выявлять узкий спектр возможных ошибок.

Динамические инструменты во время выполнения программы осуществляют запись истории обращений по всем адресам памяти и запись операций синхронизации для дальнейшего анализа. Подобные средства пытаются выявить гонки, которые произошли во время выполнения программы. По данной причине область поиска гонок ограничена путями выполнения программы, которое на практике увеличивается с увеличением числа тестов. Основной проблемой данного класса инструментов являются накладные расходы во время выполнения ПО и необходимость контролирования полноты тестовых сценариев. Детекторы Thread Checker [2], Helgrind снижают производительность анализируемой программы в 10–200 раз [1]. Несмотря на отсутствие проверки всевозможных состояний программы во время выполнения, данный метод широко используется для выявления гонок в ПО [9].

Нагрузочное тестирование является общим случаем динамического анализа гонок. Для анализируемого ПО осуществляется реализация тестовых сценариев с интенсивным взаимодействием между потоками выполнения. Так как при существовании гонки в ПО поведение программы недетерминировано, то необходимо производить тестирование на различных аппаратных и программных конфигурациях. При экстренном завершении программы в ходе тестирования невозможно автоматически сообщить является ли гонка причиной критической ошибки и место данной гонки.

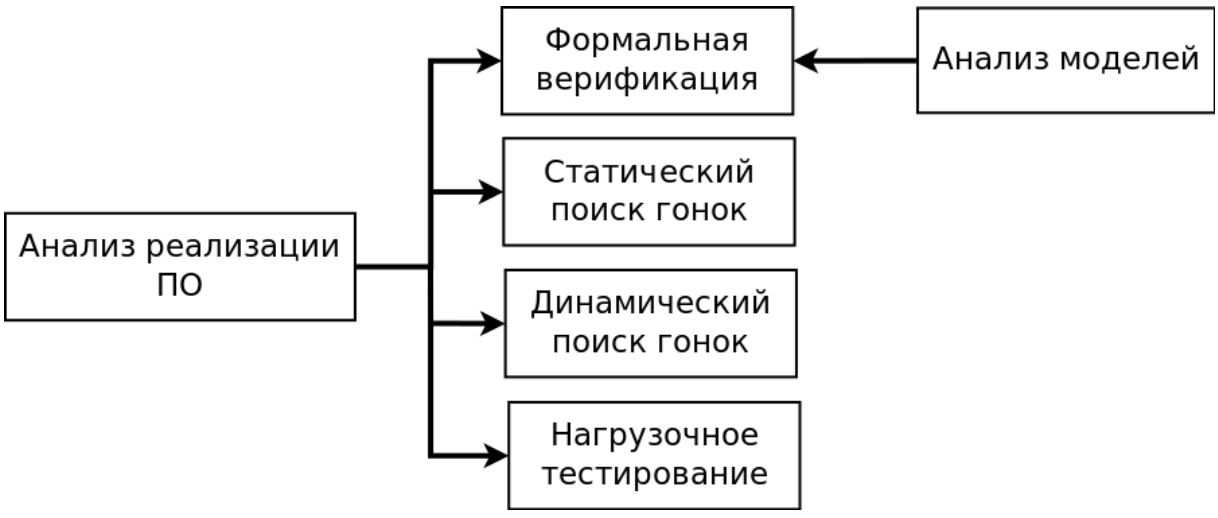

Рисунок 1.2 — Подходы к поиску гонок

Таким образом, если необходимо провести поиск гонок в ПО, то предпочтительно использование инструментов основанных на динамическом анализе, так как данный подход позволяет произвести проверку конкретной реализации.

1.3 Общий подход к динамическому поиску гонок

Динамические средства поиска гонок производят анализ программы во время ее выполнения (рисунок 1.3). При исполнении программы в некоторой среде выполнения происходит журналирование обращений по всем адресам памяти и операций с примитивами синхронизации. Анализ журнала выполнения может производиться во время выполнения программы или после её завершения. Подход с анализом данных после завершения программы применяется для анализа сложных параллельных программ на системах с низким количеством памяти [10]. Результатом анализа является сообщения о найденных гонках, включающий в себя стектрейсы потоков выполнения, которые привели к состоянию гонки при обращению к какой-либо области памяти.

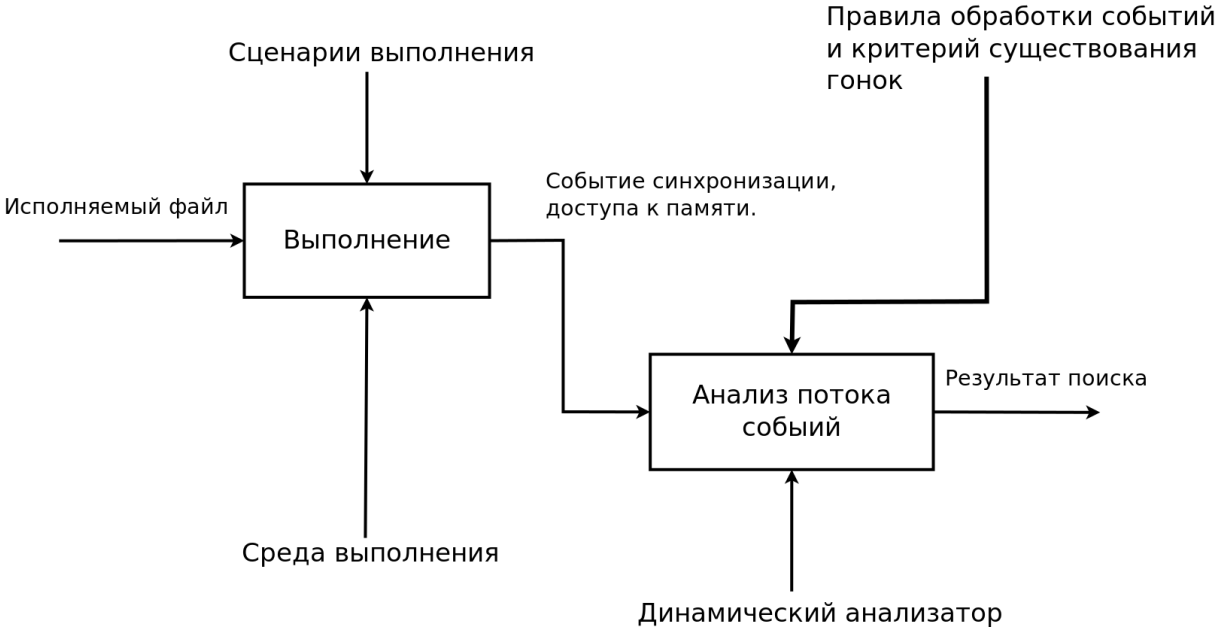

Рисунок 1.3 — Динамический поиск гонок

Кроме синхронизированного доступа к памяти необходимо учитывать особенности реализации примитивов синхронизации. Многопроцессорная среда в общем случае не гарантирует, что операции с памятью вы-

полняются в том порядке в котором они произошли при выполнении потоков. Несмотря на то, что доступ к переменной синхронизирован, возможно получение неконсистентных данных. Для решения данной проблемы процессоры реализуют механизм барьера памяти, осуществляющий упорядочивание доступа к памяти между процессорами [11].

Стандарт POSIX гарантирует, что реализация функций синхронизации, указанных в таблице 1.1, производит упорядочивание обращений к памяти между процессорами через барьеры [12].

Таблица 1.1 — Список POSIX функций, использующих барьер памяти

Назначение	Имя функции
создание процессов и ожидание событий от процессов	**fork** **wait** **waitpid**
синхронизация на барьере	**pthread_barrier_wait**
взаимодействие с условными переменными	**pthread_cond_broadcast** **pthread_cond_signal** **pthread_cond_timedwait** **pthread_cond_wait**
создание и ожидание завершения потока	**pthread_create** **pthread_join**
захват и освобождение мьютексов	**pthread_mutex_lock** **pthread_mutex_timedlock** **pthread_mutex_trylock** **pthread_mutex_unlock**
захват и освобождение rw-мьютексов	**pthread_rwlock_rdlock** **pthread_rwlock_tryrdlock** **pthread_rwlock_trywrlock** **pthread_rwlock_unlock** **pthread_rwlock_wrlock**
захват и освобождение семофоров	**sem_post** **sem_trywait** **sem_wait**

Таким образом, в POSIX окружении, если обращение к памяти реализовано примитивы синхронизации, то нет необходимости учитывать возможное неупорядоченные выполнение инструкций процессорами. Поэтому

при динамическом поиске гонок рассматриваются только обращения к памяти и события связанные с примитивами синхронизации.

1.4 Генерация потока событий

Динамические анализаторы основаны на анализе потока событий, генерируемых программой во время выполнения. Программа является «черным ящиком», подающий поток событий $\langle e_i \rangle$ в регистратор событий. Регистратор событий производит обработку полученных событий (например, поддержка актуальности векторных часов) и сохранение события в журнал выполнения потока. Далее анализатор по журналу выполнения потоков осуществляет проверку существования гонок, если текущее событие может быть источником гонок(рисунок 1.4).

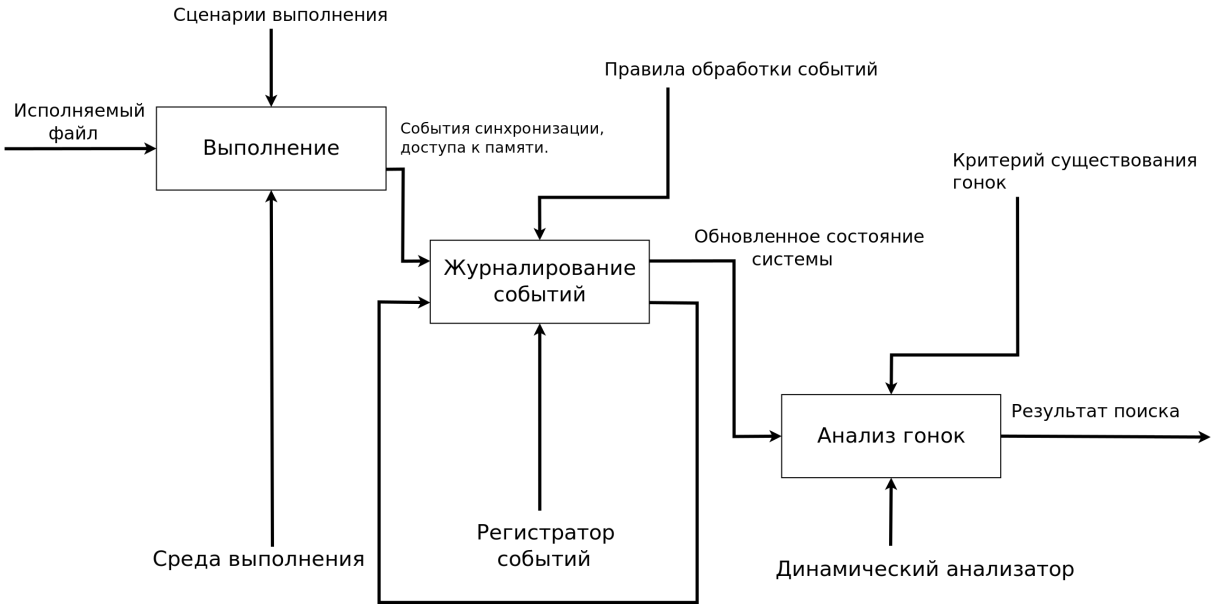

Рисунок 1.4 — Схема генерации и обработки событий

Выделим возможные характеристики, описывающие события.

а) M — множество адресов памяти. Память по данным адресам изменяемая и доступна всем потокам.

б) L — множество блокировок. В **POSIX** это адреса мьютексов.

в) T — множество дескрипторов потоков. В **POSIX** это адрес структуры потока **pthread_t**.

г) G — множество уникальных идентификаторов “сообщений” потоков, таких как создание и завершение потоков, разблокировка пото-

ков (**pthread_cond_signal**), блокировка потоков на переменной состояния (**pthread_cond_wait**), и тд.

д) A — тип доступа к памяти (чтение или запись). Возможные значения: $READ, WRITE$.

События, анализируемые динамическими средствами поиска гонок можно разделить на три класса: доступ к памяти на чтение или запись, работа с блокировами, получение сообщения от потока или отправка сообщения другому потока. Обозначим данные события следующим образом.

— **Доступ к памяти:** $Mem(m, a, t)$, где $m \in M$, $a \in A$ и $t \in T$.

— **Захват блокировки:** $ACQ(l, t)$, где $l \in L$, $t \in T$. Данное событие показывает, что поток t захватил блокировку l. Для рекурсивных блокировок событие $ACQ(l, t)$ повторно не генерируется, если поток захватил блокировку l ранее.

— **Освобождение блокировки:** $REL(l, t)$, где $l \in L$, $t \in T$. Данное событие показывает, что поток t освободил блокировку l.

— **Отправка сообщения:** $SND(g, t)$, где $g \in G$, $t \in T$. Указывает что поток t отправил сообщение g некоторому потоку.

— **Прием сообщения:** $RCV(g, t)$, где $g \in G$, $t \in T$. Данный тип сообщения означает приём потоком t сообщения g.

1.5 Обзор алгоритмов динамического поиска гонок

Средства динамического поиска гонок построены на базе двух подходов: анализ набора блокировок (*lockset-based analysis*) и анализ на основе отношения предшествования (*happens-before analysis*).

В подходе на основанном на базе анализа набора блокировок проверяется соответствие программы определенной дисциплине блокировок. Самой простой дисциплиной является проверка, что доступ к памяти всегда осуществляется через блокировки. Подход на базе отношения предшествования сообщает о найденной гонке, если два (или более) потоков используют общую память, причем, доступы к памяти причинно неупорядочены (*causally unordered*) по определению Лампорта [13].

При рассмотрении алгоритмы будут адаптированы для применения в динамическом анализа программ, написанных на языке Си в с использованием **POSIX API**.

Для простоты изложения примем, что программа выполняется последовательно и в один момент времени запущен только один поток, который может генерировать поток событий.

1.6 Алгоритм на основе множества блокировок

Алгоритм на основе множества блокировок производит анализ гонок на основе событий двух типов: обращение к памяти на чтение или запись, захват или освобождение блокировок.

Данный алгоритм может быть реализован с низкими издержками, но не позволяет анализировать взаимодействие через условные переменные, события создания потоков и ожидания завершения потоков [14].

1.6.1 Описание алгоритма

Алгоритм на основе множества блокировок требует поддержки списка захваченных блокировок для каждого потока и историю доступа к областям памяти. Для заданного потока t на основе потока событий $\langle e_i \rangle$ могут быть вычислены захваченные блокировки $L_i(t)$ в момент события e_i:

$$L_i(t) = \{l \| \exists a : a < i \wedge e_a = ACQ(l,t) \wedge (\nexists r : a < r < i \wedge e_r = REL(l,t))\}.$$

Для каждого потока производится анализ событий освобождения, захвата блокировок и формируется список активных блокировок в момент времени e_i.

Алгоритм базируется на гипотезе, что если два или более потоков используют общую память и как минимум один из них производит запись, то пересечение множеств захваченных блокировок непустое. Таким образом наличие гонки между двумя событиями e_i и e_j вычисляется следующим

образом:

$$\begin{aligned} LocksetRaceExists(i,j) \Longleftarrow & (e_i = MEM(m_i, a_i, t_i) \wedge e_j = MEM(m_j, a_j, t_j)) \\ & \wedge t_i \neq t_j \wedge m_i = m_j \\ & \wedge (a_i = WRITE \vee a_j = WRITE) \\ & \wedge L_i(t_i) \cap L_j(t_j) = \emptyset. \end{aligned}$$

Алгоритм может быть реализован на основе схемы, изображенной на рисунке 1.5. Далее представлено описание обработки нового события от потоков выполнения.

а) На вход поступает событие обращения к памяти или освобождения/захвата блокировки.

б) Если произошло событие доступа к памяти $MEM(m_i, a_i, t_i)$:

1) необходимо в историю обращений к области памяти m_i произвести запись полученного события;
2) произвести анализ наличия гонок по гипотезе описанной ранее;
3) если гонки найдены, сообщить об успешном обнаружении гонок при обращении к памяти m_i.

в) Если произошло событие освобождения/захвата блокировки $ACQ/REL(l_i, t_i)$, необходимо актуализировать список активных блокировок потока t_i.

