

# **Инструмент автоматического обнаружения гонок в параллельных сборках на основе Make**

Метод и реализация

Артем Климов, Владислав Иванишин, Александр Монаков

4 декабря 2023 г.



- **Введение.** Состояния гонок в Makefile. Примеры и формулировка проблемы.
- **Теоретический метод** автоматического обнаружения состояний гонок.
- **Реализация** метода в виде санитизатора для Make.
- **Оценка** санитизатора на реальных проектах, **сравнение** с существующими решениями. **Заключение.**

## **Введение**

**Состояние гонки** — ситуация, когда несколько потоков работают с одним и тем же ресурсом одновременно, и результат выполнения программы зависит от порядка выполнения потоков.

- Одновременное чтение и запись в одно и то же место памяти.
- Создание каталога и одновременная работа с его содержимым.
- Удаление ресурса без ожидания окончания его использования.

## Введение: Состояния гонок

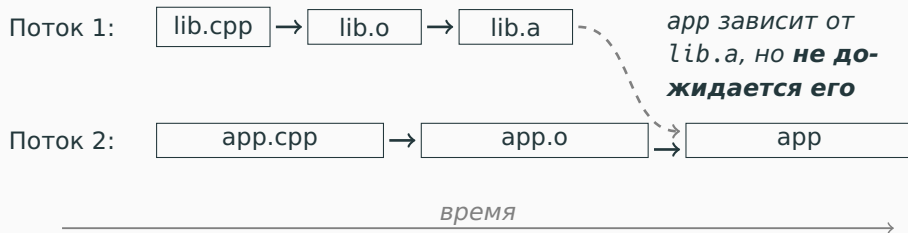
Пример: Два потока пытаются увеличить переменную counter, если она меньше 1.

```
1  std::atomic<int> counter = 0;
2
3  void thread1() {
4      if (counter < 1)
5          counter++;
6  }
7
8  void thread2() {
9      if (counter < 1)
10         counter++;
11 }
```

Если выполнение будет запланировано неудачно, counter может быть увеличено дважды, и окончательное значение будет равно 2.

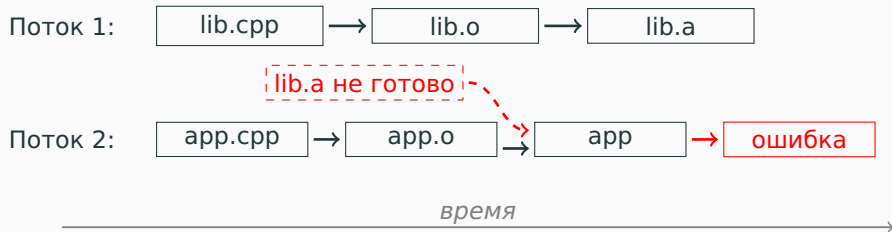
## Введение: Состояния гонки в схемах сборки

- Системы сборки могут компилировать несколько целей одновременно.
- Если две цели взаимодействуют с одним и тем же файлом во время параллельной сборки, может возникнуть состояние гонки.



## Введение: Состояния гонки в схемах сборки

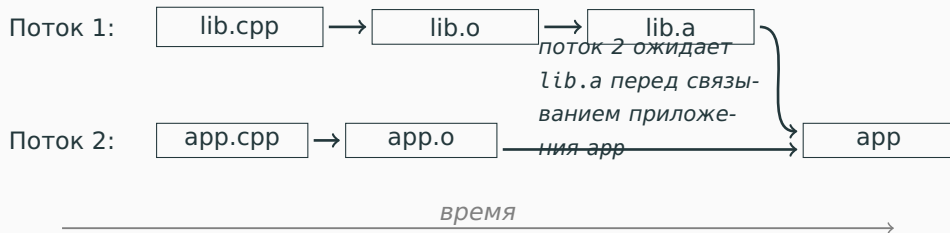
- Системы сборки могут компилировать несколько целей одновременно.
- Если две цели взаимодействуют с одним и тем же файлом во время параллельной сборки, может возникнуть состояние гонки.



- Что если сборка библиотеки займет больше времени?

## Введение: Состояния гонки в схемах сборки

- Системы сборки могут компилировать несколько целей одновременно.
- Если две цели взаимодействуют с одним и тем же файлом во время параллельной сборки, может возникнуть состояние гонки.



- Что если сборка библиотеки займет больше времени?
- **Решение:** Определить пропущенную зависимость и добавить ее в схему сборки.



Такие проблемы иногда встречаются даже в крупных проектах. Они могут быть подтверждены следующими сообщениями об ошибках:

- Vim:

```
cp: cannot create regular file '../bin/vimtutor': No such file or directory
make[1]: *** [Makefile:2546: installtutorbin] Error 1
make[1]: Leaving directory '/build/source/src'
```

- strace:

```
In file included from ./defs.h:2041,
                  from mpers-m32/struct_blk_user_trace_setup.c:10:
./syscall.h:14:11: fatal error: sys_func.h: No such file or directory
   14 | # include "sys_func.h"
      |           ^~~~~~
```

Состояния гонок в схемах сборки могут быть **трудны для отладки:**

- Сценарий гонки может редко воспроизводиться.
- `make --shuffle` — случайная переупорядочивание независимых целей. Это помогает, но не гарантирует обнаружение гонки.

Состояния гонок в схемах сборки могут быть **трудны для отладки:**

- Сценарий гонки может редко воспроизводиться.
- `make --shuffle` — случайная переупорядочивание независимых целей. Это помогает, но не гарантирует обнаружение гонки.

