Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа прикладной математики и информатики
Кафедра системного программирования ИСП РАН
Отдел компиляторных технологий

Выпускная квалификационная работа бакалавра

Автоматическое обнаружение гонок при параллельной сборке с использованием утилиты Make

Автор:
Студент группы Б05-032
Климов Артем Юрьевич

Научный руководитель:
Мельник Дмитрий Михайлович

Научный консультант:
Иванишин Владислав Анатольевич

Научный консультант:
Монаков Александр Владимирович

Москва 2024

Аннотация

Состояния гонки в схемах сборки программных проектов являются распространённой проблемой. Существующие решения не всегда позволяют искать их эффективно. В этой работе представлен процесс разработки нового санитайзера, позволяющего автоматически обнаруживать гонки в схемах сборки для систем, основанных на Make. Разработанный санитайзер доказал свою эффективность, обнаружив <X> новых гонок в <Y> проектах с открытым исходным кодом.

Содержание

1 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.

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.

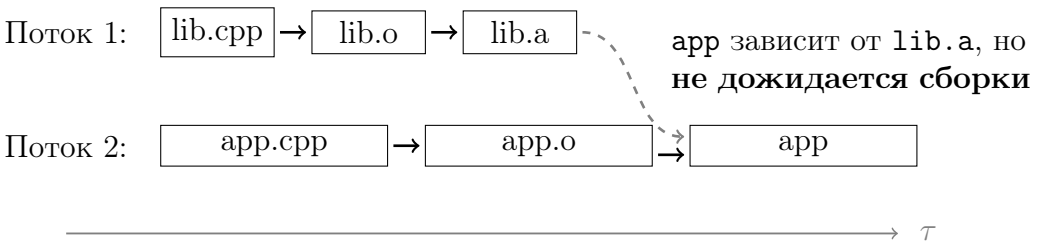

Рис. 1: Процесс сборки проекта с состоянияем гонки в схеме сборки

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

На рисунке сверху это не приводит к ошибке, поскольку библиотека сама собой успела собраться быстрее, чем она потребовалась. Однако, это не всегда может быть так. Выход из строя секторов диска, расширение самой библиотеки и множество других непредсказуемых причин могут привести к увеличению времени сборки библиотеки.

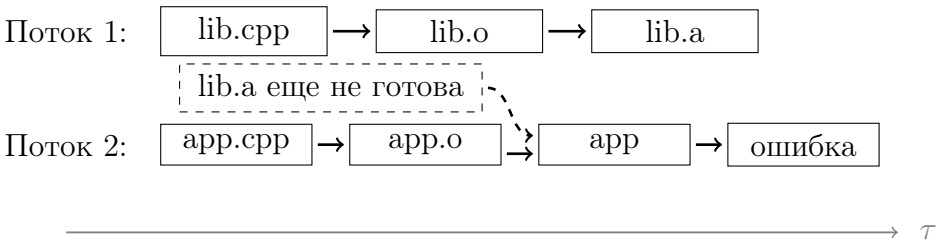

Рис. 2: Процесс сборки проекта с состоянияем гонки в схеме сборки

В такой ситуации перед разработчиком стоит выбор: попробовать собрать проект повторно, или потратить время на поиск недостающей зависимости и исправление схемы.

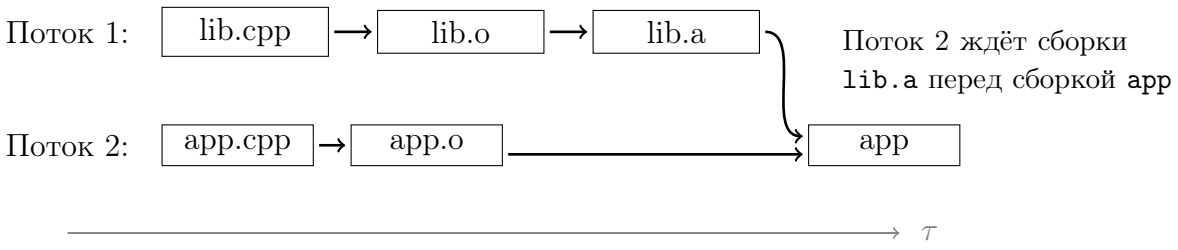

Рис. 3: Исправленная схема сборки без состояния гонки

Настоящие схемы сборки, как правило, выглядят значительно сложнее, и найти в них недостающую зависимость становится трудно. В связи 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 [1].

Опасность этих гонок заключается в том, что оставаясь скрытыми, они могут проявляться самым нежелательным образом. Наиболее частый симптом, наблюдаемый при наличии такой проблемы в схеме — 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.

2 Постановка задачи

Ручное исправление состояний гонок в схемах сборки — трудный процесс. Цель этой работы — предоставить решение, которое бы позволило его упростить. Для этого предлагается разработать автоматический инструмент — санитайзер для параллельных сборок. Он должен отвечать следующим требованиям:

- Инструмент должен обнаруживать все гонки, связанные 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 — сократить время, требуемое разработчику для поиска гонок.

