Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)
Факультет управления и прикладной математики
Кафедра информатики

Коновалов Андрей Дмитриевич

Автоматический поиск состояний гонок в ядре ОС Linux

03.04.01 — Прикладные математика и физика

Выпускная квалификационная работа магистра

Научный руководитель:
к.ф.-м.н. Хорошилов Алексей Владимирович

Долгопрудный
2016 г.

Содержание

1 Введение

При написании программ разработчики часто допускают различные ошибки. Такие ошибки могут приводить к неправильному или непредсказуемому поведению программ: замедлению скорости работы, аварийному завершению исполнения, порче данных и т.п.. Результатом подобного поведения могут быть большие экономические потери и даже гибель людей [1].

Даже при отсутствии ошибок в написанной разработчиком программе во время ее выполнения все равно могут произойти ошибки. Виной этому могут быть ошибки в каком-либо из компонентов операционной системы, в том числе в ее ядре. Ядро операционной системы Linux является одним из самых больших и сложных проектов с открытым исходным кодом на текущий момент. Поиск и исправление ошибок в ядре является актуальной задачей, поскольку в настоящее время система Linux лидирует на рынках смартфонов, интернет-серверов и суперкомпьютеров [2].

Ошибки могут быть разных типов: ошибки в логике работы программы, ошибки синхронизации потоков, ошибки работы с памятью и т.п. Поиск ошибок вручную не всегда бывает эффективен. Например, ошибки могут наблюдаться очень редко или не проявляться какое-то время после того, как они произошли. Для решения этой проблемы процесс поиска ошибок можно автоматизировать. В данной работе рассматривается автоматический поиск состояний гонок в ядре Linux. Согласно исследованию ИСП РАН [3], среди типовых ошибок в ядре наиболее часто встречаются именно состояния гонок.

Для автоматического поиска ошибок в существующих программах, используются специальные приложения, которые называются *детекторами* (англ. *detectors*). Детекторы ошибок в основном разделяются на статические и динамические [4]. Статические детекторы анализируют исходный код программы, не осуществляя ее запуск. Иногда простые статические детекторы встроены прямо в компилятор, который выдает предупреждения во время компиляции программы. Напротив, динамические детекторы анализируют поведение программы по мере ее исполнения и обнаруживают ошибку, только если она в действительности произошла.

Недостатками статического анализа является большая вычислительная сложность и низкая точность. Часто поиск нетривиальных ошибок статическим анализом с достаточной точностью возможен только для отдельных изолированных модулей

программы и требует особой организации ее исходного кода [5]. К недостаткам динамического анализа относится необходимость наличия тестов, возникающая вследствие того, что обнаружение ошибки происходит только во время исполнения кода, который ее содержит.

В рамках данной работы был выбран динамический анализ, поскольку существенное редактирование и реорганизация исходного кода ядра Linux (более 15 млн. строк кода на языке C) за разумное время представлялись невозможными. Однако поскольку при работе ядра исполняется лишь ограниченный набор драйверов (исполняются только драйвера, отвечающие за работу присоединенных к тестирующему компьютеру устройств), обнаружение ошибок с помощью динамических детекторов в драйверах является затруднительным.

В большинстве современных операционных систем ядро операционной системы и программы пользователя имеют разные привилегии, т.е. обладают разными правами на исполнение инструкций, чтение памяти и т.п. В операционной системе Linux различают два режима выполнения пользовательского процесса: *режим ядра* (англ. *kernel mode*) и *режим пользователя* (англ. *user mode*). Процесс начинает выполнение в режиме пользователя. Когда процесс производит обращение к операционной системе, режим выполнения процесса переключается с режима пользователя на режим ядра: операционная система пытается обслужить запрос пользователя, возвращая код ошибки в случае неудачного завершения операции.

В дальнейшем детекторы, которые ищут ошибки в приложениях пользователя, будем называть *детекторами ошибок для приложений пользователя* или *детекторами ошибок в пространстве пользователя*, а детекторы, которые ищут ошибки в ядре и его компонентах, – *детекторами ошибок для ядра* или *детекторами ошибок в пространстве ядра*.

Сейчас существует несколько довольно эффективных и точных детекторов состояний гонок в пространстве пользователя [6, 7]. Однако существующие детекторы состояний гонок в пространстве ядра обладают рядом ограничений. Они могут тестировать лишь отдельные компоненты ядра, а также обладают большим количеством ложных срабатываний и неудобны в использовании. Цель данной работы: разработка нового метода поиска состояний гонок для ядра Linux, применимого для всего кода ядра.

2 Состояния гонок

Одними из самых распространенных, опасных и трудноуловимых ошибок в многопоточных программах являются состояния гонок.

Определение. ***Состоянием гонки*** *(англ. data race) называется ситуация, когда два или более потока исполнения программы могут одновременно осуществить доступ к одной области памяти (например, переменной) и как минимум один из этих доступов является записью.*

Считается, что любые два доступа к памяти из двух потоков могут произойти одновременно, если обратное не гарантируется применяемой в программе дисциплиной синхронизации.

Следует отметить, что в англоязычной литературе существует два различных понятия, которые на русский язык переводятся как состояние гонки [5]. Одно из них, называемое “race condition”, относится к недетерминированности исполнения параллельных процессов. Другое, более узкое, называемое “data race”, относится к частному случаю недетерминизма, возникающему в многопоточных программах при неправильной работе потоков с *разделяемой памятью* (англ. *shared memory*) и соответствует определению выше.

Состояние гонки является серьезной ошибкой программирования – такие ошибки могут приводить к нарушению целостности данных и аварийному завершению программы. Подобные ошибки бывает очень трудно найти, так как они происходят лишь при определенных условиях, которые зачастую сложно воспроизвести специально, даже если знать эти условия – например, особая последовательность переключений исполнения двух потоков.

Состояния гонок приводят к негативным последствиям вследствие двух причин. Во-первых, результат исполнения программы с состояниями гонок может зависеть от того, в каком порядке инструкции программы исполняются различными потоками. Такую ситуацию бывает чрезвычайно трудно воспроизвести при использовании обычных способов отладки. Во-вторых, компилятор при оптимизации кода программы с состоянием гонки может преобразовать его таким образом, что он становится очевидно ошибочным [8].

Существует понятие так называемых *безобидных состояний гонок* (англ. *benign*

data race). К ним относится, например, использование разделямой целочисленной переменной для подсчета статистики. Многие программисты не считают подобные состояния гонок ошибками. Однако вследствие оптимизаций компилятора, такие состояния гонок приводят к настоящим ошибкам [8].

2.1 Неопределенное поведение

Некоторые ошибки приводят к *неопределенному поведению* (англ. *undefined behaviour*) программы. Это состояние возникает при нарушении программистом стандарта языка или спецификаций используемых программных функций или библиотек [5]. В этом случае компилятор или библиотека вправе выполнять некорректные и неожиданные действия.

Особенностью таких ошибок является то, что обычно они не имеют наблюдаемых последствий, но иногда могут приводить к серьезным сбоям. При этом условия возникновения этих ошибок могут быть трудновоспроизводимыми – например, зависеть от версии компилятора, библиотеки или операционной системы; частоты процессора, количества ядер или даже его температуры. Более того, многие такие ошибки приводят не к мгновенным последствиям, а к порче данных, эффект от которой сможет наблюдаться лишь через некоторое время, затрудняя понимание и обнаружение ошибки.

Состояния гонок относятся к ошибкам, порождающим неопределенное поведение. Начиная с версии C/C++ 11, стандарт языка явно об этом говорит [9].

