DOI: 10.15514/ISPRAS-2019-1(2)-1

Автоматическое обнаружение гонок при параллельной сборке с использованием утилиты Make

[1,2]А.Ю. Климов, ORCID: 0009-0004-3719-5873, <klimov.aiu@ispras.ru>
[1]В.А. Иванишин, ORCID: 0000-0002-9784-2911, <vlad@ispras.ru>
[1]А.В. Монаков, ORCID: 0000-0001-5118-0054, <amonakov@ispras.ru>
[1]Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2]Московский физико-технический институт, 141701, Россия, Московская Обл., Долгопрудный, Институтский пер., 9.

Аннотация. В этой работе представлен метод и программная реализация автоматического обнаружения потенциальных состояний гонки при сборке программных проектов, использующих систему сборки Make. Представлены результаты применения этого метода на открытых проектах и проведено сравнение с существующим решением для обнаружения состояний гонки.

Ключевые слова: состояние гонки; отладка; многопоточная сборка; make

Для цитирования: Климов А.Ю., Иванишин В.А., Монаков А.В. Разработка инструмента автоматического обнаружения гонок при параллельной сборке с использованием утилиты Make. Труды ИСП РАН, том 1, вып. 2, 2019 г., стр. 15-19. DOI: 10.15514/ISPRAS–2019–1(2)-1

Automated Race Condition Detection in Parallel Make-Based Builds

[1,2]A.Yu. Klimov, ORCID: 0009-0004-3719-5873, <klimov.aiu@ispras.ru>
[1]V.A. Ivanishin, ORCID: 0000-0002-9784-2911, <vlad@ispras.ru>
[1]A.V. Monakov, ORCID: 0000-0001-5118-0054, <amonakov@ispras.ru>
[1]Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2]Moscow Institute of Physics and Technology, 141701, Russia, Moscow region, Dolgoprudny, Institutskiy lane, 9.

Abstract. This research paper introduces a novel approach and software implementation for automatic identification of potential race conditions in Make-based build systems. The method is applied to various open-source projects, and the developed tool is compared against an existing race condition detection method. The proposed technique has been shown to be effective in finding race conditions.

Keywords: race condition; debugging; parallel build; make

For citation: Klimov A.Yu. Ivanishin V.A., Monakov A.V. Development of an Automated Race Detection Tool for Parallel Make-Based Builds. Trudy ISP RAN/Proc. ISP RAS, vol. 1, issue 2, 2019. pp. 15-19 (in Russian). DOI: 10.15514/ISPRAS-2019-1(2)-1

1. Глоссарий

- Санитайзер — инструмент для обнаружения проблем и уязвимостей в коде программ;

- Makefile — файл, задающий схему сборки проекта или модуля;
- Цель сборки — элемент схемы сборки, имеющий свой рецепт и список пререквизитов;
- Пререквизит — файл или цель, от которой зависит другая цель;
- Index node (inode) — структура данных, владеющая содержимым файла в файловой системе. Номер inode (или серийный номер файла, см. опр. 3.176 в стандарте POSIX [1]) — уникальный идентификатор файла в файловой системе;
- Directory entry — опр. 3.130 в стандарте POSIX [1] — структура данных, соотносящая путь в файловой системе с inode;
- Glob-выражение — шаблон для выбора нескольких имен файлов или каталогов, например, `*.txt`;
- Регулярный язык — множество слов, принимаемых некоторым регулярным выражением;
- МПДКА — минимальный полный детерминированный конечный автомат. Структура данных, позволяющая определить, относится ли слово к некоторому регулярному языку за время $O(n)$, где n — длина слова.

2. Введение

Состояние гонки — это ситуация, при которой поведение программы зависит от относительного порядка выполнения двух или более параллельных операций, и может меняться в зависимости от последовательности их выполнения. Это приводит к непредсказуемому поведению программы, и обусловлено, как правило, отсутствием синхронизации между потоками.

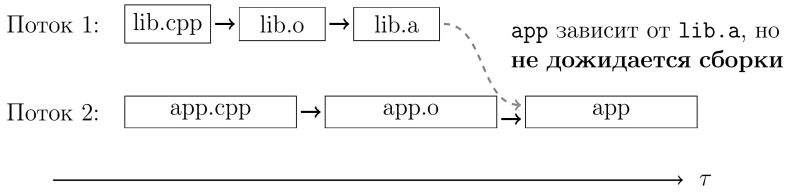

Рис. 1. Процесс сборки проекта с состоянием гонки в схеме сборки
Fig. 1. Building a project with a race condition in the build scheme

При рассмотрении проблематики состояний гонки в основном фокусируются на языках программирования прикладного уровня, таких как C++ или Java. Однако, такие проблемы также могут возникать в процессе сборки программного обеспечения, где примитивами синхронизации выступают зависимости между целями сборки. Отсутствие необходимой зависимости может привести к состоянию гонки, аналогично отсутствующей синхронизации между процессами.

Выше изображен процесс сборки проекта. В нём исходный код приложения может собираться параллельно с библиотекой, которую он использует. Это позволяет ускорить сборку всего проекта. Однако в этой схеме не указано, что перед компоновкой всего приложения необходимо дождаться, пока библиотека будет готова.

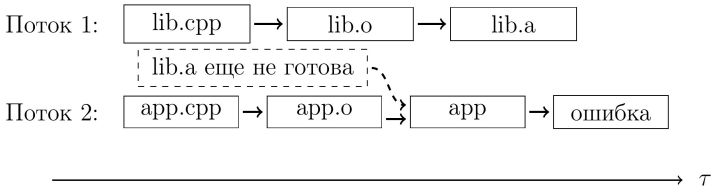

Рис. 2. Ошибка сборки, вызванная отсутствием зависимости

Fig. 2. Build error caused by a missing dependency

Такие ошибки можно исправить тремя способами:

1) Найти и добавить недостающую зависимость;
2) Перезапустить сборку, если ошибка редко воспроизводится;
3) Отключить многопоточность (указать `-j1`). Make собирает пререквизиты в таком же порядке, в котором они указаны в Makefile, поэтому отключение многопоточности исключит вероятность возникновения гонок. В некоторых проектах такое решение из временного становится постоянным, например, как в проекте `netpbm`. Для больших проектов такое решение неприменимо, поскольку в однопоточном режиме сборка может занимать в несколько раз больше времени.

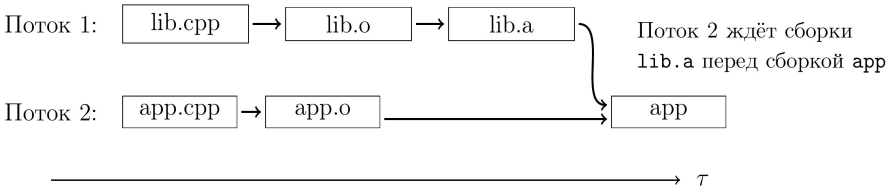

Рис. 3. Исправленная схема сборки без состояния гонки
Fig. 3. Fixed build scheme without a race condition

Настоящие схемы сборки, как правило, выглядят значительно сложнее, и найти в них недостающую зависимость становится трудно. В связи с этим, такие проблемы в проектах могут долго оставаться неисправленными. Подтверждение этому можно найти на форуме Gentoo, где перечислены открытые обсуждения, связанные с ошибками при параллельной сборке пакетов для этой системы [2].

Опасность этих гонок заключается в том, что оставаясь скрытыми, они могут проявляться непредсказуемым образом. Часто при наличии такой проблемы в схеме возникает спонтанная ошибка при сборке, которая исчезает при повторной попытке собрать проект. Существует и более опасный сценарий, при котором такая ошибка может приводить к скрытым проблемам. Например, к некорректно собранным файлам локализации или к уязвимости в распространяемом исполняемом файле.

3. Постановка задачи

Ручное исправление состояний гонок в схемах сборки является трудным процессом. Целью этой работы является предоставление решения, которое упростит этот процесс. Для достижения этой цели предлагается разработать автоматический инструмент, аналогичный Thread Sanitizer для параллельных сборок. Инструмент должен соответствовать следующим требованиям:

- Инструмент должен обнаруживать как можно больше гонок, связанных с ошибками в схеме сборки.
- Алгоритм поиска состояний гонок не должен носить вероятностный характер. Последовательные запуски инструмента на одном и том же проекте должны сообщать об одних и тех же гонках.
- Инструмент должен быть легко встраиваем в существующие проекты, не должен требовать значительных изменений в проект и не должен вмешиваться в процесс сборки.
- Не должны требоваться многократные пересборки проекта или отключение многопоточности (`-j1`), не должен значительно замедляться сам процесс сборки проекта.

Основная задача, из которой следуют все вышеперечисленные пункты, состоит в сокращении времени, требуемого разработчику для поиска гонок.