Основным преимуществом алгоритма на основе множества блокировок являются низкие накладные расходы по памяти и процессору при реализации. Существуют работы указывающие возможность аппаратной реализации данного алгоритма с накладными расходами $0.1 - 2.6\%$ [14]. К недостаткам относится генерация ложных срабатываний вызванных различными типичными техниками программирования (например, использование пула объектов) и невозможностью учитывать такие события создания, ожидания завершения потоков и взаимодействие через условные переменные.

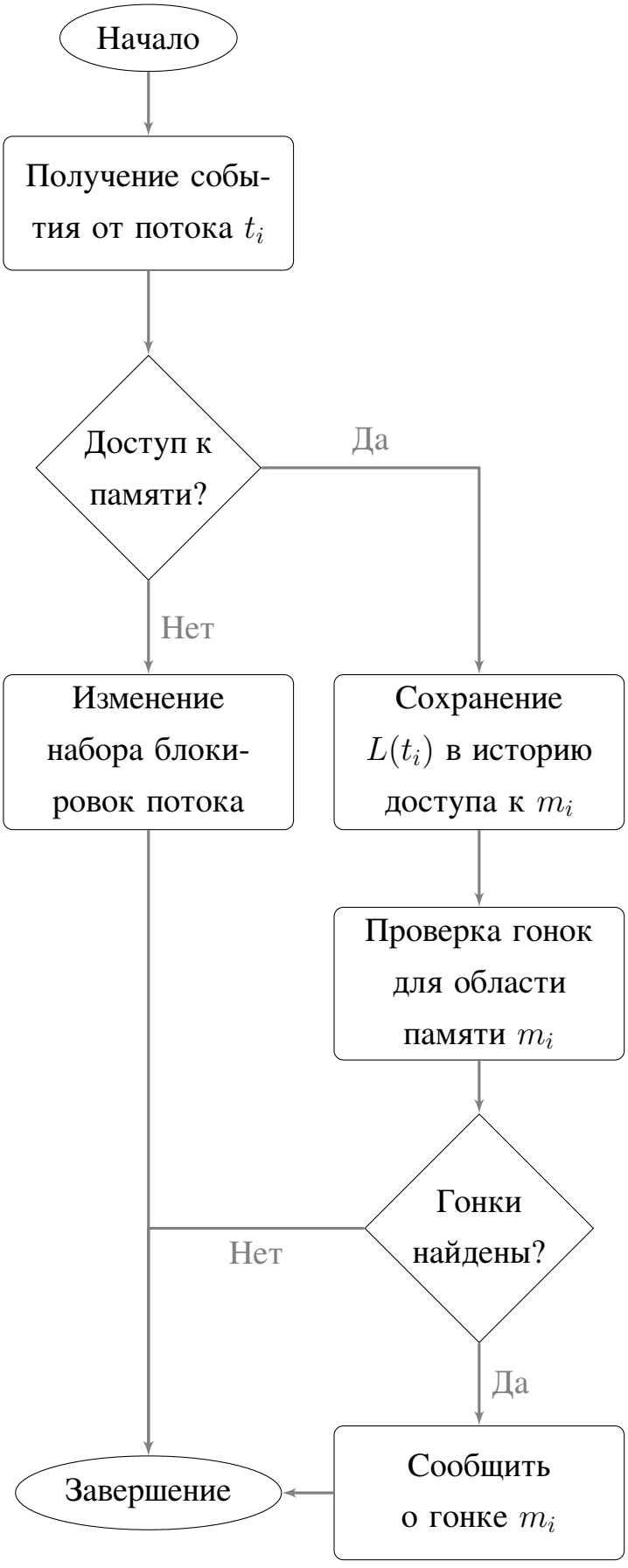

Рисунок 1.5 — Поиск гонок на основе множества блокировок

1.6.2 Проблема ложных срабатываний

При применении алгоритмов динамического поиска гонок возникают ошибки двух видов:

— невыявление существующей гонки в программе;

— сообщение о гонке, которая на самом деле отсутствует в программе (далее ложные срабатывания).

В зависимости от используемого алгоритма данные ошибки могут проявляться в различных ситуациях, поэтому рассмотрим ситуации при которых происходят ложные срабатывания.

Проведем анализ ложных срабатываний генерируемых алгоритмом на основе множества блокировок.

```
// Поток 1                                  // Поток 2
int global_variable;                        void thread2_loop() {
                                                // Ложное срабатывание
int main() {                                    global_variable = 100;
    global_variable = 1;                        ...
    run_thread(thread2_loop);                 }
    ...
}
```

Рисунок 1.6 — Пример кода, вызывающий ложное срабатывание алгоритма на основе множества блокировок

На рисунке 1.6 представлен псевдокод программы в которой алгоритм на основе множества блокировок сообщает о гонке при доступе к переменной **global_variable**. Рассмотрим по шагам работу данной программы.

а) Поток 1 устанавливает значение переменной **global_variable** равной 1 и генерирует событие $MEM(\&global\_variable, WRITE, t_1)$.

б) Активные блокировки у потока отсутствуют. Поэтому в истории обращений к адресу переменной **global_variable** добавляется новая запись,

об изменении переменной потоком 1 с пустым набором блокировок:

$$History = (WRITE, t_1, L = \emptyset).$$

в) Поток 1 запускает функцию `thread2_loop` в отдельном потоке 2.

г) Поток 2 устанавливает значение переменной **global_variable** в 100 и генерируется событие $MEM(\&global\_variable, WRITE, t_2)$.

После установки потоков 2 значение переменной **global_variable** детектор сообщит о гонке при обращении к переменной `global_variable`. История обращения к **global_variable** выглядит следующим образом:

$$\begin{aligned} History = &[(WRITE, t_1, L = \emptyset), \\ &(WRITE, t_2, L = \emptyset)]. \end{aligned}$$

Пересечение активных блокировок пустое при доступе к **global_variable**, поэтому детектор сообщает о гонке. Ложное срабатывание происходит, так как данный алгоритм не позволяет учитывать такие событие как, например, запуск и ожидание завершения потока, что позволило бы упорядочить события во времени и тем самым исключить рассматриваемое ложное срабатывание.

Другой пример ложного срабатывания связан с повторным использованием одних и тех же объектов с целью уменьшения количества выделений памяти и повторных инициализаций. Потоки используют общий пул однотипных объектов, например, пул соединений с БД, и при необходимости извлекают из него объект. После того как действия с этим объектом произведены, объект добавляется обратно в пул. Извлечение и добавление объектов в пул происходит безопасно. На рисунке 1.7 приведен описанный пример.

Данный пример рассматривает программу с кодом, в котором отсутствуют гонки, но алгоритм на основе множества блокировок сообщит о гонке при работе с объектом **obj**.

Следующий пример связан со способом хранения истории доступа к памяти. Для каждой области памяти m необходимо хранить историю доступа и список активных блокировок потока при каждом доступе.

```
// Поток 1                          // Поток 2
Object* obj = new Object();         Object* obj = ObjectsPool.pop(obj);
big.action(...);                    obj.action(...);
...                                 ...
ObjectsPool.put(obj);               ObjectsPool.put(obj);
```

Рисунок 1.7 — Пример кода, вызывающего ложное срабатывание алгоритма на основе множества блокировок

Вместо хранения списков блокировок потоков можно хранить только пересечение данных списков. Данный вариант дает ложные срабатывания (рисунок 1.8), но имеет низкий расход памяти [9].

```
// Поток 1          // Поток 2          // Поток 3
lock(l1);           lock(l2);           lock(l1);
lock(l2);           lock(l3);           lock(l3);
action(obj);        action(obj);        action(obj);
unlock(l2);         unlock(l3);         unlock(l3);
unlock(l1);         unlock(l2);         unlock(l1);
```

Рисунок 1.8 — Пример кода, вызывающий ложное срабатывание при хранении пересечений активных блокировок для области памяти

Алгоритм на основе множества блокировок имеет ошибки только одного класса, а именно сообщения о гонках, которые на самом деле отсутствует в программе. Для устранения ошибочных детектирований при анализе программы, части кода программы, которые считаются безопасными, убираются из рассмотрения анализатора гонок.

1.7 Алгоритм на основе отношения предшествования

Алгоритм на базе отношения предшествования основывается на информации о частичной упорядоченности событий доступов к памяти во времени [13]. Формирование частично упорядоченного множества обращений к данной области памяти m базируется на информации полученной следующим образом:

— анализ работы с примитивоми синхронизации;

— анализ создания и завершения потоков.

Примем, что при взаимодействии потоков через примитивы синхронизации, создании и завершении потоков происходит передача сообщений или их прием(RCV, SND). Событие потока t является передающим SND, если оно несет какую-то информацию о потоке t. Событие в дальнейшем может быть прочитано любым потоком, в том числе и t. Событие потока t является принимающим, если оно считывает информацию которая была ранее сгенерирована событием SND. Событие не может быть одновременно обоих типов. Отдельно стоит отметить, что информация посланная событием SND может быть считана несколькими событиями RCV и событие RCV может считать информацию о нескольких SND событиях за раз [15].

Далее определим правила генерации событий. Рассмотрим два потока $t1$ и $t2$.

— **Создание потока**. При создании потоком t_1 потока t_2, поток t_1 генерирует событие $SND(g, t_1)$, где g некоторый идентификатор события. Поток t_2 при запуске осуществляет прием события $RCV(g, t_2)$.

— **Завершение потока**. При завершении потока t происходит генерация события $SND(g, t)$.

— **Ожидание завершения потока**. Если поток t_1 ожидает завершения потока t_2, то после завершения потока t_2 происходит событие $RCV(g, t_1)$.

— **Оповещение о событие**. При оповещении о событие(например, $pthread\_cond\_signal$) потоком t_1 генерируется событие $SND(g, t_1)$ и поток, ожидающий данное событие (например, $pthread\_cond\_wait$) сгенерирует событие $RCV(g, t_2)$.

— **Работа с блокировками**. При освобождении блокировки потоком t_1 генерируется событие $SND(g, t_1)$, а при захвате потоком t_2 — событие $RCV(g, t_2)$.

События генерируются потоком выполнения в определенном порядке и делят процесс выполнения потока на сегменты. Таким образом сегментом является это максимальная последовательность инструкций выполнения потока, которая содержит единственное событие и им заверша-

ется. Если сегмент S заканчивается событием A, то будем считать, что сегмент S определен событием A. На рисунке 1.9 сегмент S_1 потока $t1$ определен событием A. Тогда поток выполнения разбивается на сегменты, опре-

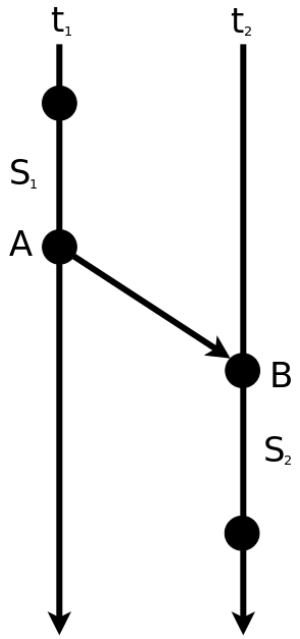

Рисунок 1.9 — Сегмент S_1 потока t_1 предшествует сегменту S_2 потока t_2

деляемые событиями генерированными данным потоком (рисунок 1.10).

Каждый сегмент потока t имеет информацию о том сколько сегментов выполнилось до него данным потоком. В тоже время сегмент обладает лишь частичным знанием о том сколько сегментов выполнилось на других потоках. Данное знание основано на полученных потоком событий и дальнейший анализ будет строиться на этом приведенном неполном знании.

Далее введем отношение порядка $\prec$ на множестве сегментов выполнения [15]. Пусть S_1 сегмент потока T_1 и S_2 сегмент потока T_2, тогда $S_1 \prec S_2$ если выполняется одно из следующих условий:

— $T_1 = T_2$ и сегмент S_1 выполнился раньше S_2;

— сегмент S_1 определен передающим данные событием(SND) A, а сегмент S_2 определен принимающим событием B, считывающем данные о событие A (см. 1.9);

— отношение $\prec$ является транзитивным, таким образом:

$$S_1 \prec S_2 \wedge S_2 \prec S_3 \Rightarrow S_1 \prec S_3$$

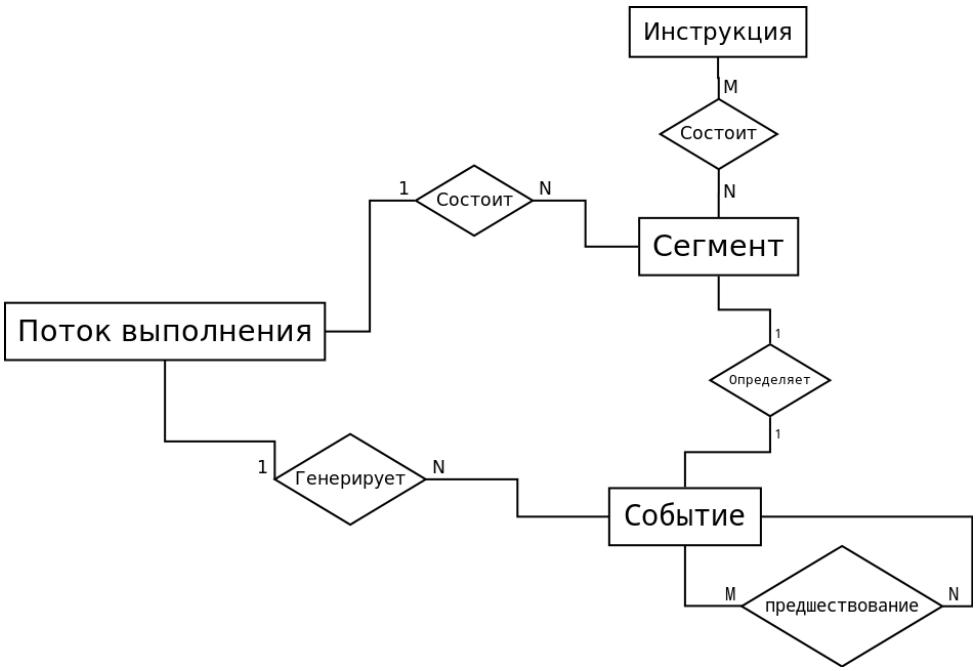

Рисунок 1.10 — Диаграмма сущностей исполняемой программы

Из определения следуют следующие следствия:

а) если $S_1 \prec S_2$, тогда сегмент S_1 завершился раньше чем сегмент S_2 запустился;

б) если сегмент S_1 завершился раньше чем сегмент S_2 запустился, то это не является достаточным условием для $S_1 \prec S_2$;

в) если время выполнения сегментов S_1 и S_2 перекрывается или S_2 завершился до запуска сегмента S_1, то $S_2 \not\prec S_1$;

г) если сегменты S_1 и S_2 выполняются на одним потоком t, то $S_1 \prec S_2$ тогда и только тогда когда S_1 выполнился раньше S_2.

Таким образом, два сегмента являются параллельными ($S_1||S_2$) тогда и только тогда когда $S_1 \not\prec S_2 \wedge S_2 \not\prec S_1$.

Для практического использования информации о параллельном выполнение сегментов необходимо дать численное выражение частичному отношению порядка $\prec$.

1.7.1 Векторные часы

Векторными часами называется алгоритм получения частичного упорядочения событий в распределённой системе. Векторные часы в си-

стеме потоков — это массив или вектор из логических часов, одни часы на поток. Локальный экземпляр вектора с наименьшими возможными значениями часов для каждого процесса строится следующим образом [16].

— Изначально все значения часов равны 0.

— В случае внутреннего события счётчик текущего потоку увеличивается на 1.

— Перед отправкой сообщения внутренний счётчик, соответствующий текущему потоку, увеличивается на 1, и вектор целиком прикрепляется к сообщению.

— При получении сообщения счётчик текущего процесса увеличивается на 1, далее значения в текущем векторе выставляются в максимум от текущего и полученногом [16].

Таким образом, для векторных часов V потока t значение $V[i]$ означает последнее событие потока t_i, которое могло повлиять на поток t. Так как сегмент определяется одним событием, то для упорядочивания сегментов достаточно упорядочить события их образующие. Для этого при генерации события к нему прикрепляются векторные часы генерирующего событие потока.