3 Обзор существующих решений

Современные системы сборки предпринимают меры для борьбы с гонками. Например, система Bazel собирает каждую цель в отдельной песочнице, в которой есть

только те файлы, которые соответствуют зависимостям этой цели сборки [2]. С таким ограничением любая схема обязана иметь все необходимые зависимости, чтобы успешно собраться. 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`, недавно добавленный в GNU Make [3]. 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 [4], 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.

4 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.

4.1 Гонка на содержимом файла

Листинг 1: Пример Makefile с гонкой на содержимом объектных файлов

```
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
```

В этом примере между целями `compile` и `link` не хватает зависимости. Аналогично примеру из вступления, 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. Самый

простой способ увидеть, как процесс работает с файлами — запустить его под утилитой strace.

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

```
$ 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
```

В 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.

4.1.1 Сопоставление операций над файлами с целями сборки

Чтобы сопоставить номера процессов с целями сборки предлагается модифицировать саму утилиту Make. В качестве подопытного был взят проект remake. Он реализует тот же функционал, что и GNU Make, но требует значительно меньше усилий для сборки из исходного кода. После внесения изменений (см. приложение 6.1) в логе сборки появятся строки с информацией о том, какие процессы порождаются Make, и каким целям они соответствуют.

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

```
$ 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
...
```

Можно заметить, что ни один процесс `gcc`, 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`.

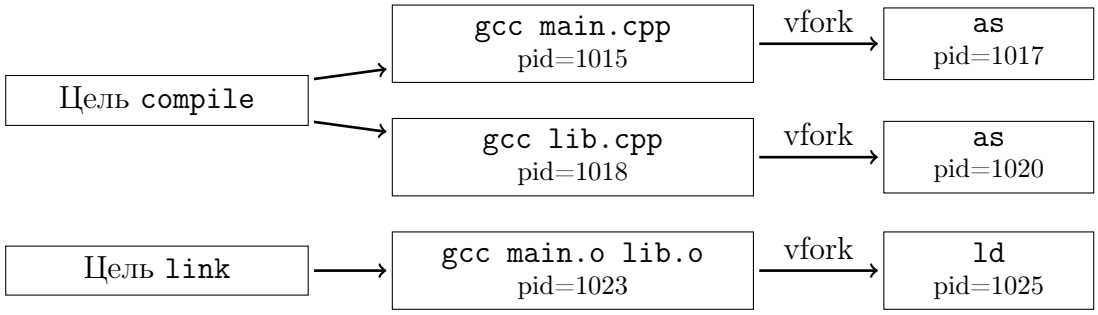

Рис. 4: Дерево процессов при сборке Makefile из листинга 1

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

Рассмотрим процессы 1020 и 1025. Согласно схеме 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 — критическая операция, результат чтения может поменяться если поменять порядок этих операций. 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.

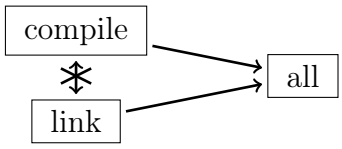

Рис. 5: Граф зависимостей Makefile из листинга 1

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 общем случае выявление конфликтующих доступов — не такая тривиальная задача. Более подробно она будет рассмотрена в подглаве 4.4.

4.1.2 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 жесткие ссылки.

В системе Linux у каждого файла или директории существует ассоциированная с ним index node (inode). Получить её номер можно из поля `st_ino` структуры `stat`. Согласно стандарту ядра, жесткие ссылки внутри одной файловой системы ссылаются на одну и ту же inode [5]. Если речь идёт о нескольких файловых системах, то потребуется обратить внимание ещё и на device number (поле `st_dev` из той же структуры `stat`). В разработанном инструменте это учтено, однако для простоты далее в этой работе device number будет опускаться.

Номера inode могут быть переиспользованы системой, когда все жесткие 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 переиспользовании inode.

Прежде наш алгоритм полагался на парсинг лога 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 жестких 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 «`.`» — жесткую 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;

4.2 Гонка на пути к файлу

Листинг 4: Пример Makefile с гонкой 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
```

В предыдущей главе мы строили алгоритм для ситуации, в которой гонка происходит 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` выбрали одно и то же имя для своих временных файлов. Если бы сборка этих целей была запущена параллельно, то мог бы возникнуть конфликт.

Значительная часть предыдущей главы была уделена борбье с жесткими ссылками. Это связано с тем, что файл может иметь несколько абсолютных путей. Для директорий это неверно, поскольку целью жестких 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 папкой 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.

4.3 Гонка между созданием директории и файла внутри неё

Листинг 5: Пример Makefile с гонкой третьей категории