4. Обзор существующих решений

Современные системы сборки предпринимают меры для борьбы с гонками. Например, система Bazel собирает каждую цель в отдельной виртуальной файловой системе, в которой есть только файлы, собранные зависимостям этой цели сборки. Такое ограничение исключает возникновение гонок: с каждой виртуальной файловой системой одновременно может работать одна цель сборки, в определённом фиксированном порядке. Однако, подобные системы пока не заместили собой стандартные, более простые утилиты, такие как Make и Ninja. Последние по-прежнему широко используются в современных проектах как непосредственно, так и в виде бекэнда для других, более высокоуровневых систем.

Для сборок на основе Make в настоящее время существует единственное решение поставленной проблемы — флаг `--shuffle`, добавленный в GNU Make 4.4 в 2022 году. Принцип его работы заключается в случайной перестановке порядка сборки независимых целей. Такой подход увеличивает вероятность того, что существующая гонка проявится и приведёт к сбою. Полученная ошибка может помочь разработчику найти и исправить гонку.

Это решение легко встраивается в существующие проекты посредством добавления флага `--shuffle` в аргументы Make или в переменную окружения `GNUMAKEFLAGS`. Если окружение не позволяет указывать переменные окружения или параметры командной строки, можно применить патч для Make, активирующий режим `--shuffle` по умолчанию.

Однако, в основе режима Make `--shuffle` лежит случайный алгоритм. Это значит, что разработчику, вероятно, придётся полностью пересобрать проект много раз, прежде чем гонка себя проявит. Кроме этого, этим решением нельзя обнаружить гонки, которые проявляются только при параллельном выполнении целей. Распространённая причина появления таких гонок заключается в том, что несколько независимых целей могут использовать временный файл по одному и тому же пути. Это может привести к ошибке или к повреждению данных, если эти цели будут собираться одновременно. Далее в этой работе такой вид гонок будет отнесён к классу “Гонки на пути к файлу”. Случайная перестановка сборки независимых целей в режиме `--shuffle` не способствует проявлению таких гонок.

5. Исследование и построение решения задачи

Самые распространённые гонки, встречающиеся в реальных проектах, можно разделить на три категории. Далее, по ходу их рассмотрения, будут предложены алгоритмы для их автоматического обнаружения. Представленная классификация не является полной, однако, как будет показано в пункте 6.6, она покрывает большую часть известных гонок.

5.1 Гонка на содержимом файла

```
all: compile link

compile:
  gcc main.c -o main.o
  gcc lib.c -o lib.o

link:
  gcc main.o lib.o -o a.out
```

Листинг 1. Пример Makefile с гонкой на содержимом объектных файлов

Listing 1. Example Makefile with a race condition on object file contents

В этом примере между целями `compile` и `link` не хватает зависимости. Аналогично примеру из введения, при многопоточной сборке цель `link` может попытаться скомпоновать

объектные файлы, которых ещё не существует, или использовать старый, ещё не обновленный объектный файл.

Основная идея автоматического обнаружения гонок заключается в отслеживании операций с файлами и сопоставление их с графом зависимостей системы сборки. Можно увидеть, какие файлы открывают процессы, если запустить сборку под утилитой strace.

```
$ strace -f -e trace=%file make
...
[pid 1017] openat(AT_FDCWD, "main.o", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
...
[pid 1020] openat(AT_FDCWD, "lib.o", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
...
[pid 1025] openat(AT_FDCWD, "main.o", O_RDONLY) = 7
[pid 1025] openat(AT_FDCWD, "lib.o", O_RDONLY) = 8
[pid 1025] openat(AT_FDCWD, "lib.o", O_RDONLY) = 9
```

Листинг 2. Фрагмент лога strace при сборке Makefile из листинга 1

Listing 2. Fragment of the strace log when building the Makefile from listing 1

Во фрагменте полученного лога можно видеть, как процессы 1017, 1020 и 1025 открывают одни и те же объектные файлы с помощью системного вызова `openat`, причём первые два — на запись, а последний — на чтение. Однако этой информации мало: из лога нельзя понять, какие цели сборки скрываются за этими номерами.

5.1.1 Сопоставление операций над файлами с целями сборки

В этой работе вместо GNU Make будет использована её вариация — remake. В ней реализованы те же функции, что и GNU Make, но она требует значительно меньше усилий для сборки из исходного кода.

Чтобы получить информацию о том, какие процессы порождаются Make и каким целям они соответствуют, необходимо модифицировать саму утилиту Make.

```
$ strace -f -e trace=%file make
...
remake: Spawned process, ppid=1014, pid=1015, target=compile
...
[pid 1015] vfork() = 1017
...
[pid 1017] openat(AT_FDCWD, "main.o", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
...
remake: Spawned process, ppid=1014, pid=1018, target=compile
...
[pid 1018] vfork() = 1020
...
[pid 1020] openat(AT_FDCWD, "lib.o", O_WRONLY|O_CREAT|O_APPEND, 0666) = 3
...
remake: Spawned process, ppid=1014, pid=1023, target=link
...
[pid 1023] vfork() = 1025
...
[pid 1025] openat(AT_FDCWD, "main.o", O_RDONLY) = 7
[pid 1025] openat(AT_FDCWD, "lib.o", O_RDONLY) = 8
[pid 1025] openat(AT_FDCWD, "lib.o", O_RDONLY) = 9
...
```

Листинг 3. Фрагмент лога сборки Makefile из листинга 1 с модифицированным remake}

Listing 3. Fragment of the build log of the Makefile from listing 1 with modified remake

Можно заметить, что ни один процесс `gcc`, который запускается Make, не работает с файлами проекта напрямую. `gcc` — не компилятор, а драйвер, который запускает нужные

компиляторы и компоновщики. Создание `main.o` и `lib.o` ведётся дочерними процессами `gcc`. В нашем случае это процессы `as`, порождённые системным вызовом `vfork`. Они генерируют объектные файлы на основе ассемблера, в который компилируется Си с помощью `cc1` — другого дочернего процесса `gcc`.

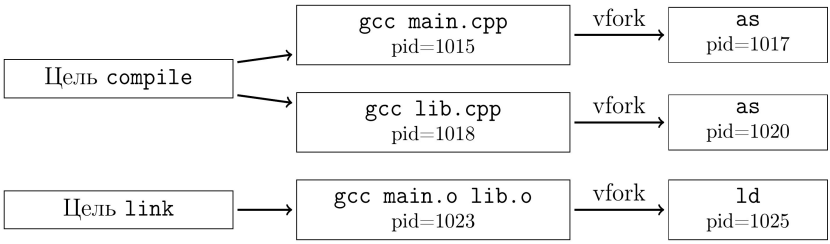

Рис. 4. Дерево процессов при сборке Makefile из листинга 1
Fig. 4. Process tree when building the Makefile from listing 1

Схему выше (кроме названий процессов) можно построить на данных из листинга 3. В ней, как и во фрагменте лога, опущены процессы `cc1`, поскольку они не производят доступов к интересующим нас объектным файлам.

Рассмотрим процессы 1020 и 1025. Согласно рис. 4, им соответствуют цели `compile` и `link`. Из фрагмента лога в листинге 3 можно установить, что процесс 1020 производит запись в файл `lib.o`, а процесс 1025 — чтение того же файла. Запись и чтение нельзя менять местами, поскольку результат чтения может поменяться. Следовательно, процессы 1020 и 1025 должны запускаться строго друг за другом. Иными словами, между соответствующими целями — `compile` и `link` — должна быть зависимость. Проверим это, обратившись к графу зависимостей схемы.

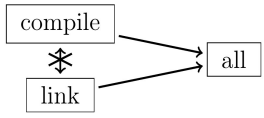

Рис. 5. Граф зависимостей Makefile из листинга 1
Fig. 5. Dependency graph of the Makefile from listing 1

Легко убедиться в том, что схема сборки из примера не содержит такой зависимости: между целями `link` и `compile` нет ориентированного пути. Соответственно, в схеме сборки присутствует гонка. Теперь можно составить первый вариант алгоритма автоматического поиска состояний подобных гонок:

1) Произвести сборку с использованием strace и модифицированного remake;
2) Получить соответствие между pid и целями сборки;
3) Получить список доступов к файлам для каждой известной цели;
4) Найти конфликтующие доступы к одному и тому же пути из разных целей;
5) Убедиться в том, что в схеме сборки существуют зависимости между целями, производящими конфликтующие доступы.

В силу простоты примера конфликтующие доступы было найти достаточно легко. Однако, в общем случае выявление конфликтующих доступов — не такая тривиальная задача. Более подробно она будет рассмотрена в пункте 5.4.

5.1.2 Мотивация использования номеров inode

В таком виде у алгоритма есть одно ограничение. Если цели `compile` и `link` будут использовать жёсткие ссылки на объектные файлы (например, `main.o.0` и `main.o.1`, гонка

останется, но в логе доступов будут фигурировать пути от разных жёстких ссылок. Алгоритм выше не обнаружит такую гонку, поскольку полагается на совпадение путей как строк. Вместо этого нужно использовать какой-то другой способ сравнения, который бы учитывал жёсткие ссылки.

