DOI: 10.15514/ISPRAS-2025-37(6)-38

Адаптация алгоритма ThreadSanitizer для обнаружения гонок по данным в ядре ОСРВ

Е.С. Ельчинов, ORCID: 0000-0003-4555-1204 <elchinov@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Дизайн и реализация корректных алгоритмов многопоточной синхронизации являются неотъемлемой частью разработки современных операционных систем реального времени. Тестирование корректности алгоритма в модели памяти языка – одна из важнейших задач на этом пути. В статье описывается интеграция широко используемого алгоритма обнаружения гонок данных ThreadSanitizer из программной инфраструктуры LLVM в систему сборки и тестирования ядра операционной системы реального времени и его преимущества и недостатки в сравнении с другими подходами обнаружения ошибок многопоточной синхронизации. Среди прочего рассматривается определение семантики управления прерываниями и работы с физическими ядрами в контексте синхронизации в модели «выполняется прежде» (happens-before). В заключение приводятся результаты интеграции инструмента ThreadSanitizer в ядро операционной системы реального времени CLOS в сравнении с существующими подходами обнаружения ошибок в ядре данной операционной системы.

Ключевые слова: многопоточная синхронизация; динамический анализ; операционные системы; алгоритм thread sanitizer; гонки по данным.

Для цитирования: Ельчинов Е.С. Адаптация алгоритма ThreadSanitizer для обнаружения гонок по данным в ядре ОСРВ. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 91–108. DOI: 10.15514/ISPRAS–2025–37(6)–38.

Adaptation of the ThreadSanitizer algorithm for data race detection in a RTOS kernel

E.S. Elchinov, ORCID: 0000-0003-4555-1204 <elchinov@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Correct design and implementation of concurrent algorithms is a crucial part of modern real-time operating system development. One of the main steps along this way is a verification of such algorithms within the programming language memory model. The article describes an integration of the ThreadSanitizer – broadly used LLVM tool for data race detection – into the RTOS kernel environment and discusses its advantages and disadvantages over other tools for data race detection. Among other things, the semantics of context switches and interrupt management within the happens-before synchronization model is considered. In conclusion the results of a ThreadSanitizer tool integration are provided compared to current approaches of concurrency bugs detection in RTOS kernel.

Keywords: mutlithreaded synchronization; dynamic analysis; operating systems; thread sanitizer algorithm; data races.

For citation: Elchinov E.S. Adaptation of the ThreadSanitizer algorithm for data race detection in a RTOS kernel. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, psrt 3, 2025, pp. 91-108 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-38.

1. Введение

Алгоритмы многопоточной синхронизации находят широкое применение в разработке современных операционных систем, в особенности, поддерживающих симметричную многопроцессорность и вытеснение на уровне задач ядра. В то же время, нетривиальные алгоритмы многопоточной синхронизации существенно усложняют процесс тестирования кода ядра и могут содержать трудно отлаживаемые без дополнительных инструментов ошибки.

В пользовательском окружении для отладки и верификации систем процессов, коммуницирующих через разделяемую память, существует множество подходов, использующих различные методы статического и динамического анализа кода и его исполнений. Некоторые операционные системы общего назначения, например, Linux, поддерживают собственные инструменты и для анализа кода ядра (такие как LKMM [1] или KCSAN [2]).

В данной работе рассматривается расширение возможностей динамического анализа применительно к поиску ошибок многопоточной синхронизации в ядре операционных систем реального времени. В сравнении с операционной системой (ОС) общего назначения, ОС реального времени (ОСРВ) характерно имеют статическую настройку виртуальной памяти, меньший объём кода ядра и строгие ограничения на время исполнения задач ядра и затрачиваемую память. Рассматриваемая в статье ОС имеет все описанные характерные для систем реального времени черты. С учётом особенностей пространства ядра ОС и, в частности, требований операционных систем реального времени, существующие подходы к динамическому анализу кода требуют адаптации и переработки.

В данной работе описывается адаптация алгоритма детектора гонок данных ThreadSanitizer [3] и необходимые оптимизации его библиотеки времени исполнения для работы в пространстве ядра упомянутой ОСРВ, а также результаты его интеграции в систему тестирования ядра. Для определения требований к алгоритму вначале определяется семантика управления прерываниями в терминах отношения *«выполняется прежде»*

(*happens-before*). Также описывается влияние свойств систем реального времени на успешность применения различных подходов к поиску ошибок синхронизации.

2. Гонки по данным

Гонки по данным – один из наиболее распространённых типов ошибок в алгоритмах многопоточной синхронизации. По стандарту языков C и C++, на которых написана значительная часть системного программного обеспечения, они приводят к неопределённому поведению [4]. Для определения корректности программы в конкурентном исполнении многие современные языки программирования внедрили в свои стандарты так называемые модели памяти – наборы правил для каждой операции чтения, определяющих все возможные модифицирующие операции, результаты которых она может вернуть. Вторая задача модели памяти – определить все корректные исполнения программы с точки зрения многопоточной синхронизации.

2.1 Математические порядки над множеством операций над памятью

В модели памяти языка C для каждого исполнения программы определяется несколько частичных порядков над выполняемыми операциями над памятью.

Для дальнейшего изложения понадобятся следующие отношения и определения:

- *happens-before* – отношение «выполняется прежде» – для двух операций **A** и **B** верно **A** *happens-before* **B**, если из результата исполнения операции **B** следует факт завершения операции **A**.
- *sequenced-before* (также, *program order*) – отношение «предшествует в одном потоке» – для двух операций **A** и **B** верно **A** *sequenced-before* **B**, если операции **A** и **B** исполняются в одном потоке, причём операция **A** предшествует операции **B**.
- *synchronizes-with* – отношение «причинности» операций синхронизации – для двух операций **A** и **B** над одной переменной синхронизации верно **A** *synchronizes-with* **B**, если **A** – операция чтения некоторого состояния, **B** – операция записи (изменения) состояния, и операция **A** читает результат записи операции **B**.
- *atomic*-операции – специальный тип операций над памятью, имеющих, согласно стандарту, определённое поведение при конкурентном исполнении конфликтующих операций.
- *release*-операции – операции записи (изменения) состояния некоторого объекта синхронизации.
- *acquire*-операции – операции чтения состояния некоторого объекта синхронизации.

Согласно стандарту, если **B** – *release*-операция и **A** – *acquire*-операция, читающая значение, записанное операцией **B**, то выполняется отношение **A** *synchronizes-with* **B**.