```
all: build build/a.out

build:
    mkdir -p build
```

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

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` [6]. Предыдущие алгоритмы не помогут в поиске таких гонок.

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 родительской папке при любом обращении к лежащему в ней файлу.

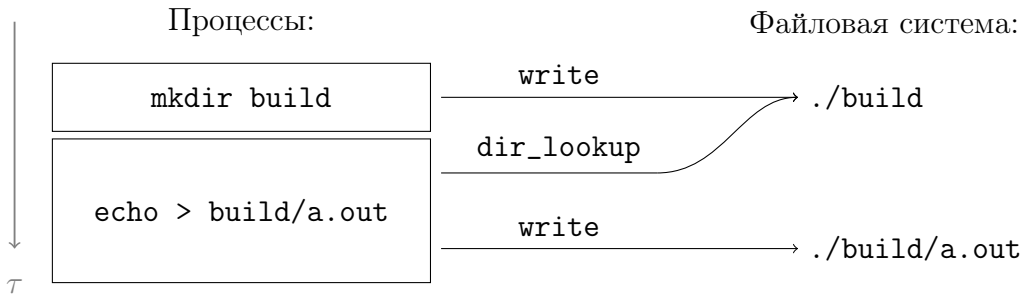

Рис. 6: Операции над файлами при сборке Makefile из листинга 5

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.

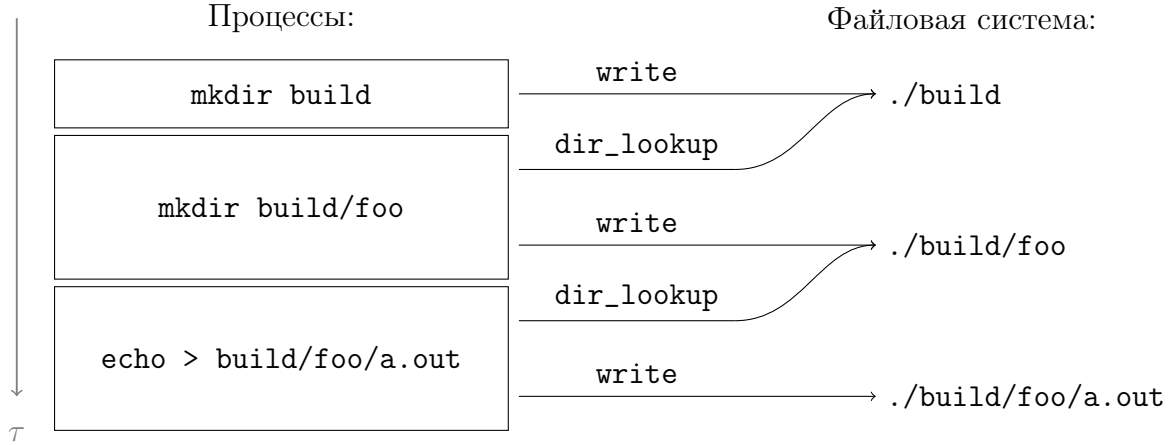

Рис. 7: Операции над файлами для большего числа вложенных директорий

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 соотвествующей 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.

4.3.1 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.

Листинг 6: Пример Makefile с созданием директории build из нескольких целей