Согласно стандарту POSIX [1], каждый файл или директория имеет связанный с ним серийный номер, уникально идентифицирующий его в файловой системе. Его можно узнать из поля `st_ino` структуры `stat`. Иногда этот номер называют номером index node, или, сокращённо, inode. Чтобы уникально идентифицировать файл во всей системе, нужно использовать его в паре с `st_dev` — идентификатором файловой системы (device number). В разработанном инструменте это учтено, однако для простоты далее в этой работе device number будет опускаться.

Номера inode могут быть использованы системой повторно, когда все жёсткие ссылки на файл оказываются удалены. Это может привести к тому, что разные файлы будут отражены в логе одними и теми же номерами inode, в результате чего алгоритм выдаст ложные срабатывания. Если добавить в лог события освобождения inode, скрипт для поиска гонок сможет отличать их поколения, и не выдавать ложных срабатываний при повторном использовании номера inode.

Прежде наш алгоритм полагался на разбор вывода strace. К сожалению, эта утилита не позволяет производить такие сложные проверки. Для этой цели лучше подходит `ptrace` — системный вызов для трассировки процессов, на основе которого реализован отладчик GDB, а так же сам strace. `ptrace` позволяет перехватывать управление процессом перед любыми системными вызовами, которые он совершает. Собственный трассировщик на основе `ptrace` позволит производить более сложные проверки и составлять более информативные логи, чем `strace`.

Перехватив управление процессом перед удалением файла или директории (системные вызовы `unlink(at)` или `rmdir`), трассировщик может проверить, что оно приведёт к освобождению номера inode. Linux указывает количество жёстких ссылок на файл в поле `st_nlink` структуры `stat`. Перед удалением последней жёсткой ссылки (и, соответственно, перед освобождением номера inode) `st_nlink` равняется 1 для файлов и 2 для директорий (каждая директория содержит «`.`» — жёсткую ссылку на себя). Произведя такую проверку, трассировщик сможет вывести в лог событие освобождения номера inode.

Таким образом, после всех исправлений, алгоритм приобретает следующий вид:

1) Произвести сборку с использованием модифицированного remake и трассировщика на Си, использующего `ptrace`;
2) Получить соответствие между pid и целями сборки;
3) Получить список доступов к inode для каждой известной цели;
4) Найти конфликтующие доступы к одному и тому же поколению inode из разных целей;
5) Убедиться в том, что в схеме сборки существуют зависимости между целями, производящими конфликтующие доступы к одному и тому же поколению inode.

5.2 Гонка на пути к файлу

```
all: something something_else

something:
  generate_something > tmp_file
  do_something_with tmp_file
  rm tmp_file

something_else:
  generate_something_else > tmp_file
```

```
  do_something_else_with tmp_file
  rm tmp_file
```

Листинг 4. Пример Makefile с гонкой на пути к файлу

Listing 4. Example Makefile with a race on file path

В предыдущей главе был построен алгоритм для обнаружения состояния гонок на содержимом одного и того же файла. Здесь же речь пойдёт о разных файлах, которые были доступны по одному и тому же пути в разные моменты времени. В листинге 4 представлен распространённый сценарий гонки: независимые цели `something` и `something_else` используют одно и то же имя для своих временных файлов. Если запустить сборку этих целей параллельно, то может возникнуть конфликт.

Значительная часть предыдущей главы была уделена корректной обработке жёстких ссылок, из-за которых файл может иметь несколько абсолютных путей. При работе с директориями в этом нет необходимости — целью жёстких ссылок могут быть только файлы (за исключением «.» и «..», которые не используются в абсолютных путях). Следовательно, для директорий корректно использовать их абсолютные пути в качестве уникального идентификатора. Стоит оговориться, что для этого нужно использовать разрешённый путь, то есть не содержащий переходов по символическим ссылкам. Далее под путями будут подразумеваться абсолютные разрешенные пути.

Пусть в директории по пути d существует сущность (directory entry) с именем n. Directory entry может быть каталогом или жёсткой ссылкой на inode (на файл). Поскольку путь к директории уникален, получить доступ к этой directory entry можно только через путь d/n. Не может быть такого, что файл был удалён по пути d_1/n, и перестал быть доступен по пути d_2/n, где $d_1 \neq d_2$. Таким образом, путь является уникальным идентификатором не только директорий, но и любой directory entry.

Поскольку гонка из примера связана с пересозданием directory entry по какому-либо пути, для её обнаружения достаточно проверять, что между целями, которые производят удаление (`unlink`) и повторное создание (`write`) этой directory entry, существует зависимость.

Операция удаления может образовать гонку не только с записью, но и с чтением, если оно произошло первым и использовало тот же путь, что и операция удаления. Такие гонки тоже можно отнести к этой категории. Никакие другие типы гонок не связаны с операцией удаления, поэтому разделение на категории останется корректным.

5.3 Гонка между созданием директории и файла внутри неё

```
all: build build/a.out

build:
  mkdir -p build

build/a.out:
  echo "a" > build/a.out
```

Листинг 5. Пример Makefile с гонкой на директории

Listing 5. Example Makefile with a race on directory

В листинге 5 цели `build` и `build/a.out` не зависят друг от друга. Если цель `build/a.out` начнёт собираться раньше, она не сможет создать файл в директории, которой ещё не существует. Такая гонка была обнаружена в проекте GPM с помощью `make --shuffle` [7].

Директория и файл внутри неё — разные элементы файловой системы, имеющие разные пути и разные номера inode, поэтому предыдущие алгоритмы не смогут обнаружить эту гонку. Простое решение — фиксировать доступ специального вида (directory lookup) к родительскому каталогу при любом обращении к находящемуся в нём файлу.

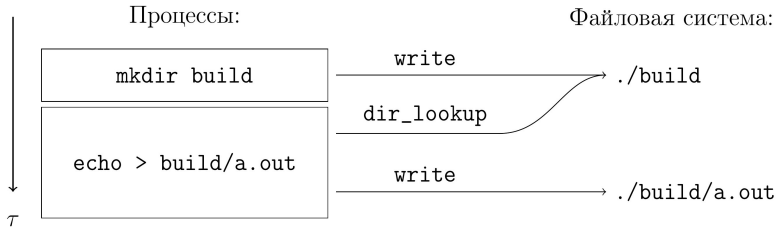

Рис. 6. Операции над файлами при сборке Makefile из листинга 5
Fig. 6. File operations performed during the build of the Makefile from listing. 5

Операция `dir_lookup` позволила связать процесс `echo` из цели `build/a.out` с процессом `mkdir` из цели `build`. Поскольку теперь они производят чтение и запись на одной и той же директории, алгоритм поиска гонок из первой главы проверит наличие зависимости между их целями. Несмотря на то, что доступ `dir_lookup` фиксируется только к ближайшему родительскому каталогу, этот принцип применим и для большего числа вложенных директорий.

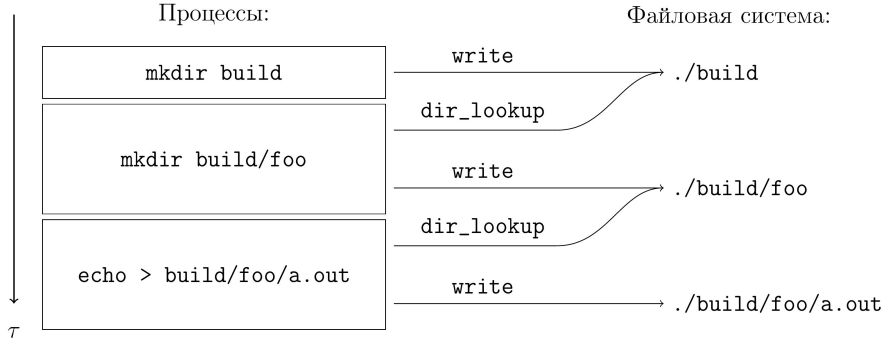

Рис. 7. Операции над файлами для большего числа вложенных директорий
Fig. 7. File operations for a greater depth of nested directories

При создании множественных вложенных директорий, доступ `dir_lookup` связывает между собой все «соседние» процессы. Если окажется, что все процессы, которые создают цепочку вложенных директорий, связаны соответствующей цепочкой зависимостей, то гонки будут исключены. Верно и обратное: если, например, `mkdir build` и `mkdir build/foo` не связаны зависимостью (цепочка зависимостей разорвана), то присутствует гонка — вторая цель может исполниться раньше первой, что приведёт к ошибке.