Для демонстрации последствий неопределенного поведения рассмотрим пример программы [5], приведенный на листинге 1. В этой программе на последней, 64-й, итерации цикла происходит переполнение целочисленной знаковой переменной `value`. По стандарту C/C++ результат переполнения знаковых целочисленных переменных неопределен. Это очень распространенный тип ошибок, который нередко приводит к серьезным уязвимостям программного обеспечения [10].

```
#include <stdio.h>

void foo(int* array) {
  int value = 0x03020100;
  for (int i = 0; i < 64; i++) {
    printf("%d␣", i);
    array[i] = value;
    value += 0x04040404;
  }
  printf("\n");
}

int main() {
  int values[64];
  foo(values);
  return 0;
}
```

Листинг 1: Пример программы с переполнением целочисленной переменной

Ожидаемый результат работы функции `foo` при запуске такого кода – печать значений от 0 до 63, а также заполнение аргумента-массива значениями. Этот результат действительно наблюдается при использовании компилятора GCC [11] версии 4.8.1 с настройками по умолчанию, как показано на листинге 2.

```
$ g++ test.cpp && ./a.out
0 1 2 ... 61 62 63
```

Листинг 2: Запуск программы с переполнением целочисленной переменной

Однако при включении компиляторных оптимизаций на 64-битной архитектуре результат получается другой. В данном случае печать значений продолжается и после 63 до тех пор, пока не произойдет ошибка сегментации, как показано на листинге 3.

```
$ g++ -O2 test.cpp && ./a.out
0 1 2 ... 61 62 63 64 65 66 ... 2062 2063 2064
Segmentation fault
```

Листинг 3: Запуск программы с переполнением целочисленной переменной, скомпилированной со включением компиляторных оптимизаций

Такое поведение программы связано с наличием в ней неопределенного поведения, поскольку в этом случае компилятор вправе породить произвольный код после

возникновения ошибки. При компиляции примера из листинга 1 получается ассемблерный код, приведенный в листинге 4. Видно, что в полученном коде нет условных переходов, которые могли бы прекратить исполнение цикла; также нет и инструкций возврата. Когда о таком поведении было сообщено разработчикам GCC, они отказались что-либо менять, сославшись на неопределенное поведение [12].

```
.LC0:
  .string "%d␣"
foo:
  push    %r12
  mov     %rdi, %r12
  push    %rbp
  mov     $0x3020100, %ebp
  push    %rbx
  xor     %ebx, %ebx
  xchg    %ax, %ax
.L2:
  mov     %ebx, %edx
  mov     .LC0, %esi
  mov     $0x1, %edi
  xor     %eax, %eax
  callq   __printf_chks
  mov     %ebp, (%r12,%rbx,4)
  add     $0x4040404, %ebp
  add     $0x1, %rbx
  jmp     .L2
```

Листинг 4: Ассемблерный код, получаемый при компиляции функции с переполнением целочисленной переменной

Таким образом, следует отметить, что неопределенное поведение может приводить не только к некорректному результату ошибочной операции, но и к произвольным результатам любых дальнейших операций. Понятно, что если одна и та же ошибка по-разному проявляется при использовании одного и того же компилятора с разными настройками, то подобные ошибки могут становиться наблюдаемыми при переходе на новую версию компилятора, процессора и вообще программного и аппаратного обеспечения. Другими словами, при наличии в программе ошибки, приводящей к неопределенному поведению, даже если сегодня программа работает корректно, то никто не может гарантировать, что программа продолжит работать корректно завтра.

2.2 Поиск состояний гонок

Для поиска состояний гонок динамические детекторы отслеживают обращения программы к памяти (переменным) и использование программой примитивов синхронизации во время исполнения. Для проверки того, находятся ли два доступа к памяти в состоянии гонки в соответствие с определением, детектору необходимо уметь проверять возможность одновременности двух событий. Существует два классических алгоритма, используемых в динамических детекторах для поиска состояний гонок: Lockset и Happens-before. Каждый из этих алгоритмов по-своему определяет понятие одновременности и в соответствие со своим определением ищет состояния гонок.

2.2.1 Алгоритм Lockset

Алгоритм Lockset строится из предположения, что каждой переменной, которая может быть использована из нескольких потоков, должен соответствовать синхронизирующий доступы к ней мьютекс. Данный алгоритм был разработан для динамического детектора Eraser, применявшегося для тестирования программного обеспечения серверов поисковой компании AltaVista в середине 1990-х годов [13].

Определение. ***Мьютекс*** *(англ. mutex от "mutual exclusion", "взаимное исключение") – это такой объект, который находится в одном из двух состояний: свободный или захвачен (взят) потоком T. Над свободным мьютексом M поток T может выполнить операцию захвата (англ. lock), а затем этот же поток может выполнить операцию освобождения (англ. unlock). При попытке потока T захватить мьютекс M, уже захваченный другим потоком T', поток T блокируется до возможности захватить мьютекс M.*

Определенный таким образом мьютекс обладает свойством *взаимного исключения* (англ. *mutual exclusion*), то есть он может быть захвачен только одним потоком. Алгоритм Lockset считает, что два доступа к памяти могут произойти одновременно тогда и только тогда, когда не существует мьютекса, взятого при обоих доступах.

Алгоритм Lockset работают следующим образом [5]. Алгоритм отслеживает операции взятия мьютексов и обращения к разделяемым переменным осуществляемые потоками исполнения. Каждой разделяемой переменной V ставится в соответствие

множество мьютексов (англ. *lockset*) $LS(V)$, которые были взяты при каждом обращении к этой переменной. Первоначальное значение $LS(V)$ для каждой переменной задается как множество всех возможных мьютексов. При каждом обращении потоком T к разделяемой переменной V в $LS(V)$ записывается пересечение старого значения $LS(V)$ и множества взятых потоком T мьютексов. Если множество $LS(V)$ становится пустым, то это значит, то не существует ни одного мьютекса, который бы синхронизировал доступы к V. В этом случае выводится сообщение об ошибке. Псевдокод алгоритма представлен в листинге 5.

```
1  function InitializeShadowValue(V) {
2    LS(V) = GetMaximumLockset();
3  }
4
5  function HandleMemoryAccess(T, V) {
6    LS(V) = LS(V).Intersect(GetCurrentLockset(T));
7    if (LS(V).Empty()) {
8      ReportRace(T, V);
9    }
10 }
```

Листинг 5: Псевдокод алгоритма Lockset

Алгоритм Lockset не пропускает ошибок, то есть если при исполнении многопоточного кода возникло состояние гонки, оно будет обнаружено детектором. Если некоторый мьютекс $M \in LS(V)$, то либо он был захвачен при всех доступах к V, либо к V еще не было ни одного доступа. А значит, если алгоритм Lockset не обнаружил состояние гонки на переменной V, то к этой переменной либо не было доступов, либо при выполнении всех доступов был захвачен хотя бы один мьютекс. Таким образом, состояние гонки невозможно по определению.

К сожалению, алгоритм Lockset имеет ложные срабатывания, то есть если детектор выводит сообщение о найденной ошибке, то это не значит что ошибка на самом деле произошла. В частности ошибки происходят при использовании в программе средств синхронизации, отличных от мьютексов (семафоры, атомарные переменные, и т.д.).

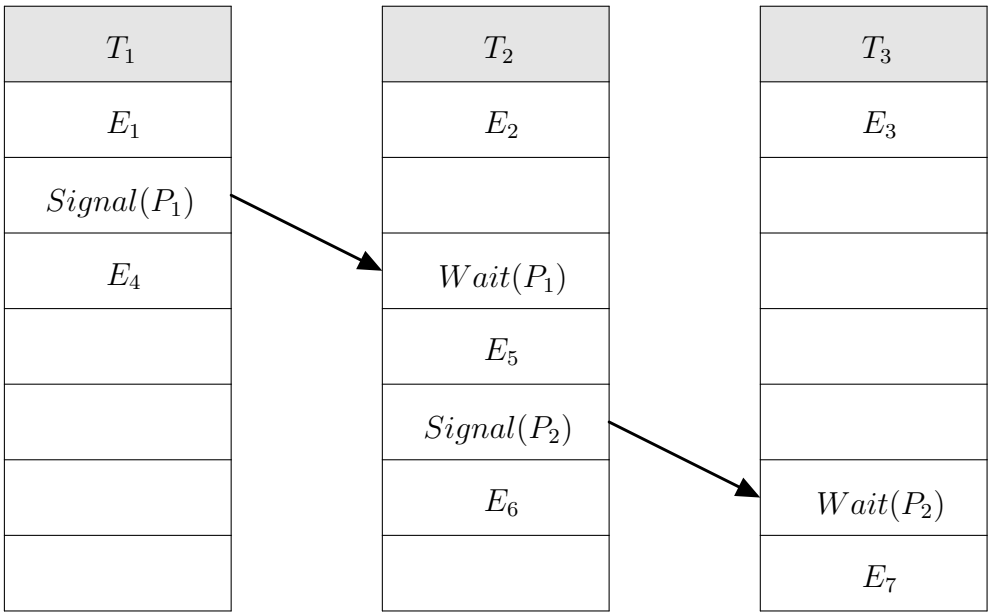

Рис. 1: Иллюстрация отношения предшествования

2.2.2 Алгоритм Happens-before

Алгоритм Happens-before использует отношение частичного порядка *"предшествует"* (англ. *happens-before*) для проверки одновременности событий [14]. Это отношение определяется следующим образом [5].