```
all: lib1 ... lib9

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

lib9:
    mkdir -p build
    build_library build/lib9.a
```

В Makefile, изображённом на листинге 6, несколько целей самостоятельно создают папку `build`, а затем используют её для сборки. Предложенный выше алгоритм выдаст ложные срабатывания, зафиксировав первую цель `libN`, которая первая создала директорию `build`, и ошибочно потребовав от всех остальных целей `lib(M` $\neq$ `N)` иметь зависимость с `libN`.

В исправленной версии алгоритма операция `dir_access` должна требовать наличие зависимости не с единственным успешным созданием директории, а хотя бы с одной, любой попыткой это сделать. Для этого нужно также модифицировать трассировщик системных вызовов: он должен сообщать о тех `mkdir`, которые завершились с `EEXIST`. Если для наглядности добавить в схему из листинга 6 десятую библиотеку, которая не делает `mkdir -p build`, новый алгоритм найдёт её и корректно сообщит о гонке.

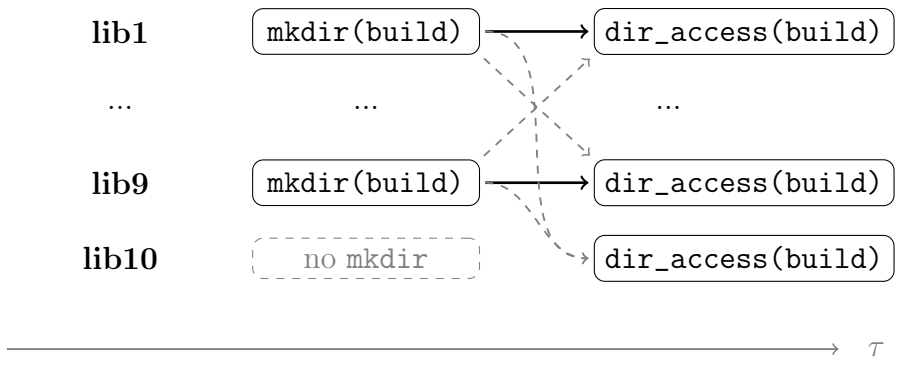

Рис. 8: Обнаружение гонок на основе попыток создания директории

4.4 Поиск конфликтующих доступов

Базовый алгоритм поиска гонок, сформулированный в главе 4.1, 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. 4.1.2)
- `dir_access(path)` — Доступ к директории перед обращением к файлу в ней (см. 4.3)

Рассмотрим гонки на содержимом файла (см. 4.1). 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 жестких ссылок на номер inode. Гонки, вовлекающие операцию удаления, относятся к следующей категории, поэтому в рамках этого пункта будут рассмотрены только операции `read` и `write`.

Для любых двух соседних 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 целую таблицу:

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

Таблица 1: Таблица конфликтов между операциями для гонок типа 4.1

Для работы с любым 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 схеме 9 представлен простой контрпример.

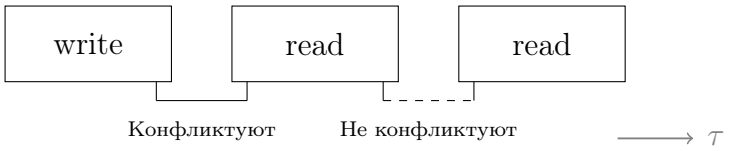

Рис. 9: Контрпример для наивного алгоритма

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. Несмотря на то, что такое решение корректно сработает на примере выше, проверять все пары доступов требует квадратичного времени.

4.4.1 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 10 изображена некоторая последовательность доступов к файлу. Стрелками связаны те доступы, для которых в таблице 1 указан конфликт:

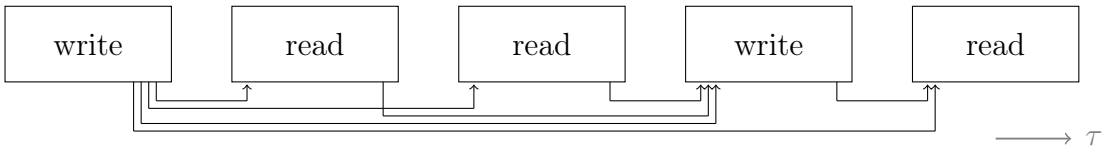

Рис. 10: Последовательность доступов с указанными конфликтами

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:

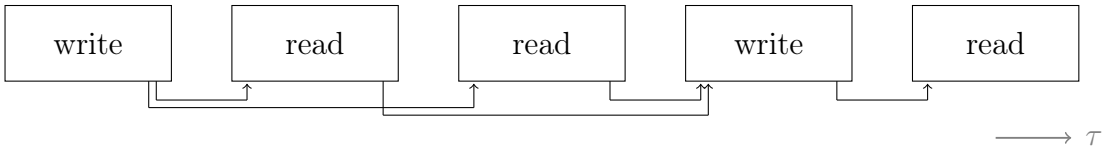

Рис. 11: Последовательность доступов с удалёнными избыточными конфликтами

4.4.2 Доказательство линейности числа проверок

Утверждается, что после удаления всех избыточных проверок количество проверок сократится с квадратичного до линейного. Обозначим проверяемые пары доступов как множество рёбер $P : \{(A_i, A_j), j > i\}$ на вершинах доступов A. По условию, P не содержит избыточных рёбер, то есть $\forall i < j < k (A_i, A_j) \in P \wedge (A_j, A_k) \in P \implies (A_i, A_k) \notin P$. Иными словами, P можно рассматривать как антитранзитивное отношение.

Выделим в A максимальные цепочки последовательных неконфликтующих доступов. Будем обозначать их как $N^k = (N^k_1, N^k_2, \ldots, N^k_n)$. Пустые промежутки между элементами N^k объединим в цепочки $C^k = (C^k_1, C^k_2, \ldots, C^k_n)$. Таким образом, $A = \sqcup A^i$, где каждый A^i является цепочкой либо конфликтующих (C^k), либо неконфликтующих (N^k) соседних доступов. Рассмотрим оба типа цепочек и оценим число рёбер в подграфах, образованных ими.

- Подграф на вершинах $(C^k_1, C^k_2, ...C^k_n)$ содержит только рёбра между соседними доступами C^k_i и C^k_{i+1}. Любые другие рёбра будут избыточными. Общее число рёбер в этом подграфе составит $|C^k| - 1$.

- В цепочке $(N^k_1, N^k_2, ...N^k_n)$ все соседние доступы являются неконфликтующими, а значит, все N_i являются операциями чтения (согласно таблице 1). Следовательно, любые произвольные два элемента из этой последовательности будут неконфликтующими, и число рёбер в подграфе на этих вершинах будет равно нулю.

Рис. 12: Рёбра P на цепочках C и N

