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.

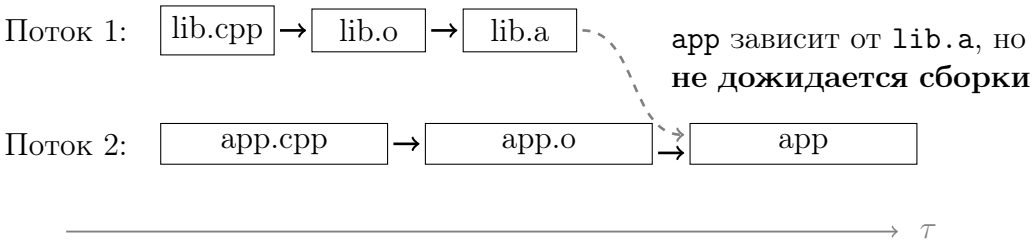
Содержание

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 являются зависимости между целями сборки. Отсутствие необходимой зависимости может привести к состоянию гонки, аналогично отсутствующей синхронизации между процессами.

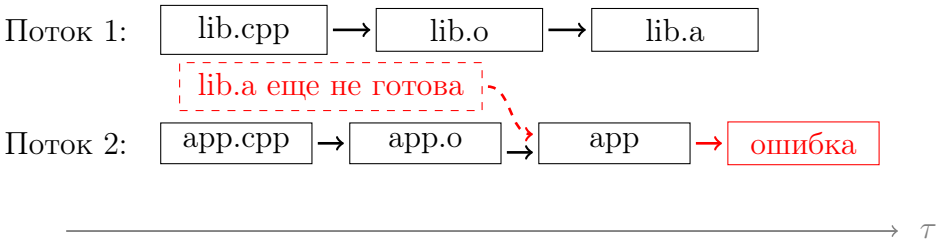
Рис. 1: Процесс сборки проекта с состоянияем гонки в схеме сборки



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.

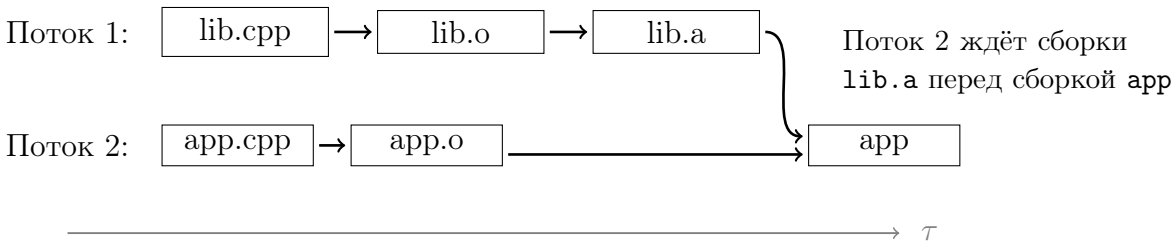
Если бы для сборки библиотеки потребовалось бы больше времени, это бы привело к ошибке сборки всего проекта. Такое может произойти по непредсказуемым причинам, таким как расширение самой библиотеки, выход из строя секторов диска, или высокая нагрузка на систему.

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 с этим состояния гонки могут долго оставаться неисправленными. Как будет продемонстрировано в дальнейшем в этой работе, даже такие проекты, как Vim и Strace, над которыми трудятся тысячи разработчиков по всему миру, имеют состояния гонок в своих схемах сборки.

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

Процесс поиска состояний гонок в схемах сборки является сложным и трудоёмким. Цель этой работы — предоставить решение, которое бы позволило упростить этот процесс. 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

Меры для борьбы с гонками уже принимаются в современных системах сборки. Например, система Bazel создаёт песочницу — виртуальную файловую систему, отдельную для каждой цели. В этой песочнице есть только те файлы, которые соответствуют зависимостям этой цели сборки [1]. 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`, добавленный в GNU Make 4.4 в 2022 году [2]. 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 [3], 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, это решение имеет ограничение в том, что не может обнаружить гонки, проявляющиеся только при параллельном выполнении целей. Распространённая причина появления таких гонок заключается в том, что несколько независимых целей используют 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". Режим `--shuffle` утилиты `Make` не помогает обнаружить гонки этого класса. Он нацелен на изменение порядка сборки целей, и это не позволяет проверить, может ли любой поднабор независимых целей работать одновременно без конфликтов.

4 Исследование и построение решения задачи

Самые распространённые гонки, встречающиеся в реальных проектах, можно разделить на три категории. Ниже они рассмотрены по отдельности.

4.1 Гонка на содержимом файла

Листинг 1: Пример Makefile с гонкой на содержимом файла race_file

```
all: write_first_part write_second_part
write_first_part:
	echo 'a' > file.out
write_second_part:
	echo 'b' >> file.out
```

В этом примере цели `write_first_part` и `write_second_part` записывают текст в один и тот же файл. Между этими целями нет зависимостей, соответственно, они могут быть исполнены в любом порядке. При многопоточной сборке содержимое результирующего файла `file.out` будет неопределённым (либо `"ab"`, либо `"a"`).

Аналогичная проблема может произойти, если цели `write_a` и `write_b` будут использовать жёсткие ссылки на файл `file.out`. Тогда несмотря на то, что при сборке записи будут произведены по разным путям, они всё равно образуют состояние гонки на содержимом файла. Из этого следует, что при поиске таких гонок важно учитывать не пути к файлам, а идентификатор содержимого файла. В качестве такого идентификатора предлагается использовать номер inode.

4.2 Гонка на пути к файлу

Этот класс гонок отличается тем, что для его поиска необходимо использовать пути к файлам, а не номера inode.

Листинг 2: Пример Makefile с гонкой на пути

```
all: file copy_1 copy_2
file:
	echo 'a' > file
copy_1: file
	cp file file\_copy
	# Something is done with file_copy
	rm file_copy
copy_2: file
	cp file file_copy
	# Something else is done with file_copy
	rm file_copy
```

В этом примере между целями `copy_1` и `copy_2` нет зависимости, однако они создают и удаляют файл по одному и тому же пути. Если бы сборка этих целей была запущена параллельно, то мог бы возникнуть конфликт.

При пересоздании файла его номер inode может измениться. В этом примере при сборке целей `copy_1` и `copy_2` файл `file_copy` имеет различные номера inode, поэтому обнаружить гонку, полагаясь только на эту информацию, невозможно. Для обнаружения гонок с участием удаления необходимо полагаться именно на пути к файлам.

4.3 Гонка между созданием директории и файла внутри

Листинг 3: Пример Makefile с гонкой третьей категории

```
all: build build/file.out

build:
    mkdir -p build

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

5 Описание практической части

6 Заключение

Список литературы

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

[2] *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].

[3] 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].