2.2 Определение гонки данных

Стандарты языка C, начиная с версии C11 определяют гонку данных как два одновременных конфликтующих неатомарных обращения к памяти. Согласно стандарту, две попытки доступа конфликтуют, если соответствующие им ячейки памяти имеют непустое пересечение, и хотя бы одна из попыток модифицирующая [4]. Две операции могут считаться одновременными в случае, когда они не упорядочены порядком *happens-before*, согласно его определению в модели памяти – иными словами, если в исполнении в промежутке между этими операциями нет наблюдаемой синхронизации соответствующих им потоков.

3. Динамические методы обнаружения гонок данных

Для поиска гонок данных в коде применяется два основных класса методов – алгоритмы статического и динамического анализа кода.

Алгоритмы статического анализа кода ищут ошибки, опираясь на исходный код, без непосредственного исполнения алгоритма. Среди статических инструментов для поиска гонок данных в коде ядра ОС можно выделить LKMM [1] из ядра ОС Linux.

В отличие от статических методов, при использовании динамического анализа алгоритм обнаружения ошибок внедряется в анализируемый код на этапе сборки и производит поиск ошибок на основе анализа текущего исполнения на некоторых тестовых сценариях.

Среди основных преимуществ динамического анализа:

- Динамический анализ позволяет практически исключить получение ложноположительных результатов, ограничиваясь небольшим (в сравнении со статическим анализом) количеством аннотаций к анализируемому коду.
- Многие динамические алгоритмы требуют затрат по памяти и времени исполнения, пропорциональных требованиям анализируемой системы, что позволяет анализировать сложные исполнения.

Из минусов динамического подхода, можно отметить:

- Так как динамические методы рассматривают лишь конкретное исполнение, качество анализа напрямую зависит от качества тестового покрытия анализируемого кода.
- Динамический анализ не способен обеспечить строгие гарантии корректности кода, так как в тестовых сценариях могут реализовываться не все возможные исполнения.

Для динамического поиска гонок в коде ядра ОС Linux поддерживается инструмент KCSAN (kernel consurrency sanitizer) [2]. Для пользовательских приложений инфраструктура сборки LLVM предоставляет инструмент динамического анализа Thread Sanitizer [5].

На момент написания статьи существует несколько подходов к динамическому анализу программ на предмет гонок данных, среди которых можно выделить алгоритмы, основанные на использовании точек останова по данным (watchpoint-based) и алгоритмы, использующие для поиска гонок упомянутое ранее отношение *happens-before* над операциями с памятью, а также метод, основанный на построении множества активных критических секций (lockset). По определению, гонка данных – это одновременный конфликтующий небезопасный доступ. Упомянутые подходы различаются алгоритмом обнаружения одновременных операций над одной ячейкой памяти.

Основанные на точках останова (watchpoint-based) алгоритмы находят одновременные доступы путём приостановки программы в местах доступа к памяти на псевдослучайный промежуток времени и обработки всех доступов, произошедших к данной ячейке памяти в обозначенный промежуток, как одновременных с первым доступом. Для доступов к интересующему адресу такие алгоритмы используют механизм точек останова по данным (watchpoint).

Детекторы гонок по данным, основанные на построении порядка операций (happens-before-детекторы), производят поиск одновременных доступов по определению из стандарта, то есть поддерживают в памяти часть истории доступов к памяти и сжатое представление порядка *happens-before* и проверяют, упорядочен ли каждый следующий доступ с конфликтующими с ним сохранёнными доступами.

Алгоритм построения множества активных критических секций (lockset) для каждого доступа к памяти определяет все активные на этот момент критические секции и проверяет для каждой переменной, что все конфликтующие доступы к ней разделяют между собой хотя бы одну общую критическую секцию.

Соответственно, преимуществами подхода, основанного на точках останова являются:

- Минимальные затраты памяти – необходимо лишь хранить информацию о текущих точках останова по данным;
- Отсутствие необходимости в аннотациях к кодовой базе, так как доступы, происходящие в момент ожидания на точках останова, всегда не упорядочены с обрабатываемым доступом в смысле порядка *happens-before* в модели памяти.

В свою очередь, из плюсов подхода, основанного на построении во время исполнения представления частичного порядка *happens-before*, можно отметить:

- Способность обнаруживать гонки по их определению в модели памяти языка – то есть независимо от целевой платформы и её модели памяти.
- Обработка каждой операции над памятью детерминированным образом – то есть для верификации конкретного сценария исполнения программы его достаточно проанализировать один раз.

Таким образом, детекторы, основанные на построении порядка *happens-before*, в сравнении с подходом, основанным на точках останова, позволяют независимо от текущей целевой платформы находить более широкий спектр ошибок, в том числе гонок данных, не приводящих к некорректным значениям в анализируемом исполнении, но требуют более детальных аннотаций и дополнительных ресурсов по времени и памяти.

По сравнению с алгоритмами, основанными на точках останова по данным либо построении порядка *happens-before*, метод, основанный на построении множества активных критических секций (lockset) обеспечивает меньшие затраты по памяти, чем метод, основаный на *happens-before* и, в отличие от детекторов, основанных на точках останова, позволяет детектировать потенциальные гонки, не реализовавшиеся в анализируемом исполнении. Однако, этот метод не позволяет корректно анализировать алгоритмы неблокирующей синхронизации, а также, по сути подхода, способен генерировать ложные сообщения об ошибках.

К подходам, основанным на точках останова, относятся такие алгоритмы как, например, RaceHound [6] и Kernel Concurrency Sanitizer [2] в ядре Linux. Алгоритм Thread Sanitizer [3], в свою очередь, является детектором, основанным на построении порядка *happens-before*. Среди детекторов, использующих построение множества активных критических секций, можно выделить Eraser [7].

4. Алгоритм Thread Sanitizer

Алгоритм поиска гонок Thread Sanitizer относится к классу динамических алгоритмов поиска гонок, основанных на анализе порядка операций *happens-before* и, соответственно, должен поддерживать в своей памяти некоторое сжатое представление упомянутого порядка над всеми хранимыми операциями с памятью. С этой целью каждой операции с памятью ставится в соответствие её эпоха – целое неотрицательное число, а каждому потоку и переменным синхронизации соответствуют векторные часы [8].

4.1 Модель синхронизации алгоритма ThreadSanitizer

В модели памяти языка С операции внутри одного потока упорядочены полным порядком *sequenced-before* (см. п. 2.1), известным также как *program order*. Отношение *happens-before* в модели памяти определяется как транзитивное замыкание *sequenced-before*, *synchronizes-with* и ещё нескольких порядков, где каждая дуга *synchronizes-with* формируется *release*-операцией и соответствующей ей (наблюдающей её эффект) последующей *acquire*-операцией над некоторой переменной синхронизации.

В модели синхронизации алгоритма Thread Sanitizer порядок *happens-before* выражается в виде транзитивного замыкания порядков *sequenced-before* и *synchronizes-with*, что является упрощением модели памяти языка С.