По построению, две соседних цепочки A^k и A^{k+1} всегда будут иметь противоположные типы. Рассмотрим случай (C^k, N^{k+1}). Тогда, согласно таблице, последний C_i^k конфликтует со всеми чтениями из N^{k+1}. Поскольку C_i^k является последним конфликтующим доступом перед N^{k+1}, такое ребро не может быть избыточным, следовательно, оно лежит в P. С другой стороны, никакой другой C_n^k нельзя связать ребром напрямую с каким-либо чтением из N^{k+1}. Такое ребро дублировало бы собой существующий путь: $C_n^k \to C_{n+1}^k \to \ldots \to C_i^k \to N_j^{k+1}$. Аналогично, в случае (N^k, C^{k+1}), первый доступ C_j^{k+1} был бы связан ребром со всеми N_i^k.

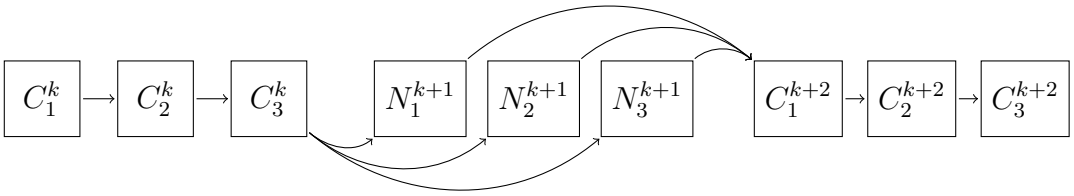

Рис. 13: Рёбра P на последовательности цепочек вида (C, N, C)

Таким образом, между вершинами из соседних цепочек (N^k, C^{k+1}) или (C^k, N^{k+1}) присутствует $|N|$ рёбер. Их наличие гарантирует существование пути между любым элементом из первой и любым элементом из второй цепочки. Поскольку это верно для всех соседних цепочек, а весь набор доступов A состоит из цепочек чередующихся типов, то любые два доступа из любых двух разных цепочек будут связаны односторонним путём. Следовательно, любые другие рёбра между цепочками будут избыточными.

В общей сложности число рёбер в графе не превысит $2 \cdot \Sigma_{N^i}|N^i| + \Sigma_{C^i}|C^i| \leq 2 \cdot \Sigma_{A^i}|A^i| \leq 2 \cdot |A|$.

Более того, такое множество P образует минимальный набор проверок, которые необходимо выполнить для поиска потенциальных гонок. Поскольку никакое ребро в P не является избыточным, удаление любого одного ребра приведёт к тому что два доступа A_i и A_j, которые были им соединены, больше не будут иметь ориентированного пути, и не будут проверены на зависимость, несмотря на то что они должны быть проверены согласно таблице 1.

4.4.3 Введение метода критических доступов и его применение для гонок вида 4.1

Помимо линейности числа доступов, в предыдущем пункте было также показано, какой вид будет иметь интересующее нас множество рёбер для проверки (P):

- Все доступы внутри цепочек вида (C) связаны друг с другом линейно;
- Все доступы из цепочек вида (N) связаны слева и справа с ближайшими соседями из соседних цепочек;

Эти правила легко реализовать алгоритмически. Необходимо лишь знать, где проходят границы между цепочками. Будем называть доступ критическим, если он принадлежит цепочке вида C и некритическим, если он принадлежит цепочке вида N. В предыдущем пункте было замечено, что в контексте поиска гонок вида 4.1, если доступ является некритическим, то он является чтением. В обратную сторону это не всегда верно: если среди двух операций записи встретится одна операция чтения, она не образует цепочку вида N, поскольку будет зависеть от обоих своих соседей.

Однако этот крайний случай не мешает провести однозначное соответствие между видами доступов и критичностью. Даже если на месте единичного чтения вставить N—цепочку длины 1, итоговое множество P не изменится:

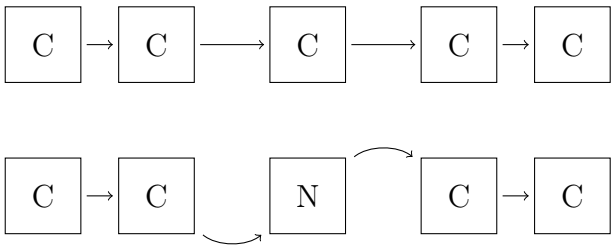

Рис. 14: Независимость множества P от наличия N-цепочек длины 1

Таким образом, если вместо таблицы конфликтов использовать разделение доступов на критические и некритические, то можно проверять лишь минимальный набор зависимостей, имеющий линейный от числа доступов размер. Этот алгоритм и будет применяться в дальшейшем.

4.4.4 Применение метода критических доступов для гонок вида 4.2

Прежде были рассмотрены только гонки на содержимом файла (см. 4.1). Для них критическим был определен доступ `write`, а некритическим — `read`. Остальные виды доступов не рассматривались, поскольку являлись служебными или не были связаны с номерами inode.

В отличие от предыдущего случая, поиск гонок на пути к файлу требует список операций на пути, а не на номере inode (см. 4.2). Среди всего списка доступов (см. 4.4), с путями работают `read`, `write` и `unlink`. Есть также доступ типа `dir_lookup`, но он предназначен только для поиска гонок с созданием директорий (см. 4.3) и не будет рассмотрен здесь.

Осталось разделить выбранные типы доступов на критические и некритические. Обратимся к пункту 4.2 и составим таблицу конфликтов для этого типа гонок.

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

Таблица 2: Таблица конфликтов между операциями для гонок вида 4.2