5.3.1 Обработка множественных попыток создания директории

На практике представленный алгоритм часто выдаёт ложные срабатывания. Проблема в том, что хоть две записи в файл и являются критическими операциями (поскольку влияют на содержимое файла) и требуют наличия зависимости, две попытки создания одной и той же директории могут быть безопасно переставлены местами. Результат не поменяется — директория всё равно будет создана в тот же момент времени.

```
all: lib1 lib2

lib1:
  mkdir -p build
  build_library build/lib1.a

lib2:
```

```
mkdir -p build
build_library build/lib2.a
```

Листинг 6. Пример Makefile с созданием директории build из нескольких целей
Listing 6. Multiple Makefile targets creating the same directory

В Makefile, изображённом на листинге 6, несколько целей самостоятельно создают каталог `build`, а затем используют его для сборки. На рис. 8 представлен сценарий сборки этого Makefile, при котором алгоритм выдаст ложные срабатывание.

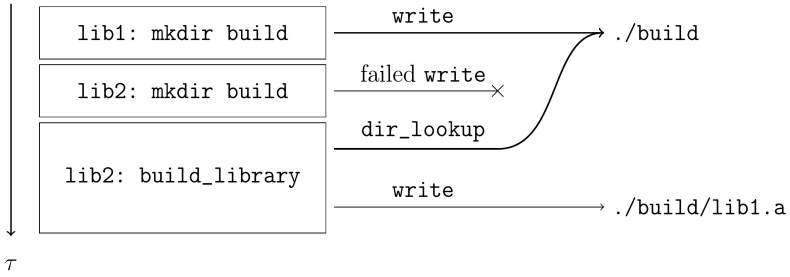

Рис. 8. Возможный сценарий сборки Makefile из листинга 6
Fig. 8. Possible build scenario for the Makefile from listing 6

Цель `lib1` выполнила `mkdir` первой. Когда `lib2` выполнила свой `mkdir`, он завершился ошибкой `EEXIST`, поскольку директория уже была создана. В связи с этим все дальнейшие `dir_lookup` будут проверяться на зависимость с первым успешным `mkdir`, совершенным целью `lib1`. Это приведёт к ложным срабатываниям. В связи с этим, метод, описанный в предыдущих главах, не подходит для обнаружения гонок на каталоге.

Чтобы утверждать о отсутствии гонки, достаточно убедиться, что перед любой операцией `dir_lookup` директория гарантированно будет создана. Для этого нужно проверить, что существует хотя бы одна операция `write` (вероятно, неуспешная), которая гарантированно произойдет раньше этого `dir_lookup`. Чтобы реализовать такую проверку, нужно начать учитывать неуспешные `mkdir`, которые завершились с `EEXIST`, как на рис. 9.

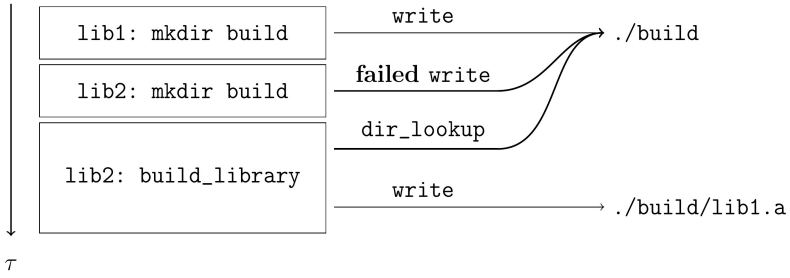

Рис. 9. Учитывание неуспешных mkdir
Fig. 9. Accounting for failed mkdir operations

В примере выше `dir_lookup` можно связать с неуспешным `mkdir`, который был произведён раньше той же целью. Таким образом, ложное срабатывание будет исключено.

Необходимо также учесть случай, когда `dir_lookup` происходит раньше любого `write`. Это означает, что директория была создана до начала сборки, и сообщать о гонке не нужно.

5.4 Поиск конфликтующих доступов

Базовый алгоритм поиска гонок, сформулированный в главе 5.1, содержит пункт «Найти конфликтующие доступы к одному и тому же пути (поколению inode) из разных целей». В этой части работы будет подробно рассмотрена задача поиска таких доступов.

Определим, какие виды доступов требуется поддерживать, чтобы обнаруживать все представленные ранее виды гонок.

- `read(path, inode)` — Чтение файла на заданном пути с заданным номером inode
- `write(path, inode)` — Запись файла / создание директории на заданном пути с заданным номером inode.
- `unlink_path(path)` — Удаление указанного пути из файловой системы
- `release_inode(inode)` — Освобождение номера inode (см. 5.1.2)
- `dir_lookup(path)` — Доступ к директории перед обращением к файлу в ней (см. 5.3)

Рассмотрим гонки на содержимом файла (см. 5.1). Для их поиска требуются только номера inode. Соответственно, актуальными для них могут являться только доступы `read`, `write` и `release_inode`. Последний вид доступа — удаление всех жёстких ссылок на inode. Гонки, вовлекающие операцию удаления, относятся к следующей категории, поэтому в рамках этого пункта будут рассмотрены только операции `read` и `write`.

Для любых двух последовательных операций на одном и том же номере inode можно легко сказать, являются ли они конфликтующими. Например, для пары операций чтения это неверно: результат будет одним и тем же вне зависимости от порядка. Чтение и запись могут образовать конфликт, поскольку от порядка будет зависеть результат чтения. Рассуждая аналогично, можно построить таблицу:

Табл. 1. Конфликты между операциями для гонок на содержимом файла
Table 1. Conflicts between operations for file content races

	`read`	`write`
`read`	-	+
`write`	+	+

Для работы с произвольным числом доступов рассуждения требуется обобщить. Один из простых способов — применить таблицу 1 ко всем соседним доступам на один и тот же номер inode. Однако такое обобщение не является корректным. На рис. 10 представлен простой контрпример.

Рис. 10. Контрпример для наивного алгоритма
Fig. 10. Counterexample for the naive algorithm

Такой алгоритм сочтёт конфликтующими только первую запись и первое чтение, несмотря на то, что второе чтение тоже обязано произойти строго позже записи в файл. Следовательно, проверять только соседние доступы недостаточно.

Другой простой способ обобщить рассуждения для любого числа доступов — применять таблицу ко всем возможным парам доступов. Такое решение корректно сработает на примере выше, однако не будет являться оптимальным: проверка всех возможных пар доступов требует $O(n^2)$ операций.

5.4.1 Улучшенный алгоритм перебора доступов

Отношение зависимости целей ($\rightarrow$) является транзитивным. Если цели A, B и C такие, что $A \rightarrow B$ и $B \rightarrow C$, то верно, что $A \rightarrow C$. Это рассуждение позволяет свести квадратичный перебор к линейному.

На схеме 11 изображена некоторая последовательность доступов к файлу. Стрелками связаны те доступы, для которых в таблице 1 указан конфликт:

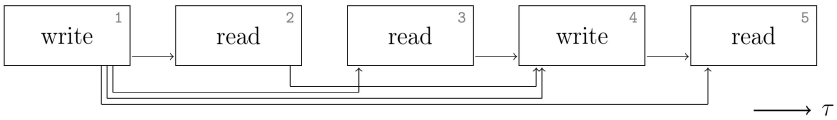

Рис. 11. Последовательность доступов с указанными конфликтами
Fig. 11. Sequence of accesses with specified conflicts

Можно заметить, что не обязательно проверять на зависимость все операции, связанные стрелками. Например, поскольку проверяется пара доступов **(1,4)** и **(4,5)**, проверять **(1,5)** необязательно — зависимость между ними будет следовать из зависимости первых двух. Если подобным образом удалить из все «избыточные» стрелки, то схема приобретёт следующий вид:

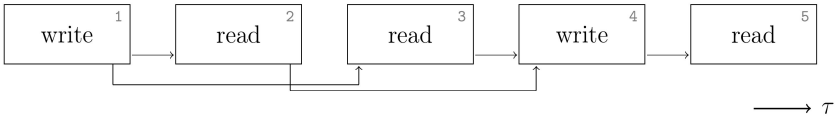

Рис. 12. Последовательность доступов с удалёнными избыточными конфликтами
Fig. 12. Sequence of accesses without redundant conflicts

5.4.2 Введение метода критических доступов

Можно показать, что после удаления всех избыточных конфликтов их количество сократится с квадратичного до линейного, а само множество проверок окажется минимальным.

Построить множество пар для проверки можно тривиально, если разделить доступы на две категории:

- Критические доступы (*C*) — барьеры, которые нельзя менять местами ни с какими другими операциями.
- Некритические доступы (*N*) — те, которые можно менять местами с некритическими соседями. Например, операции чтения.

Тогда само множество проверок будет иметь следующий вид:

- Критические доступы связаны с своими непосредственными критическими соседями;
- Некритические доступы связаны с своими ближайшими критическими соседями.