Ниже приведен алгоритм расчета векторных часов $V_i(t)$ потока t в момент генерации события e_i с прикрепленными к нему векторными часами $V(g)$.

$$
\begin{aligned}
V_o(t)(t) &= 1 \\
V_o(t_1)(t_2) &= 0, t_1 \neq t_2 \\
V_i(t)(t) &= V_{i-1}(t)(t) + 1, \text{если } e_i = SND(g,t) \\
V_i(t_1)(t_2) &= max(V_{i-1}(t_1)(t_2), V(g)(t_2)), \\
&\quad e_i = RCV(g,t_1) \wedge t_1 \neq t_2 \\
V_i(t_1)(t_2) &= V_{i-1}(t_1)(t_2), \text{иначе} \\
V(g) &= V_i(t), e_i = SND(g,t)
\end{aligned}
$$

Алгоритмическая сложность поддержки векторных линейно зависит от числа когда либо запущенных потоков в процессе ($O(|T|)$) и требует

$O(|T|)$ памяти для каждого потока и для каждого события, которое может быть получено в будущем.

1.7.2 Обнаружение гонок

Гонка возникает между сегментами S_1 и S_2 при доступе к области памяти m, если эти сегменты параллельны $S_1||S_2$ и доступ одного из сегментов к m на запись.

Пусть S_1 сегмент потока t_1 и S_2 сегмент потока t_2. Обозначим через R_{s_1} множество областей памяти чтение которых осуществлялась в сегменте S_1, множество областей в которую осуществлялось чтение обозначим через W_{s_1}. Аналогично для сегмента S_2: R_{s_2} и W_{s_2}. Тогда гонка между сегментами S_1 и S_2 возникает при выполнении следующих условий:

— $S_1||S_2$
— $(R_{s_1} \cup W_{s_1}) \cap W_{s_2} \neq \emptyset \vee (R_{s_2} \cup W_{s_2}) \cap W_{s_1} \neq \emptyset$

Данный алгоритм в отличие от алгоритма на основе множества блокировок не дает ложных срабатываний, но обнаруживает не все гонки. На рисунке 1.11 показан пример гонки, которая не будет обнаружена алгоритмом на основе отношения предшествования. Гонка возникает при доступе к переменной x.

Рассмотрим более детально выполнение данной программы (таблица 1.2). В столбце $V(t_i)$ указано значение векторных часов после завершения сегмента потоком t_i (в $V(e)$ векторные часы прикрепленные к сгенерированному событию).

Изначально значение векторных часов равно $(1, 0)$ и $(0, 1)$ для потоков t_1 и t_2 соответсвенно. Далее следуя правилам генерации событий (см. раздел 1.7) значение векторных часов становятся $(2, 0)$ и $(2, 2)$ соответсвенно для t_1 и t_2.

Покажем, что данный алгоритм не обнаруживает гонку при доступе к переменной x между сегментами S_1^1 и S_2^3. Значение векторных часов сегмента $S_1^1 = (1, 0)$, сегмента $S_2^3 = (2, 2)$. Таким образом, $S_1^1 \prec S_2^3$, то есть доступ к переменной x из сегментов S_1^1 и S_2^3 является упорядоченным, поэтому детектор не сообщает об обнаружении гонки. Проблема заключается

Таблица 1.2 — Посегментное выполнение примера

Сегмент	$V(t_1)$	$V(t_2)$	Тип события e	$V(e)$
S_1^1	$(1,0)$	$(0,1)$	—	—
S_1^2	$(1,0)$	$(0,1)$	SND	(2,0)
S_2^1	$(2,0)$	$(0,1)$	RCV	(2,0)
S_2^2	$(2,0)$	$(2,1)$	SND	(2,2)
S_2^3	$(2,0)$	$(2,2)$	SND	(2,3)

в том, что отношение предшествования не всегда соответствует тому, что доступ к данным синхронизирован.

1.8 Среда выполнения при динамическом поиске гонок

Для динамического поиска гонок необходимо генерировать события обращений к памяти и события работы с примитивами синхронизации. На основе данных событий в зависимости от используемого алгоритма поиска гонок осуществляется генерация сегментов выполнения потоков, поддержка векторных часов потоков и списков активных блокировок потока. Генерация потока событий реализуются двумя основными подходами:

— реализация на базе виртуальной машины;

— модификация кода программы во время компиляции и переопределение системных функций по работе с примитивами синхронизации.

Реализация на базе виртуальной машины осуществляет интерпретацию исполняемого файла, что позволяет производить анализ программы и используемых ей библиотек без необходимости перекомпиляции и внесения изменений в бинарные файлы. Такие анализаторы как `Helgrind` и `Thread Checker` реализованы таким образом [1, 2].

ПО `Helgrind` является расширением виртуальной машины `Valgrind` [9]. `Valgrind` использует JIT-компиляцию. То есть, оригинальная программа не выполняется непосредственно на основном процессоре. Вместо этого, Valgrind сначала транслирует программу во

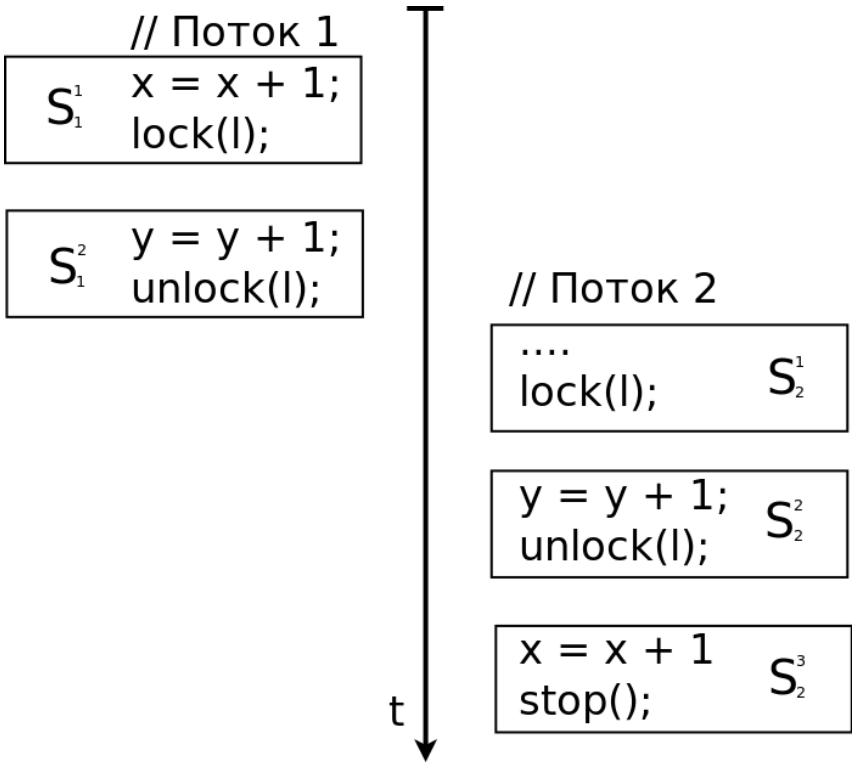

Рисунок 1.11 — Пример кода в котором не происходит обнаружение гонки алгоритмом на основе отношения

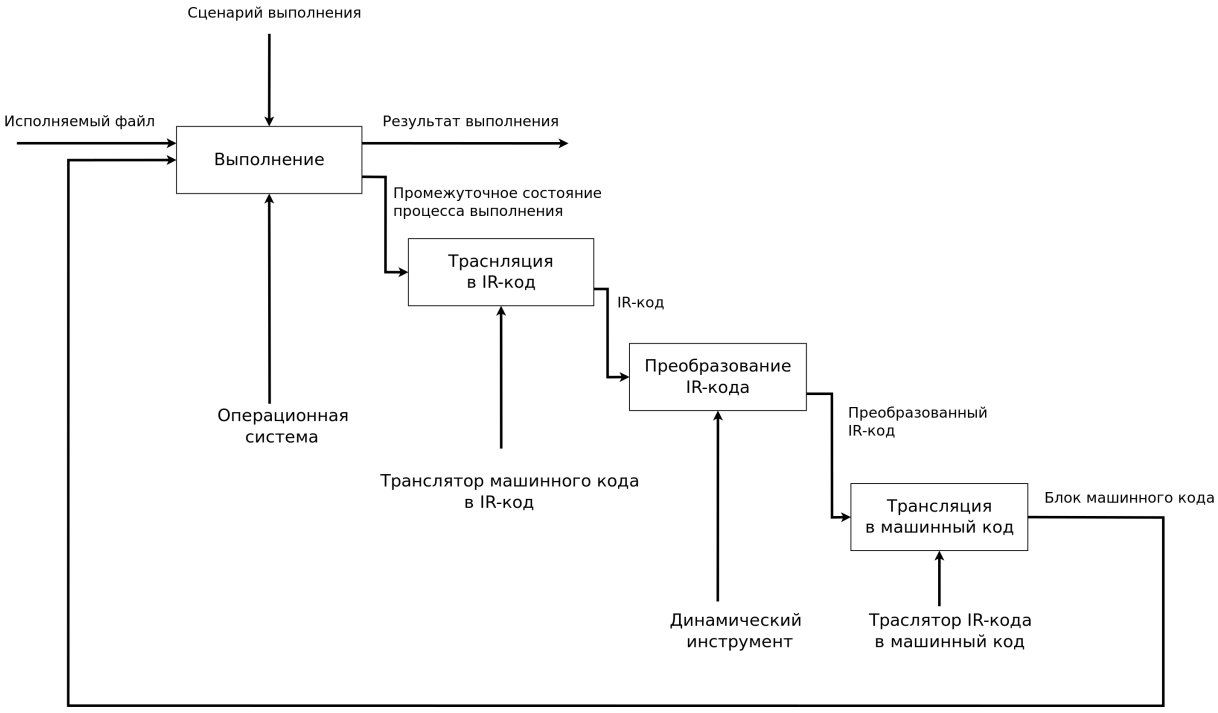

Рисунок 1.12 — Схема выполнения программы в Valgrind

временную, более простую форму, называемую промежуточным представлением, которая сама по себе не зависима от процессора. После

преобразования инструмент может выполнять любое необходимое преобразование над IR-кодом, до того как `Valgrind` оттранслирует IR-код обратно в машинный код и позволит основному процессору его исполнить. `Valgrind` перекомпилирует двоичный код для запуска на основном и целевом (или его симуляторе) процессорах одинаковой архитектуры. Несмотря на использование различных методик ускорения, исполняемый файл, запущенный под `Valgrind` и "пустым" (не выполняющим никаких функций) инструментом, работает в 4-5 раз медленее по сравнению с исполнением кода напрямую [17].

Второй подход заключается в модификации кода программы на этапе компиляции в машинный код или внесение изменений в исполняемый файл. В исполняемый файл вносятся инструкции по генерации событий при доступе к памяти. Для генерации событий при работе с примитивами синхронизации производится переопределение системный функций. Таким образом события, необходимые для алгоритма поиска гонок, можно разделить на два класса по источнику их возникновения:

— события, генерируемые инструкциями вставленными компилятором в исполняемый код программы на этапе компиляции;

— события генерируемые переопределенными системными функциями.

На вход модифицированному компилятору поступает файл с исходным текстом компилируемого модуля (рисунок 1.13). Компилятор производит модификацию кода программы во время компиляции и осуществляет вставку инструкций следующих типов:

— генерация событий обращений к памяти;

— генерация новых сегментов выполнения, в случае использования алгоритма на основе отношения предшествования.

Для генерации необходимых событий при использовании функций POSIX API по работе с потоками и примитивами синхронизации применяется механизм переопределения системных функций на уровне компоновщика объектных файлов (рисунок 1.13).

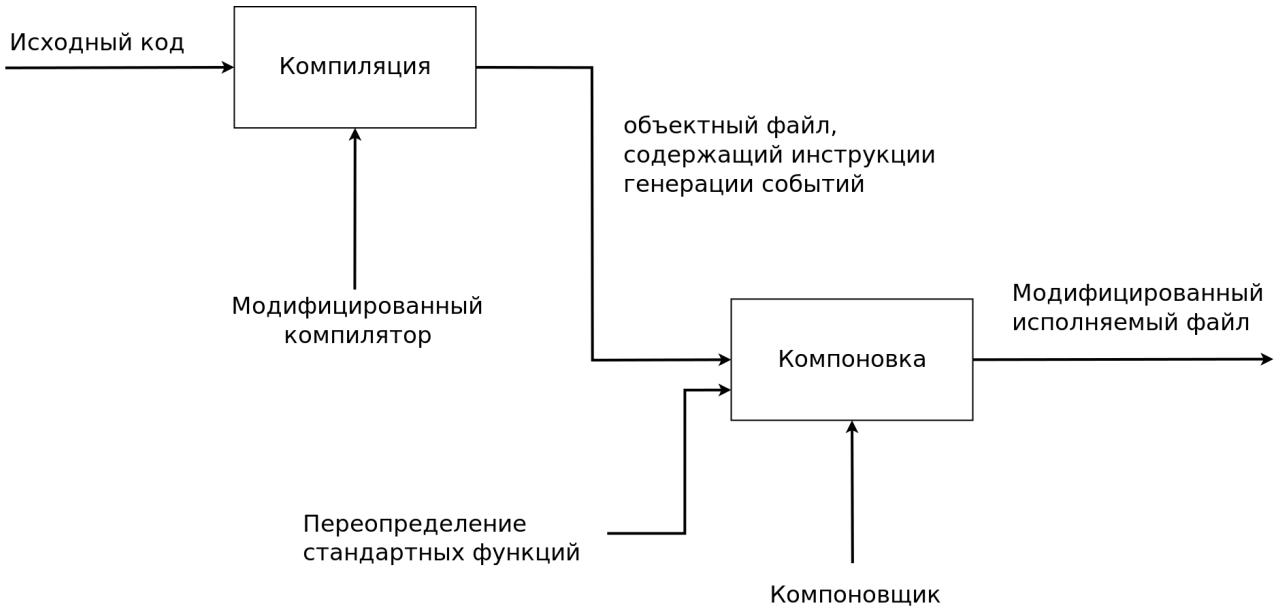

Рисунок 1.13 — Схема получения модифицированного исполняемого файла

Таким образом генерация исполняемого файла происходит следующим образом:

— генерация компилятором модифицированных объектных файлов модулей ПО из исходного кода;

— генерация объектного файла с переопределенными стандартными механизмами синхронизации и функциями анализа гонок;

— компоновка полученных объектный файлов.

Полученный модифицированный исполняемый файл при выполнении генерирует поток событий и производит анализ существования гонок на основе данных событий.

Плюсами данного подхода являются:

— высокая производительность;

— при компиляции исходного кода программы происходит анализ сущностей уровня языка, а не машинных команд, что позволяет производить дополнительный анализ.

Минусом реализации динамического анализатора с применением модифицированного компилятора и переопределенный системных библио-

тек является необходимость перекомпиляции используемых анализируемой программой библиотек.

Выводы

В ходе проведенного анализа рассмотрены базовые алгоритмы применяемые в при динамическом поиске гонок:

— алгоритм на базе множества блокировок;
— алгоритм на базе отношения предшествования.

Алгоритм на основе множества блокировок дает ложные срабатывания, но позволяет реализовать эффективный анализатор. Ложные срабатывания возникают, так как данный алгоритм производит анализ основываясь только на событиях доступа к памяти и захвата или освобождения блокировок, но не позволяет упорядочить события во времени и учитывать операции работы с потоками и условными переменными. Алгоритм на основе отношения предшествования не сообщает ложные срабатывания, но некоторые типы гонок не обнаруживает. Алгоритмическая сложность поддержки векторных часов и ведения списка сегментов, которые имели доступ к данной области памяти зависит линейно от количества потоков и числа обращений к памяти.

Также рассмотрены подходы к реализации динамических анализаторов для генерации потока событий во время выполнения программы для его дальнейшего анализа:

— реализация на базе виртуальной машины;
— модификация кода программы на этапе компиляции и переопределение системных функций.

Основным преимуществом первого подхода является возможность анализа программы без предворительной подготовки исполняемых файлов и используемых ей библиотек. Недостатком является значительное снижение скорости выполнения анализируемой программы.

2 Метод динамического поиска гонок в программах, реализованных на языке Си

В разделе описан метод динамического поиска гонок в программах, реализованных на языке Си и использующих POSIX API для работы с потоками и примитивами синхронизации. Метод динамического поиска гонок включает в себя принципиальную схему реализации анализатора и алгоритм поиска гонок.

2.1 Гибридный алгоритм поиска гонок

Алгоритм на основе множества блокировок генерирует ложные срабатывания и позволяет осуществлять их фильтрацию только исключением из анализа части исходного кода программы, в то же время алгоритм на основе отношения предшествования не находит гонки некоторых типа.