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 операцей, и следовательно, является критической.

Правильный выбор кофликтующих 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 4.2 (на содержимом файла) и как гонка вида 4.1 (на пути к файлу).

4.4.5 Алгоритм поиска гонок вида 4.3

Согласно пункту 4.3, 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.

5 Описание практической части

Согласно разработанной архитектуре, трассировщик и санитайзер разделены на 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).

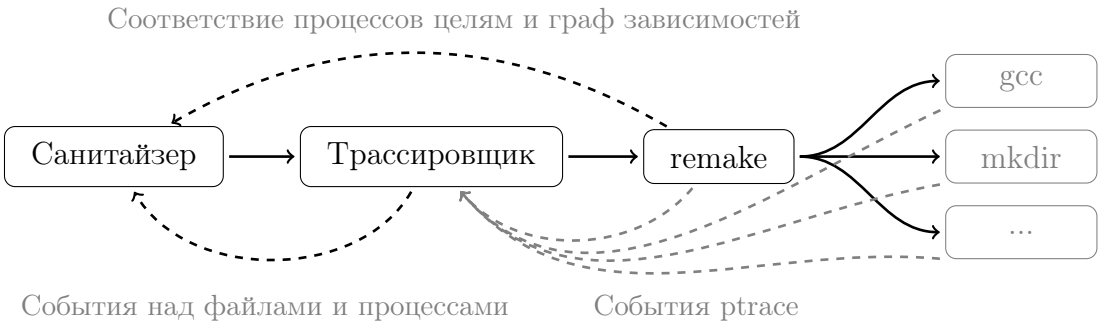

Рис. 15: Схема межпроцессного взаимодействия при сборке под санитайзером

На рис. 15 сплошные стрелки связывают родительские процессы с дочерними, пунктирные стрелки указывают передачу данных через Unix-сокеты, и пунктирные серые стрелки показывают, как трассировщик перехватывает системные вызовы, производимые процессами правее, используя системный вызов `ptrace`.

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++.

5.1 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. 4.1.1).

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 упустить гонку. Поэтому цель, которой соответствует процесс, должна быть получена санитайзером раньше, чем этот процесс будет запущен. Это позволяет сделать 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.

5.2 Обработка вложенных Make

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

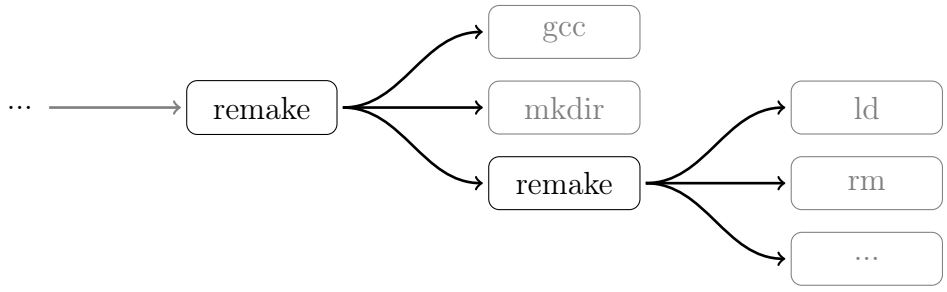

Рис. 16: Дерево процессов при исползовании вложенных Make

Для корректной обработки таких случаев санитайзер производит поиск гонок в контексте каждого Make-процесса отдельно. При этом учитываются те цели и зависимости, которые определены в схеме сборки, обрабатываемой именно этим Make-процессом. Каждый «контекст» производит поиск гонок на основе доступов к файлам, производимым ниже по дереву процессов.

Рассмотрим процесс `ld` из рис. 16. Предположим, что он был порождён целью сборки `target2`, а вложенный Make, являющийся его родительским процессом, порождён целью `target1` во внешнем Make, как на рис. 17.

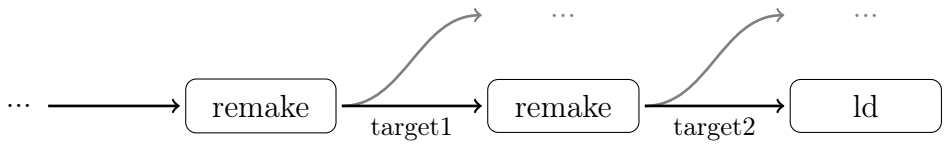

Рис. 17: Пример распространения событий при вложенных Make

Если процесс `ld` на рис. 17 произведёт чтение файла `libfoo.so`, то для вложенного Make этот доступ будет зарегистрирован как доступ к `libfoo.so` из цели `target2`, а для внешнего Make этот доступ будет зарегистрирован как доступ из цели `target1`.

Если бы система Linux позволяла устанавливать процессу множество отладчиков, то предложенный подход был бы эквивалентен запуску отдельной пары трассировщик-санитайзер для каждого вложенного Make. Это бы приравняло запуск вложенного Make к запуску обычного процесса. Все доступы к файлам, которые бы производились ниже, в контексте этого санитайзера присваивались бы к той цели, которая породила этот вложенный Make.

Альтернативное обоснование выбранного подхода следует из соображения о том, что любая цель, собираемая во вложенном Make, имеет зависимость от цели, которая породила сам вложенный Make-процесс. Честная проверка наличия зависимости между целью `A` из вложенного Make и целью `B` из внешнего Make сводилась бы к проверке наличия зависимости между целью, породившей вложенный Make и целью `B`.

5.3 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.

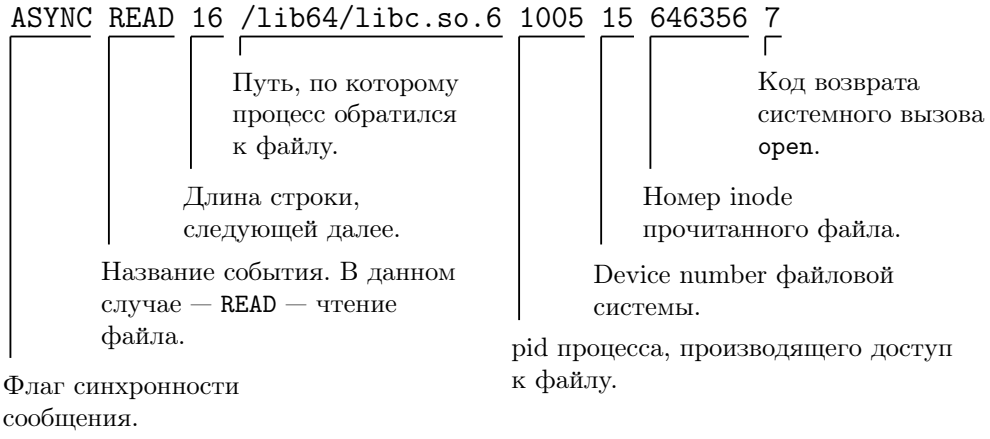

Рис. 18: Пример сообщения, передаваемого санитайзеру трассировщиком

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. В пункте 5.3 был приведён пример 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. 5.3.
3. `READ` <Путь к файлу> `<pid> <devnum> <inum>` <код возврата> — событие открытия процессом файла на чтение.
4. `WRITE` <Путь к файлу> `<pid> <devnum> <inum>` <код возврата> — событие открытия процессом файла на запись.
5. `UNLINK` <Путь к файлу> `<pid> <devnum> <inum>` <код возврата> — событие удаления пути (directory entry)
6. `INODE_UNLINK` <Путь к файлу> `<pid> <devnum> <inum>` <код возврата> — дублирует сообщение `UNLINK` в случае, если удалённый путь был последней жесткой ссылкой на свою inode.
7. `DIE <pid>` — синхронное сообщение. 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. 5.3.
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` [7]. 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. [7].