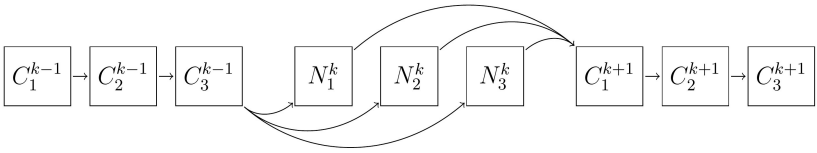

Рис. 13. Рёбра P на последовательности цепочек вида C, N, C
Fig. 13. Edges P on a sequence of chains of the form C, N, C

В зависимости от того, какие именно доступы помечаются критическими, а какие — некритическими, представленный алгоритм может обнаруживать разные виды гонок. Согласно таблице 1, для гонок на содержимом файла критической нужно определить операцию `write`, а некритической — `read`.

Поиск гонок на пути к файлу требует список операций на пути, а не на номере inode (см. 5.2). Среди всего списка доступов (см. 5.4), с путями работают `read`, `write` и `unlink`. Есть также доступ типа `dir_lookup`, но он предназначен только для поиска гонок с созданием директорий (см. 5.3), который будет рассмотрен ниже.

Осталось разделить выбранные типы доступов на критические и некритические. Обратимся к пункту 5.2 и составим таблицу конфликтов для этого типа гонок.

Табл. 2. Конфликты между операциями для гонок на пути
Table 2. Conflicts between operations for file path races

	`read`	`write`	`unlink`
`read`	-	-	+
`write`	-	-	+
`unlink`	?	+	?

Поскольку после удаления пути может быть только повторное его создание, пары (`unlink`, `unlink`) и (`unlink`, `read`) помечены знаком вопроса.

Можно заметить, что операции `read` и `write` не конфликтуют ни в каком сочетании. Это значит, что их можно отнести к некритическим доступам. Операция `unlink`, напротив, конфликтует с любой другой операцией, и следовательно, является критической.

Правильный выбор конфликтующих доступов важен для корректного поиска гонок. Если бы все операции кроме чтения (`unlink` и `write`) были помечены критическими, как в предыдущем пункте, гонкой на пути могли бы считаться две операции записи. На практике это привело бы к нежелательному дублированию: такая гонка обнаружилась бы и как гонка вида 5.2 (на содержимом файла) и как гонка вида 5.1 (на пути к файлу).

5.4.3 Алгоритм поиска гонок вида 5.3

Согласно пункту 5.3, для поиска гонок этой категории требуется особый набор проверок, связанных через логическое «или». Метод критических доступов не подходит для этого.

С другой стороны, алгоритм был уже почти полностью описан ранее. Каждую операцию `dir_lookup` нужно проверить на наличие зависимости хотя бы с одной предшествующей попыткой создания (операцией `write`, быть может, неуспешной), произошедшую после последнего `unlink`. Наивная реализация этого алгоритма будет предполагать квадратичный перебор, однако его можно сократить до линейного, если использовать ленивый алгоритм и кешировать те цели, из которых `dir_lookup` не порождает гонку.

6. Описание практической части

Согласно разработанной архитектуре, трассировщик и санитайзер представляют собой два отдельных процесса. Используя Unix-сокеты, санитайзер получает от трассировщика информацию о доступах к файлам и о порождении новых процессов, а от remake — дерево зависимостей и соответствие целей сборки номерам процессов (pid).

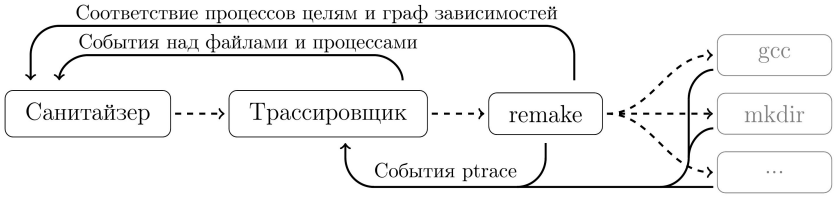

Рис. 14. Взаимодействие процессов санитайзера
Fig. 14. Interaction of sanitizer processes

Пунктирные стрелки на рис. 14 связывают родительские процессы с дочерними.

Выбранная архитектура позволяет изолировать санитайзер от непосредственной работы с файловой системой и с дочерними процессами. Результат работы санитайзера определяется лишь сообщениями, которые он получает от трассировщика и от процессов remake. Позже это решение также позволит значительно ускорить итеративную отладку гонок.

Дерево зависимостей передаётся санитайзеру из Make-процесса. Это позволяет избежать написания своего синтаксического анализатора для Makefile. В дополнение, такой подход гарантирует, что граф зависимостей, на основании которого производится поиск гонок, в

точности соответствует настоящему графу зависимостей, который используется самой утилитой Make.

Поскольку трассировщик является сравнительно тонкой обёрткой над системным вызовом `ptrace`, он был написан на языке Си. Для санитайзера, как для более сложного проекта, был выбран язык C++.

6.1 Построение дерева процессов

Информация о процессах приходит санитайзеру из двух источников. Первым трассировщик сообщает о создании нового процесса путём отслеживания системных вызовов `clone`, `spawn` и `fork`. Если процесс был порождён процессом remake, он должен передать информацию о цели, которой этот процесс соответствует. Это позволит санитайзеру соотнести операции над файлами с целями сборки (см. 8).

По умолчанию трассировщик останавливает все новые процессы на первой инструкции. Это необходимо для того, чтобы успеть настроить перехват системных вызовов и передать санитайзеру информацию о созданном процессе раньше, чем тот начнёт совершать какие-либо действия. Когда санитайзер получает сообщение о создании нового процесса, он добавляет его в общее дерево и отправляет подтверждение трассировщику. Дождавшись ответа, трассировщик позволяет процессу начать работу.

Если процесс успеет открыть какой-то файл раньше, чем санитайзер узнает его цель, он не сможет отнести этот доступ к правильной цели, и может пропустить гонку. Поэтому цель, которой соответствует процесс, должна быть получена санитайзером раньше, чем этот процесс будет запущен. Это позволяет реализовать схема `fork/exec`. После вызова `fork` pid процесса становится известен. В этот момент remake отправляет его вместе с названием цели санитайзеру. Дождавшись ответного сообщения, и убедившись что санитайзер установил соответствие между процессом и целью, remake заканчивает создание нового процесса вызовом `exec`.

Утилита remake поддерживает несколько способов порождения новых процессов. Кроме классического `fork/exec` поддерживается и более эффективный метод, использующий `posix_spawn`. Этот способ, однако, не позволяет узнать pid нового процесса перед его запуском. Его нужно отключить, указав флаг `--disable-posix-spawn` в фазе configure.

Сообщая о гонке, санитайзер указывает два конфликтующих доступа, которые могут произойти в обратном порядке. Один из этих доступов может быть произведён процессом, который уже успел завершиться. В этом случае его pid может быть использован ядром повторно для другого процесса. В связи с этим для идентификации процессов в внутренних структурах санитайзера используется пара чисел (pid, эпоха). Каждый раз, когда pid используется повторно, эпоха инкрементируется. Это позволяет избежать конфликтов и однозначно ссылаться на уже завершённые процессы.

6.2 Отслеживание операций над файлами

Табл. 3. Системные вызовы над файлами, отслеживаемые трассировщиком.
Table 3. File system calls tracked by the tracer

Системный вызов	**Событие для санитайзера**
`open(at)(at2)`	`read` или `write`
`mkdir(at)`	`write`
`creat`	`write`
`rmdir`	`unlink`
`unlink(at)`	`unlink`
`rename(at)(at2)`	`unlink` целевого пути, если он существует.

Трассировщик перехватывает операции над файлами и сообщает о них санитайзеру. На таблице 3 перечислены системные вызовы, соответствующие разным типам событий для санитайзера.

Флаги, переданные системному вызову `open`, могут указать, открывает ли процесс файл на чтение (`O_RD`) или на запись (`O_WR`). Если файл открыт на чтение и запись (`O_RDWR`), трассировщик сообщает только о записи. Этого оказывается достаточно, поскольку ни в одном из рассматриваемых типов гонок чтение из файла не породит новых конфликтов.

Санитайзер работает в предположении, что если файл был открыт на чтение или запись, то процесс обязательно произведёт это чтение или запись. Это может быть верно не всегда: процесс может открыть файл в режиме `O_RDWR`, и использовать полученный дескриптор только для чтения. Единственный способ избежать этой проблемы — отслеживать системные вызовы `read` и `write`.

На практике такое решение оказывается неоправданным. На одном открытом файле `read` и `write` может быть вызван множество раз. Кроме операций над файлами, эти системные вызовы используются для работы с сокетами, вывода в консоль, чтения `signalfd`, и прочим. Перехват таких нагруженных системных вызовов многократно увеличивает объём обрабатываемых данных и замедляет работу санитайзера.

Поскольку санитайзер предполагает «наихудший» сценарий, подход с использованием флагов `open` не может привести к пропуску гонок, только к ложным срабатываниям. Несмотря на это, при тестировании санитайзера на реальных проектах таких случаев не было выявлено. Все утилиты gcc и стандартные утилиты Linux используют правильные флаги для `open`.