4.2 Эпохи и векторные часы

Поскольку в модели синхронизации внутри каждого потока отношение *sequenced-before* определяет полный порядок на операциях, алгоритм ThreadSanitizer сопоставляет каждой операции её эпоху – некоторое целое неотрицательное число, являющееся мерой прогресса потока к моменту данной операции, аналогично логическим часам Лэмпорта [9]. В дескрипторе каждой обрабатываемой операции над памятью, Thread Sanitizer, помимо прочих характеристик, сохраняет эпоху этой операции и идентификатор её потока.

В качестве сжатого представления порядка *happens-before* в ThreadSanitizer используются векторные часы [8], являющиеся широко используемым обобщением логических часов Лэмпорта. Если **T** – некоторый поток исполнения, то векторные часы потока **T** – список, хранящий для каждого активного потока в системе эпоху его последней операции, упорядоченной до текущей операции в потоке **T** в смысле порядка *happens-before*.

4.3 Синхронизация

Для корректной работы алгоритма необходим способ пересчёта сжатого представления графа отношения *happens-before* после операций синхронизации (то есть *release* и *acquire* операций над переменными синхронизации). С этой целью каждой переменной синхронизации – примитивам синхронизации и атомарным (*atomic*) переменным – также ставятся в соответствие векторные часы, хранящие для каждого потока последнюю его эпоху, предшествующую или равную (в смысле отношения *happens-before*) некоторой *release*-операции над этой переменной.

Для каждой *release*-операции потока **T** над переменной синхронизации **S** её векторные часы $\mathbf{V_S}$ (рис. 1) обновляются поэлементным максимумом (далее – операция $\max^*$) с векторными часами $\mathbf{V_T}$ потока **T**. Для *acquire*-операций, наоборот, векторные часы $\mathbf{V_T}$ обновляются поэлементным максимумом с $\mathbf{V_S}$. Таким образом, для дуги отношения *synchronizes-with* между операциями *release*(**S**) (далее – сокр. **rel**) в **T1** и *acquire*(**S**) (далее – сокр. **acq**) в **T2** верно $\mathbf{V'_{T2}} = \max^*(\mathbf{V_{T2}}, \mathbf{V'_S}) > \max^*(\mathbf{V_{T2}}, \max^*(\mathbf{V_{T1}}, \mathbf{V_S})) > \mathbf{V_{T1}}$ (где $\mathbf{V'_{T2}}$ и $\mathbf{V'_S}$ – обновлённые значения $\mathbf{V_{T2}}$ и $\mathbf{V_S}$ соответственно) (рис. 1).

Такое поведение согласуется с определениями векторных часов и отношений *synchronizes-with* и *happens-before* в модели памяти.

4.4 Теневая память и поиск гонок

Для хранения информации об обработанных доступах к памяти Thread Sanitizer использует отдельный регион памяти – так называемую теневую память – такой, что каждой ячейке машинного слова в основной памяти соответствует ячейка теневой памяти.

Каждая ячейка теневой памяти хранит необходимую информацию про несколько (в реализации в LLVM – 4) последних доступов к соответствующему машинному слову в основной памяти. Для каждой операции (листинг 1) в теневой памяти сохраняется её идентификатор потока, текущая эпоха этого потока, затронутые байты и тип доступа (рис. 2).

Таким образом, в момент обработки текущей операции **X** над памятью теневая память определяет потенциально конфликтующие предшествующие операции $\mathbf{Y_i}$, а векторные часы позволяют проверить, что между каждой конфликтующей операцией $\mathbf{Y_i}$ и **X** присутствует отношение *happens-before* (то есть, **X** *happens-before* $\mathbf{Y_i}$). В противном случае инструмент генерирует сообщение об ошибке.

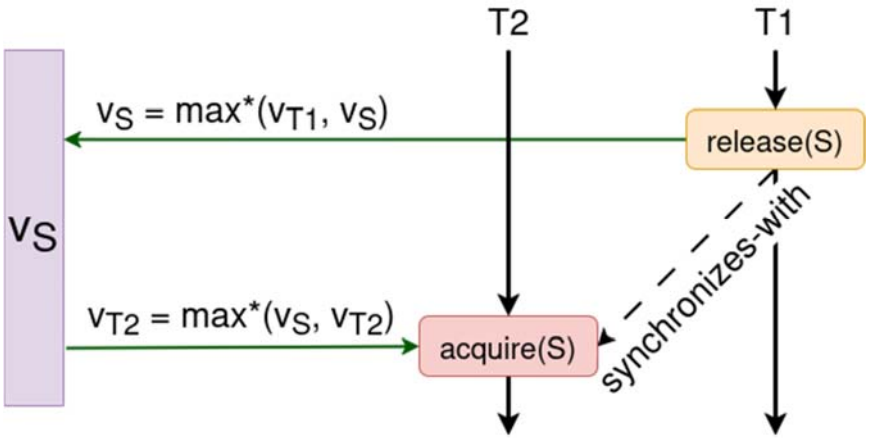

Рис. 1. Векторные часы.
Fig. 1. Vector clock.

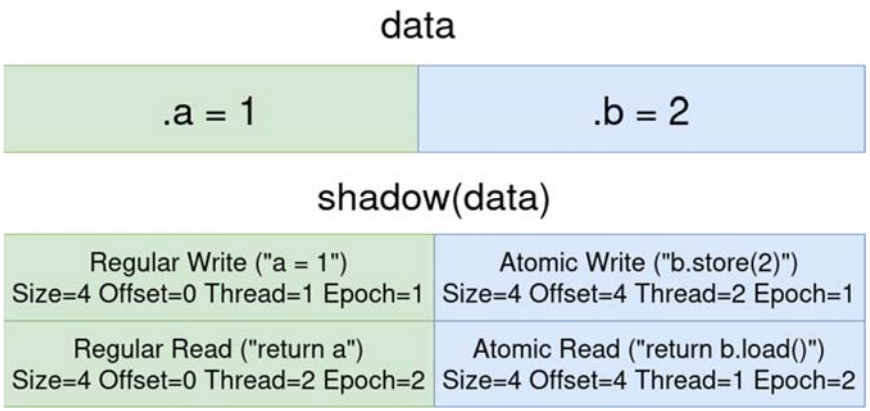

Рис. 2. Теневая память.
Fig. 2. Shadow memory.

```
struct {
    int a;
    std::atomic<int> b;
} data;
int thread1 () {
    data.a = 1;
    return data.b.load();
}
int thread2 () {
    data.b.store(2);
    return data.a;
}
```

Листинг 1. Пример заполнения теневой памяти.
Listing 1. Shadow memory update example.