Определение. *Событие A* ***предшествует*** *(англ. happens-before) событию B ($A \prec B$), если выполнено хотя бы одно из следующих утверждений:*

- *A и B были выполнены одним и тем же потоком исполнения и A произошло раньше B;*
- *A является* ***уведомлением*** *(англ. signal) некоторого примитива синхронизации P, при этом B является* ***ожиданием*** *(англ. wait) на этом примитиве;*
- *Существуют такие события A' и B', что $A \preceq A' \prec B' \preceq B$ (свойство транзитивности).*

На рисунке 1 иллюстрируется это определение. В трех потоках исполнения T_1, T_2 и T_3 происходят события E_1, E_2, ..., E_7. Кроме того, имеется два примитива синхронизации P_1 и P_2 на которых происходят события уведомления и ожидания.

В соответствие с определением:

- $E_1 \prec E_4$, так как E_1 произошло раньше E_4 в одном потоке;
- $Signal(P_1) \prec Wait(P_1)$ и $Signal(P_2) \prec Wait(P_2)$, так как события являются уведомлением и ожиданием на одном и том же примитиве;
- $E_1 \prec E_7$ по свойству транзитивности;
- $E_4 \not\prec E_2$, $E_2 \not\prec E_3$, так как события не упорядочены.

Детекторы, работа которых основана на этом алгоритме, во время исполнения программы отслеживают доступы к памяти и использование примитивов синхронизации. В процессе исполнения, детектор строит определенное выше отношение предшествования на множестве произошедших событий. Если обнаруживаются два доступа по одному адресу памяти, хотя бы один из которых является записью и ни один из них не предшествует другому, выводится сообщение о состоянии гонки.

Существует несколько способов реализации вычисления отношения предшествования на множестве произошедших событий. Один из них – это построение ориентированного графа событий. Ребра проводятся между последовательно происходящими событиями отдельных потоков, а также между парами событий уведомление-ожидание. Для проверки отношения предшествования для двух отдельно взятых событий остается обойти этот граф. К сожалению обход графа для проверки каждого доступа к памяти приводит к большим численным расходам.

Существуют и другие алгоритмы вычисления отношения предшествования, эквивалентные обходу графа событий. Чаще всего используются более эффективные алгоритмы, использующие *векторные часы* (англ. *vector clock*):

Определение. ***Векторное время*** *(англ. vector clock) V – это отображение из множества потоков во множество неотрицательных целых чисел $V : \tau \to Z_+$. Переменные, хранящие значение типа векторное время, будем называть векторными часами.*

Каждому потоку в программе T ставятся в соответствие свои векторные часы V_T. Кроме того, каждому примитиву синхронизации P в программе также ставятся в соответствие свои векторные часы V_P. Число, которое является образом $V_T(T)$ для потока T и его векторных часов V_T называется собственным временем этого потока.

Значения векторных часов определяются следующим образом:

- *При создании потока T его векторные часы получают значение*

$$V_T(t) = \begin{cases} 1, & t = T, \\ 0, & t \neq T. \end{cases}$$

- *При обработке доступа к памяти, совершенного потоком T, часы этого потока «тикают» (собственное время потока инкрементируется):*

$$V'_T(t) = Tick(V_T, T)(t) = \begin{cases} V_T(t) + 1, & t = T, \\ V_T(t), & t \neq T. \end{cases}$$

- *При создании нового примитива синхронизации P, его векторные часы хранят нулевое значение для всех потоков:*

$$V_P(t) = 0, \quad \forall\ t \in \tau.$$

- *При уведомлении примитива P потоком T, векторные часы этого примитива обновляется следующим образом:*

$$V'_P(t) = Join(V_T, V_P)(t) \equiv max(V_P(t), V_T(t)), \quad \forall\ t \in \tau.$$

Эта операция называется «объединением» векторных часов. В этом случае говорят, что векторные часы потока T уведомляют векторные часы примитива P.

После этого часы самого потока T «тикают»:

$$V'_T = Tick(V_T, T).$$

- *При возникновении события ожидания на примитиве P потоком T, новое значение векторных часов потока равно*

$$V'_T = Join(V_T, V_P)$$

В этом случае говорят, что векторные часы потока T ожидают векторные часы примитива P.

Важной деталью реализации детекторов, основанных на алгоритме Happens-before, является трактовка событий, связанных с операциями на мьютексах. Обычно считается, что событие освобождения мьютекса $Unlock(m)$ является уведомлением примитива m, а событие взятия $Lock(m)$ является ожиданием m. Векторные часы в этих случаях обновляются соответственно.

Фактически, элементу векторных часов $V_T(t)$, соответствующих потоку T, который является образом другого потока t, соответствует собственное время потока t на тот момент, когда поток T последний раз с ним синхронизировался. Пользуясь этим, возможно осуществить проверку того, являются ли два события синхронизированными в соответствии с алгоритмом Happens-before:

Утверждение. *Пусть события A и B произошли в потоках T_1 и T_2 соответственно, а V_{T_1} и V_{T_2} - это их векторные часы на момент возникновения событий. Тогда:*

$$(A \prec B) \quad <=> \quad V_{T_1}(T_1) < V_{T_2}(T_1)$$

3 Обзор динамических детекторов

3.1 Применение детекторов

Обнаружение ошибки с помощью динамического анализатора возможно только во время исполнения участка кода, в котором эта ошибка допущена. Вследствие этого для эффективного применения динамического тестирования необходимо иметь возможность исполнять максимальное количество различных участков кода, относящихся к программе.

Одним из возможных способов исполнения различных участков кода является использование *автоматического модульного тестирования* (англ. *unit testing*). Модульные тесты – это небольшие программы или специальные функции программы-теста, которые обычно выполняют небольшое количество простых действий с программной компонентой, после чего сравнивают полученный результат с ожидаемым [5]. Модульные тесты хороши тем, что их можно многократно перезапустить в случае возникновения недетерминированных ложных срабатываний или пропусков ошибок. К сожалению, такой подход наследует от модульного тестирования основной недостаток – невозможность нахождения ошибок в коде, не исполняемом тестами.

Другим способом является использование *рандомизированного тестирования* (англ. *fuzz testing*). Рандомизированное тестирование заключается в автоматической генерации случайных данных, передаче их на вход программе и дальнейшее отслеживание появления исключительных ситуаций, например аварийного завершения [15]. Особенностью такого подхода является возможность находить ошибки, происходящие при определенных сложных условиях, которые не удается придумать программистам и тестировщикам.

При тестировании программ сами детекторы могут совершать ошибки. В соответствии с классификацией ошибок [5], *ошибкой первого рода* или *ложным срабатыванием* (англ. *false positive*) называется такая ситуация, когда в результате работы детектора пользователь получает отчет об ошибке, которой на самом деле в программе нет. *Ошибкой второго рода* или *пропуском ошибки* (англ. *false negative*) называется такая ситуация, когда в результате работы детектора пользователь не получает отчета об ошибке, которая на самом деле в программе есть. Следует отметить, что под ошибками первого и второго рода подразумеваются особенности поведения конкретного алгоритма детектора, а не ошибки в исследуемых программах.