2.1.1 Описание гибридного алгоритма

Для того чтобы исключить ложные срабатывания генерируемые алгоритмом на основе множества блокировок добавим поддержку упорядочивания частей выполнения программы с помощью отношения предшествования в алгоритм на основе множества блокировок.

Для этого система должна для каждого потока выполнения t поддерживать список захваченных блокировок $L(t)$.

Так же как и в алгоритме описанном в разделе 1.7, выполнение программы разбивается на сегменты выполнения. Сегменты определяют области потока выполнения, содержащие только обращения к памяти и завершаются событием SND или RCV. Для каждого сегмента записывается список активных блокировок потока, выполняющего сегмент.

Для каждой области памяти m ведется два списка сегментов из которых осуществлялся доступ к ней: R и W. R — список сегментов из которых осуществлялось чтение, W — список сегментов из которых осуществлялась запись.

Гонка при доступе к памяти m между сегментами S_i и S_j происходит при следующем условии:

$$S_i || S_j \wedge L_i \cap L_j = \emptyset.$$

Таким образом, необходимо хранить состояние для каждого потока и каждой области памяти к которой осуществлялся доступ. Состояние потока содержит в себе векторные часы потока и захваченные блокировки. Состояние ячейки памяти содержит список R и W. Каждый сегмент описывается списком блокировок и векторными часами потока на момент выполнения данного сегмента.

Для поддержки актуальности состояния системы необходимо осуществлять следующие действия:

— обработка доступа к памяти m;
— обработка захвата/обработки блокировок;
— обработка событий SND/RCV;
— генерация нового сегмента S;
— проверка существования гонок.

Алгоритм обработки событий, генерируемых анализируемой программой, представлен далее (рисунок 2.1).

а) Если произошел доступ к памяти перейти на шаг г.

б) Если произошел захват или освобождение блокировки, произвести обновление списка активных блокировок потока и перейти на шаг г.

в) Произошло событие SND или RCV. Необходимо обновить векторные часы потока и перейти на шаг г.

г) Произвести обработку события доступа к памяти:

1) обновить историю доступа к данной области памяти;
2) произвести поиск гонок;
3) перейти на шаг г.

д) Завершение.

При обработки события $MEM(m_i, a_i, t_i)$ осуществляются следующие действия:

Рисунок 2.1 — Обработка потока событий гибридным алгоритмом

а) Если произошла запись, то в список сегментов W добавляется текущий сегмент S потока t_i. Если чтение произошло чтение, то происходит добавление сегмента в список R.

б) Вызывается проверка гонок для данной области m_i.

Такая реализация требует добавления записи в список сегментов при каждом обращении к области памяти.

Гибридный алгоритм также как и алгоритм на множестве блокировок генерирует ложные срабатывания. Но введение отношения предшествования позволяет ограничить их введением аннотаций кода, производящих генерацию и прием сообщений для упорядочивания сегментов выполнения и устранения ложного срабатывания.

2.1.2 Анализ потока событий гибридным алгоритмом

Рассмотрим примеры кода, приведенные при разборе алгоритмов на основе множества блокировок и отношения предшествования.

Код, представленный на рисунке 1.6, генерирует ложное срабатывание при использовании алгоритма на основе множества блокировок. Однако при использовании отношения предшествования ложное срабатывание не происходит. Гибридный алгоритм при анализе данного кода также не генерирует ошибочное сообщение о найденном состоянии гонок. При применении гибридного алгоритма доступ к переменной $global\_variable$ является упорядоченным, так как при создании потока генерируется событие SND и RCV, и происходит увеличение значения векторный часов, поэтому сегменты в которых происходит обращение к переменной $global\_variable$ находятся в отношении предшествования.

На рисунке 1.11 алгоритм на основе отношения предшествования не обнаруживал гонку при доступе к переменной x. Гибридный алгоритм данную ошибку обнаружит, поскольку векторные часы во время выполнения данных фрагментов кода не изменяются. Таким образом, сегменты S_1^1 и S_2^3 являются параллельными ($S_1^1 || S_2^3$), а множества блокировок данных сегментов пустые. Поэтому детектор сообщит об обнаружении гонки при доступе к переменной x между сегментами S_1^1 и S_2^3.

При использовании пулов объектов (рисунок 1.7) гибридный алгоритм генерирует ложное срабатывание, так как доступ к переменной obj неупорядоченный, то есть сегменты из которых осуществляется доступ к данной переменной параллельны, а пересечение множеств блокировок сегментов пустое. Решением данной проблемы является внесение в код специ-

альных аннотаций кода, генерирующих события SND и RCV для упорядочивания сегментов. В общем случае аннотирование позволяет избежать все ложные срабатывания.

Таким образом, в качестве альтернативы алгоритму на основе множества блокировок и алгоритму на основе отношения предшествования предложен гибридный алгоритм, основанный на множестве блокировок, но использующий отношение предшествования для упорядочивания сегментов выполнения анализируемой программы. Данное решение позволяет исключить ложные срабатывания и ввести аннотации в код, осуществляющие фильтрацию ложных срабатываний. Так как данный алгоритм производит подмножество событий генерируемых алгоритмом на основе отношения предшествования, то поддержка векторных часов потоков выполнения требует меньших накладных расходов.

2.2 Ограничение глубины истории доступа к памяти

При использовании гибридного алгоритма поиска гонок для каждой области m необходимо хранить два списка сегментов (R и W) для хранения сегментов из которых осуществлялась запись и чтение, то есть для каждой области памяти хранится история всех обращений. Таким образом создаются дополнительные накладные расходы для поддержания актуальности истории обращений анализируемой программы.

Пусть дана последовательность сегментов S из которых осуществлялся доступ к области памяти m. Гонка может возникать между любыми двумя сегментами из списка. Поэтому ограничение количества элементов в списке сегментов памяти m не позволит выявить ряд гонок, причем поведение анализатора будет недетерминированым и зависеть от порядка в каком выполняются потоки [15].

Для уменьшения глубины истории в списке сегментов из которых осуществлялось чтение R хранятся только такие сегменты, что любой сегмент из множества R произошел после любого сегмента из W или параллелен с ними:

$$\forall S' \in R, S'' \in W : S' \not\preceq S''.$$

Добавление нового сегмента S при записи в множество сегментов из которых осуществляется запись W происходит следующим образом:

$$R \leftarrow \{s : s \in R \wedge s \not\trianglelefteq S\},$$
$$W \leftarrow \{s : s \in W \wedge s \not\trianglelefteq S\} \cup \{S\}.$$

При добавлении сегмента S при записи в список W происходит очистка W и R от элементов, которые произошли раньше сегмента S [9].

Добавление нового сегмента S в множество сегментов R из которых осуществлялась чтение происходит следующим образом:

$$R \leftarrow \{s : s \in R \wedge s \not\trianglelefteq S\} \cup \{S\}.$$

На рисунке 2.2 представлен алгоритм обновления списков истории доступа к памяти. Рассмотрим данный алгоритм по шагам. Пусть произошел доступ к памяти m из сегмента S_a.

а) Получить список сегментов W и R из которых осуществлялся доступ на запись и на чтение памяти m соответсвенно.

б) Если произошла запись, то произвести обновление списков W и R.

1) Каждый элемент S из списка R сравнить с добавляемым сегментом S_a, если $S \not\trianglelefteq S_a$, тогда сегмент S в списке R остается, иначе удаляется из него.
2) Каждый элемент S из списка W сравнить с добавляемым сегментом S_a, если $S \not\trianglelefteq S_a$, тогда сегмент S в списке W остается, иначе удаляется из него.
3) Добавить сегмент S_a в список W.
4) Перейти на шаг г.

в) Если произошло чтение, то необходимо произвести обновление списка R.

1) Каждый элемент S из списка R сравнить с добавляемым сегментом S_a, если $S \not\trianglelefteq S_a$, тогда сегмент S в списке R остается, иначе удаляется из него.
2) Добавить сегмент S_a в список R.

г) Завершение.

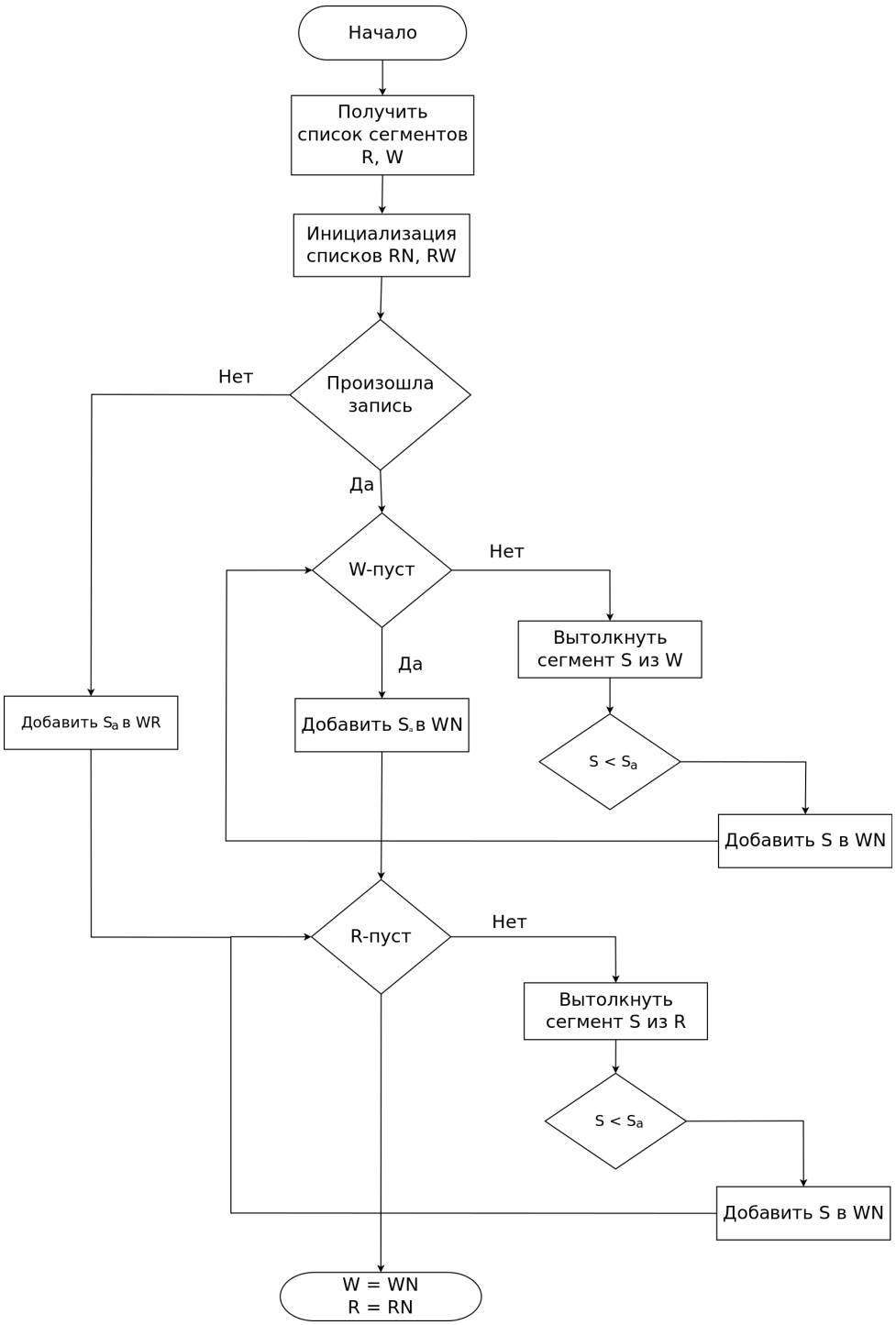

Рисунок 2.2 — Схема алгоритма обновления списков истории доступа к памяти

Алгоритм позволяет ограничить глубину истории, что позволяет уменьшить накладные расходы памяти и уменьшить число необходимый проверок состояния гонок между сегментами в гибридном алгоритме.

2.3 Разделение потока выполнения на сегменты

Анализатор сообщает о гонке при текущем доступе к памяти m и каким-то сегментом, который обращался к данной области ранее. Таким образом можно сообщить точное место в коде текущего обращения, но точность сообщения о предыдущем доступе зависит от размера сегмента с которым произошли гонки.

Каноническим разделением на сегменты является создание сегментов при генерации событий синхронизации RCV и SND. Сегмент в таком случае может покрывать различные функции программы, что усложняет процесс анализа сообщения о гонке и локализацию ошибки.

Рассмотрим альтернативные варианты разбиения на сегменты [9].

— При каждом доступе к памяти создается новый сегмент. Данное решение позволяет точно показать место предыдущего доступа к данной области памяти, но требует дополнительные расходы как процессорного времени так и памяти.

— Создание сегмента при входе в базовый блок (один вход и один выход) или при событии синхронизации. В этом случае сегмент покрывает достаточно небольшую область кода.

Последний из вариантов является наиболее предпочтительным как по используемой памяти, так и по вычислительной сложности. В то же время размер блока достаточно маленький и покрывает логически связную область кода, что упрощает анализ отчета о найденной гонке.

2.4 Выбор среды выполнения

Для генерации потока событий для гибридного алгоритма поиска гонок необходима модифицированная среда выполнения. В качестве подхода к реализации динамического анализатора в разработанном методе выбрана модификация кода на этапе компиляции модулей программы с переопределением POSIX функций по работе с примитивами синхронизаций и потоками. Данное решение позволяет реализовать анализатор с низкими накладными расходами из-за отсутствия интерпретации машинного кода, используемой при построении анализаторов на базе виртуальной машины.

Таким образом, события используемые гибридным алгоритмом имеют два истоничка (рисунок 2.3):

— обращения к памяти и генерация сегментов выполнения потоков генерируются инструкциями вставленными компилятором в исполняемый код программы на этапе компиляции;

— события SND, RCV и создание или освобождение блокировок происходят в переопределенных POSIX функциях.

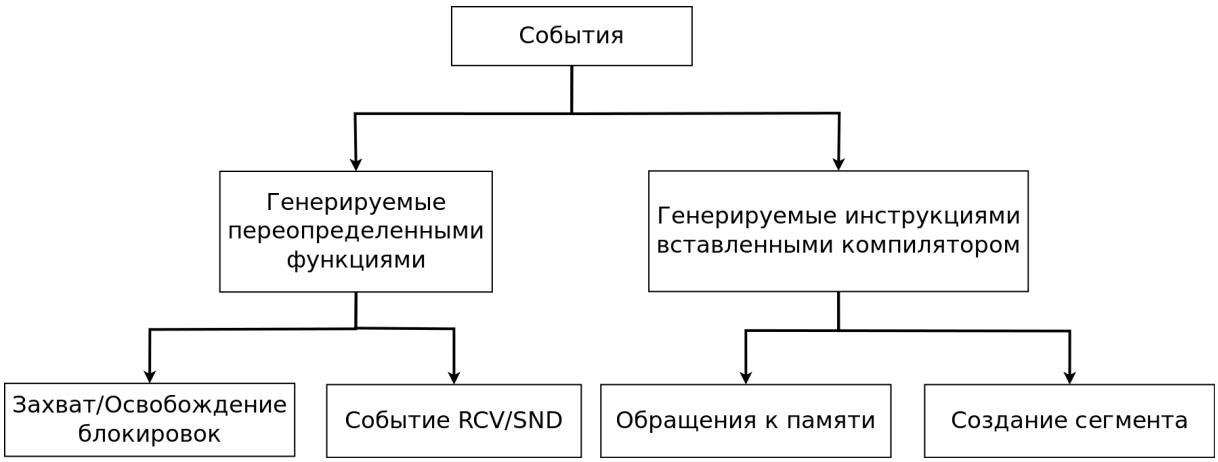

Рисунок 2.3 — Типы событий

Схема выполнение динамического поиска гонок в программах на языке Си в разработанном методе состоит из следующих этапов (рисунок 2.4).

— Компиляция кода программы модифицированным компилятором языка Си;

— Компоновка модулей анализируемой программы с модифицированной реализацией POSIX API.

— Запуск тестовых сценариев программы.

— Программа во время своего выполнения генерирует поток событий доступа к памяти и событий операций с примитивами синхронизации.

— Анализатор на основе гибридного алгоритма поиска гонок осуществляет обработку событий и обнаружение гонок.