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

5.4 Режим интерактивной отладки

При отладке состояний гонок разработчикам часто требуется многократно пересобирать проект, устанавливая отладочные выводы на сборке определённых целей. В разработанном инструменте предусмотрен режим интерактивной отладки. Он позволяет экономить время одновременно двумя способами:

1. Поскольку санитайзер опирается только на получаемые им сообщения, выполнять поиск гонок можно отложенно. Если перенаправить все входящие сообщения в файл, и уже после передать их на вход санитайзера, то результат его работы будет таким же, как и если бы сборка происходила прямо сейчас. Такое «воспроизведение» записи занимает значительно меньше времени, чем обычная пересборка проекта.

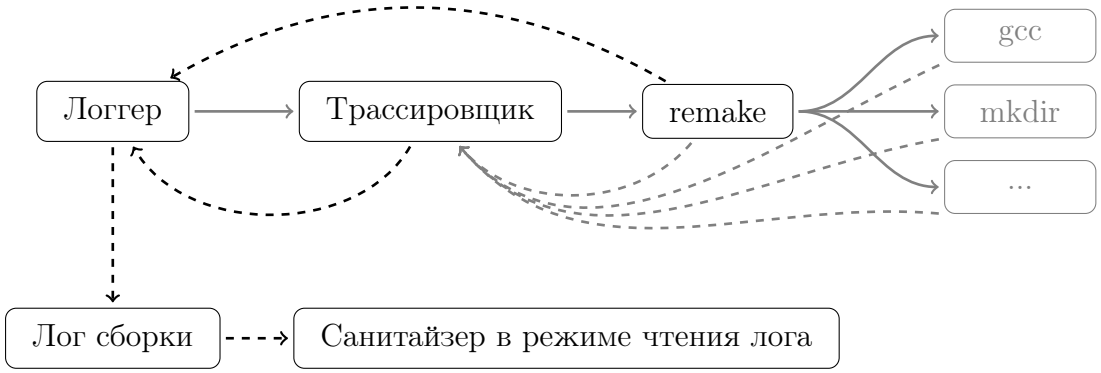

Рис. 19: Запись и воспроизведение лога сборки

   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.

2. Режим отладки позволяет устанавливать точки останова при обнаружении гонки или на определённых действиях с файлами. Разработчику не требуется искать и открывать нужный Makefile и искать цели, работающие с этими файлами.

Для реализации точек останова в санитайзер был добавлен режим отладки. При его активации, все передаваемые по сокету сообщения становятся синхронными. При срабатывании точки останова на определённом доступе к файлу санитайзер перестаёт отправлять `ACK`-сообщения. Трассировщик не позволит процессу продолжить работу, пока не получит ответное сообщение, и сборка остановится.

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 этот доступ;
- Процесс, непосредственно производящий 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.

Точка останова может быть привязана к произвольному множеству путей, задаваемым одним или нескольскими 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»).

5.5 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, производящиеся 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 [3]. 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

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. Воизбежание 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` [8]. 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 10 новых.

5.6 Пример новой обнаруженной гонки

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

Листинг 7: Диагностика санитайзера при сборке проекта Vim

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

Диагностика, изображенная на листинге 7, сообщает о гонке на пути к файлу (4.1). Как было замечено в пункте 3, гонки этой категории нельзя обнаружить режимом `-shuffle`. Обратимся к рецептам целей, указанных в диагностике (`src/po/Makefile:216`).

Листинг 8: Фрагмент схемы сборки Vim

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

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

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 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 и доказана его корректность применительно к обнаружению гонок двух типов. 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 2:

- 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 путём использования модифицированного Make;
- В сравнении с Make `-shuffle`, санитайзер требует произвести лишь единоразовую сборку проекта. Отладка гонок может производиться отложенно с использованием собранной трассы, что значительно сокращает время ожидания.

Список литературы

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

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

[3] *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].

[4] 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].

[5] Dynamic Structures. — https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html?highlight=inode#directory-entries. — [Online; accessed 18-March-2024].

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

[7] socket(2) — Linux manual page. — https://man7.org/linux/man-pages/man2/socket.2.html. — [Online; accessed 6-April-2024].

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

Приложение

6.1 Патч для remake, реализующий печать соответствий pid и целей сборки

Патч применяется к remake 4.3, commit 7619a01217cf84c409a3ebc98fd3a732f72a4ce6

