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.

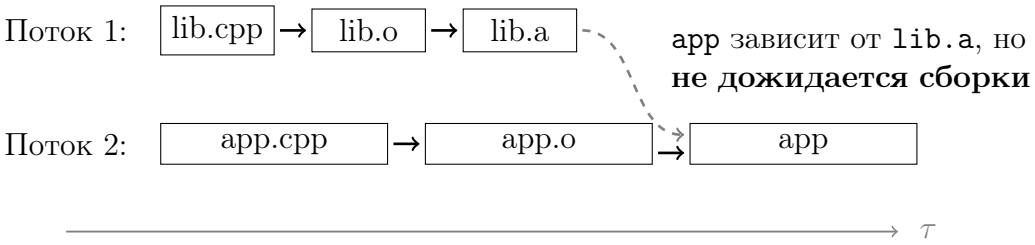
Содержание

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.

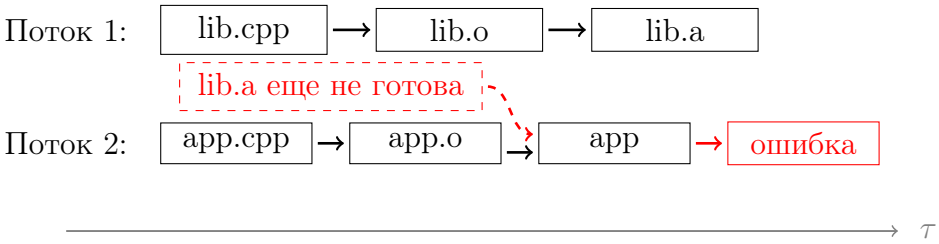
Рис. 1: Процесс сборки проекта с состоянияем гонки в схеме сборки



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.

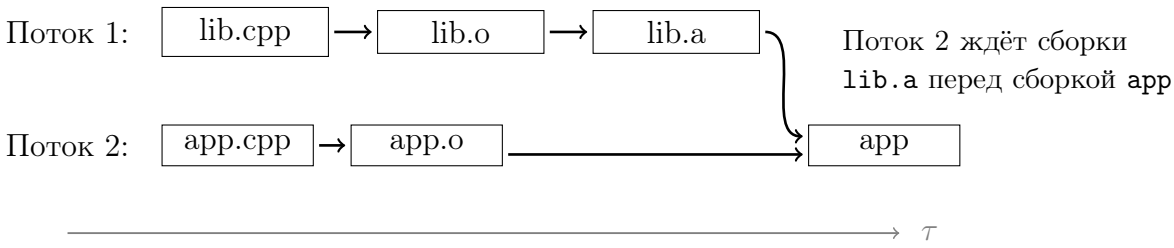
Если бы для сборки библиотеки потребовалось бы больше времени, это бы привело к ошибке сборки всего проекта. Такое может произойти по непредсказуемым причинам, таким как расширение самой библиотеки, выход из строя секторов диска, или высокая нагрузка на систему.

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. 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

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 [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`, добавленный в GNU Make 4.4 в 2022 году [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, 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. 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 файла file.out

```
all: write_a append_b

write_a:
	echo 'a' > file.out

append_b:
	echo 'b' >> file.out
```

В этом примере цели `write_a` и `write_b` 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"`).

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 - это запустить его под утилитой strace.

Листинг 2: Вывод утилиты strace при сборке Makefile из листинга 1

```
$ strace -f -e trace=%file make
...
strace: Process 1060 attached
[pid 1060] execve("/bin/sh", ["/bin/sh", "-c", "echo 'a' > file.out"], /* ... */)
    = 0
...
[pid 1060] openat(AT_FDCWD, "file.out", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
[pid 1060] +++ exited with 0 +++
...
strace: Process 1061 attached
[pid 1061] execve("/bin/sh", ["/bin/sh", "-c", "echo 'b' >> file.out"], /* ... */)
     = 0
...
[pid 1061] openat(AT_FDCWD, "file.out", O_WRONLY|O_CREAT|O_APPEND, 0666) = 3
[pid 1061] +++ exited with 0 +++
...
```

В полученном логе можно видеть, как процессы 1060 и 1061 открывают file.out с помощью системного вызова `openat`. Однако этой информации мало: неизвестно, какие цели сборки скрываются за этими номерами.

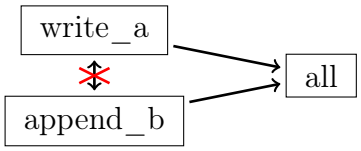
Чтобы сопоставить номера процессов с целями сборки предлагается модифицировать саму утилиту Make. В качестве подопытного был взят проект remake. Он реализует тот же функционал, что и GNU Make, но требует значительно меньше усилий для сборки из исходного кода. После внесения изменений (см. приложение 6.1) в логе появилась недостающая информация:

Листинг 3: Лог сборки Makefile из листинга 1 с модифицированным remake

```
$ strace -f -e trace=%file make
...
strace: Process 1060 attached
remake: Spawned process, ppid=1059, pid=1060, target=write_a
[pid 1060] execve("/bin/sh", ["/bin/sh", "-c", "echo␣'a'␣>␣file.out"], /* ... */)
    = 0
...
[pid 1060] openat(AT_FDCWD, "file.out", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
[pid 1060] +++ exited with 0 +++
...
strace: Process 1061 attached
remake: Spawned process, ppid=1059, pid=1061, target=append_b
[pid 1061] execve("/bin/sh", ["/bin/sh", "-c", "echo␣'b'␣>>␣file.out"], /* ... */)
     = 0
...
[pid 1061] openat(AT_FDCWD, "file.out", O_WRONLY|O_CREAT|O_APPEND, 0666) = 3
[pid 1061] +++ exited with 0 +++
...
```

Обработав полученный лог примитивным скриптом, можно установить, что процессы 1060 и 1061, которым соответствуют цели `write_a` и `append_b`, производят запись по одному и тому же пути - file.out. Из этого следует, что воизбежание гонок указанные цели должны выполняться в одном и том же порядке. Иными словами - между ними должна быть зависимость.

Рис. 4: Граф зависимостей Makefile из листинга 1



Легко убедиться в том, что схема сборки из примера не содержит такой зависимости: между целями `write_a` и `append_b` нет ориентированного пути, соответственно гонка имеет место. Таким образом, шаг за шагом, был построен алгоритм автоматического поиска состояний гонок первой категории:

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 графе зависимостей существуют зависимости между целями, производящими конфликтующие доступы;

В таком виде у алгоритма есть одно ограничение. Если цели `write_a` и `append_b` будут использовать жёсткие ссылки на файл `file.out` (например, `file.out.link1` и `file.out.link2`), состояние гонки будет по-прежнему присутствовать, но в логе доступов будут фигурировать пути от разных жёстких ссылок. Поскольку алгоритм полагается на совпадение путей, эта гонка не будет выявлена. Чтобы это исправить, нужно использовать некоторый «уникальный идентификатор» файла, который позволит прозрачно сравнивать между собой жесткие ссылки.

В Linux в качестве такого идентификатора можно использовать номер index node (inode). Узнать его можно через системный вызов `stat`, обратившись к полю `st_ino`. 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 далее в этой работе device number будет опускаться для простоты.

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 4: Пример Makefile с гонкой на пути

```
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 этом примере между целями `something` и `something_else` 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 `something` и `something` файл `tmp_file` 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 неё

Листинг 5: Пример Makefile с гонкой третьей категории language

```
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] 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].

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);
```