Переименование файла `A` в `B`, контринтуитивно, не эквивалентно удалению пути `A` и созданию пути `B`. При перемещении файла с помощью системного вызова `rename` его номер inode не изменяется. При этом, если путь назначения (`B`) существовал, то inode, на которую он ссылался, теряет ссылку. Это эквивалентно удалению пути `B`, а не `A`. Санитайзер учитывает переименование как удаление пути назначения. Это позволяет искать гонки на содержимом файла даже если он был переименован.

6.3 Обработка вложенных Make

Крупные проекты бывают разделены на несколько схем сборки, каждая из которых отвечает за свой модуль. «Корневой» Makefile запускает их сборку, вызывая вложенные Make. Таким образом, проект можно представить как дерево из модулей. Гонки могут происходить между разными модулями одного проекта.

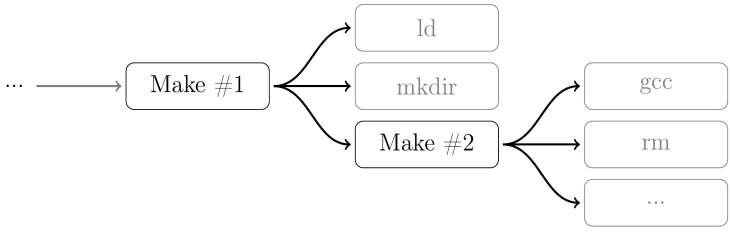

Рис. 15. Дерево процессов при использовании вложенных Make
Fig. 15. Process tree when using submake

Рассмотрим процессы `gcc` и `ld` из рис. 15. Предположим, что процесс `gcc` запускается в ходе сборки библиотеки `libfoo.a` одноимённой целью, `ld` — целью `a.out` а Make #2 — целью `libfoo`, как на рис. 16.

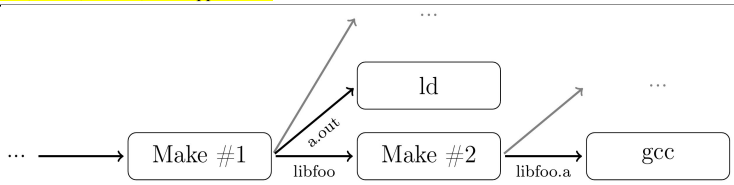

Рис. 16. Дерево процессов с целями при использовании вложенных Make
Fig. 16. Process tree with goals when using submake

Предположим, что упомянутые процессы работают с одним и тем же файлом `libfoo.a`. Процесс `gcc` — на запись, `ld` — на чтение. Чтобы проверить наличие гонки, необходимо установить, есть ли зависимость между целями `libfoo.a` и `a.out`. Найти эту зависимость напрямую невозможно, поскольку эти цели принадлежат разным Makefile.

Как изображено на рис. 17, цели вложенного Make-процесса выполняются в интервале времени работы внешней цели, породившей этот вложенный Make. Если между `libfoo` и `a.out` есть зависимость, это значит, что все цели вложенного Make (включая `libfoo.a`) закончат выполнение раньше, чем запустится `a.out`.

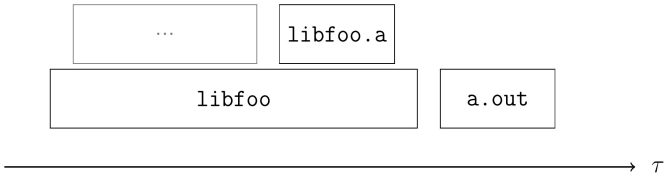

Рис. 17. Временная линия сборки при использовании вложенных Make
Fig. 17. Build timeline when using submake

В случае, если две цели из разных Makefile производят доступ к одному и тому же файлу, санитайзер «поднимется» на тот уровень, который породил обе этих цели, и будет производить поиск зависимости между соответствующими целями верхнего уровня. Этот «подъём» известен как задача поиска Lowest Common Ancestor (LCA). Она имеет несколько on-line решений:

1) Линейный подъём, использующий глубину узлов дерева — $O(n)$ в худшем случае.
2) Двоичный подъём, требующий предварительного подсчёта ссылок на родителей уровней 2^i — $O(\log n)$ в худшем случае.

Оба алгоритма имеют свои достоинства и недостатки. Линейный подъём дольше поднимается на большие расстояния, в то время как двоичный подъём требует предварительного подсчёта, что усложняет процедуру добавления новых процессов в дерево. Таким образом, оптимальный выбор алгоритма зависит от того, насколько часто санитайзеру придётся подниматься на большую высоту.

6.3.1 Оценка средней дистанции подъёма

Для выбора оптимального алгоритма был проведён эксперимент на нескольких проектах. Наибольшая средняя высота подъёма составила 0.74 при сборке gcc версии 13.1.0 для платформы x86-64 с флагом `--disable-multilib`. Подробные результаты представлены на табл. 4.

Табл. 4. Количество подъёмов при сборке `gcc` версии 13.1.0
Table 4. Number of ascents when building `gcc` version 13.1.0

Высота подъёма	Количество	% от общего числа
0	1270422	63.50%
1	314306	15.71%

Высота подъёма	Количество	% от общего числа
2	176906	8.84%
3	188222	9.40%
4	34932	1.74%
5	15562	0.77%
6	260	0.0013%

Среди проектов, выбранных для эксперимента, gcc является самым крупным. При сборке небольших проектов средняя высота подъёма меньше в сотни раз: на проекте Vim она составляет 0.004, а на strace — 0.0028. Это указывает на то, что наиболее оптимальным алгоритмом будет метод линейного подъёма.

6.4 Протокол взаимодействия

Архитектура инструмента предполагает обмен данными между процессами посредством сокета. Ниже представлен пример сообщения, которым трассировщик сообщает санитайзеру, что определённый процесс открыл файл в режиме чтения.

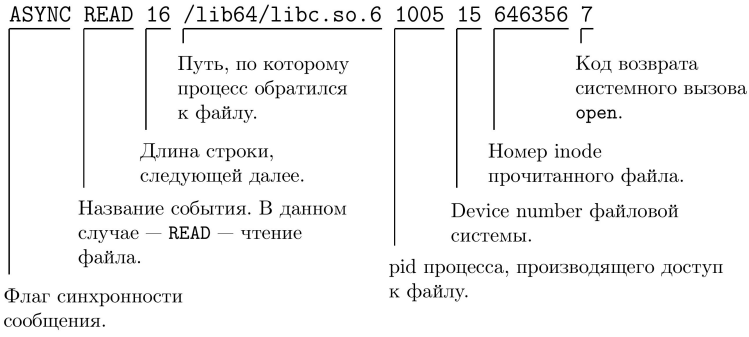

Рис. 18. Пример сообщения, передаваемого санитайзеру трассировщиком
Fig. 18. Example message sent from the tracer to the sanitizer

Любое сообщение, отправляемое санитайзеру, должно начинаться с флага синхронности (слова `SYNC` или `ASYNC`). Он указывает, ожидает ли отправляющая сторона ответное сообщение, как подтверждение того что событие было обработано. Пункт 6.1 содержит примеры некоторых типов сообщений, требующих подобной синхронизации.

Следующее слово определяет тип передаваемого события. Каждый тип имеет свой набор аргументов. Если аргументом является строка (например, путь к файлу), то перед её началом передаётся её длина и один разделяющий символ пробела. Такой простой формат упрощает составление и разбор сообщений до нескольких вызовов `memcpy` и `sprintf` / `sscanf`.

Перечень сообщений, отправляемых трассировщиком санитайзеру:

1) `INIT TRACER` — инициализирующее сообщение.
2) `CHILD <pid> <ppid> <cmdline>` — Сообщает о создании нового процесса в дереве, или о том, что существующий процесс сменил свою `cmdline`, вызвав `exec`. Это сообщение всегда является синхронным, см. 6.1.
3) `READ <path> <pid> <devnum> <inum>` <код возврата> — событие открытия процессом файла на чтение.
4) `WRITE <path> <pid> <devnum> <inum>` <код возврата> — событие открытия процессом файла на запись.
5) `UNLINK <path> <pid> <devnum> <inum>` <код возврата> — событие удаления пути (directory entry)

6) `INODE_UNLINK <path> <pid> <devnum> <inum> <код возврата>}` — дублирует сообщение `UNLINK` в случае, если удалённый путь был последней жёсткой ссылкой на свою inode.

7) `DIE <pid>` — сообщает о завершении работы процесса с указанным pid. Если свою работу завершает сам трассировщик, он отправляет это сообщение с собственным pid. В этом случае сообщение должно быть синхронным. Это будет гарантировать, что санитайзер успеет обработать все предыдущие сообщения от трассировщика прежде, чем получит сигнал `SIGCHLD` и остановится.

Перечень сообщений, отправляемых процессом remake санитайзеру:

1) `INIT MAKE` — инициализирующее сообщение.