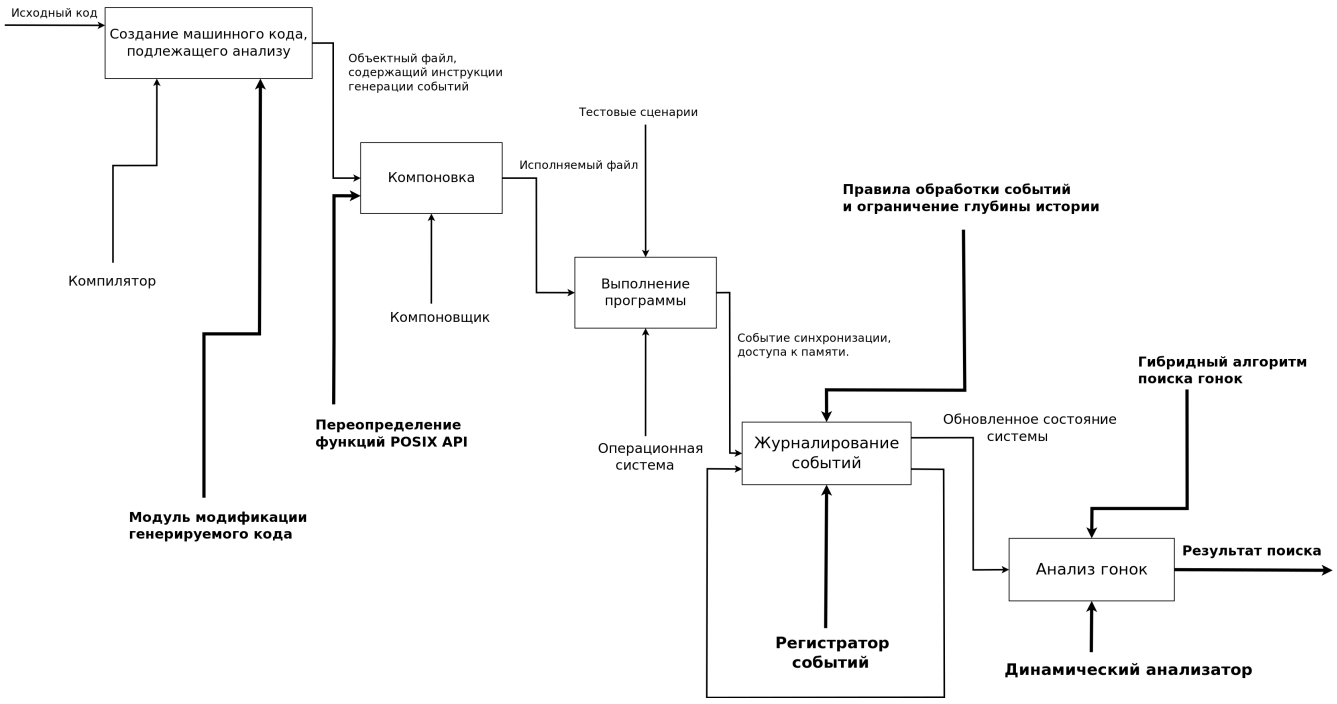

Рисунок 2.4 — Схема метода динамического поиска гонок в программах на языке Си

Выводы

В предложенном методе динамического поиска гонок в программах на языке Си используется гибридный алгоритм поиска гонок с ограничением глубины истории обращений памяти. Данный алгоритм базируется на алгоритме на основе множества блокировок с использованием отношения предшествования для упорядочивания сегментов выполнения потоков, что позволяет исключить ряд ложных срабатываний. Ограничение глубины истории позволяет использовать предложенный метод при анализе программ с интенсивной работой с памятью. Генерация сегментов выполнения производится при входе в базовые блоки кода программы. Такое разделение на сегменты при формировании отчета о найденных гонках позволяет сформировать корректный стектрейс предыдущих обращений к памяти и указать номер строки начала базового блока.

Для реализации динамического анализатора в методе используется модифицированный компилятор языка Си и переопределение POSIX функций по работе с потоками и примитивами синхронизации.

3 Проектирование и реализация динамического анализатора

В данной главе рассмотрено проектирование анализатора поиска гонок и его реализация. Анализатор реализует метод динамический поиск гонок в программах на языке Си с использованием гибридного алгоритма поиска гонок с ограниченной глубиной истории.

3.1 Общая схема анализатора

Реализация динамический анализатор состоит из следующих компонентов:

— Модуль компилятора, осуществляющий модификацию генерируемого машинного кода;

— библиотека переопределенных POSIX функций для работы с потоками и примитивами синхронизации;

— библиотека регистрации событий;

— библиотека содержащая реализацию гибридного алгоритма поиска гонок.

На рисунке 3.1 показаны связи между данными компонентами. Модуль компилятора, осуществляет генерацию модифицированного исполняемого файла, который во время своего выполнения производит генерацию событий. Библиотека, содержащая переопределенные POSIX функции для работы с потоками и примитивами синхронизации, необходима для генерации событий типа SND и RCV для упорядочивания сегментов выполнения потоков. Функции данной библиотеки вызываются во время выполнения анализируемой программы. Библиотека регистрации событий предоставляет интерфейс через который осуществляется фиксация событий в журнале выполнения потоков. Библиотека, содержащая реализацию гибридного алгоритма поиска гонок на основе журнала выполнения потоков, осуществляет поиск гонок и формирование отчета на основе журнала выполнения.

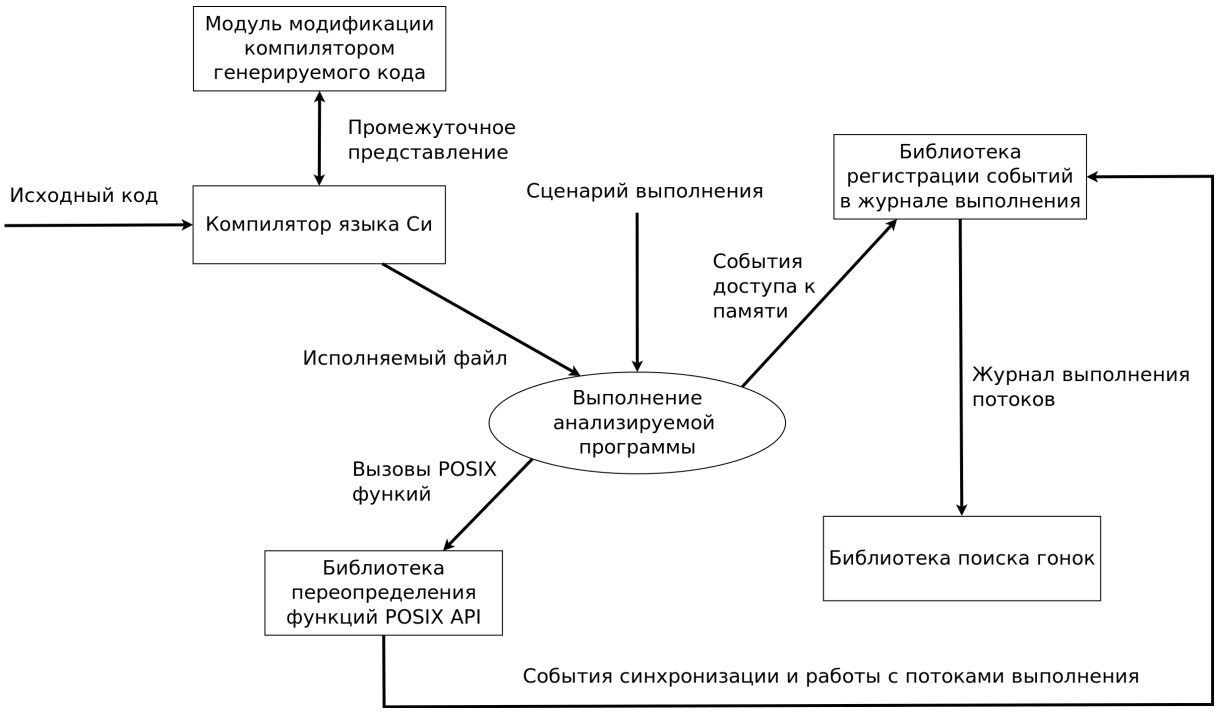

Рисунок 3.1 — Основные компоненты динамического анализатора и связь между ними

В разрабатываемом анализаторе переопределены следующие функции:

— **pthread_create** — создание потока;
— **pthread_join** — ожидание завершения потока;
— **pthread_mutex_lock** — захват мьютекса;
— **pthread_mutex_unlock** — освобождение мьютекса;
— **pthread_mutex_trylock** — неблокирующий захват мьютекса;
— **pthread_cond_wait** — блокировка на условной переменной;
— **pthread_cond_signal** — разблокировка потока заблокированного на условной переменной;

Рассмотрим обработку некоторых из них. На рисунке 3.2 показана диаграмма последовательности при создании нового потока выполнения t_2 потоком t_1 и ожидания завершения потока t_2 потоком t_1. Сначала в потоке выполнения t_1 происходит вызов переопределенной POSIX функции **pthread_create** служащей для создания нового потока. В переопределенной функции происходит генерация события SND и вызов исходной функции **pthread_create** из стандартной библиотеки Си. Далее происхо-

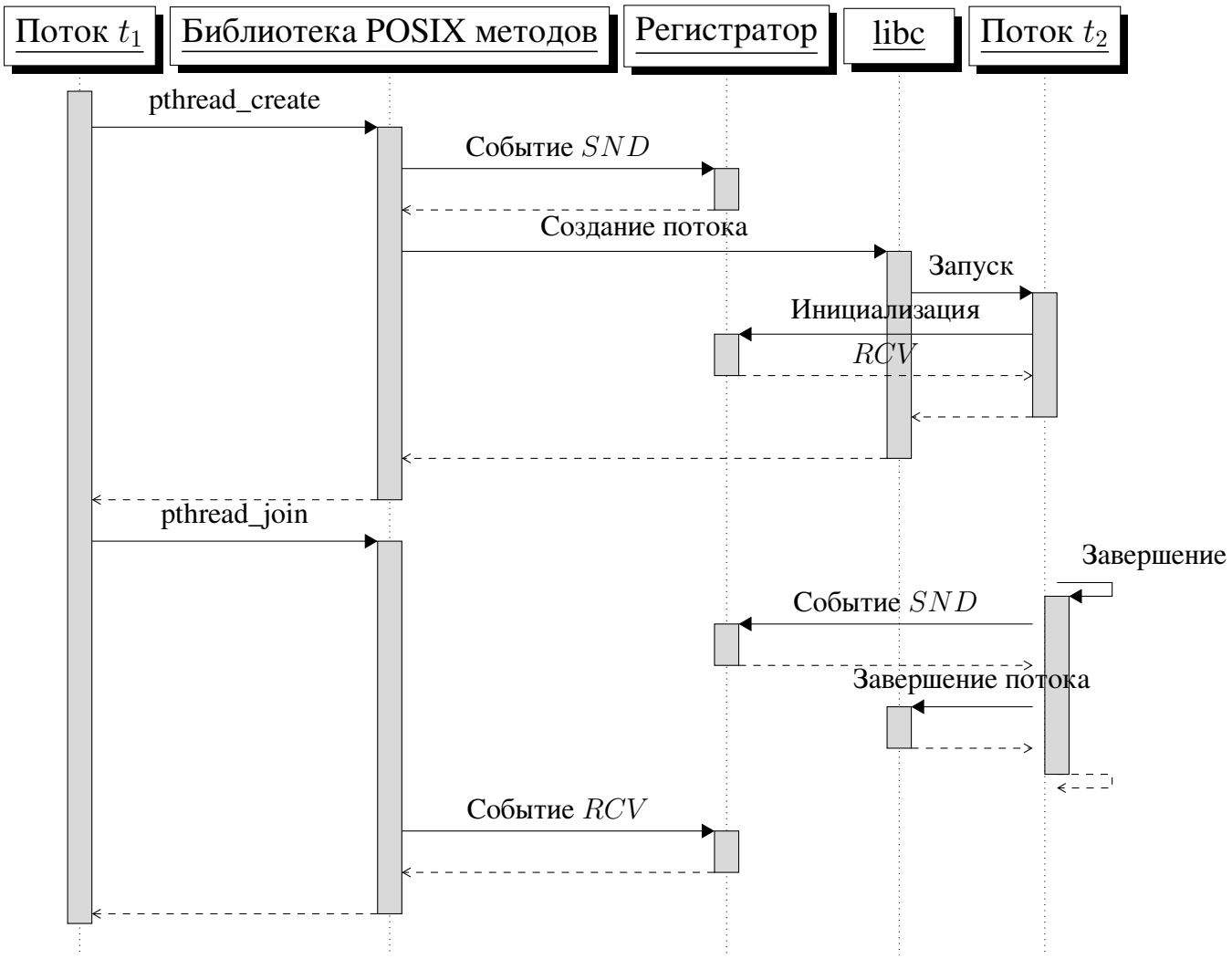

Рисунок 3.2 — Диаграмма создание и завершения потока

дит запуск потока выполнения t_2. При старте поток t_2 осуществляет вызов библиотеки регистрации событий для инициализации журнала выполнения. При завершении потока осуществляется журналирование события SND через библиотеку регистрации событий. Если какой-либо поток ожидал завершение остановленного потока, то через вызов переопределенной функции **pthread_join** осуществляется регистрация события RCV в журнале выполнения ожидающего потока.

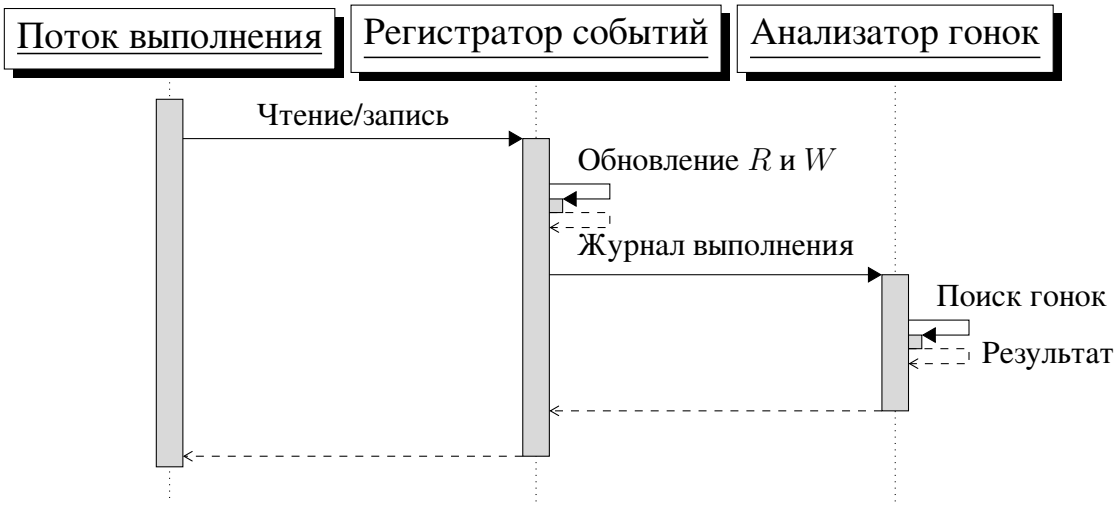

Рисунок 3.3 — Диаграмма обработки доступа к памяти

На рисунке 3.3 показана диаграмма последовательности при доступе к памяти. При доступе потоком выполнения к памяти на чтение или запись происходит вызов функции регистрации доступа к памяти. Регистратор события производит обновление списков сегментов из которых осуществлялся доступ к данной области памяти в соответствии с алгоритмом поддержки ограниченной истории. После того как состояние обновлено, происходит поиск гонок гибридным алгоритмом и сообщение о гонке, если она найдена.

На рисунке 3.4 показана диаграмма последовательности при захвате **pthread_mutex_lock** или освобождения **pthread_mutex_unlock** блокировки. При вызове данных функций потоком выполнения происходит вызов обработчика в регистраторе событий. Регистратор событий производит обновление списка активных блокировок потока, который произвел вызов функции.

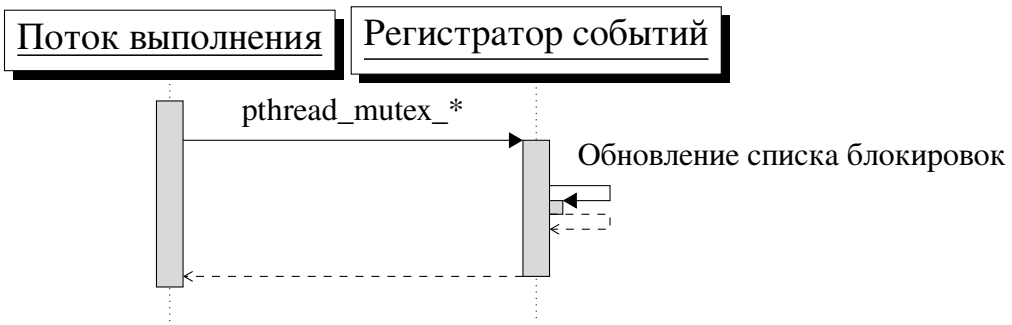

Рисунок 3.4 — Диаграмма обработки захвата и освобождения блокировок

3.2 Выбор компилятора для модификации

Для генерации событий доступа к памяти и генерации сегментов выполнения потока, необходимо произвести модификацию компилятора языка Си. В настоящее время существует ряд компиляторов языка Си с открытым исходным кодом. Самыми популярными являются:

— **GCC** (GNU C Compiler) — открытый компилятор языка Си, разрабатываемый проектом **GNU**.

