

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	3
2 Постановка задачи	4
3 Обзор существующих решений	4
4 Исследование и построение решения задачи	5
4.1 Гонка на содержимом файла	5
4.1.1 Мотивация использования номеров inode	7
4.2 Гонка на пути к файлу	8
4.3 Гонка между созданием директории и файла внутри неё	9
4.3.1 Обработка множественных попыток создания директории	10
4.4 Поиск конфликтующих доступов	11
4.4.1 Улучшенный алгоритм перебора доступов	12
4.4.2 Доказательство линейности числа проверок	13
4.4.3 Введение метода критических доступов и его применение для гонок вида 4.1	14
4.4.4 Применение метода критических доступов для гонок вида 4.2	15
4.4.5 Алгоритм поиска гонок вида 4.3	16
5 Описание практической части	16
6 Заключение	16
Приложение	17
6.1 Патч для remake, реализующий печать соответствий pid и целей сборки .	17

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.

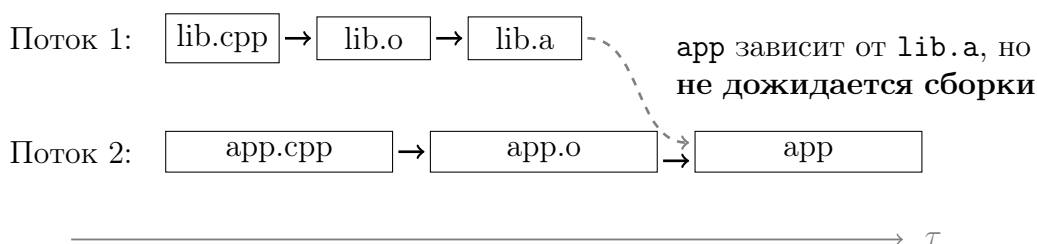


Рис. 1: Процесс сборки проекта с состоянием гонки в схеме сборки

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.

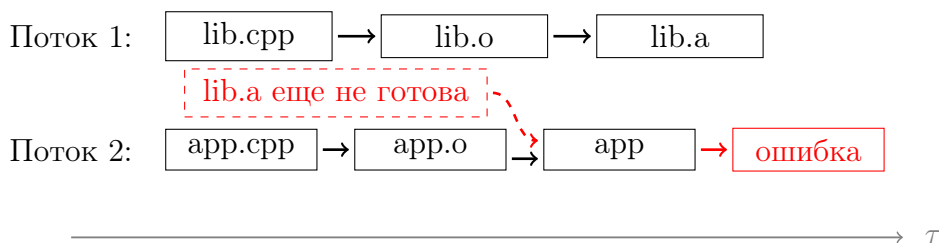


Рис. 2: Процесс сборки проекта с состоянием гонки в схеме сборки

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

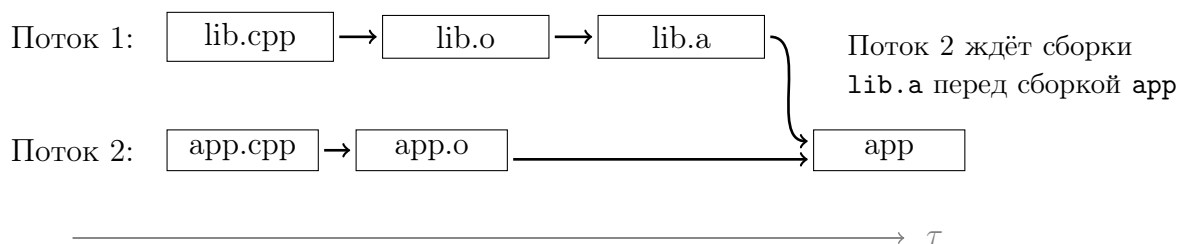


Рис. 3: Исправленная схема сборки без состояния гонки

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.
- Поиск гонок не должен отнимать у разработчика много времени. Не должны требоваться многократные пересборки проекта или отключение многопоточности (`-j1`), не должен значительно замедляться сам процесс сборки проекта.

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 скопировать объектные файлы, которых ещё не существует, или использовать старый, ещё не обновлённый 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.

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`.

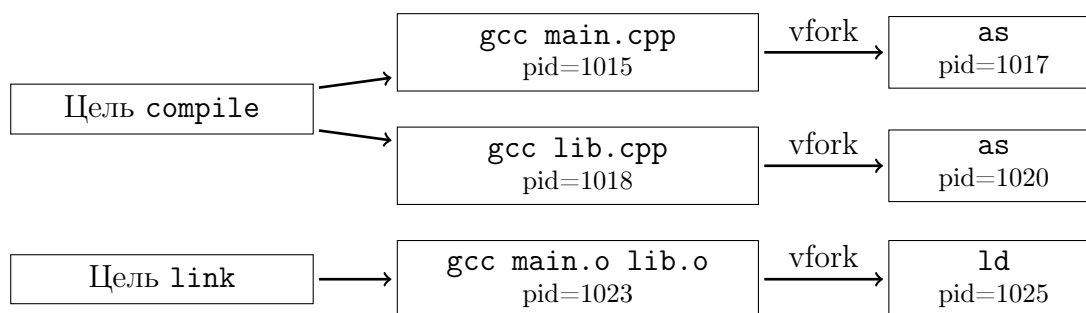


Рис. 4: Дерево процессов при сборке Makefile из листинга 1

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.