5. Оптимизации, используемые в LLVM Thread Sanitizer

С целью уменьшения расходов по памяти и поддержки потенциально неограниченного числа потоков в текущей версии алгоритма LLVM Thread Sanitizer используется несколько механизмов, в числе которых обновление теневой памяти, трассировка операций и слоты синхронизации.

5.1 Трассировка операций и теневой стек

В трассах операций для каждого потока сохраняется порядок операций над памятью относительно вызовов процедур, что, среди прочего, позволяет восстановить стек вызовов любой операции из теневой памяти. Теневой стек поддерживает текущий стек вызовов в виде массива адресов в коде, предоставляя возможность быстро генерировать сообщения об ошибках (рис. 3).

Thread Sanitizer внедряет в код вызовы библиотеки времени исполнения в момент входа в каждую функцию и возврата из неё, что позволяет поддерживать теневой стек и учитывать вызовы и возвраты из функций в трассах потоков.

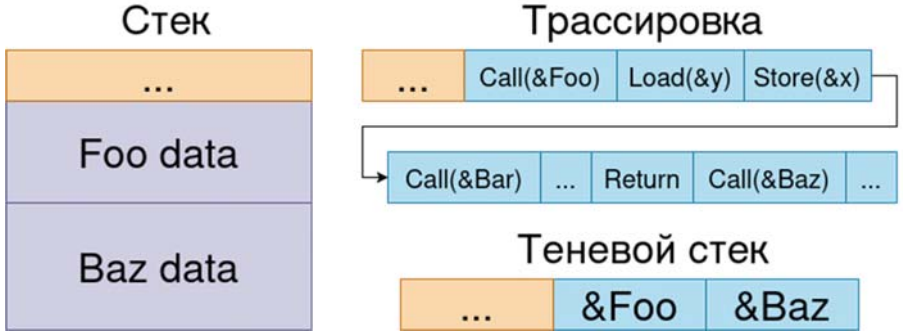

Рис. 3. Трассировка и теневой стек.
Fig. 3. Tracing and shadow stack.

5.2 Слоты синхронизации

Для эффективной обработки большого количества потоков в последней версии библиотеки времени исполнения Thread Sanitizer была сделана оптимизация, позволяющая ограничить размер векторных часов и количество бит идентификатора потока в теневой памяти.

Вместо потенциально неограниченного по величине целочисленного идентификатора потока реализация ThreadSanitizer поддерживает фиксированное количество так называемых слотов – логических единиц исполнения, имеющих локальный счётчик эпохи (рис. 4). Библиотека времени исполнения ThreadSanitizer ставит каждому исполняемому на данный момент потоку в соответствие некоторый слот. Каждый слот, в свою очередь, поддерживает историю потоков, когда-либо исполнявшихся в привязке к этому слоту. Эпоха слота считается эпохой привязанного к нему потока, а векторные часы потоков и переменных синхронизации хранят последнюю наблюдаемую эпоху каждого слота. В теневой памяти вместо идентификатора потока хранится индекс слота, соответствующего потоку, исполнившему операцию.

В такой модели наличие дуги отношения *synchronizes-with* между некоторыми потоками **T1** и **T2** влечёт за собой наличие той же дуги между соответствующими слотами потоков. В случае если количество потоков превосходит количество слотов, такой подход вызывает паразитную синхронизацию между потоками, разделяющими один слот в течение времени исполнения, что может привести к ложноотрицательным вердиктам в некоторых

исполнениях. Однако, если количество активных потоков не превосходит количества слотов, данный подход не приводит к потере информации в сжатом представлении порядка *happens-before* текущего исполнения.

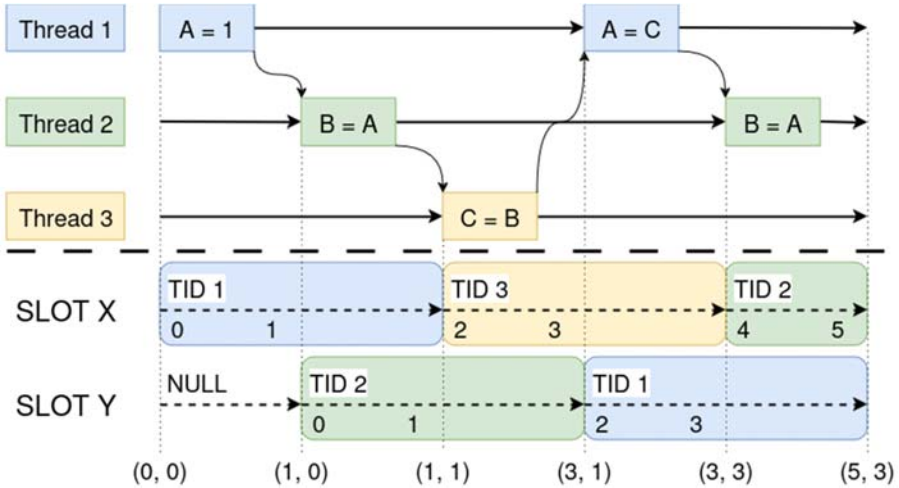

Рис. 4. Слоты синхронизации.
Fig. 4. Synchronization slots.

6. Интеграция Thread Sanitizer в ядро ОСРВ

6.1 Требования при интеграции в ядро ОСРВ

Особенности пространства ядра операционной системы накладывают ряд требований на реализацию алгоритма Thread Sanitizer:

- Большое количество потоков в сравнении с пространством пользователя – требуется умение обрабатывать синхронизацию сотен потоков, не увеличивая затраты памяти и времени для поддержания механизма векторных часов.
- Использование нестандартных механизмов синхронизации (барьеров памяти и управления прерываниями) – требуется определить их семантику в отношении порядка *happens-before* и интегрировать поддержку описанных механизмов в библиотеку времени исполнения.
- Переход в пространство пользователя и обработка асинхронных прерываний – требуется поддержать механизм поиска гонок между обработчиком прерывания и кодом прерванного потока.

Помимо описанных требований, свойства ОС реального времени накладывают на реализацию дополнительные ограничения:

- Ограниченный и фиксированный объём физической и виртуальной памяти – объём теневой памяти в отношении к основной памяти ядра не должен превосходить 1:1.
- Ограничения по затратам по времени в худшем случае – алгоритмы, опирающиеся на гарантии реального времени, должны корректно работать в комбинации с инструментом динамического анализа.

Для уменьшения требований алгоритма по дополнительной памяти за основу реализации библиотеки времени исполнения была взята описанная выше версия библиотеки времени

исполнения LLVM Thread Sanitizer, поддерживающая трассировку операций и слоты синхронизации, которая позволяет поддерживать произвольно большое количество логических потоков при фиксированном размере векторных часов и корректно обрабатывать переполнение счётчиков эпох потоков.