— **CLang** — транслятор языка Си в промежуточное представление виртуальной машины LLVM.

Оба компилятора имеют широкое распространение и поддерживают существующие стандарты.

Компилятор GCC

Компилятор **GCC** является лидером по количеству процессоров и операционных систем, которые он поддерживает. Поэтому модификация **GCC** позволит производить запуск динамического анализатора гонок на различных платформах. К минусам относится сложность внесения изменений в исходный код компилятора, так как не предоставляется удобное API для реализации собственных оптимизаторов.

Компилятор CLang

$LowLevelVirtualMachine$ ($LLVM$) — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с $RISC$-подобными инструкциями. Может использоваться как оп-

тимизирующий компилятор этого байткода в машинный код для различных архитектур либо для его интерпретации и JIT-компиляции.

Компиляция из языка программирования в промежуточное представление реализуется в соответсвующем фронтенде $LLVM$. Для языка C в качестве фронтенда используется $CLang$. После преобразования фронтендом исходного кода в $IR-$ происходит наложение цепочки оптимизаторов и получение объектного файла для заданной архитектуры. Далее для получения исполняемого файла происходит компоновка (например, с помощью ld) объектных файлов с необходимыми библиотеками (3.5).

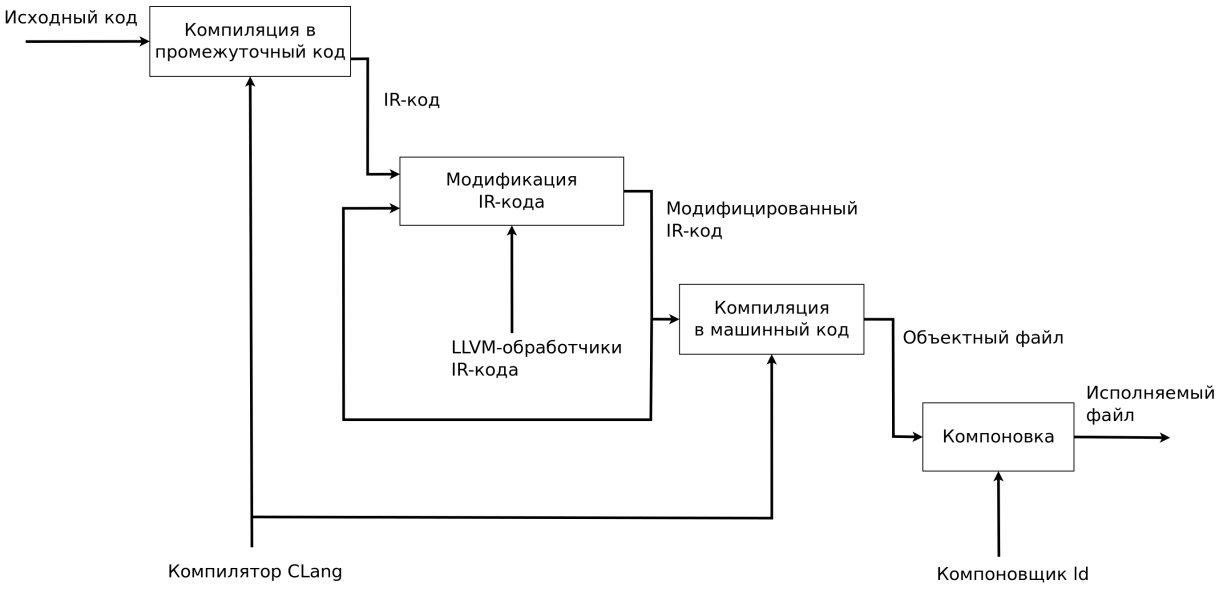

Рисунок 3.5 — Компиляция исходного кода CLang компилятором

$LLVM$ предоставляет возможность включения собственных обработчиков AST-деревьев в процесс компиляции IR-кода в машинный код. Обработчики производят преобразования и оптимизации кода, сбор аналитических данных во время компиляции. $LLVM$ предоставляет гибкое и документированное API для реализации собственных обработчиков.

Таким образом, для генерации модифицированного исполняемого файла реализован обработчик AST-деревьев в рамках платформы $LLVM$.

Все обработчики $LLVM$ являются наследниками класса $Pass$, которые реализует необходимую функциональность переопределением виртуальных функций отнаследованных от класса $Pass$. В зависимости от цели оптимизатора $LLVM$ представляет несколько типов базовых

проходов: $ModulePass$, $CallGraphSCCPass$, $FunctionPass$, $LoopPass$, $RegionPass$, $BasicBlockPass$.

$ModulePass$ является наиболее общим проходом, позволяющий добавлять, удалять функции или изменять код любой функции. Для определения корректного прохода уровня модуля необходимо создать класс-наследник от $ModulePass$ и переопределить функцию $runOnModule$:

```
virtual bool runOnModule(Module &M) = 0;
```

В данной функции необходимо производить все необходимые модификации кода. В случае если код изменен, то отптимизатор должен вернуть $true$.

3.3 Переопределение стандартных функций

Для генерации событий захвата, освобождения блокировок и событий RCV, SND при создании, завершения потоков или при работе с условными переменными необходимо переопределить POSIX функции. Для переопределения функций используется флаг компоновщика ld $--$**wrap=symbol**. Данный идентификатор **symbol** будет привязан к переопределенной функции **__wrap_symbol**, а неопределенный идентификатор **__real_symbol** к исходной реализации **symbol**.

Таким образом, при вызове системной функции **symbol** будет вызвана переопределенная **__wrap_symbol**, а для вызова исходной необходимо вызвать **__real_symbol**.

Далее приведен пример переопределения функции `malloc`:

```
void *
__wrap_malloc (size_t c)
{
    printf ("malloc called with %zu\n", c);
            return __real_malloc (c);
}
```

При вызове компоновщика с флагом $--$**wrap malloc** в полученном объектном файле вместо системного вызова **malloc** будет происходить вызов обертки **__wrap_malloc**.

3.4 Модификация кода во время компиляции

Для модификации анализируемой программы во время компиляции определен обработчик в платформе **LLVM** осуществляющий добавление вызовов следующих функций:

— **void new_segment(int line, char* file, char* dir)** — генерация события входа в новых сегмент выполнения программы;

— **void mem_access(int* addr, short is_write)** — генерация события обращения к памяти на чтение или запись;

— **void enter_function(char* name)** — генерация события входа в функцию;

— **void leave_function()** — генерация события о выходе из функции.

Разработанный обработчик производит анализ на уровне модулей программы, то есть является наследником **ModulePass**. Модификация модуля программы происходит в два этапа:

— добавление деклараций функций, вызовы которых будут вставлены в код;

— проход по всем функциям модуля для вставки необходимых инструкций.

3.4.1 Добавление нового сегмента

Генерация нового сегмента происходит при входе в базовый блок или после выполнения функций синхронизации. Для первого случая необходимо в исполняемый код произвести вставку вызова функции **new_segment** для генерации нового сегмента. Во втором случае вызов **new_segment** происходит в переопределенных **POSIX** функциях. Базовый блок состоит из инструкций LLVM ассемблера и имеет один вход (то есть код внутри блока не может быть назначением инструкции перехода), один выход и не содержит инструкций передачи управления.

Так как анализатор сообщает о гонках, между сегментами, то для построения информативного отчета необходимо иметь связь между сег-

ментом и позицией в исходном коде программы. Функция **new_segment** принимает следующие параметры:

— **char* dir** — путь до каталога с исходным текстом программы, которому принадлежит генерируемый сегмент;

— **char* file** — имя файла с исходным текстом, которому принадлежит сегмент;

— **int line** — номер строки в файле с исходным текстом, где начинается сегмент.

Данные параметры могут быть получены во время компиляции модуля программы, если программа компилируется с отладочной информацией. В **LLVM** базовый блок не имеет отладочной информации, отладочная информация привязывается к инструкциям из которых он состоит, причем не все инструкции **LLVM** ассемблера ее содержат. Отдельно стоит отметить, что параметры **dir** и **file** имеют составной тип и не могут быть явным образом переданы в инструкцию **CallInst**, а только через дополнительные переменные, содержащие адреса нужных строк, поэтому значение строк должны быть записаны в сегмент данных.

Таким образом, добавление вызова функции **new_segment** в компилируемый модуль, происходит следующим образом:

а) **BasicBlock** — базовый блок.
б) **BI** — первая инструкция базового блока **BasicBlock**.
в) Если **BI** имеет отладочную информацию DBG:
   1) Считать путь до каталога с исходным текстом компилируемого модуля **DBG.getDirectory()**. Если путь не связан ни с какой глобальной переменной, то создать глобальную переменную содержащую полученный путь.
   2) Считать имя файла с исходным текстом компилируемого модуля **DBG.getFilename()**. Если имя файла не связано ни с какой глобальной переменной, то создать глобальную переменную содержащую полученное имя файла.
   3) Считать номер строки **DBG.getLineNumber()**;
   4) Перейти на шаг е.

г) Если **BI** не последняя инструкция базового блока **BasicBlock**, то присвоить переменной **BI** последующую инструкцию и перейти на шаг в.

д) Если **BI** последняя инструкция базового блока, то отладочная информация не найдена и сегмент будет создан без нее.

е) Произвести генерацию инструкций:

1) Генерация инструкций получения указателей на строки, содержащих путь до каталога и имя файла.
2) Произвести генерацию инструкции **CallInst** для вызова функции **new_segment** с полученными параметрами.

Таким образом, вызов функции генерации сегмента состоит из трех инструкций **LLVM** ассемблера:

```
1  %1 = getelementptr [8 x i8]* @0, i8 0, i8 0
2  %2 = getelementptr [59 x i8]* @1, i8 0, i8 0
3  call void @new_segment(i32 59, i8* %1, i8* %2)
```

Кроме того, если базовый блок является первым блоком функции или завершается инструкциями выхода из функции, то происходит вставка вызовов функций: **enter_function**, **leave_function**. В функцию **enter_function** передается имя текущей функции, **leave_function** параметров не принимает. Данные функции необходимы для поддержки стека вызовов, чтобы кроме позиции сегмента в исходном коде была возможность сообщить о пути выполнения программы, приведшем к появлению гонок. Данный подход не позволяет правильно обрабатывать изменения и восстановления контекста выполнения программы.

3.4.2 Обработка обращений к памяти

Перед командой обращения к памяти происходит вставка инструкции вызова функции **mem_access**. В **LLVM** ассемблере для чтения значения по указанному адресу используется инструкция **load**, а для записи инструкция **store**. Функция **mem_access** принимает адрес памяти к которой происходит обращение и флаг указывающий тип обращения — чтение или запись. В листинге 3.1 приведен псевдокод добавления обработки инструкций обращений к памяти. В каждом базовом блоке модуля производится

поиск инструкций **store** и **load** и осуществляется добавление инструкции **CallInst**.

Листинг 3.1 — Добавление вызовов функции mem_access

```
for Function in Module:
    for BasicBlock in Function:
        for Instruction in BasicBlock:
            if not Instruction.type in ['store', 'load']:
                continue
            is_write = False
            if Istruction.type == 'store':
                is_write = True
            InsertCallInstrBefore(Instruction, Instruction.address,
               is_write)
```

3.4.3 Пример модификации программы

Листинг 3.2 — Минимальная программа на языке Си

```
int main() {
    return 0;
}
```

В листинге 3.2 приведен минимальный пример программы на Си. Программа после запуска сразу же производит завершение с кодом 0. При компиляции данного примера модифицированным компилятором генерируется промежуточное представление **LLVM**, показанное в листинге 3.3.

Функция **main** состоит из единственного базового блока. Блок начинается в начале функции и завершается инструкцией **ret**, возвращающей результат выполнения функции.

Листинг 3.3 — Фрагмент промежуточного представления программы

```
@0 = internal constant [7 x i8] c"test.c\00"
@1 = internal constant [59 x i8] c"/tmp/\00"
@2 = internal constant [5 x i8] c"main\00"

define i32 @main() nounwind {
  %1 = getelementptr [7 x i8]* @0, i8 0, i8 0
  %2 = getelementptr [59 x i8]* @1, i8 0, i8 0
  call void @new_segment(i32 2, i8* %1, i8* %2)
  %3 = getelementptr [5 x i8]* @2, i8 0, i8 0
  call void @enter_function(i8* %3)
  %4 = alloca i32, align 4
```

```
12    call void @mem_access(i32* %4, i8 1)
13    store i32 0, i32* %4
14    call void @leave_function()
15    ret i32 0, !dbg !12
16  }
17
18  declare void @mem_access(i32*, i8)
19  declare void @enter_function(i8*)
20  declare void @leave_function()
21  declare void @new_segment(i32, i8*, i8*)
```

В модуле программы добавлены объявления строковых констант, использующиеся для вызовов функций **new_segment** и **enter_function**:

— “test.c” — имя файла;

— “/tmp/” — путь до каталога с исходным кодом;

— “main” — имя функции.

Добавлены декларации функций, реализованных в библиотеке регистрации событий в журнале выполнения:

— declare void @mem_access(i32*, i8);

— declare void @enter_function(i8*);

— declare void @leave_function();

— declare void @new_segment(i32, i8*, i8*);

В начало функции **main** добавлены инструкции вызова функции создания нового сегмента. Далее производится вызов функции **enter_function** для обновления стека вызовов текущего сегмента. Перед командой записи результата выполнения функции **main** произведена вставка инструкции обработки обращения к памяти. Перед командой **ret** производится в вызов функции **leave_function**, сообщающей, что производится завершение текущей функции.

3.4.4 Компиляция анализируемых программ

CLang не позволяет параметрически задавать список обработчиков, которые необходимо применять при генерации машинного кода, по этой причине нет возможности автоматически применять разработанный

модуль, модифицирующий AST-дерево. Поэтому процесс компиляции модифицированной программы состоит из следующих этапов.

— Компиляция исходного кода программы в байт-код **LLVM** с помощью **CLang**.

— Применение необходимого обработчика с помощью утилиты **opt**, входящей в состав **LLVM**, для генерации файла, содержащего модифицированный байт-код.

— Компоновка программы с необходимыми библиотеками и флагами для переопределения **POSIX** функций.

Для скрытия этих этапов разработана вспомогательная утилита **clang_race**, реализующая данные этапы автоматически и имеющая такие же входные параметры как **CLang**.

3.5 Тестирование

Тестирование разработанного анализатора динамического поиска гонок производилось со следующими версиями ПО:

— ОС Ubuntu 12.04;
— LLVM 3.0;
— CLang 3.0;
— GNU ld 2.22.

Процесс тестирования включает в себя модульное и системное тестирования. Для запусков тестов необходимо в каталоге исходных кодов произвести выполнение команды `make test`. В рамках модульного тестирования производится проверка обработки событий обращений к памяти, создания и завершения потоков, генерацию сегментов выполнения и операции с векторными часами. В рамках системного тестирования производится запуск тестовых программ на предмет поиска гонок.

Кроме того для поиска ошибок использования памяти и обнаружения утечек памяти используется приложение **Valgrind** при запуске тестов. На листинге 3.4 показан вывод **Valgrind** при прохождении тестов, указы-

вающий на отсутствие найденных проблем при работе анализатора с памятью.