2) `TARGET_PID <pid> <ppid> <cmdline>` — Устанавливает соответствие между процессом с указанным pid и целью, породившей его. Это сообщение всегда является синхронным, см. 6.1.

3) `DEPENDENCY <target_a> <target_b>` — Сообщает о наличии зависимости между двумя целями. Это сообщение должно являться синхронным, поскольку санитайзер должен получить весь граф зависимостей, прежде чем сможет находить потенциальные гонки между получаемыми им доступами.

В обратную сторону санитайзер может отправить только слово `ACK` (от англ. — Acknowledged, принято). Оно отправляется как подтверждение завершения обработки предыдущего сообщения, если это требуется флагом синхронности.

Санитайзер должен знать, какой pid имеет отправитель каждого сообщения. Ядро Linux позволяет получить эти данные, используя системный вызов `getsockopt` и флаг `SO_PEERCRED` (см. [8]). Это избавляет от необходимости передавать эти данные по сокету.

В качестве режима для сокета был выбран `SOCK_SEQPACKET`. Он гарантирует порядок доставки и сохранение границ сообщений (см. пункт 2.10.6 в стандарте POSIX [1]).

Имена событий подобраны таким образом, чтобы их можно было отличать по первому символу (сообщение `INIT` является исключением, но его отличает то, что оно всегда идёт первым). Это позволяет использовать `switch` для вызова нужного обработчика. Несмотря на то, что передаваемые сообщения являются человекочитаемыми, а не бинарными, простота протокола позволяет избежать значительных потерь в производительности.

6.5 Режим интерактивной отладки

При отладке состояний гонок разработчикам часто требуется многократно пересобирать проект, устанавливая отладочные выводы на сборке определённых целей. Разработанный инструмент предлагает несколько функций, упрощающих этот процесс:

1) Точки останова. Санитайзер позволяет приостанавливать сборку на определённых действиях с файлами или при обнаружении гонки. При срабатывании точки останова санитайзер предоставляет пользователю интерактивную консоль, с помощью которой можно вывести информацию о текущем состоянии сборки, а именно:
    - Цель, при сборке которой произошел последний доступ к файлу;
    - Процесс, который произвёл этот доступ;
    - Информацию по любому процессу, даже завершённому:
        - Командную строку процесса;
        - Список целей его Makefile, если он является Make-процессом;
        - Цепочку его родительских процессов;
        - Поддерево процессов, порожденных им.

2) Быстрое воспроизведение записи сборки. Если перенаправить в файл все сообщения, поступающие санитайзеру, и передать их на вход повторно, то результат его работы будет таким же, как и во время настоящей сборки.

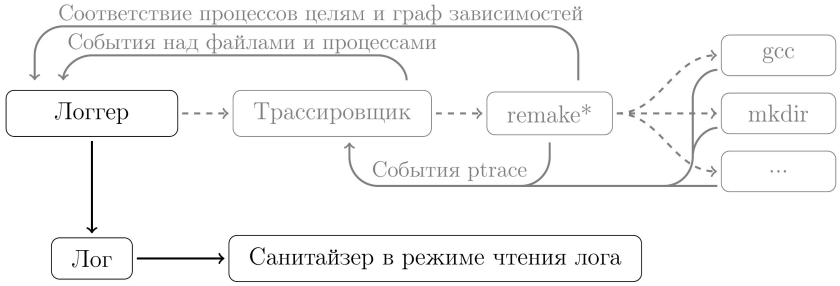

Рис. 19. Запись и воспроизведение лога сборки
Fig. 19. Recording and replaying build traces

Такое «воспроизведение» быстрее обычной пересборки проекта. Чтение записи сборки gcc 13.1.0 занимает у санитайзера 70 секунд, в то время как пересборка проекта с использованием восьми потоков занимает 50 минут.

В сочетании с предыдущим пунктом эта функция позволяет производить итеративную отладку. После изменения точек останова, разработчик может быстро перезапустить запись, и не ждать полной пересборки.

В случае, если санитайзер читает лог сборки, а не получает сообщения из сокета, использовать `SO_PEERCRED` для получения pid отправителя невозможно. Эти данные указываются в логе явно, в начале каждого сообщения.

Для реализации точек останова в санитайзер был добавлен отладочный режим. При его активации, все передаваемые по сокету сообщения становятся синхронными. При срабатывании точки останова на определённом доступе к файлу санитайзер перестаёт отправлять `ACK`-сообщения. Это приводит к тому, что трассировщик не позволяет процессам продолжить работу, и останавливает сборку.

Точка останова может быть привязана к произвольному множеству путей, задаваемым одним или несколькими glob–выражениями. Санитайзер конвертирует их в МПДКА, поэтому сложность проверки не растёт с увеличением количества точек останова. Санитайзер может добавить к существующему автомату новый, отвечающий за срабатывание на новой точке останова, с использованием логического «ИЛИ». Затем сумму этих автоматов можно привести к новому МПДКА. Это позволяет добавлять и удалять точки останова в любой момент отладки без ухудшения производительности (удаление точки останова эквивалентно добавлению инвертированного автомата с логическим «И»).

6.6 Тестирование и сравнение

По мере разработки проекта был разработан набор из 23 автоматических тестов, проверяющих поведение санитайзера в известных крайних случаях. Среди них:

- Обнаружение гонок, включающих:
  - Обновление Makefile, влекущее перезапуск Make;
  - Доступы, производимые самим Make-процессом.
  - Создание вложенных директорий;
  - Указание более чем одной цели для сборки Make-процессом;
  - Использование шаблонных правил;
  - Запуск вложенных Make.

- Разрешение символических ссылок при доступе к файлу, но игнорирование их при удалении;
- Корректная нормализация пути (удаление `.` и `..`);

Тестирование производится автоматически с использованием CI на LXC-контейнерах с тремя разными системами:

- Ubuntu Mantic;
- OpenRC Gentoo 6.8.1;
- Alpine 3.19.

Сергей Трофимович, разработчик режима Make `--shuffle`, опубликовал в своём блоге список проектов, в которых ему удалось обнаружить гонки с использованием этого режима [5]. Оценка эффективности разработанного санитайзера производилась путём запуска его на проектах из этого же списка и сравнении полученных гонок с перечнем известных, обнаруженных ранее.

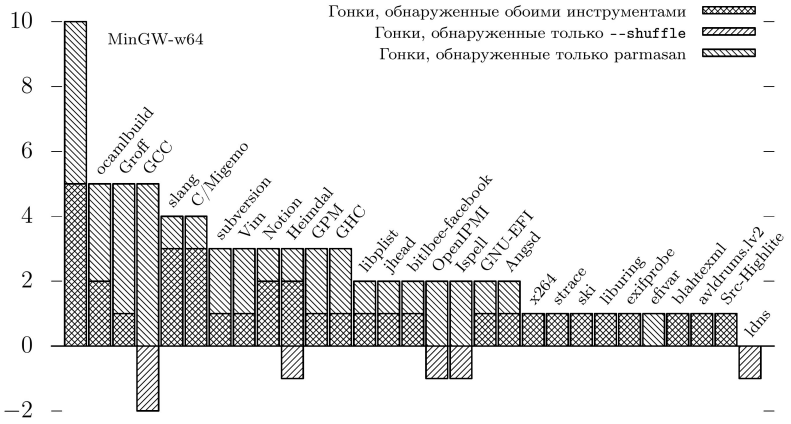

Рис. 20. Результаты тестирования инструмента на проектах с известными гонками
Fig. 20. Testing results of the tool on projects with known data races

Следует уточнить, что «количество обнаруженных гонок» не является формальной величиной. Санитайзер может обнаружить сотни и тысячи гонок, которые в действительности могут быть устранены одним исправлением в схеме сборки. В связи с этим, гонки, обнаруженные санитайзером, были сгруппированы вручную по схожести имён участвующих в них файлов и целей. Во избежание завышения оценок, в спорных случаях гонки также группировались вместе, а ложные срабатывания, по возможности, не учитывались.

Например, гонка в проекте strace происходила из-за того, что шаблонное правило `mpers-m%.stamp` не содержало зависимость с файлом `sys_func.h` [9]. В таком случае санитайзер выведет отдельные сообщения о гонках для каждого файла, собираемого этим правилом. На рис. 20 все такие гонки были сгруппированы вместе.

Диаграмма на рис. 20 позволяет оценить эффективность разработанного инструмента. Среди 35 гонок, о которых сообщалось изначально, санитайзер успешно обнаружил 29, и сообщил о 39 новых.