Ложные срабатывания затрудняют применение детекторов, поскольку на анализ каждого отчета о найденной ошибке человек может тратить значительное количество времени. Пропуски ошибок, в свою очередь, снижают пользу от использования детекторов, так как означают, что некоторые ошибки так и не будут найдены.

3.2 Внутреннее устройство детекторов

Многие динамические детекторы ошибок работают следующим образом. Детектор отслеживает и анализирует осуществляемые программой действия при исполнении. В зависимости от информации, необходимой для поиска того или иного типа ошибки, он может отслеживать различные события: обращения к памяти, выделение и освобождение участков памяти, использование примитивов синхронизации и т.д. При обнаружении ошибки детектор выводит отчет, сообщающий о найденной ошибке и содержащий информацию, полезную для локализации и исправления этой ошибки.

3.2.1 Теневая память

Многие детекторы для каждой ячейки памяти приложения хранят связанные с ней дополнительные данные. Дополнительные данные могут содержать, например, информацию о доступности этой ячейки памяти для чтения или записи или информацию о нескольких последних доступах к данной ячейке памяти. Память, хранящая эти дополнительные данные, называется *теневой памятью* (англ. *shadow memory*).

Для доступа к дополнительным данным для данной ячейки памяти приложения детектор должен вычислять адрес соответствующей ей ячейки теневой памяти. Чем проще процедура вычисления этого адреса, тем выше производительность детектора. Другими словами, для эффективной работы детектора требуется, чтобы можно было несложной процедурой получать адрес теневой памяти, зная адрес анализируемой ячейки памяти.

Одним из самых простых вариантов хранения теневой памяти является использование непрерывного участка адресного пространства, на который все адресное пространство отображается с помощью сжатия и сдвига. Такой вариант устройства теневой памяти используется в детекторах AddressSanitizer [16] и ThreadSanitizer [6]. Другие детекторы [17] могут использовать более сложную многоуровневую систему организации теневой памяти.

3.2.2 Инструментация

При каждом приложения к памяти детекторы часто осуществляют работу с данными, хранящимися в теневой памяти. Обычно, для этого перед каждым таким обращением добавляется дополнительный код. Процесс внедрения дополнительного кода в программу без изменения ее основной функциональности называется *инструментированием* или *инструментацией* (англ. *instrumentation*) [5].

Инструментация выполняется либо во время компиляции, либо перед исполнением программы, либо во время ее исполнения, и называется *компиляторной*, *статической* и *динамической* инструментацией, соответственно [5]. Для приложений пользователя известными примерами систем, использующих динамическую инструментацию, являются Valgrind [18], Pin [19] и DynamoRIO [20], популярными примерами использования компиляторной инструментации являются утилита исследования покрытия кода тестами gcov [21] и детектор ошибок mudflap [22], а в качестве примера

использования статической инструментации можно привести PEBIL [23]. Для ядра Linux примерами использования динамической инструментации являются PinOS [24], Kprobes [25] и KernInst [26], а в качестве примера использования компиляторной инструментации можно привести ftrace [27].

3.2.3 Библиотека времени исполнения

Для работы детектора в тестируемое приложение необходимо включить реализацию алгоритма детектора. Кроме того, при запуске приложения надо выделить и подготовить к использованию теневую память. Зачастую для этих целей одной только инструментации недостаточно. В программу необходимо также встроить *библиотеку времени исполнения* (англ. *runtime library*). Она может содержать новые функции и заменять реализации существующих.

Новые функции встраиваются в программу для того, чтобы их можно было вызывать из внедренного инструментацией кода. Примером такой функции является функция печати отчета о найденной ошибке. Замена существующих функций бывает полезна, когда проще предоставить новую реализацию, чем заниматься инструментацией.

Библиотека времени исполнения может внедряться в программу на стадии компиляции (при использовании компиляторной инструментации), на стадии компоновки (например при использовании статической инструментации) или при запуске (при использовании динамической инструментации) [5].

3.3 Существующие детекторы в пространстве ядра

Существует несколько детекторов состояний гонок в ядре Linux. Детекторы RaceHound [28] и KernelStrider [29] используют динамический анализ. Существует статический детектор состояний гонок CPALockator [30]. Этим детекторам присущи следующие недостатки: возможность искать состояния гонок только в отдельных компонентах ядра, а не во всем ядре в целом, наличие сложных срабатываний и сложность использования.

Детектор RaceHound [28] работает следующим образом. Во время исполнения ядро прерывается в случайные моменты времени при обращении к памяти. После прерывания, на адрес памяти, к которому осуществляет доступ текущая инструкция,

ставится *аппаратная точка прерывания* (англ. *hardware breakpoint*). Далее делается небольшая задержка в ожидании того, что какой-нибудь другой поток обратится к этой области памяти. Если это произойдет, то аппаратная точка прерывания сработает и RaceHound сообщит о найденном состоянии гонки.

Подход, используемый RaceHound, плохо работает для поиска состояний гонок в больших по размеру модулях ядра. Дело в том, что, чтобы RaceHound нашел состояние гонки, ядро должно быть прервано в том момент, когда исполняется одна из инструкций, осуществляющая несинхронизованный доступ к памяти. Даже если повторять процедуру прерывания очень часто, это событие довольно маловероятно. Однако RaceHound можно использовать для того, чтобы подтвердить потенциальное состояние гонки, на которое укажет какой-нибудь другой детектор.

Детектор KernelStrider [29] использует динамическую инструментацию для сбора информации об анализируемом модуле ядра в процессе его работы. Он отслеживает такие события как обращения к памяти и использование примитивов синхронизации и сохраняет последовательность событий для последующего анализа с помощью детектора ThreadSanitizer [6].

К сожалению, при использовании динамической бинарной инструментации невозможно отследить использование некоторых низкоуровневых примитивов синхронизации (атомарные операции, барьеры доступов к памяти), поскольку часть информации теряется на этапе компиляции. Вследствие этого, KernelStrider имеет ложные срабатывания, для устранения которых необходимо добавлять в код анализируемого модуля специальные аннотации. Кроме ложных срабатываний, недостатками детектора KernelStrider являются сложность настройки и запуска, а также возможность анализировать только отдельные модули.

Детектор CPALockator [30], построенный на основе CPAchecker [31], использует статический анализ. Этот детектор использует алгоритм Lockset для поиска состояний гонок для поиска состояний гонок. Недостатком детектора CPALockator является наличие ложных срабатываний вследствие использования статического анализа.

4 Детектор KernelThreadSanitizer

Существующие детекторы состояний гонок в ядре обладают существенными недостатками. Они нетривиальны в использовании, имеют ложные срабатывания и способны находить состояния гонок лишь в отдельных компонентах ядра. Эти недостатки были устранены в разработанном детекторе KernelThreadSanitizer, алгоритм которого основан на алгоритме детектора состояний гонок для приложений пользователя ThreadSanitizer, хорошо показавшего себя на практике.

4.1 Устройство детектора KernelThreadSanitizer

Детектор KernelThreadSanitizer состоит из двух частей: одна является составляющей компилятора, а другая является компонентом ядра. Компиляторная составляющая осуществляет интрументацию доступов к памяти, а компонент ядра содержит в себе аннотации для используемых в ядре примитивов синхронизации и реализацию анализирующего алгоритма. Во время исполнения ядра детектор отслеживает обращения к памяти и использование примитивов синхронизации, которые происходят в ядре, и передает их анализирующему алгоритму. Анализирующий алгоритм основан на алгоритме Happens-before и использует концепцию векторных часов.

4.1.1 Теневая память

KernelThreadSanitizer использует теневую память для хранения дополнительной информации о памяти ядра. На каждую 8-байтную ячейку памяти ядра, детектор хранит 32 байта теневой памяти. Эти 32 байта предсталуют собой 4 ячейки по 8 байт, каждая из которых описывает одно из последних обращений к соответствующий ячейке памяти ядра.