Листинг 3.4 — Вывод Valgrind при прохождении тестов

```
1 ==22696==
2 ==22696== HEAP SUMMARY:
3 ==22696==     in use at exit: 0 bytes in 0 blocks
4 ==22696==   total heap usage: 199 allocs, 199 frees, 11924 bytes allocated
5 ==22696==
6 ==22696== All heap blocks were freed -- no leaks are possible
7 ==22696==
8 ==22696== For counts of detected and suppressed errors, rerun with: -v
9 ==22696== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Выводы

Спроектирован и разработан динамический анализатор, осуществляющий поиск гонок в программах на языке Си. Для генерации модифицированного исполняемого файла, осуществляющего генерацию событий необходимых анализатору для поиска гонок, разработан обработчик AST-деревьев в для платформы $LLVM$. Также реализованы:

— библиотека регистрации событий, осуществляющая обработку событий и реализующая ограничение истории обращений к памяти;

— библиотека переопределенных POSIX функций, реализующая генерацию событий связанных с управлением потоками и работой с примитивами синхронизации;

— реализован гибридный алгоритм поиска гонок, осуществляющий анализ журнала выполнения программы.

4 Проведение эксперимента

В данной главе проведены эксперименты по поиску гонок и анализу выполнения программ, запущенных под динамическим анализатором.

4.1 Обнаружение гонок

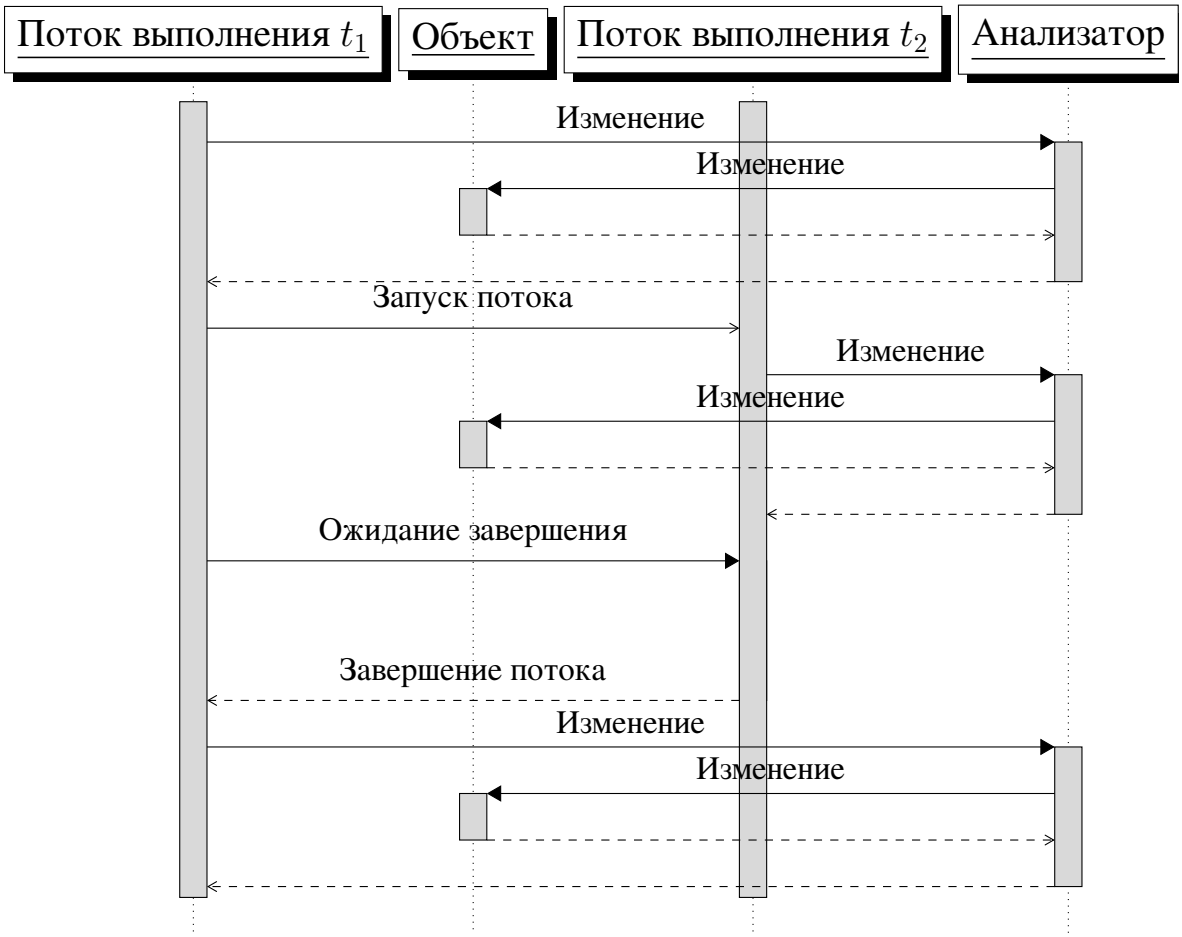

Рисунок 4.1 — Пример схемы сценария работы многопоточного приложения

Рассмотрим основные сценарии работы потоков с памятью и примитивами синхронизации. На диаграмме 4.1 отражен следующий сценарий:

а) поток выполнения t_1 производит работу с заданной областью памяти m;

б) поток выполнения t_1 осуществляет запуск потока t_2;

в) поток выполнения t_2 выполняет задачи связанные с областью памяти m;

г) в тоже время потока t_1 выполняет другие задачи и затем ожидает завершения выполнения t_2;

д) после завершения t_2, поток t_1 осуществляет обращения к области памяти m.

Примером программ, реализующих подобного рода сценарий, являются программы, в которых основной поток выполнения осуществляет подготовку вычислительных задач для дополнительных потоков, выполняющих необходимые вычисления.

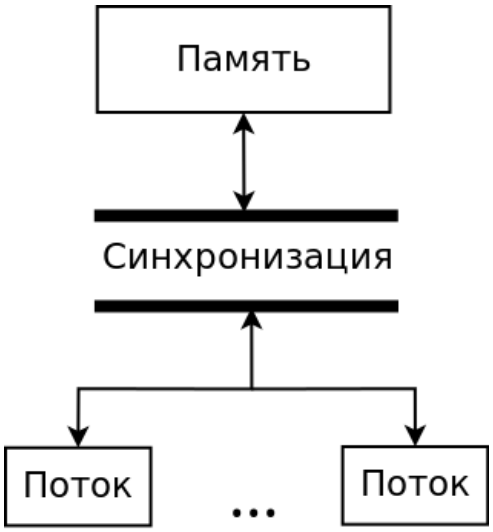

Рисунок 4.2 — Доступ к памяти через примитивы синхронизации

Следующим сценарием является использование потоками совместно одних и тех же областей памяти (рисунок 4.2), но доступ к памяти осуществляется с использованием примитивов синхронизации. Данный сценарий используется для совместной работы потоков с общими данными, реализации очередей сообщений и средств управления потоками.

Также в многопоточном программировании широко используется повторное использование одних и тех же объектов с целью уменьшения количества выделений памяти и повторных инициализаций. На схеме 4.3 представлен пример сценария работы нескольких потоков с пулом объектов. По необходимости поток извлекает из пула объект и производит с ним некоторую работу. После того как все необходимые действия с этим объектом произведены, объект добавляется обратно в пул. Извлечение и добавление объектов в пул происходит безопасно. Соответственно разные

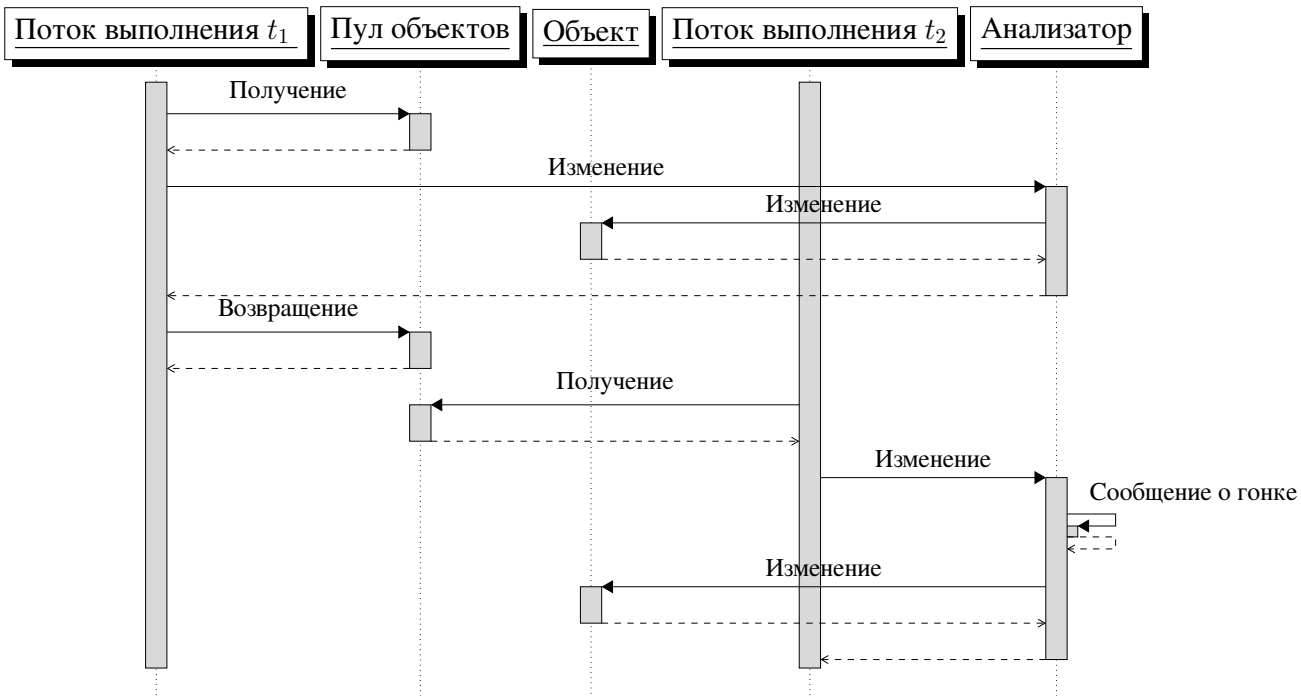

Рисунок 4.3 — Пример схемы сценария работы многопоточного приложения с пулом объектов

потоки в разные моменты времени могут использовать одни и те же ресурсы, но ситуация гонки не возникает, так как в каждый момент времени гарантировано только один поток владеет объектом.

Рассмотрим выполнение тестовых программ, реализующих описанные сценарии, с использованием разработанного динамического анализатора поиска гонок.

Листинг 4.1 — Пример отсутствия гонок при упорядочивание обращений к памяти событием создания потока

```
1  int global_variable;
2
3  void thread2_loop() {
4      global_variable = 100;
5  }
6
7  int main() {
8      global_variable = 1;
9      run_thread(thread_loop);
10     ...
11 }
```

В листинге 4.1 представлен псевдокод программы осуществляющей следующие действия.

— Установка переменной $global\_variable$ в значение 1 в основном потоке.

— Запуск дополнительного потока t_2.

— Установка в дополнительном потоке переменной $global_variable$ в значение 100.

В данном примере гонки отсутствуют и разработанный анализатор не генерирует ложное срабатывание, так как обращения к переменной $global\_variable$ упорядочены во времени событием создания нового потока. Таким образом модификация переменной $global\_variable$ в основном потоке произошла раньше, чем изменение в дополнительном.

Листинг 4.2 — Пример отсутствия гонок при упорядочивание обращений к памяти событием ожидания завершения потока

```
1  int global_variable;
2
3  void thread2_loop() {
4      global_variable = 100;
5  }
6
7  int main() {
8      global_variable = 1;
9      run_thread(thread_loop);
10     wait_thread(...);
11     global_variable = 1;
12 }
```

В листинге 4.2 представлен псевдокод программы осуществляющей следующие действия.

— Установка переменной $global\_variable$ в значение 1 в основном потоке.

— Запуск дополнительного потока t_2.

— Установка в дополнительном потоке переменной $global_variable$ в значение 100.

— Ожидание основным потоком завершения дополнительного потока.

— Установка переменной $global\_variable$ в значение 1 в основном потоке.

В данном примере гонки также отсутствуют и разработанный анализатор не генерирует ложное срабатывание. Все обращения к переменной $global\_variable$ упорядочены во времени событиями создания и ожидания завершения потока и значения векторных часов в сегментах выполнения, где происходили обращения находятся в отношении порядка.

Листинг 4.3 — Пример кода на Си ведущего к гонкам

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  int global_variable;
5
6  void *thread2_loop(void *data)
7  {
8      global_variable = 2;
9  }
10
11
12 int main() {
13     pthread_t thread;
14     pthread_create(&thread, NULL, thread2_loop, (void *)NULL);
15     global_variable = 1;
16     pthread_join(thread, NULL);
17 }
```

В листинге 4.3 представлен код программы на языке Си. При запуске программы происходит запуск дополнительного потока и запись значений в переменную $global\_variable$ в основном и дополнительном потоках. Данный код содержит гонки, так как доступ к переменной $global\_variable$ несинхронизирован. Разработанный анализатор обнаруживает данный тип гонок. В листинге 4.4 показано сообщение генерируемое анализатором при выполнение данного примера. Отчет включает в себя информацию о типе гонки, адрес памяти к которому произошло обращение и данные о сегментах выполнения между которыми произошла, найденная гонка. Для сегментов выводится информацию о идентификаторе потока, местоположении начала сегмента в коде, трассировка стека и значение векторных часов.

Листинг 4.4 — Вывод сообщения о найденной гонке

```
1  =============== RACE ===============
2  Type: WRITE-WRITE
3  Race on location: 0x8049afc
4  Segment of thread: 0x40356f40
5  Dir: /home/arhibot/workspace/diploma_12_d-kovega/code/clangtest
6  File: test3.c
7  Line: 13
8  >>>>> Traceback <<<<<
9  > main
10 >>>>> Vector clocks <<<<<
11 Thread: 40356f40 Value: 1
12 Thread: 40559b40 Value: 0
13 ===============
14 Segment of thread: 0x40559b40
15 Dir: /home/arhibot/workspace/diploma_12_d-kovega/code/clangtest
16 File: test3.c
17 Line: 6
18 >>>>> Traceback <<<<<
19 > thread2_loop
20 >>>>> Vector clocks <<<<<
21 Thread: 40356f40 Value: 0
22 Thread: 40559b40 Value: 1
23 ====================================
```

В случае, если доступ к переменной $global\_variable$ в программе, представленной в листинге 4.3 осуществляется с помощью синхронизации с помощью мьютекса, то гонки в коде отсутствуют и анализатор не сгенерирует ложное срабатывание.

Проанализируем тестовую программу использующую общий пул объектов. В качестве пула объектов используется список массивов, размер пула равен количеству потоков. Каждый поток с использованием примитивов синхронизации удаляет элемент из пула объектов и производит цикл записей и чтений в полученный массив. Далее поток помещает массив обратно в пул объектов и извлекает какой-либо другой массив. Динамический анализатор при повторном выполнении потоком цикла записей и чтений сообщает о найденной гонке. Таким образом, разработанный метод может генерировать ложные срабатывания при использовании одних и тех же объектов, если повторные использования не упорядочены событиями SND или RCV. Так как гибридный алгоритм поиска гонок не производит генерацию данных событий при работе с мьютексами для того чтобы ис-

ключить возможность пропуска возможных гонок, то необходимо ручное аннотирование кода, для генерации события RCV при взятии объекты из пула и SND при извлечении.

4.2 Анализ выполнения циклических программ

Для анализа потребления накладных расходов создаваемых динамическим анализатором проведен эксперименты в рамках который сравнивается потребление памяти и процессорного времени во время выполнения одной и той же тестовой программы:

— запуск программы без анализатора;

— запуск программы с динамическим анализатором с ограничением глубины истории обращений к памяти;

— запуск программы с динамическим анализатором, но без ограничение глубины истории обращений к памяти.