6.2 Оптимизация расходов по памяти

В текущей реализации LLVM Thread Sanitizer теневая память состоит из двух регионов: в первом, в соотношении 2:1 к основной памяти, хранится информация о последних доступах к памяти, во втором – метаинформация о переменных синхронизации, в соотношении 1:1 к основной памяти. В пространстве ядра ОСРВ нет механизмов динамического управления виртуальной памятью, а объём физической памяти на целевых платформах зачастую не позволяет иметь теневую память в соотношении большем чем 1:1 к основной.

Трёхкратная экономия потребляемой теневой памяти достигается за счёт уменьшения количества слотов потоков и максимального значения эпохи потока, а также замены теневого региона для метаинформации хеш-таблицей, хранящей для каждого адреса переменной синхронизации её векторные часы.

6.3 Обработка барьеров памяти и ассемблерного кода

Барьеры памяти, почти не используемые в пользовательском коде, находят широкое применение в коде ядра и должны учитываться библиотекой времени исполнения как точки синхронизации. Принцип обработки барьеров описан авторами KTSAN [10] и был адаптирован для текущей версии библиотеки времени исполнения.

С целью корректной обработки ассемблерного и прочего кода, не подвергающегося автоматическому аннотированию во время сборки (т.н. инструментации), Thread Sanitizer поддерживает явные вызовы интерфейса библиотеки времени исполнения – аннотации.

6.4 Оптимизация обработки переполнения счётчика эпохи

Для корректности свойств векторных часов и алгоритма обнаружения гонок данных требуется монотонность эпох операций в каждом слоте. Поскольку, с целью оптимизации затрат по памяти, размер счётчика эпохи в текущей реализации, в сравнении с LLVM ThreadSanitizer, был уменьшен до 6 бит, на любом реальном исполнении будет возникать переполнение счётчика эпохи. После переполнения счётчика эпохи нарушается свойство монотонности, и вся сохранённая информация, содержащая номера старых эпох и данные векторных часов, становится несогласованной с текущим состоянием структур Thread Sanitizer. Таким образом, для корректной обработки переполнения эпохи требуется выполнить сброс всего текущего состояния.

Обработка переполнения счётчика эпохи происходит в два этапа: сначала производится очистка трасс потоков и таблицы переменных синхронизации, затем операция обновления тени.

Меньший диапазон возможных эпох потоков, помимо уменьшения размера ячейки теневой памяти, даёт разумное ограничение на размер хеш-таблицы для метаинформации. Однако такое решение приводит к частым операциям обновления внутреннего состояния санитайзера, в том числе обнуления (сброса) теневой памяти с целью обработки переполнения счётчика эпохи потока. В LLVM Thread Sanitizer для Linux обнуление теневой памяти реализовано через системный вызов mmap. ОСРВ не поддерживает такое решение в силу статической конфигурации виртуальной памяти.

В связи с этим, для сброса теневой памяти состояние страниц тени (уже обновлена страница, или требует обнуления) эмулируется программным образом с использованием буфера флагов (рис. 5). При необходимости доступа к странице, требующей сброса данных, поток сначала

обнуляет её данные, затем сбрасывает флаг состояния страницы. Так как сбросы страниц теневой памяти конфликтуют с чтением информации о доступах из тени, возможные ложные гонки фильтруются с помощью подсистемы трассировки доступов к памяти.

Описанная реализация позволяет не останавливать прогресс системы во время операции обновления тени, что важно для динамического анализа кода, опирающегося на гарантии реального времени.

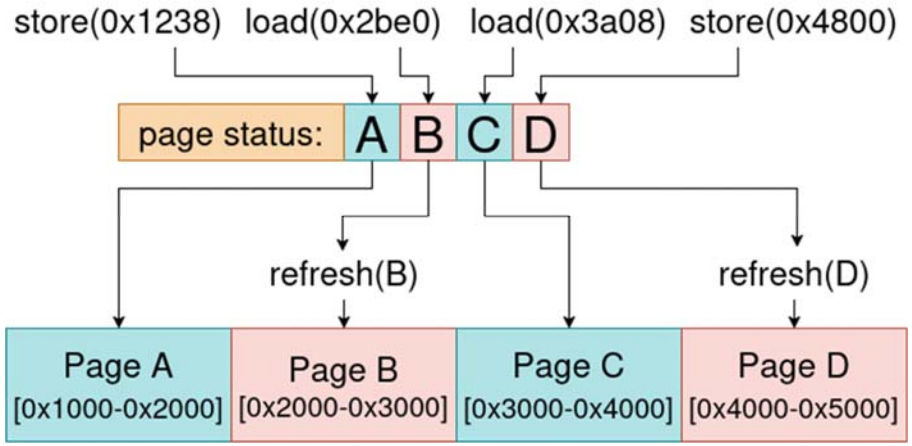

Рис. 5. Постраничное обновление тени.
Fig. 5. Per-page shadow reset.

7. Поддержка прерываний в реализации ThreadSanitizer

Одним из ключевых отличий пространства ядра от пользовательского пространства является наличие прерываний и управления контекстом исполнения.

7.1 Обработка процедур обработчиков прерываний

Поскольку внутри ядра ОС основная часть взаимодействия с пользователем и аппаратурой построена на механизме прерываний, код обработки прерываний может как вносить вклад в синхронизацию потоков, так и содержать гонки данных, которые необходимо своевременно обнаруживать. Таким образом, ThreadSanitizer обязан внедрять соответствующие проверки в код, исполняемый в процессе обработки прерываний.

Так как текущая реализация алгоритма ThreadSanitizer содержит блокировки и критические секции, вызовы библиотеки времени исполнения должны исполняться с запретом асинхронных прерываний. Запрет прерываний на время вызовов функций библиотеки времени исполнения Thread Sanitizer гарантирует, что на одном физическом ядре обработчики асинхронных прерывания обрабатываются в изоляции относительно кода проверок из библиотеки времени исполнения Thread Sanitizer для родительского потока.

Также, процедуры переключения контекстов нарушают предположение подсистемы трассировки вызовов Thread Sanitizer, что вызов функции и возврата из неё происходят в одном потоке и требуют отключения внедрения проверок в их коде.

7.2 Динамический анализ обработчиков асинхронных прерываний

Поскольку асинхронные прерывания исполняются конкурентно с кодом родительского потока, между ними существует возможность возникновения гонок по данным. С этой целью

обработчик прерывания следует рассматривать как отдельный логический поток, имеющий другой идентификатор доступов, хранящихся в теневой памяти, то есть привязанный к отдельному слоту потока.