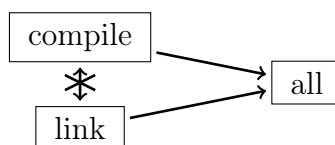


Рис. 5: Граф зависимостей Makefile из листинга 1

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

4.1.1 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 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 потребуются 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. Таким образом, можно заменить звено `strace` на собственный трассировщик на основе `ptrace`. Он позволит производить более сложные проверки и составлять более информативные логи.

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 `gmake` 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 борьбе 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.

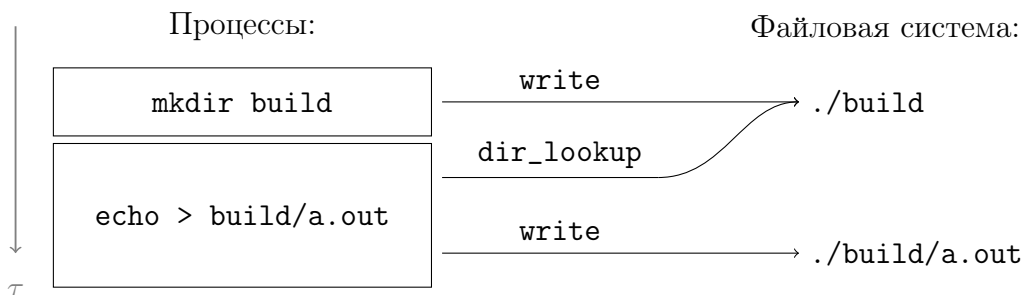


Рис. 6: Операции над файлами при сборке Makefile из листинга 5

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.

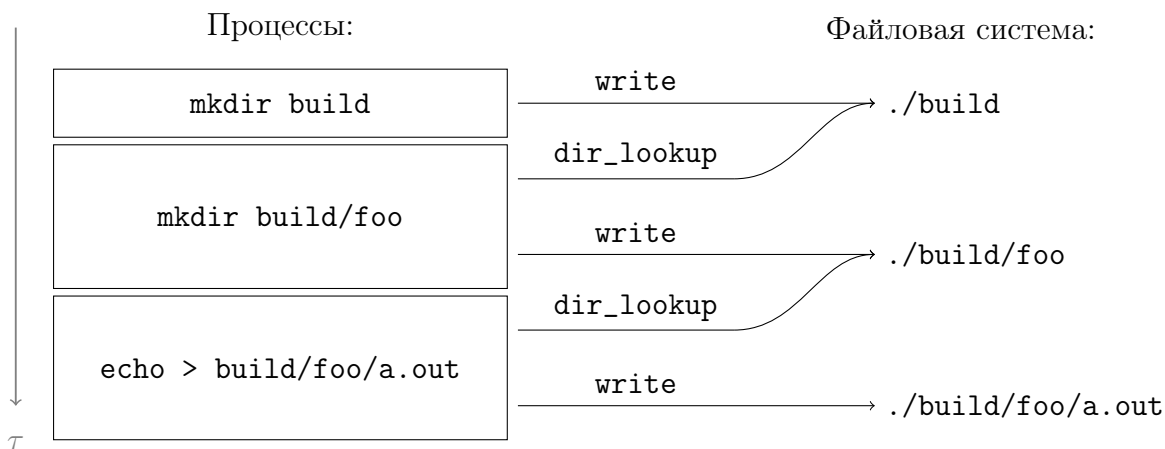


Рис. 7: Операции над файлами для большего числа вложенных директорий

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 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.

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 ≠ 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.

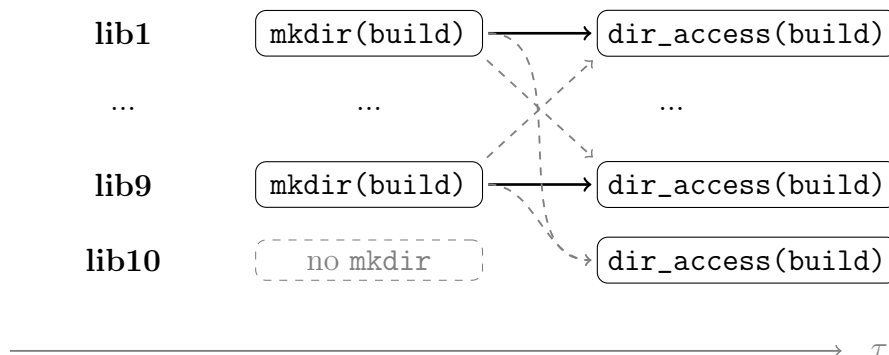


Рис. 8: Обнаружение гонок на основе попыток создания директории

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. 4.1.1)
- `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 номер 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.