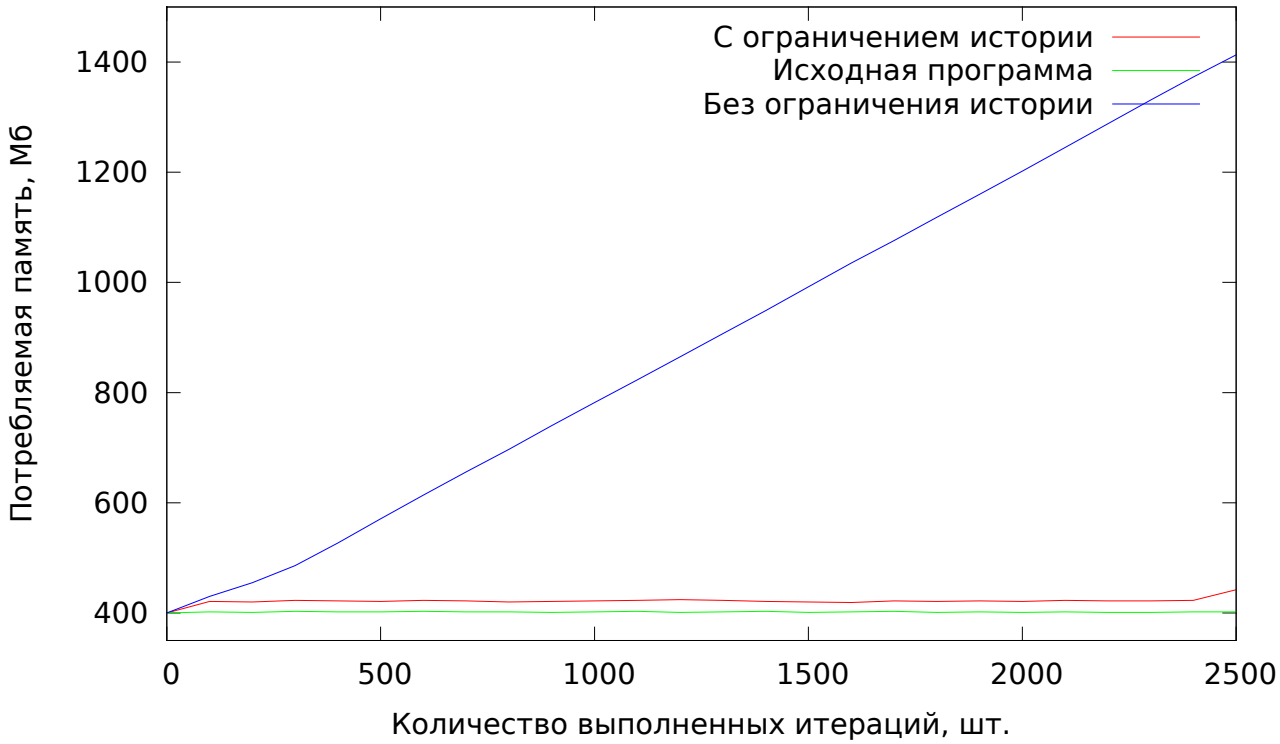

Рисунок 4.4 — Потребление памяти анализируемой программой

Тестовая программа запускает 20 потоков выполнения и в каждом потоке выполнения выделяется 20 мегабайт памяти. Далее по 1000 слу-

чайным адресам, принадлежащим областям памяти, выделенных в каждом потоке, происходит запись или чтение.

На графике 4.4 показано использование памяти при работе тестовых программ. Видно, что тестовая программа без ограничения глубины истории практически линейно увеличивает потребляемую память. В то же время, программа с ограничением глубины истории имеет практически константное потребление памяти и превышает в среднем на 20 мегабайт, потребление памяти программы, запущенной без анализатора.

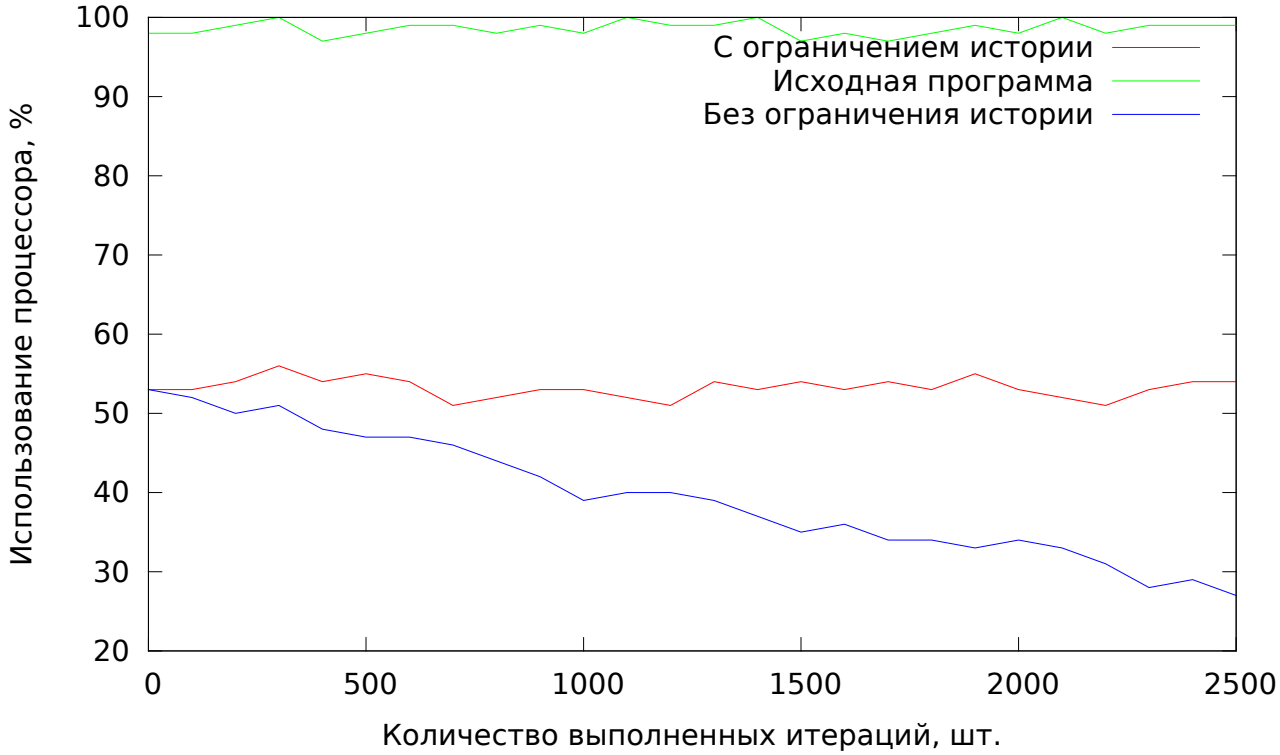

Рисунок 4.5 — Потребление процессорного времени анализируемой программой

Рассмотрим использование процессорного времени тестовыми программами. Эксперимент проводился на компьютере с четырьмя ядрами. Так как тестовая программа многопоточная, то ожидается, что во время выполнения будут использованы все доступные ядра. На графике 4.5 показано использование процессора, рассматриваемыми программами. Видно, что программа без использования анализатора занимает все ядра и загружает процессор практически на 100%.

Программы, запущенные с анализатором, используют не больше 55% процессорного времени. Это связано с использованием глобальных блокировок между потоками в анализаторе гонок для того чтобы безопасно добавлять события в журнал выполнения программы и осуществлять поиск гонок. Среднее использование процессора программой, где использовалось ограничение истории, составляет 52% и практически не меняется со временем. В тоже время из графика видно, что использование процессора программой, где история не ограничивается падает с 55% до 27%. Это обусловлено тем, что процедура анализа журнала выполнения происходит эксклюзивно одним потоком, а так как размер журнала выполнения в программе без использования ограничении истории растет линейно от числа обращений к памяти, то большее время потоки ожидают получение доступа к журналу.

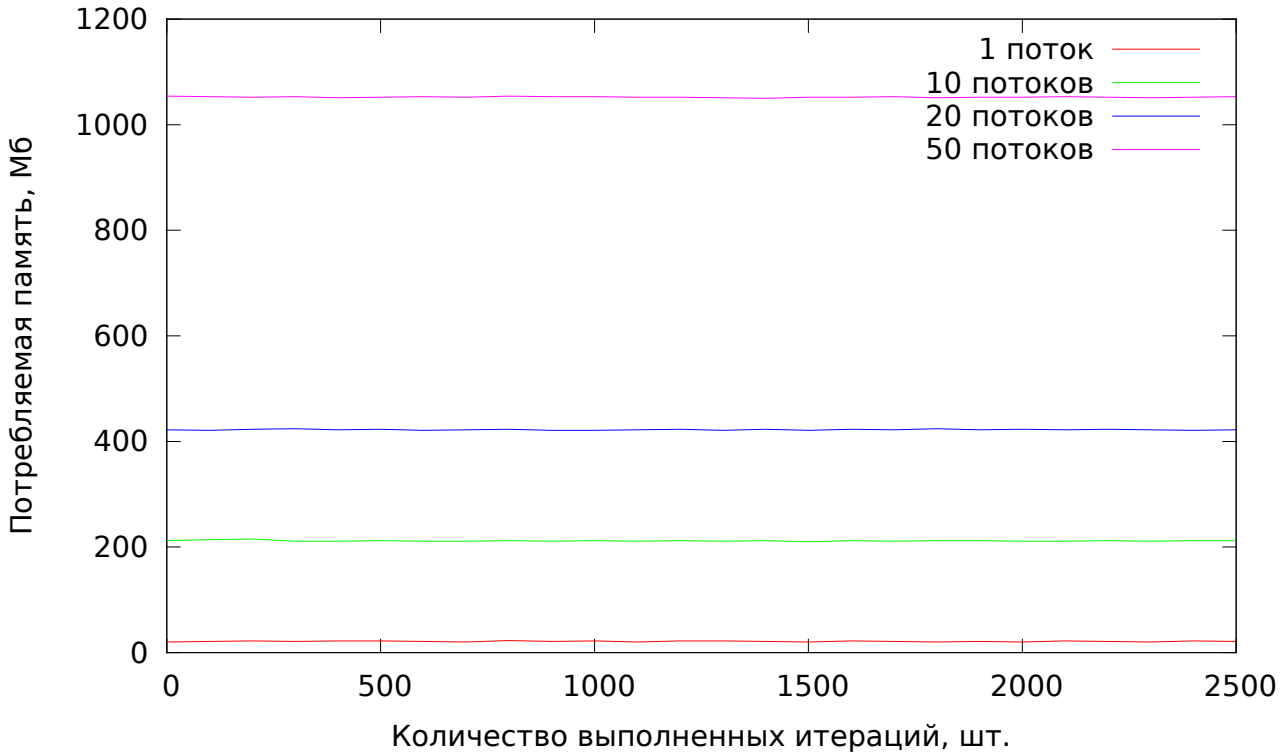

Рисунок 4.6 — Зависимость потребления памяти от числа запущенных потоков

Установим зависимость накладных расходов, вызванных использованием анализатора, от количества запущенных потоков выполнения. Тестовая программа запускает заданное число потоков выполнения и в каж-

дом потоке выделяется 20 мегабайт памяти. Далее по 1000 случайным адресам, принадлежащим областям памяти, выделенных в каждом потоке, происходит запись или чтение. На графике представлено потребление памяти при запуске тестовый программы с 1, 10, 20, 50 потоками выполнения. С увеличением числа запущенных потоков потребление памяти растет линейно и накладные расходы, вызванные использованием динамического анализатора, составляют 0.5 Мб на каждый поток в данной тестовой программе.

Выводы

В результате проведенных экспериментов подтверждено, что разработанный динамический анализатор может использоваться для динамического поиска гонок в программах, разработанных на языке Си. Анализатор выявляет гонки связанные с несинхронизированным доступом к памяти потоками выполнения, но может генерировать ложные срабатывания, связанные с повторным использованием объектов различными потоками. Данная проблема может быть решена модификацией исходного кода программы с целью внесения дополнительных аннотаций, сообщающих анализатору об упорядоченности событий извлечения и добавления объектов в пул объектов.

Также были проведены эксперименты, выявляющие накладные расходы вносимые динамическим анализом. Выявлено, что использование процессорного времени падает в среднем в 1.8 раз. Падение эффективности использования процессора связано с применением глобальных блокировок при журналировании обращений к памяти и при поиске гонок.

Заключение

В результате проделанной работы, разработан метод динамического поиска гонок в программах, реализованных на языка Си. Разработанный метод включает в себя гибридный алгоритм поиска гонок с ограничением истории доступов к памяти и подход к генерации событий для анализа.

Гибридный алгоритм базируется на алгоритме на основе множества блокировок с использованием отношения предшествования для упорядочивания сегментов выполнения потоков. Ограничение глубины истории позволяет использовать предложенный метод при анализе программ с интенсивной работой с памятью. Генерация сегментов выполнения в разработанном методе производится при входе в базовые блоки кода программы. Такое разделение на сегменты при формировании отчета о найденных гонках позволяет сформировать корректный стектрейс предыдущих обращений к памяти и указать номер строки начала базового блока.

Для реализации динамического анализатора в методе используется модифицированный компилятор языка Си и переопределение POSIX функций по работе с потоками и примитивами синхронизации. В качестве компилятора выбран CLang и был разработан обработчик для платформы LLVM осуществляющий модификацию машинного кода, генерируемого CLang.

Проведены эксперименты подтверждающие возможность использования разработанного метода для поиска гонок в программах, реализованных на языке Си. Показаны случаи правильного срабатывания анализатора и отражен случай генерации ложного срабатывания. Ложное срабатывание вызвано повторным использованием объектов потоками выполнения. Были проведены эксперименты, выявляющие накладные расходы, вносимые динамическим анализатором. Данные эксперименты показали падение использования процессорного времени в среднем в 1.8 раз.

Список использованных источников

1. *Marino, Daniel*. LiteRace: effective sampling for lightweight data-race detection / Daniel Marino, Madanlal Musuvathi, Satish Narayanasamy // In PLDI. — 2009.

2. Accurate and efficient filtering for the Intel thread checker race detector / Paul Sack, Brian E. Bliss, Zhiqiang Ma et al. // Proceedings of the 1st workshop on Architectural and system support for improving software dependability. — ASID '06. — New York, NY, USA: ACM, 2006. — Pp. 34–41. http://doi.acm.org/10.1145/1181309.1181315.

3. Верификация моделей программ / Э.М. Кларк, О. Грамберг, Д. Пелед, В. Захаров. — Изд-во Моск. центра непрерыв. мат. образования, 2002. http://books.google.ru/books?id=0dv4AAAACAAJ.

4. *Boyapati, Chandrasekhar*. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. — 2002.

5. *Flanagan, Cormac*. Type-based race detection for Java / Cormac Flanagan, Stephen N. Freund // IN PROCEEDINGS OF THE SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION. — Pp. 219–232.

6. Automated type-based analysis of data races and atomicity / Amit Sasturkar, Rahul Agarwal, Liqiang Wang, Scott D. Stoller // Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. — PPoPP '05. — New York, NY, USA: ACM, 2005. — Pp. 83–94. http://doi.acm.org/10.1145/1065944.1065956.

7. *Holzmann, Gerard J.* Logic Verification of ANSI-C Code with SPIN / Gerard J. Holzmann // Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. — London, UK, UK: Springer-Verlag, 2000. — Pp. 131–147. http://dl.acm.org/citation.cfm?id=645880.672083.

8. *Kars, Pim*. The Application of Promela and Spin in the BOS Project. — 1996.

9. *Serebryany, Konstantin*. ThreadSanitizer: data race detection in practice / Konstantin Serebryany, Timur Iskhodzhanov // Proceedings of the

Workshop on Binary Instrumentation and Applications. — WBIA '09. — New York, NY, USA: ACM, 2009. — Pp. 62–71. http://doi.acm.org/10.1145/1791194.1791203.

10. Detecting Data Races on Weak Memory Systems / Sarita V. Adve, Mark D. Hill, Barton P. Miller, Robert H. B. Netzer // IN PROCEEDINGS OF THE 18TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE. — 1991. — Pp. 234–243.

11. *Butenhof, David R.* Programming with POSIX threads / David R. Butenhof. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

12. *The IEEE and The Open Group*. The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition / The IEEE and The Open Group. — New York, NY, USA: IEEE, 2004. http://www.opengroup.org/onlinepubs/009695399/.

13. *Gaines, R. Stockton*. Time, Clocks, and the Ordering of Events in a Distributed System. — 1973.

14. *Zhou, Pin*. HARD: Hardware-Assisted Lockset-based Race Detection / Pin Zhou, Radu Teodorescu, Yuanyuan Zhou // Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture. — HPCA '07. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 121–132. http://dx.doi.org/10.1109/HPCA.2007.346191.

15. A theory of data race detection / Utpal Banerjee, Brian Bliss, Zhiqiang Ma, Paul Petersen // Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging. — PADTAD '06. — New York, NY, USA: ACM, 2006. — Pp. 69–78. http://doi.acm.org/10.1145/1147403.1147416.

16. *Fidge, C J.* Timestamps in message-passing systems that preserve the partial ordering / C J Fidge // *Proceedings of the 11th Australian Computer Science Conference*. — 1988. — Vol. 10, no. 1. — P. 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf.

17. *Nethercote, Nicholas*. Valgrind: A framework for heavyweight dynamic binary instrumentation / Nicholas Nethercote, Julian Seward // In

Proceedings of the 2007 Programming Language Design and Implementation Conference. — 2007.