6.7 Пример новой обнаруженной гонки

Рассмотрим одну из новых гонок, обнаруженных санитайзером в проекте Vim.

```
race found at file 'src/po/LINGUAS':
 - unlink at target gvim.desktop
 - write at target vim.desktop
```

Листинг 7. Диагностика санитайзера при сборке проекта Vim

Listing 7. A diagnostic produced by the sanitizer when building Vim

Диагностика, изображенная на листинге 7, сообщает о гонке на пути к файлу (5.1). Как было замечено в пункте 8, гонки этой категории нельзя обнаружить режимом `--shuffle`. Обратимся к рецептам целей, указанных в диагностике (`src/po/Makefile:216`).

```
vim.desktop: ...
  echo ... > LINGUAS
  $(MSGFMT) ...
  rm -f LINGUAS

gvim.desktop: ...
  echo ... > LINGUAS
  $(MSGFMT) ...
  rm -f LINGUAS
```

Листинг 8. Фрагмент схемы сборки Vim

Listing 8. A fragment of the Vim Makefile

Цели, указанные в диагностике, используют одно и то же имя (`LINGUAS`) для временного файла, что приводит к гонке на этом пути.

В версии 9.1.0108 эта гонка была исправлена путём добавления зависимости между целями `vim.desktop` и `gvim.desktop`. Для проведения справедливого сравнения санитайзер тестировался на версии 8.2.4595, которую использовал Сергей Трофимович при тестировании режима Make `--shuffle` и составлении публикации в своём блоге.

6.8 Оценка замедления сборки

Для оценки скорости работы разработанного инструмента было проведено сравнение времени сборки пяти разных проектов с использованием и без использования санитайзера. Для сборки использовалось 8 потоков. Результаты представлены на таблице 5.

Табл. 5. Сравнение времени сборки проектов

Table 5. Comparison of project build times

Программа	Время сборки	Время сборки с санитайзером	Замедление
vim	0:18,3	0:20,7	13,1%
x264	0:29,6	0:33,0	11,5%
exifprobe	0:01,36	0:01,53	12,6%
angsd	0:21,3	0:23,8	11,7%
gcc	50:28,0	60:53,5	20,6%

Согласно результатам тестирования, замедление сборки составляет от 12% до 20%, что является хорошим результатом для инструментов такого класса. Например, Thread Sanitizer, предназначенный для обнаружения гонок в коде на C и C++, замедляет программу в 5–15 раз [10], а Address Sanitizer, который считается быстрым санитайзером, замедляет программу в среднем в 2 раза [11].

При сборке gcc замедление составило 20.6%, что заметно выше остальных измерений. Профилирование показало, что дополнительные 8% замедления возникают на этапе перехвата системных вызовов. Время, которое требуется алгоритмам поиска гонок на обработку всей трассы, не зависит от объёма проекта и составляет около 2%. В случае gcc, это время составляет 1:06, что в 45 раз быстрее обычной пересборки проекта.

Табл. 6. Сравнение времени сборки и времени обработки трассы

Table 6. Comparison of build time and trace processing time

Программа	Время сборки	Время обработки трассы	Соотношение
vim	0:18,3	0:00,367	2%
x264	0:29,6	0:00,218	0,7%

Программа	Время сборки	Время обработки трассы	Соотношение
exifprobe	0:01,36	0:00,023	1,6%
angsd	0:21,3	0:00,392	1,8%
gcc	50:28,0	1:06,618	2,2%

Согласно таблице 6, время обработки трассы проекта x264 меньше, чем в остальных измерениях. Это связано с тем, что этот проект реже обращается к файлам при сборке. За 29.6 секунд он производит 117510 операций с файлами, что соответствует 3969 операциям в секунду. Для сравнения, сопоставимые по времени сборки проекты vim и angsd совершают 10204 и 10122 операций в секунду соответственно.

7. Заключение

В рамках этой работы была рассмотрена проблема состояний гонок при параллельной сборке. Был проведён анализ проблемы и приведены недостатки существующих решений. Для разработки нового решения была представлена классификация распространённых типов гонок и разработаны методики для их автоматического обнаружения. Разработан метод критических доступов, позволяющего производить линейное количество проверок. В практической части представлена архитектура программной реализации представленных алгоритмов.

Согласно гистограмме на рис. 20, санитайзер не является полноценной заменой режима `--shuffle` утилиты Make. Некоторые существующие в проектах гонки не обнаруживаются санитайзером. Несмотря на корректность представленных алгоритмов, представленная в работе классификация не покрывает все возможные гонки.

Существует по крайней мере ещё один тип гонок, который не может быть обнаружен ни одним из представленных алгоритмов — гонка между созданием и использованием символической ссылки. Известно, что некоторые из пропущенных санитайзером гонок относятся именно к этому типу. Составить алгоритм, который бы позволил производить нужные проверки для обнаружения таких гонок за допустимое время, к текущему времени не удалось. Даже с добавлением такого алгоритма в санитайзер нельзя было бы гарантировать полное покрытие всех возможных типов гонок.

Несмотря на это, в сравнении с режимом `--shuffle` утилиты Make, разработанный инструмент в большей степени отвечает требованиям, сформулированным в главе 9:

- Согласно данным тестирования, в среднем санитайзер обнаруживает больше гонок, чем позволяет Make `--shuffle`;
- В отличие от существующего решения, алгоритмы поиска, используемые санитайзером, не носят вероятностный характер;
- Инструмент может быть встроен в любой проект, использующий Make, путём использования его модифицированной версии. Согласно результатам тестирования, замедление сборки составляет от 12% до 20% в зависимости от сложности проекта;
- В сравнении с Make `--shuffle`, санитайзер требует произвести лишь единоразовую сборку проекта. Отладка гонок может производиться отложено с использованием собранной трассы, что значительно сокращает время ожидания.

Дальнейшее улучшение санитайзера включает в себя следующие направления:

- Расширение классификации гонок и разработка алгоритмов для их обнаружения (например, гонка между созданием и использованием символической ссылки);
- Поддержка других систем сборки, таких как Ninja;
- Внедрение санитайзера в существующие build-боты и CI/CD системы;
- Добавление протокола, подобного Chrome DevTools Protocol, для подключения внешних отладчиков;

Актуальные версии санитайзера и модифицированного remake доступны на репозиториях GitHub:

- `https://github.com/ispras/parmasan`
- `https://github.com/ispras/parmasan-remake`

Список литературы / References

[1]. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7 // IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008). — 2018. — Pp. 1–3951.

[2]. Packages failing to use parallel make. — https://bugs.gentoo.org/351559. — 2011. — [Online; accessed 14-March-2024].

[3]. Serebryany, Konstantin. ThreadSanitizer – data race detection in practice. / Konstantin Serebryany, Timur Iskhodzhanov // Proceedings of the Workshop on Binary Instrumentation and Applications. — NYC, NY, U.S.A.: 2009. — Pp. 62–71. http://doi.acm.org/10.1145/1791194.1791203.

[4]. Bazel sandboxing. — https://bazel.build/docs/sandboxing. — 2024. — [Online; accessed 12-March-2024].

[5]. Trofimovich, Sergei. A small update on 'make –shuffle' mode. — https://trofi.github.io/posts/249-an-update-on-make-shuffle.html. — 2022. — [Online; accessed 11-March-2024].

[6]. Random by default patch for Make. — https://slyfox.uni.cx/distfiles/make/make-4.3.90.20220619-random-by-default.patch. — 2022. — [Online; accessed 14- March-2024].

[7]. PR: Makefile.in: guarantee directory creation at install time before file copy. — https://github.com/telmich/gpm/pull/43. — [Online; accessed 21-March-2024].

[8]. unix(7) — Linux manual page. — https://man7.org/linux/man-pages/man7/unix.7.html. — [Online; accessed 25-April-2024].

[9]. PR: mpers: add missing dependency on autogenerated sys_func.h. — https://github.com/strace/strace/pull/215. — [Online; accessed 8-April-2024].

[10]. Clang 19.0.0git documentation - Thread Sanitizer. — https://clang.llvm.org/docs/ThreadSanitizer.html. — [Online; accessed 22-April-2024].

[11]. Clang 19.0.0git documentation - Address Sanitizer. — https://clang.llvm.org/docs/AddressSanitizer.html. — [Online; accessed 22-April-2024].

Информация об авторах / Information about authors

Артем Юрьевич КЛИМОВ — Лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 4 курса Московского Физико-технического Института Физтех-школы Прикладной Математики и Информатики по направлению Информатика и Вычислительная Техника.

Artem Yurievich KLIMOV — Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 4th year student of the Moscow Institute of Physics and Technology, Department of Applied Mathematics and Informatics, field of Informatics and Computer Science.

Владислав Анатольевич ИВАНИШИН — Научный сотрудник Института системного программирования им. В.П. Иванникова Российской академии наук.

Vladislav Anatolevich IVANISHIN — Researcher at Ivannikov Institute for System Programming of the Russian Academy of Sciences.

Александр Владимирович МОНАКОВ — Старший научный сотрудник Института системного программирования им. В.П. Иванникова Российской академии наук.

Alexander Vladimirovich MONAKOV — Senior researcher at Ivannikov Institute for System Programming of the Russian Academy of Sciences.