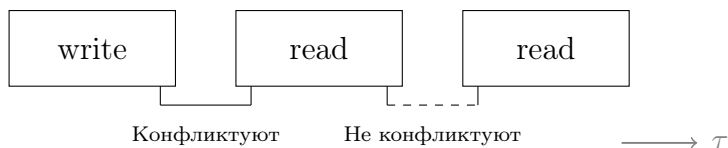


Рис. 9: Контрпример для наивного алгоритма

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:

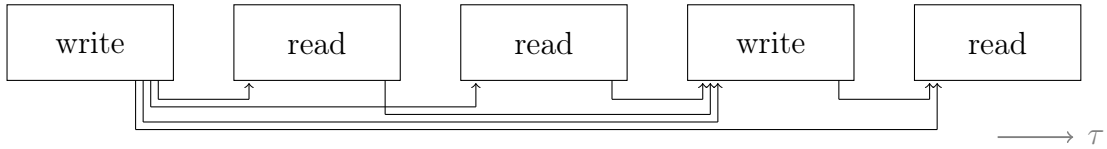


Рис. 10: Последовательность доступов с указанными конфликтами

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:

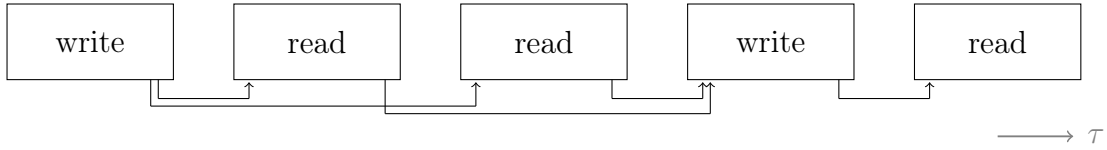


Рис. 11: Последовательность доступов с удалёнными избыточными конфликтами

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 $N^k = (N_1^k, N_2^k, \dots, N_n^k)$. Пустые промежутки между элементами N^k объединим в цепочки $C^k = (C_1^k, C_2^k, \dots, C_n^k)$. 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 $(C_1^k, C_2^k, \dots, C_n^k)$ содержит только рёбра между соседними доступами C_i^k и C_{i+1}^k . Любые другие рёбра будут избыточными. Общее число рёбер в этом подграфе составит $|C^k| - 1$.
- В цепочке $(N_1^k, N_2^k, \dots, N_n^k)$ 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: $C_n^k \rightarrow C_{n+1}^k \rightarrow \dots \rightarrow C_i^k \rightarrow N_j^{k+1}$. Аналогично, в случае (N^k, C^{k+1}) , первый доступ C_j^{k+1} был бы связан ребром со всеми N_i^k .

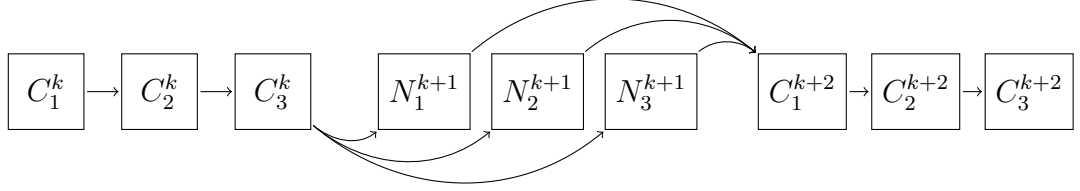


Рис. 13: Рёбра P на последовательности цепочек вида (C, N, C)

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 $2 \cdot \sum_{N^i} |N^i| + \sum_{C^i} |C^i| \leq 2 \cdot \sum_{A^i} |A^i| \leq 2 \cdot |A|$.

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:

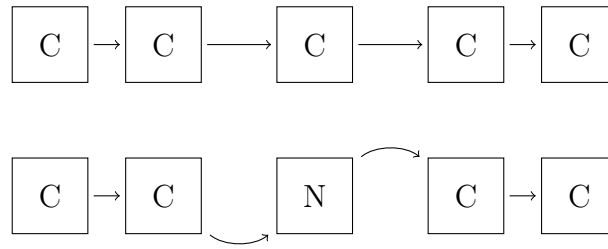


Рис. 14: Независимость множества P от наличия N -цепочек длины 1

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 дальнейшем.

4.4.4 Применение метода критических доступов для гонок вида 4.2

Прежде был рассмотрены только гонки на содержимом файла (см. 4.1). Для него критическим был определен доступ **write**, а некритическим — **read**. Остальные виды доступов не рассматриваются этим алгоритмом, поскольку являются служебными или не несут информации о номерах 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 критичности между доступами 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 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, гонку на пути смогли бы образовать две операции записи. На практике это привело бы к нежелательному дублированию: такая гонка обнаружилась бы дважды, как гонка на содержимом файла и как гонка на пути к файлу.

4.4.5 Алгоритм поиска гонок вида 4.3

Как было описано в пункте 4.3, для поиска этой гонки этой категории требуется особый набор проверок, связанных через логическое «или». Метод критических доступов не подходит для этого.

С другой стороны, алгоритм для поиска достаточно очевидный: 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

6 Заключение

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].

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);
```