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.

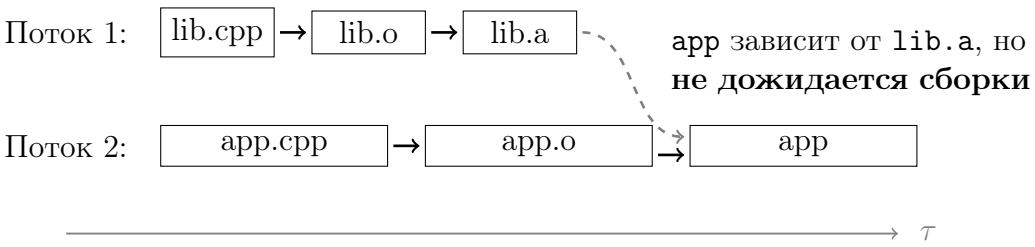
Содержание

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.

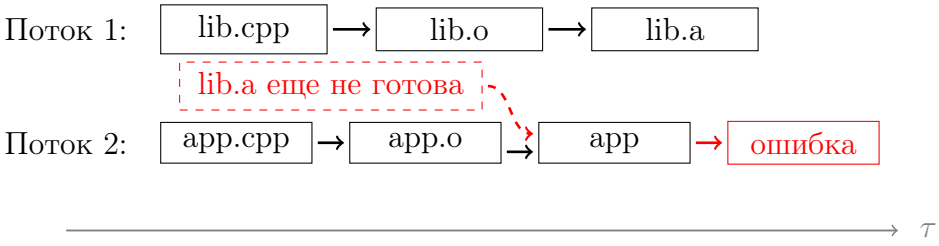
Рис. 1: Процесс сборки проекта с состоянияем гонки в схеме сборки



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.

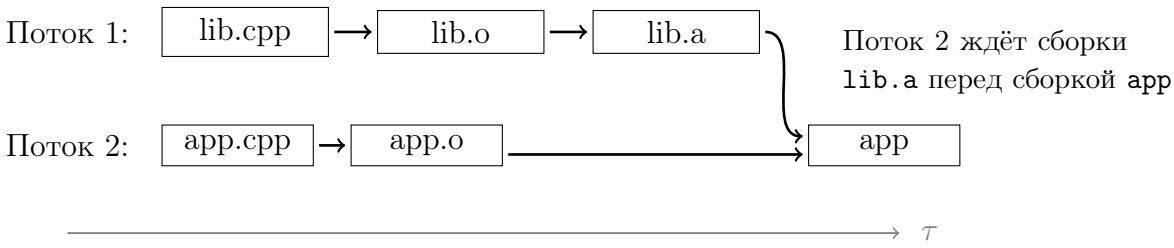
На рисунке сверху это не приводит к ошибке, поскольку библиотека сама собой успела собраться быстрее, чем она потребовалась. Однако, это не всегда может быть так. Выход из строя секторов диска, расширение самой библиотеки и множество других непредсказуемых причин могут привести к увеличению времени сборки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

Рис. 2: Ошибка при сборке 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.
- Поиск гонок не должен отнимать у разработчика много времени. Не должны требоваться многократные пересборки проекта или отключение многопоточности (`-j1`).

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.

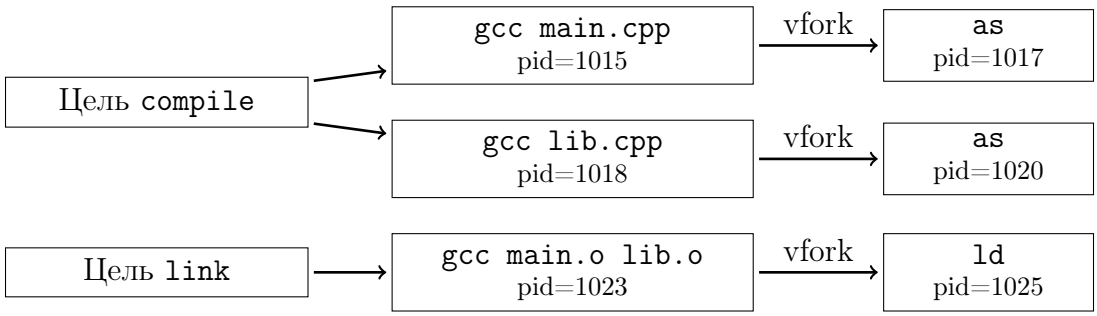
Чтобы сопоставить номера процессов с целями сборки предлагается модифицировать саму утилиту Make. В качестве подопытного был взят проект remake. Он реализует тот же функционал, что и GNU Make, но требует значительно меньше усилий для сборки из исходного кода. После внесения изменений (см. приложение 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`.

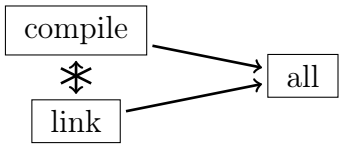
Рис. 4: Дерево процессов при сборке Makefile из листинга 1



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. Из схемы выше, им соответствуют цели `compile` и `link` соответственно. В то же время из фрагмента лога 3 можно установить, что процесс 1020 производит запись в файл `lib.o`, а процесс 1025 - чтение того же файла. Запись в файл и чтение файла нельзя менять местами, иначе результат чтения может измениться. 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 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 ассоцированная 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 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. Таким образом, можно заменить звено strace на собственный трассировщик на основе `ptrace`. Он позволить 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 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.