На данный момент теневая память выделяется каждый раз, когда ядро выделяет какое-либо количество страниц физической памяти с помощью внутреннего распределителя памяти. Таким образом, каждому блоку выделенных страниц соответствует свой блок страниц теневой памяти. Система адресации может быть реализована и более оптимально: теневая память может быть выделена непрерывным участком, который отображается на память ядра сжатием со сдвигом.

Каждая из ячеек теневой памяти описывает несколько параметров обращения к

памяти. Структура ячейки показана на листинге 6. Эта структура включает в себя идентификатор потока, совершившего доступ к памяти, его собственное время во время доступа, сдвиг от начала 8-байтовой ячейки памяти, соответствующий адресу обращения, размер доступа и два флага, говорящих о том, является ли доступ операцией записи или чтения и является ли доступ атомарным.

```
struct kt_shadow_s {
  unsigned long threadId : 12;
  unsigned long clock    : 42;
  unsigned long offset   : 3;
  unsigned long size     : 2;
  unsigned long isWrite  : 1;
  unsigned long isAtomic : 1;
};
```

Листинг 6: Структура ячейки теневой памяти в детекторе KernelThreadSanitizer

Под идентификатор потока в ячейке теневой памяти отводится 12 бит. Это означает, что максимальное количество одновременно существующих потоков, которое детектор способен обрабатывать равно $2^{12} = 4096$. Этого количества достаточно, чтобы проводить тестирование ядра и поиск состояний гонок с помощью рандомизированного тестирования, однако на реальных системах оно может достигать и больших значений.

В следующем поле структуры хранится собственное время потока, совершившего доступ к памяти на момент этого доступа. Ограничение в 42 бита означает, что максимальное значение собственного времени, которое поддерживает детектор равно 2^{42}. Это значение никогда не достигалось при тестировании, однако на практике это возможно. На данный момент детектор необходимо перезапустить, если это произошло. Возможно реализовать специальные процедуры, которые будут обрабатывать это переполнение.

При каждом обращении к ячейке памяти, соответствующие ячейки теневой памяти должным образом обновляются. Однако это не означает, что в четырех ячейках теневой памяти хранятся четыре самые последние обращения к ячейке памяти ядра. Процедура обновления содержимого ячеек теневой памяти описана ниже. Размер ячейки в 8 байт позволяет эффективно осуществлять действия с ячейкой теневой памяти с помощью одной операции записи или чтения.

4.1.2 Инструментация

KernelThreadSanitizer использует компляторную инструментацию, реализованную как проход компилятора GCC [11]. Компиляторный модуль инструментирует обращения к памяти и входы и выходы из функций. Инструментация обращений к памяти необходима непосредственно для поиска состояний гонок, а инструментация функций необходима для печати содержательного сообщения об ошибке. Пример инструментации приведен на листинге 7.

```
1 void foo(int *p) {
2   __tsan_func_entry(__builtin_return_address(0));
3
4   __tsan_write4(p);
5   *p = 42;
6
7   __tsan_func_exit();
8 }
```

Листинг 7: Пример компиляторной инструментации в детекторе KernelThreadSanitizer

4.1.3 Аннотации примитивов синхронизации

В ядре Linux используется немало различных примитивов синхронизации:

- мьютексы (mutex, rwsem, percpu_rwsem),
- спинлоки (spinlock, rwlock),
- семафоры (sema, completion),
- атомарные обращения к памяти и барьеры доступов к памяти,
- и т.д.

Для того, чтобы детектор мог отслеживать их использование, в код ядра были добавлены так называемые *аннотации* (англ. *annotation*). Аннотации представляют собой вызовы функций детектора, которые никак не влияют на работу самих примитивов синхронизации. Важно отметить, что аннотации для примитивов синхронизации были добавлены разработчиками детектора, и пользователю их добавлять

не требуется до тех пор, пока он не использует в тестируемом коде собственноручно реализованные низкоуровневые примитивы синхронизации.

Примитивов синхронизации в ядре очень много и их все необходимо аннотировать. Отсутствие аннотаций или неправильные аннотации хотя бы для одного из примитивов будут приводить к ложным срабатываниям детектора.

Функции, вызов которых добавлен в виде аннотаций в реализацию примитивов синхронизации, осуществляют операции над векторными часами потока и векторными часами примитива синхронизации. Трактовка событий, связанных со спинлоками, мьютексами и семафорами не вызывает вопросов и соответствует приведенной при описании алгоритма Happens-before. Пример аннотаций, добавленных в функцию взятия и освобождения мьютекса в ядре, можно увидеть на листинге 8.

Можно заметить, что для аннотирования мьютекса было добавлено сразу два вызова функций детектора: до осуществления блокировки и после. Функция, вызываемая после блокировки, совершает операцию над векторными часами потока и мьютекса. Наличие вызова функции до блокировки необходимо для того, чтобы отключить обработку событий использования примитивов синхронизации текущим потоком на время блокировки мьютекса. Это связано с тем, что внутренняя реализация мьютекса использует несколько более низкоуровневых примитивов синхронизации. Их обработка не привнесла бы никаких негативных эффектов с точки зрения воздействия на векторные часы, но она была отключена с целью экономии памяти.

```
1  void __sched mutex_lock(struct mutex *lock)
2  {
3    might_sleep();
4    ktsan_mtx_pre_lock(lock, true, false); // Annotation.
5    __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
6    mutex_set_owner(lock);
7    ktsan_mtx_post_lock(lock, true, false, true); // Annotation.
8  }
9
10 void __sched mutex_unlock(struct mutex *lock)
11 {
12   ktsan_mtx_pre_unlock(lock, true); // Annotation.
13   __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
14   ktsan_mtx_post_unlock(lock, true); // Annotation.
15 }
```

Листинг 8: Аннотации для мьютекса в детекторе KernelThreadSanitizer

События, связанные с *атомарными* (англ. *atomic*) доступами к памяти трактуются следующим образом. Каждая атомарная переменная является примитивом синхронизации. Осуществление атомарной операции с *семантикой* (англ. *memory order*) release соответствует уведомлению примитива, осуществление атомарной операции с семантикой acquire – ожиданию примитива, а осуществление операции с семантикой release-acquire – ожиданию и уведомлению примитива. В функции и макросы в ядре, выполняющие атомарные операции с семантиками release, acquire и release-acquire, были добавлены соответствующие аннотации.

Особую сложность вызывает аннотирование *барьеров доступов к памяти* (англ. *memory barriers*). Дело в том, что сам по себе барьер доступа к памяти не осуществляет никакой синхронизации. Однако она появляется при использовании барьеров вместе с атомарными операциями, поскольку барьеры изменяют их семантику. Например последовательное использование *барьера записи* (англ. *write barrier*) и атомарной операции записи с семантикой relaxed эквивалентно осуществлению атомарной операции записи с семантикой release. Аналогично, последовательное использование атомарной операции чтения с семантикой relaxed и *барьера чтения* (англ. *read barrier*) эквивалентно осуществлению атомарной операции чтения с семантикой acquire. Более того, один барьер может изменить семантику сразу нескольких предшествующих или последующих атомарных обращений к памяти.