Так как в каждый момент времени каждый поток исполняет либо собственный код, либо код обработчика прерывания, логический поток обработчика прерывания может разделять с родительским потоком теневой стек и буфер трассировки (рис. 6). Таким образом, поддержка асинхронных прерываний почти не требует дополнительной памяти. Также описанный подход упрощает диагностику при трассировке стека доступа, вызвавшего гонку, так как будет учтён стек вызовов не только обработчика прерывания, но и прерванного потока.

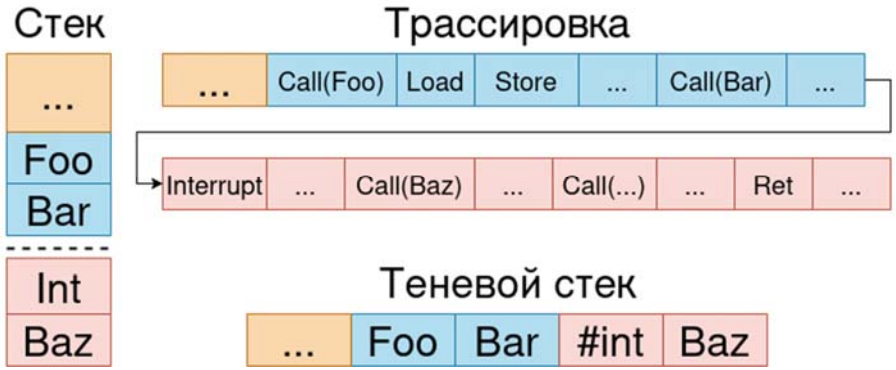

Рис. 6. Трассировка и теневой стек обработчика прерывания.
Fig. 6. Tracing and shadow stack with interrupt handlers.

В описанной реализации локальное хранилище данных структуры потока в реализации ThreadSanitizer в ядре ОС привязано к идентификатору соответствующего контекста исполнения, а текущий слот логического потока зависит от наличия прерываний.

Алгоритм, реализующий поддержку обработчиков прерываний в качестве отдельных логических потоков, должен определить семантику операций переключения контекста в ядре ОС относительно синхронизации в модели памяти. В реализации ядра ОС присутствуют три основных вида переключений контекста: переключение на исполнение другого потока ядра, переключение в пространство пользователя и, в момент возникновения прерывания, переключение на точку входа в его обработчик и возврат из прерывания. Прерывания, которые необходимо обрабатывать, могут быть как прерываниями из пространства пользователя, так и асинхронными прерываниями из пространства ядра.

7.3 Модель переключения контекстов в ядре ОС CLOS

Обработка прерываний и переключений контекстов исполнения в ядре ОС требует описания семантики работы каждой затрагиваемой операции, её предусловия и эффекта в некоторой модели состояния системы. В упрощённой модели далее рассматривается следующее состояние, локальное для каждого ядра центрального процессора:

- **context**: *integer* – ID текущего потока ядра ОС
- **ienable**: *bool* – флаг разрешения асинхронных прерываний
- **is_user**: *bool* – флаг исполнения пользовательского кода
- **async_int**: *integer* – уровень вложенности текущего обработчика асинхронного прерывания ядра (0 для случая исполнения кода потока), локальный для текущего context

Для простоты рассуждений и реализации модель не разделяет различные номера прерываний. В данной модели верны следующие инварианты:

- is_user ⇒ ienable
- async_int > 0 ⇒ ¬is_user

Определим в описанной модели обрабатываемые операции управления прерываниями (см. табл. 1). Нетрудно заметить, что вышеописанные инварианты согласуются с описанными операциями.

Табл. 1. Операции в модели переключения контекстов ОС CLOS.
Table 1. Operations in the CLOS OS model of context switching.

Операция	Описание	Предусловие	Эффект
async_enable	Разрешение асинхронных прерываний	¬is_user ∧ ¬ienable	ienable = 1
async_disable	Запрет асинхронных прерываний	¬is_user ∧ ienable	ienable = 0
switch_context(T)	Переключение на поток T	¬is_user ∧ ¬ienable	context = T ∧ async_int = async_int(T)
switch_to_user	Переключение в код пользователя	¬is_user ∧ ¬ienable	is_user = 1 ∧ ienable = 1 ∧ async_int = 0
kernel_sync_int	Вход в синхронное прерывание ядра	¬is_user	∅
kernel_sync_ret	Возврат из синхронного прерывания ядра	¬is_user	∅
kernel_async_int	Вход в асинхронное прерывание ядра	¬is_user ∧ ienable	async_int = async_int+1
kernel_async_ret	Возврат из асинхронного прерывания ядра	¬is_user ∧ async_int > 0	async_int = async_int−1
user_int	Вход в (любое) прерывание пользователя	is_user	is_user = 0 ∧ ienable
user_ret	Возврат из (любого) прерывания пользователя	¬is_user ∧ ¬ienable ∧ async_int = 0	is_user = 1 ∧ ienable = 1

7.4 Семантика асинхронных прерываний в коде ядра

Операция разрешения прерываний *async_enable* имеет семантику *release*-операции (см. определения ранее), поскольку, помимо управления прерываниями, служит аппаратным барьером для операций записи (т. н. *release*-барьером). Аналогично, операция запрета прерываний *async_disable* имеет эффект аппаратного барьера для операций чтения (т. н. *acquire*-барьера). Таким образом, точка входа в прерывание *kernel_async_int* и операция возврата из него *kernel_async_ret* имеют, соответственно, семантику *acquire* и *release* операций.

Операция *kernel_async_int* требует **¬is_user∧ienable**, а значит, наблюдает эффект **ienable = 1** при **¬is_user** некоторой операции включения прерываний *async_enable* на данном ядре.

Операция *async_disable* имеет эффект **ienable = 0**, а значит, наблюдает эффект всех обработанных прерываний на данном ядре, в силу предусловия **ienable** операции *kernel_async_int*, её эффекта **async_int = async_int + 1** и предусловия **async_int > 0** операции *kernel_async_ret*.

Операции *kernel_sync_int* и *kernel_sync_ret* не меняют состояния модели и не обрабатываются в реализации.

Таким образом, на одном ядре CPU верно, что

- *async_enable* synchronizes-with *kernel_async_int*
- *kernel_async_ret* synchëronizes-with *async_disable*

С целью поддержания этого отношения, в структуре-дескрипторе физического ядра выделяются отдельные векторные часы, **«core» vclock**, по аналогии с обработкой регулярной переменной синхронизации (рис. 7).