Если в некоторой директории d два разных файла были доступны по имени n в разные моменты времени, это означает, что между этим было произведено удаление и повторное создание этого файла. Поскольку к этой директории существует единственный путь, его удаление и создание не могло быть произведено ни по какому пути, кроме d/n. Следовательно, для поиска таких гонок достаточно искать зависимости между целями, которые удаляют и создают файлы по одному и тому же пути. В этом случае пользоваться номерами inode уже не нужно.

4.3 Гонка между созданием директории и файла внутри неё

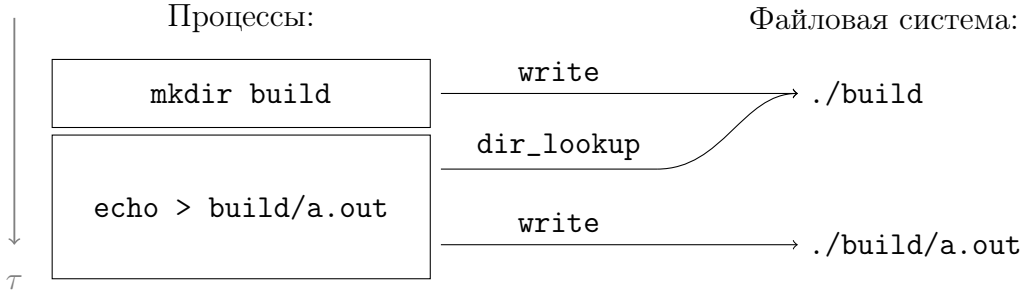
Листинг 5: Пример Makefile с гонкой третьей категории

```
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.

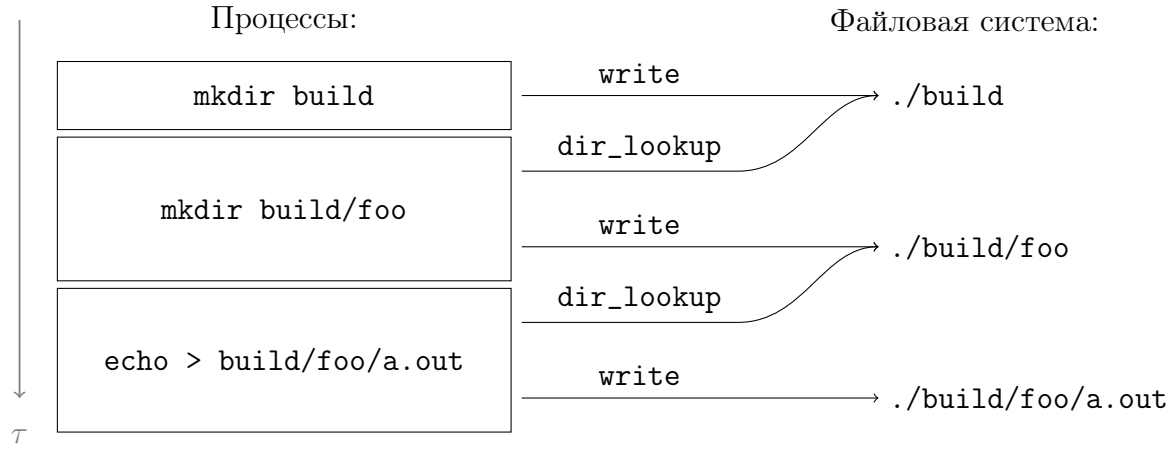
Директория и файл внутри неё — разные элементы файловой системы, имеющие разные пути и разные номера inode, поэтому предыдущие алгоритмы не смогут обнаружить эту гонку. Простое решение — фиксировать доступ специального вида (directory lookup) к родительской папке при любом обращении к лежащему в ней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

Однако на практике этот 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

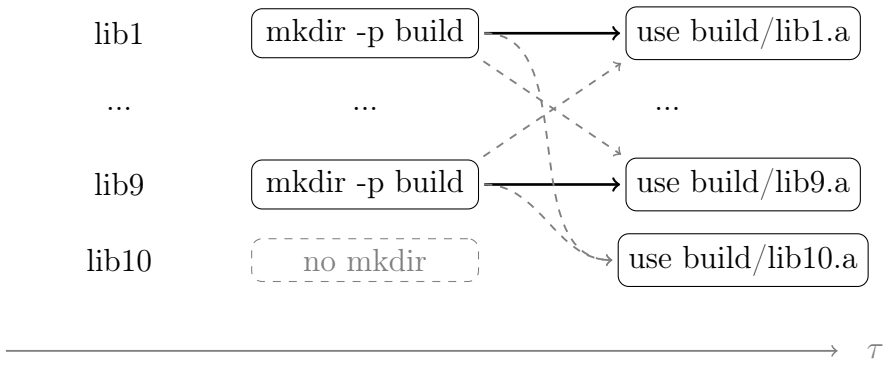
```
all: lib1 ... lib9

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 корректированной 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



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);
```