Для того, чтобы обрабатывать синхронизацию, осуществляемую с помощью барьеров и атомарных операций, был предложен следующий метод. Его основная идея состоит в том, чтобы воздействие на векторные часы потока и примитива синхронизации при последовательном использовании атомарной операции и барьера доступа к памяти было таким же, как и при использовании атомарной операции с семантикой release или acquire. Для этого кроме обычных векторных часов каждому потоку соответствует еще пара векторных часов: release векторные часы и acquire векторные часы. Процедура обновления этих дополнительных векторных часов показана в листингах 9 и 10. Семантикой барьера доступа к памяти считается release, если это барьер записи, и acquire, если это барьер чтения. Под операцией $Acquire$ для векторных часов понимается операция уведомления, то есть $VC_1.Acquire(VC_2)$ означает уведомление векторных часов VC_1 векторными часами VC_2.

```
function OnMemoryBarrier(thread, memoryOrder) {
  if (memoryOrder == Acquire || memoryOrder == AcquireRelease) {
    thread.vectorClock.Acquire(thread.acquireVectorClock);
  }

  DoMemoryBarrier(memoryOrder);

  if (memoryOrder == Release || memoryOrder == AcquireRelease) {
    thread.releaseVectorClock.Acquire(thread.vectorClock);
  }
}
```

Листинг 9: Аннотации барьеров доступов к памяти в детекторе KernelThreadSanitizer

```
function BeforeAtomicAccess(thread, atomic, memoryOrder, isWrite) {
  if (memoryOrder == Release || memoryOrder == AcquireRelease) {
    atomic.vectorClock.Acquire(thread.vectorClock);
  } else if (isWrite) {
    atomic.vectorClock.Acquire(thread.releaseVectorClock);
  }
}

function AfterAtomicAccess(thread, atomic, memoryOrder, isWrite) {
  if (memoryOrder == Acquire || memoryOrder == AcquireRelease) {
    thread.vectorClock.Acquire(atomic.vectorClock);
  } else if (!isWrite) {
    thread.acquireVectorClock.Acquire(atomic.vectorClock);
  }
}
```

Листинг 10: Аннотации атомарных операций в детекторе KernelThreadSanitizer

Для того, чтобы подробнее понять, каким образом работают данные аннотации рассмотрим пример. Пусть поток последовательно исполняет барьер записи и атомарную запись с семантикой relaxed. Данная последовательность действий эквивалентна атомарной операции записи с семантикой release. В соответствие с используемыми аннотациями, во время барьера записи, который является барьером с семантикой release, обычные векторные часы потока уведомят release векторные часы потока. Далее, во время атомарной операции записи, release векторные часы потока уведомят векторные часы примива синхронизации. Данная последовательность уведомлений эквивалентна уведомлению векторных часов примива синхронизации векторными часами потока, что и происходит при атомарной операции записи

с семантикой release. Аналогично можно рассмотреть пример, в котором поток последовательно совершает атомарную операцию чтения с семантикой relaxed и барьер чтения.

Данная реализация аннотаций позволят корректно обработать атомарные операции и барьеры доступов к памяти и не допустить ложных срабатываний. Однако она имеет небольшой недостаток: она может приводить к лишней, паразитной синхронизации с точки зрения детектора. Например, это происходит, когда поток после барьера записи спустя какое-то время совершает не связанную логически с барьером атомарную операцию записи с семантикой relaxed. Для того, чтобы уменьшить негативный эффект от подобной паразитной синхронизации, была применена следующая эвристика: спустя несколько (на данный момент 3) входов и выходов из функций после барьера записи, содержимое release векторных часов потока очищается. Аналогичная операция осуществляется для acquire векторных часов потока.

В ядре Linux присутствует большое количество функций для работы с атомарными переменными. Семантики атомарных доступов к памяти выполняемых некоторыми из этих функций незадокументированы ни в коде ядра, ни в сопутствующей документации. Не смотря на это проводятся попытки формально описать модель памяти ядра Linux [32], в соответствие с которыми и были реализованы аннотации для атомарных переменных.

4.1.4 Обработка обращения к памяти

Перед каждым обращенем к памяти детектор осуществляет проверку на возможность состояния гонки между этим обращением и одним из предыдущих обращений к той же ячейки памяти. Для этого сначала детектор проверяет, пересекаются ли участки памяти в пределах 8-байтовой ячейки, к которой произошли обращения. Если это условие выполняется, то дальнейшие проверки производятся в соответствие с определением состояния гонки. Осуществляется проверка того, что обращения произошли в разных потоках, что хотя бы одно из обращений является операцией записи и что обращения произошли без синхронизации. Проверка на наличие синхронизации делается с помощью векторных часов. В случае, если состояние гонки возможно по определению, то печатается сообщение об ошибке. Псевдокод, иллюстрирующий алгоритм обработки обращения к памяти в детекторе KernelThreadSanitizer показан

на листинге 11.

```
function OnMemoryAccess(access) {
  shadow = AddrToShadow(access.addr);
  for (previousAccess in shadow) {
    if (accessesIntersect(access, previousAccess) &&
        (access.threadId != previousAccess.threadId) &&
        access.isWrite || previousAccess.isWrite &&
        (!access.isAtomic || !previousAccess.isAtomic) &&
        !HappensBefore(previousAccess, access)) {
      ReportDataRace(access, previousAccess);
    }
  }
  UpdateShadow(shadow, access);
}

function HappensBefore(previousAccess, access) {
  oldThread = GetThread(previousAccess.threadId);
  return previousAccess.clock <=
     oldThread.vectorClock[access.threadId);
}
```

Листинг 11: Обработка обращения к памяти в детекторе KernelThreadSanitizer

Следует отметить, что состояние гонки будет найдено не только в том случае, если оно привело к каким-либо реальных ошибкам во время исполнения. Оно будет найдено, если оно потенциально возможно при наблюдаемой последовательности обращений к памяти и использования примитивов синхронизации, которые осуществляют потоки.

Проверка на возможность состояния гонки с текущим доступом к памяти осуществляется не для всех обращений, которые произошли к этой ячейке за все время работы ядра до текущего момента, а только для тех, чье описание хранится в соответствующих ячейках теневой памяти. При очередном обращении к памяти содержимое теневой памяти необходимо обновить. Процедура обновления содержимого ячеек теневой памяти показана на листинге 12. Идея алгоритма обновления состоит в том, чтобы пытаться перезаписывать те ячейки, которые описывают более слабые с точки зрения алгоритма поиска состояний гонок доступы к памяти. В случае, если такая ячейка не найдена, то переписывается одна из случайных ячеек.

```
function UpdateShadow(shadow, access) {
  for (slot in shadow) {
    if (slot.IsEmpty()) {
      slot = shadow;
      return;
    }
    previousAccess = slot;
    stored = false;
    if (access.offset == previousAccess.offset &&
        offset.size == previousAccess.size) {
      if (access.threadId == previousAccess.threadId ||
          HappensBefore(previousAccess, access)) {
        if (AccessWeakerOrEqual(previousAccess, access)) {
          slot = shadow;
          stored = true;
        }
      }
      break;
    }
  }
  if (!stored)
    shadow[Random() % 4] = access;
}

function AccessWeakerOrEqual(first, second) {
  return ((first.isAtomic << 1) + (first.isWrite ^ 1)) >=
     ((second.isAtomic << 1) + (second.isWrite ^ 1))
}
```

Листинг 12: Обновление ячеек теневой памяти в детекторе KernelThreadSanitizer

4.1.5 Поиск ошибок в планировщике

По сути, ядро представляет собой код, который исполняется на ядрах процессора. В ядре существует понятие *задания* (англ. *task*), определенное в его исходном коде, которое представляет собой поток исполнения внутри ядра. Часть ядра, которая называется *планировщиком* (англ. *scheduler*), осуществляет периодическое переключение исполнения кода на каждом ядре процессора на одно из ожидающих исполнения заданий.

В обычном режиме, детектор KernelThreadSanitizer, полагает, что задание – это и есть поток с точки зрения своего алгоритма. То есть состояния гонок ищутся между заданиями в ядре. При переключении заданий планировщик использует различ-

ные примитивы синхронизации, которые детектору KernelThreadSanitizer приходится игнорировать. В противном случае задания, которые последовательно запускаются планировщиком на одном ядре, окажутся синхронизированными. Кроме того, детектор игнорирует и обращения к памяти, которые происходят внутри планировщика. Игнорирование этих событий объясняется еще и тем, что неясно к какому потоку эти события относить, ведь они происходят между заданиями при их переключении. При таком подходе поиск состояний гонок в планировщике становится невозможным.

Тем не менее, возможен и другой подход, который позволяет искать состояния гонок в планировщике. Этот подход был реализован как отдельный режим в детекторе KernelThreadSanitizer и называется per-cpu режим. Основная его идея состоит в том, что детектор с точки зрения своего алгоритма считает потоком не задание, а ядро процессора. В результате с точки зрения детектора в ядре Linux всегда исполняется фиксированное количество потоков, которое равно количеству ядер процессора.