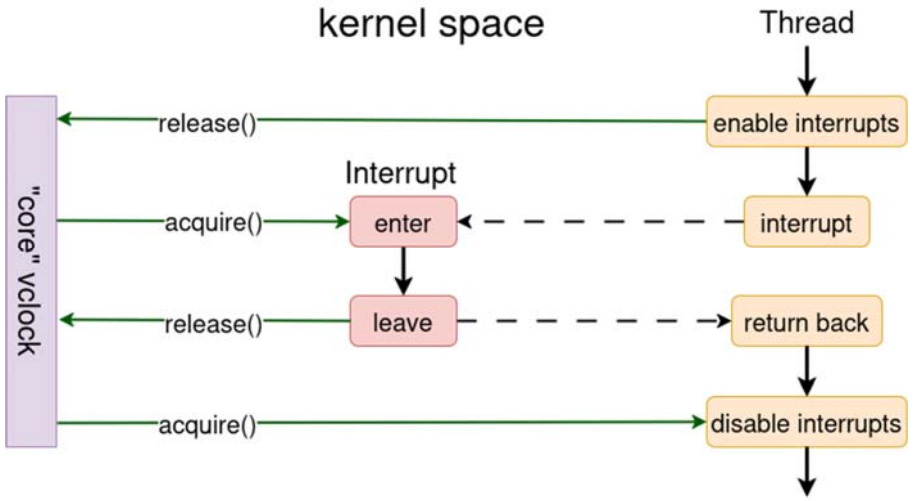

Рис. 7. Асинхронные прерывания.
Fig. 7. Asynchronous interrupts.

7.5 Семантика переключений контекста исполнения

При синхронном переключении с контекста потока **Thread 1** (далее **T1**) на контекст другого потока ядра **Thread 2** (далее **T2**) – *switch_context(***T2***)* – все последующие операции потока **T2** наблюдают эффект всех операций текущего потока **T1**, исполненных к данному моменту.

Так как для *switch_context* выполняется предусловие **¬is_user∧¬ienable**, верно, что *async_disable* sequenced-before *switch_context*, и для синхронизации потоков **T1** и **T2** возможно использовать те же векторные часы **«core» vclock**, что и в предыдущем пункте (рис .8).

7.6 Семантика прерываний в коде пользователя

Переключение в пространство пользователя останавливает прогресс кода потока ядра,д однако ядро продолжает обрабатывать прерывания пользовательского кода, как асинхронные

(например, прерывание таймера), так и синхронные (например, системные вызовы). По аналогии с обработкой прерываний в ядре, операции *switch_to_user* и *user_ret* имеют release-семантику, а операция *user_ret* имеет acquire-семантику.

Так как *user_int* имеет предусловие **is_user**, а операции *switch_to_user* и *user_ret* имеют эффект **is_user = 1**, верно, что *user_int* наблюдает эффект *switch_to_user* и *user_ret*.

Также *switch_to_user* имеет предусловие **¬is_user∧¬ienable**, то есть, наблюдают эффект **ienable = 0** операции *async_disable*. В свою очередь, *async_disable* наблюдает эффект всех обработанных прерываний на данном ядре (см. раздел 7.4).

Таким образом, на одном ядре CPU выполняется

- *switch_to_user* synchronizes-with *user_int*
- *user_ret* synchronizes-with *user_int*

Для обработки этого отношения, аналогичным с **«core» vclock** образом, в структуре-дескрипторе физического ядра выделяются ещё одни векторные часы, **«user» vclock** (рис. 9). Свойство *async_disable* synchronizes-with *switch_to_user* обеспечивается синхронизацией **«user» vclock** с **«core» vclock**.

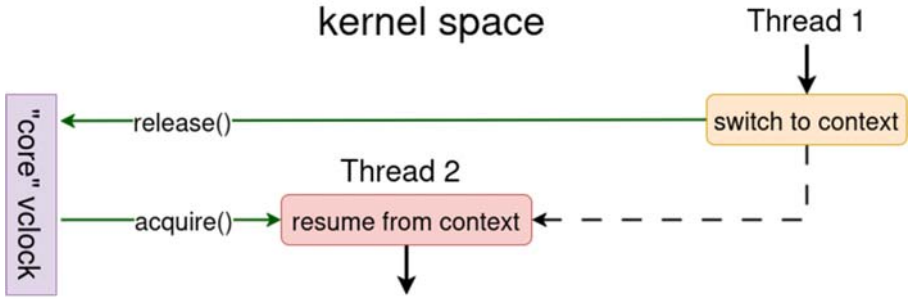

Рис. 8. Переключение контекста.
Fig. 8. Context switch.

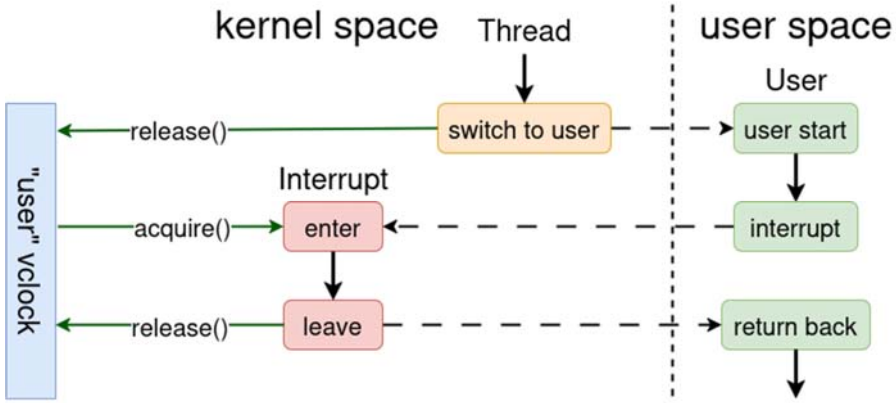

Рис. 9. Пользовательские прерывания.
Fig. 9. Userspace interrupts.

8. Результаты

8.1 Производительность

В связи с приоритетом оптимизации расходов по памяти, адаптированный к коду ядра ОСРВ алгоритм ThreadSanitizer заметно уступает по производительности версии ThreadSanitizer из набора инструментов программной инфраструктуры LLVM. На рис. 10 показано относительное увеличение времени чтения и записи сообщения размером 1024 байта для различных портов (примитивов взаимодействия между процессами в ОСРВ CLOS). Замедление интерфейсов большинства примитивов синхронизации также лежит в пределах от 100 до 150 раз по отношению к их неинструментированным версиям.

Несмотря на приоритет экономии ресурсов памяти над оптимизацией производительности, текущая реализация алгоритма позволяет успешно тестировать код ядра ОСРВ в различных сценариях работы, обнаруживать ошибки многопоточной синхронизации и предоставлять детальную диагностику их места возникновения.