```
diff --git a/src/function.c b/src/function.c
index 6a578ada..3dc44079 100644
--- a/src/function.c
+++ b/src/function.c
@@ -1712,7 +1712,7 @@ func_shell_base (char *o, char **argv, int trim_newlines)
     child.output.err = errfd;
     child.environment = envp;

-    pid = child_execute_job (&child, 1, command_argv);
+    pid = child_execute_job (&child, 1, command_argv, NULL);

     free (child.cmd_name);
   }
diff --git a/src/job.c b/src/job.c
index a4a40df4..fae40f94 100644
--- a/src/job.c
+++ b/src/job.c
@@ -1277,7 +1277,8 @@ start_job_command (child_t *child,
       jobserver_pre_child (flags & COMMANDS_RECURSE);

       child->pid = child_execute_job ((struct childbase *)child,
-                                      child->good_stdin, argv);
+                                      child->good_stdin, argv,
+                                      child->file->name);

       environ = parent_environ; /* Restore value child may have clobbered. */
       jobserver_post_child (flags & COMMANDS_RECURSE);
@@ -1998,12 +1999,13 @@ start_waiting_jobs (target_stack_node_t *p_call_stack)
    Create a child process executing the command in ARGV.
    Returns the PID or -1. */
 pid_t
-child_execute_job (struct childbase *child, int good_stdin, char **argv)
+child_execute_job (struct childbase *child, int good_stdin, char **argv, char*
    target_name)
 {
   const int fdin = good_stdin ? FD_STDIN : get_bad_stdin ();
   int fdout = FD_STDOUT;
   int fderr = FD_STDERR;
   pid_t pid;
+  pid_t ppid = getpid();
   int r;
 #if defined(USE_POSIX_SPAWN)
   char *cmd;
@@ -2026,6 +2028,8 @@ child_execute_job (struct childbase *child, int good_stdin,
    char **argv)
   pid = vfork();
   if (pid != 0)
     return pid;
+  else if (target_name)
```

```
+    printf("remake: Spawned process, ppid=%d, pid=%d, target=%s\n", ppid, getpid()
     , target_name);

   /* We are the child. */
   unblock_all_sigs ();
diff --git a/src/job.h b/src/job.h
index eaf6f8fd..ab3fa0a6 100644
--- a/src/job.h
+++ b/src/job.h
@@ -82,7 +82,8 @@ extern void start_waiting_jobs (target_stack_node_t *
    p_call_stack);
 char **construct_command_argv (char *line, char **restp, struct file *file,
                                int cmd_flags, char** batch_file);

-pid_t child_execute_job (struct childbase *child, int good_stdin, char **argv);
+pid_t child_execute_job (struct childbase *child, int good_stdin, char **argv,
+                         char* target_name);

 // void exec_command (char **argv, char **envp) NORETURN;
 void exec_command (char **argv, char **envp);
```