При таком подходе становится неважно, что планировщик заданий является частью ядра. Сами понятия планировщика и задания являются деталями реализации тестируемого кода и не влияют особенным образом на процесс обработки событий синхронизации и обращений к памяти.

Существенным преимуществом per-cpu режима является возможность поиска ошибок в планировщике. Недостатком является меньшая вероятность найти состояние гонки в остальном коде. Это связано с тем, что для того, чтобы состояние гонки было найдено, оно должно произойти между ядрами, а не между заданиями, которые могли быть запущены планировщиком на одном ядре.

4.1.6 Отчеты об ошибках

Одним из важных преимуществ разработанного детектора являются подробные сообщения об ошибках. На листингах 13 и 14 приведен отчет об ошибке, который печатает детектор KernelThreadSanitizer при обнаружении состояния гонки. В отчете приводится много информации, которая может помочь найти и исправить ошибку.

В заголовке сообщения об ошибке приводятся стеки вызовов для обращений к памяти, которые находятся в состоянии гонки. Кроме адресов внутри бинарного файла ядра и имен функций, в стеке вызовов также показаны названия файлов с исходным

кодом и номера строк в этих файлах. Более того, в стеке вызовов показываются и *встроенные* (англ. *inline*) вызовы.

В следующей части отчета перечислены мьютексы, взятые каждым из потоков, которые находятся в состоянии гонки. Для каждого из мьютексов приводится стек вызовов на момент взятия этого мьютекса.

```
1  ThreadSanitizer: data-race in ipc_obtain_object_check
2
3  Read at 0xffff88047f810f68 of size 8 by thread 2749 on CPU 5:
4   [<ffffffff8147d84d>] ipc_obtain_object_check+0x7d/0xd0
       ipc/util.c:621
5   [<     inline     >] msq_obtain_object_check ipc/msg.c:90
6   [<ffffffff8147e708>] msgctl_nolock.constprop.9+0x208/0x430
       ipc/msg.c:480
7   [<     inline     >] SYSC_msgctl ipc/msg.c:538
8   [<ffffffff8147f061>] SyS_msgctl+0xa1/0xb0 ipc/msg.c:522
9   [<ffffffff81ee3e11>] entry_SYSCALL_64_fastpath+0x31/0x95
       arch/x86/entry/entry_64.S:188
10
11 Previous write at 0xffff88047f810f68 of size 8 by thread 2755 on
      CPU 4:
12  [<ffffffff8147cf97>] ipc_addid+0x217/0x260 ipc/util.c:257
13  [<ffffffff8147eb4c>] newque+0xac/0x240 ipc/msg.c:141
14  [<     inline     >] ipcget_public ipc/util.c:355
15  [<ffffffff8147daa2>] ipcget+0x202/0x280 ipc/util.c:646
16  [<     inline     >] SYSC_msgget ipc/msg.c:255
17  [<ffffffff8147efaa>] SyS_msgget+0x7a/0x90 ipc/msg.c:241
18  [<ffffffff81ee3e11>] entry_SYSCALL_64_fastpath+0x31/0x95
       arch/x86/entry/entry_64.S:188
```

Листинг 13: Отчет детектора KernelThreadSanitizer: заголовок сообщения об ошибке и стек вызовов для обращений к памяти

```
Mutexes locked by thread 2755:
Mutex 445417 is locked here:
 [<ffffffff81ee0d45>] down_write+0x65/0x80
    kernel/locking/rwsem.c:62
 [<     inline     >] ipcget_public ipc/util.c:348
 [<ffffffff8147d90c>] ipcget+0x6c/0x280 ipc/util.c:646
 [<     inline     >] SYSC_msgget ipc/msg.c:255
 [<ffffffff8147efaa>] SyS_msgget+0x7a/0x90 ipc/msg.c:241
 [<ffffffff81ee3e11>] entry_SYSCALL_64_fastpath+0x31/0x95
    arch/x86/entry/entry_64.S:188

Mutex 453634 is locked here:
 [<     inline     >] __raw_spin_lock
    include/linux/spinlock_api_smp.h:158
 [<ffffffff81ee37d0>] _raw_spin_lock+0x50/0x70
    kernel/locking/spinlock.c:151
 [<     inline     >] spin_lock include/linux/spinlock.h:312
 [<ffffffff8147ce0e>] ipc_addid+0x8e/0x260 ipc/util.c:238
 [<ffffffff8147eb4c>] newque+0xac/0x240 ipc/msg.c:141
 [<     inline     >] ipcget_public ipc/util.c:355
 [<ffffffff8147daa2>] ipcget+0x202/0x280 ipc/util.c:646
 [<     inline     >] SYSC_msgget ipc/msg.c:255
 [<ffffffff8147efaa>] SyS_msgget+0x7a/0x90 ipc/msg.c:241
 [<ffffffff81ee3e11>] entry_SYSCALL_64_fastpath+0x31/0x95
    arch/x86/entry/entry_64.S:188
```

Листинг 14: Отчет детектора KernelThreadSanitizer: описание захваченных мьютексов

4.2 Производительность

При тестировании детектора учитывались потребление памяти и замедление производительности ядра. Максимальный расход памяти детектором достигает 16 ГБ в обычном режиме и 1 ГБ в per-cpu режиме. Производительность ядра засчет работы детектора замедляется в среднем в 6 раз при работе детектора в обычном режиме и в 2 раза в per-cpu режиме.

Большой расход памяти детектором в обычном режиме, связан с большим количеством примитивов синхронизации и большим потенциальным количеством потоков, которые создает ядро. При нормальной работе ядра количество примитивов синхронизации может достигать 10^6, а количество одновременно существующих потоков на определенных этапах загрузки близко к 10^3. Чем больше число примитивов син-

Таблица 1: Сравнение детекторов KernelStrider, RaceHound и KernelThreadSanitizer

	KernelStrider	RaceHound	KTSAN
Обработка высокоуровневой синхронизации	Да	Да	Да
Обработка низкоуровневой синхронизации	Нет	Да	Да
Отсутствие ложных срабатываний	Нет	Да	Да
Отсутствие пропусков ошибок	Нет	Нет	Нет
Поиск ошибок во всем ядре	Нет	Нет	Да
Поиск ошибок в планировщике ядра	Нет	Нет	Да

хронизации и потоков, тем больше памяти расходуется на векторные часы. Напомню, что размер векторных часов равен числу потоков, а их количество пропорционально количеству примитивов синхронизации. В per-cpu режиме, количество потоков снижается на порядки и размер памяти, необходимой для хранения векторных часов существенно уменьшается.

В таблице 1 показано сравнение детекторов KernelStrider, RaceHound и разработанного детектора KernelThreadSanitizer. Как видно из таблицы, преимуществами детектора KernelThreadSanitizer является возможность поиска состояний гонок во всем коде ядра включая планировщик и отсутствие ложных срабатываний.

4.3 Найденные ошибки

Тестирование ядра с помощью динамических детекторов осложняется отсутствием набора модульных тестов с достаточно высоким покрытием кода[1]. В результате, приходится использовать рандомизированное тестирование.

Trinity [33] – это одна из утилит, применяемая для рандомизированного тестирования ядра Linux. Принцип ее работы заключается в вызове системных вызовов со случайными аргументами. Однако, передавая в системные вызовы абсолютно случайные аргументы, их выполнение часто не проходит дальше, чем проверка этих аргументов на корректность. Trinity поступает умнее и передает не совсем случайные аргументы. Например, при запуске она генерирует список всех доступных файловых

[1] «"Regression testing"? What's that? If it compiles, it is good; if it boots up, it is perfect.», Linus Torvalds, Linux Kernel Mailing List, 1998-04-08.

дескрипторов и передает один из них в качестве аргумента тем системным вызовам, соответствующим аргументом которых должен быть именно файловый дескриптор.