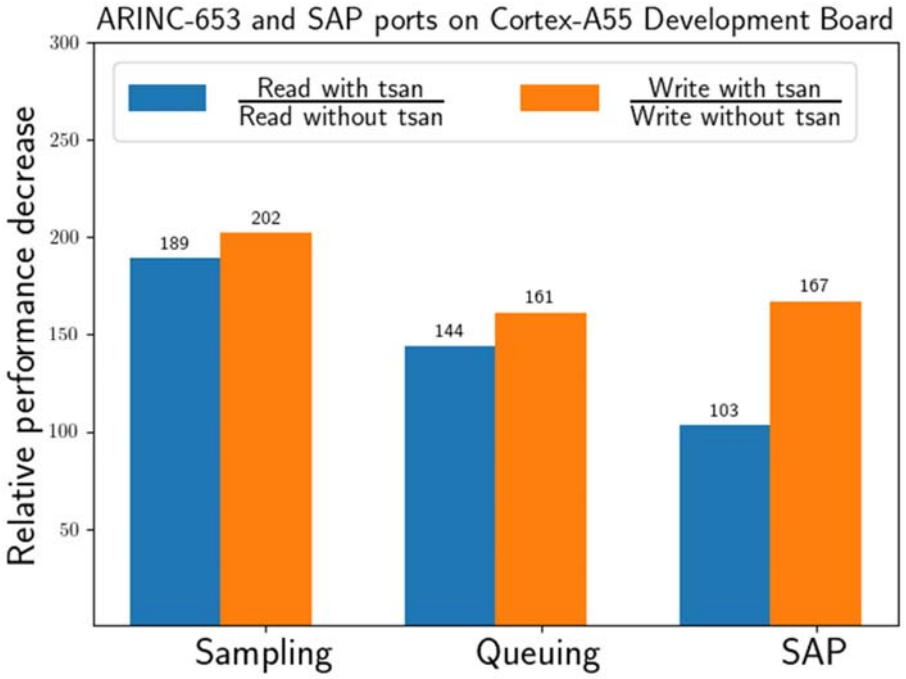

Рис. 10. Замедление процедур интерфейсов ARINC-653 и SAP портов.
Fig. 10. Performance decrease for ARINC-653 and SAP port interfaces.

8.2 Обнаруженные ошибки многопоточной синхронизации

Поскольку до интеграции Thread Sanitizer код ядра ОСРВ CLOS тестировался с применением детектора гонок по данным RaceHunter, основанного на методе точек останова по данным, а также в силу сравнительно небольшого объёма кода, алгоритм Thread Sanitizer способен найти сравнительно немного новых ошибок в коде ядра. Также, частые операции обработки переполнения эпохи теоретически способны привести к необнаружению гонок в некоторых исполнениях.

Несмотря на это, в результате интеграции в систему сборки и тестирования ядра ОС реального времени CLOS алгоритма Thread Sanitizer была подтверждена одна и обнаружены ещё две гонки по данным – как между потоками ядра, исполняющимися на различных процессорных ядрах, так и между кодом потока и обработчиком прерывания на одном ядре (листинг 2).

```
[TSAN] race condition on addr 0x822034e0 :
[tid=9 pattern = ".......X" kind="Regular Write"] ,
[tid=8 pattern = ".......X" kind="Regular Read "]
[TSAN] tid 9 backtrace
[00] 0x000000009003c8ac
[01] 0x000000009004c604
[02] 0x0000000090047bdc
[03] 0x000000009002eac8
...
[09] 0x000000009001e5c0
[10] 0x0000000000000004
[TSAN] tid 8 backtrace
[00] 0x00000000900329e8
[01] 0x00000000900522b4
[02] 0x0000000090052988
[03] 0x000000009002eac8
...
[09] 0x000000009001e5c0
[10] 0x0000000000000000
```

Листинг 2. Сообщение об обнаруженной гонке по данным.
Listing 2. Detailed log message about a data race detected.

Также, для тестирования инструмента в систему тестирования ядра ОСРВ добавлен сценарий, содержащий гонки по данным, и позволяющий проверять корректность работы Thread Sanitizer в коде ядра.

9 Заключение

В данной работе показано, что алгоритм поиска гонок по данным, основанный на последней версии LLVM ThreadSanitizer, может быть адаптирован к интеграции с системой сборки и тестирования ядра ОС реального времени. Возникающие при этом задачи соответствия жестким требованиями на предсказуемость времени исполнения и затраты требуемой памяти имеют решения, позволяющие обнаруживать гонки по данным, оказывая ограниченное влияние на эти свойства исполнения.

В сравнении с ОС Linux, где в качестве инструмента динамического анализа для поиска гонок по данным в ядре был выбран подход, основанный на точках останова по данным (KCSAN), в ОС специального назначения, в частности, в ОС реального времени, проблемы при интеграции детекторов, основанных на построении отношения *happens-before* преодолимы, что позволяет применять оба семейства детекторов гонок по данным, и анализировать больший спектр исполнений.

Список литературы / References

[1]. J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel, SIGPLAN Not., vol. 53, pp. 405–418, Mar. 2018.
[2]. M. Elver, Concurrency bugs should fear the big bad data-race detector. https://lwn.net/Articles/816850/ , 2020. [Online; accessed 21-May-2025].
[3]. K. Serebryany and T. Iskhodzhanov, Threadsanitizer – data race detection in practice, pp. 62–71, 12 2009.

[4]. ISO Central Secretary, Information technology – Programming Languages – C, Standard ISO/IEC 9899:2024, International Organization for Standardization, Geneva, CH, 2024.
[5]. D. Vyukov, Llvm thread sanitizer. https://github.com/google/sanitizers/wiki/threadsanitizercppmanual, 2020. [Online; accessed 22-May-2025].
[6]. N. Komarov, On the implementation of data-breakpoints based race detection for linux kernel modules, 2013.
[7]. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, ACM Trans. Comput. Syst., vol. 15, p. 391–411, Nov. 1997.
[8]. F. Mattern, Virtual time and global states of distributed systems, 01 2004.
[9]. L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM, vol. 21, p. 558–565, July 1978.
[10]. A. Konovalov, Kernel thread sanitizer. https://github.com/google/kernel-sanitizers/blob/master/KTSAN.md , 2015. [Online; accessed 24-April-2025].

Информация об авторах / Information about authors

Егор Сергеевич ЕЛЬЧИНОВ – старший лаборант отдела технологий программирования Института системного программирования. Сфера научных интересов: методы динамического анализа ПО, алгоритмы многопоточной синхронизации, операционные системы.

Egor Sergeevich ELCHINOV – Senior Lab Assistant of the Department of Programming Technologies of the Institute for System Programming of the RAS. Research interests: methods for software dynamic analysis, concurrency algorithms, operating systems.