В процессе тестирования ядра с применением детектора KernelThreadSanitizer и утилиты Trinity детектор нашел большое количество состояний гонок. К сожалению, большая их часть хоть и являются состояниями гонок по определению, но не приводят к существенным негативным последствиям, и разработчики ядра не заинтересованы в их исправлении. Примером такого состояния гонки является несинхронизированное обращение из нескольких потоков к переменной, которая является счетчиком какой-либо незначительной статистики.

Тем не менее, в процессе тестирования было найдено 23 настоящих ошибки: в основном обращения из нескольких потоков к разделяемой переменной, а также несколько пропущенных барьеров доступов к памяти. Часть ошибок были подтверждены (14 ошибок) и исправлены (10 ошибок) разработчиками ядра.

Одна из найденных ошибок оказалась серьезной уязвимостью подсистемы ядра IPC [34]. С помощью этой уязвимости возможно добиться повышения привилегий в системе. Уязвимы оказались несколько современных дистрибутивов на базе Linux включая Ubuntu 12.04 LTS [35].

5 Заключение

Для автоматического поиска ошибок в ядре Linux был разработан детектор KernelThreadSanitizer. Основными преимуществами нового детектора над существующими является поддержка обработки низкоуровневых примитивов синхронизации, возможность поиска состояний гонок во всем ядре и качество отчетов о найденных ошибках. В результате тестирования ядра Linux с применением разработанного детектора, были найдены более 20 ранее неизвестных дефектов в ядре Linux, часть которых была впоследствии исправлена разработчиками ядра.

- Разработан новый метод поиска состояний гонок для ядра Linux, применимый для всего кода ядра. Разработанный метод не обладает ложными срабатываниями.

- На основе разработанного метода реализован детектор состояний гонок для

ядра KernelThreadSanitizer. Исходный код детектора открыт и доступен для использования.

- С помощью разработанного детектора найдено более 20 состояний гонок в ядре Linux, часть которых подтверждена и исправлена разработчиками ядра.
- Детектор KernelThreadSanitizer внедрен в процесс тестирования ядра Linux в компании Google.
- Результаты представлены на 57-й научной конференции МФТИ [36] и на конференции LinuxCon North America 2015 [37].

Список литературы

[1] Leveson Nancy G, Turner Clark S. An investigation of the therac-25 accidents // Computer. — 1993. — V. 26, no. 7. — P. 18–41.

[2] Linux. — http://ru.wikipedia.org/wiki/Linux.

[3] ХОРОШИЛОВ АВ, МУТИЛИН ВС, НОВИКОВ ЕМ. Анализ типовых ошибок в драйверах операционной системы linux // Труды Института системного программирования РАН. — 2012. — V. 22.

[4] Ernst Michael D. Static and dynamic analysis: Synergy and duality // WODA 2003: ICSE Workshop on Dynamic Analysis / Citeseer. — 2003. — P. 24–27.

[5] Исходжанов Т.Р. Автоматический поиск ошибок в компьютерных программах с применением динамического анализа: дис. ... канд. физ.-мат. наук / Т.Р. Исходжанов. — МФТИ. — 2013.

[6] ThreadSanitizer. — https://github.com/google/sanitizers/wiki.

[7] Helgrind: a thread error detector. — http://valgrind.org/docs/manual/hg-manual.html.

[8] 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 / ACM. — 2006. — P. 69–78.

[9] Iso/iec 14882: 2011, standard for programming language c++: Rep. / Technical report, 2011. http://www.open-std.org/jtc1/sc22/wg21. — Executor: C++ Standards Committee et al.: 2011.

[10] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. — http://cwe.mitre.org/top25/.

[11] GCC, the GNU Compiler Collection. — http://gcc.gnu.org/.

[12] GCC Bugzilla — Bug 33498 — Optimizer (-O2) may convert a normal loop to infinite. — http://gcc.gnu.org/bugzilla/show_bug.cgi?id=33498.

[13] Eraser: A dynamic data race detector for multithreaded programs / Stefan Savage, Michael Burrows, Greg Nelson et al. // ACM Transactions on Computer Systems (TOCS). — 1997. — V. 15, no. 4. — P. 391–411.

[14] Lamport Leslie. Time, clocks, and the ordering of events in a distributed system // Communications of the ACM. — 1978. — V. 21, no. 7. — P. 558–565.

[15] Fuzz testing. — http://ru.wikipedia.org/wiki/Fuzz_testing.

[16] AddressSanitizer: a fast address sanity checker / Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov // Proceedings of the 2012 USENIX conference on Annual Technical Conference. — USENIX ATC'12. — Berkeley, CA, USA, 2012. — P. 28–28.

[17] Seward Julian, Nethercote Nicholas. Using Valgrind to detect undefined value errors with bit-precision // USENIX Annual Technical Conference. — 2005. — P. 17–30.

[18] Nethercote Nicholas, Seward Julian. Valgrind: A framework for heavyweight dynamic binary instrumentation // Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. — PLDI '07. — New York, NY, USA: ACM, 2007. — P. 89–100.

[19] Pin: building customized program analysis tools with dynamic instrumentation / Chi-Keung Luk, Robert Cohn, Robert Muth et al. // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. — PLDI '05. — New York, NY, USA: ACM, 2005. — P. 190–200.

[20] Bruening Derek. Efficient, Transparent, and Comprehensive Runtime Code Manipulation: Ph. D. thesis / Derek Bruening. — M.I.T. — 2004.

[21] gcov — a Test Coverage Program. — http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[22] Eigler Frank Ch. Mudflap: pointer use checking for C/C++ // GCC Developers Summit / Red Hat Inc. — 2003.

[23] Pebil: Efficient static binary instrumentation for linux / Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, Allan Snavely // Performance Analysis of

Systems & Software (ISPASS), 2010 IEEE International Symposium on / IEEE. — 2010. — P. 175–183.

[24] Bungale Prashanth P, Luk Chi-Keung. PinOS: a programmable framework for whole-system dynamic instrumentation // Proceedings of the 3rd international conference on Virtual execution environments / ACM. — 2007. — P. 137–147.

[25] Kprobes. — https://sourceware.org/systemtap/kprobes/.

[26] Tamches Ariel, Miller Barton P. Fine-grained dynamic instrumentation of commodity operating system kernels: Ph. D. thesis / Ariel Tamches, Barton P Miller. — University of Wisconsin–Madison. — 2001.

[27] ftrace - Function Tracer. — https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[28] Komarov Nikita. On the implementation of data-breakpoints based race detection for linux kernel modules // Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering / Федеральное государственное бюджетное учреждение науки Институт системного программирования Российской академии наук. — No. 7. — 2013.

[29] KernelStrider. — https://github.com/euspectre/kernel-strider.

[30] Андрианов П.С., Мутилин В.С., Хорошилов А.В. Метод легковесного статического анализа для поиска состояний гонок // Труды Института системного программирования РАН. — 2015. — V. 27, no. 5.

[31] Vadim Mutilin, Alexey Khoroshilov. An approach to lightweight static data race detection // Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering / Федеральное государственное бюджетное учреждение науки Институт системного программирования Российской академии наук. — No. 8. — 2014.

[32] Linux-Kernel Memory Model. — http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4444.html.

[33] Trinity: A Linux System call fuzz tester. — http://codemonkey.org.uk/projects/trinity/.

[34] CVE-2015-7613. — https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7613.

[35] Linux Kernel 'ipc_addid()' Function Local Memory Corruption Vulnerability. — http://www.securityfocus.com/bid/76977.

[36] Коновалов А.Д., Вьюков Д.С. Автоматический поиск состояний гонок в ядре ОС Linux // Труды 57-й научной конференции МФТИ. Управление и прикладная математика. Т. 2 / МФТИ. — 2014. — С. 149–150.

[37] KernelAddressSanitizer (KASan): a fast memory error detector for the Linux kernel. — https://events.linuxfoundation.org/sites/events/files/slides/LinuxCon%20North%20America%202015%20KernelAddressSanitizer.pdf.