**Цель исследования:** Облегчить для разработчиков поиск и отладку состояний гонок в их схемах сборки.

*В качестве целевой системы сборки была выбрана Make из-за ее популярности.*

## **Теоретический метод**

Самые частые состояния гонок, встречающиеся в реальных проектах, можно разделить на три категории:

- **Гонка на содержимом файла.**
- **Гонка на путь к файлу.**
- **Гонка между созданием каталога и файлом в нем.**

- **Гонка на содержимом файла.**
- Гонка на путь к файлу.
- Гонка между созданием каталога и файлом в нем.

## Теоретический метод: Гонка на содержимом файла

В следующем Makefile присутствует гонка на содержимом файла `file.out`

```
1 all: write_a append_b
2
3 write_a:                                'a' записывается в file.out
4     echo 'a' > file.out                из цели write_a.
5
6 append_b:                                'b' добавляется к
7     echo 'b' >> file.out                тому же файлу из цели
                                         append_b.
```

## Теоретический метод: Гонка на содержимом файла

В следующем Makefile присутствует гонка на содержимом файла `file.out`

```
1 all: write_a append_b
2
3 write_a:                                'a' записывается в file.out
4     echo 'a' > file.out                из цели write_a.
5
6 append_b:                               'b' добавляется к
7     echo 'b' >> file.out               тому же файлу из цели
                                         append_b.
```

`write_a` и `append_b` независимы, их порядок выполнения не определен.  
`file.out` может содержать `'ab'` или `'a'`.



## Теоретический метод: Гонка на содержимом файла

В следующем Makefile присутствует гонка на содержимом файла file.out

```
1 all: write_a append_b
2
3 write_a:                                'a' записывается в file.out
4     echo 'a' > file.out                из цели write_a.
5
6 append_b: write_a                       'b' добавляется к
7     echo 'b' >> file.out                тому же файлу из цели
                                         append_b.
```

write\_a и append\_b независимы, их порядок выполнения не определен.  
file.out может содержать 'ab' или 'a'.

**Исправление:** Сделать append\_b зависимым от write\_a

**Как можно обнаружить такие состояния гонок?**

## Теоретический метод: Гонка на содержимом файла

Запустите сборку под **strace**.

Strace - это утилита Linux, которая перехватывает и выводит системные вызовы и сигналы.

```
1 $ strace -f -e trace=%file make
2 ...
3 # echo 'a' > file.out:
4 [pid 1060] openat(AT_FDCWD, "file.out", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
5 [pid 1060] +++ exited with 0 +++
6 ...
7 # echo 'b' >> file.out:
8 [pid 1061] openat(AT_FDCWD, "file.out", O_WRONLY|O_CREAT|O_APPEND, 0666) = 3
9 [pid 1061] +++ exited with 0 +++
10 ...
```

Журнал strace показывает, что file.out был открыт из двух разных процессов для записи и дозаписи (как флаги O\_TRUNC и O\_APPEND указывают).

## Теоретический метод: Гонка на содержимом файла

Путем обработки журнала и сопоставления процессов с целями можно получить более информативный журнал событий.

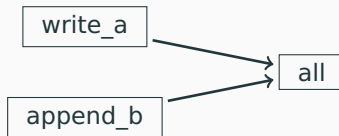
```
1  ...  
2  target write_a has performed write access on 'file.out'  
3  ...  
4  target append_b has performed read-write access on 'file.out'
```

## Теоретический метод: Гонка на содержимом файла

Путем обработки журнала и сопоставления процессов с целями можно получить более информативный журнал событий.

```
1 ...  
2 target write_a has performed write access on 'file.out'  
3 ...  
4 target append_b has performed read-write access on 'file.out'
```

Граф зависимостей схемы сборки выглядит следующим образом:

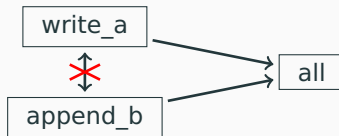


## Теоретический метод: Гонка на содержимом файла

Путем обработки журнала и сопоставления процессов с целями можно получить более информативный журнал событий.

```
1 ...  
2 target write_a has performed write access on 'file.out'  
3 ...  
4 target append_b has performed read-write access on 'file.out'
```

Граф зависимостей схемы сборки выглядит следующим образом:



Так как между `write_a` и `append_b` нет ориентированного пути, цели являются независимыми. Можно сообщить о гонке.

## Обзор алгоритма:

1. Получить журнал событий:
  - Запустите сборку под strace.
  - Сопоставьте процессы с целями.
  - Переформатируйте журнал.
2. Получите граф зависимостей схемы сборки.
3. Для каждой пары конфликтующих доступов к одному и тому же пути файла :
  - **Найти зависимость** между соответствующими целями.
  - **Если отсутствует**, сообщить о **состоянии гонки**.

## Обзор алгоритма:

1. Получить журнал событий:
  - Запустите сборку под strace.
  - Сопоставьте процессы с целями. — Как?
  - Переформатируйте журнал.
2. Получите граф зависимостей схемы сборки. — Как?
3. Для каждой пары конфликтующих доступов к одному и тому же пути файла :
  - **Найти зависимость** между соответствующими целями.
  - **Если отсутствует**, сообщить о **состоянии гонки**.



## Обзор алгоритма:

1. Получить журнал событий:
  - Запустите сборку под strace.
  - Сопоставьте процессы с целями. — Как?
  - Переформатируйте журнал.
2. Получите граф зависимостей схемы сборки. — Как?
3. Для каждой пары конфликтующих доступов к одному и тому же пути файла :  
— Что насчет жестких ссылок?
  - **Найти зависимость** между соответствующими целями.
  - **Если отсутствует**, сообщить о **состоянии гонки**.

## Теоретический метод: Гонка на содержимом файла

**Жесткие ссылки** - это несколько имен для одного и того же файла в файловой системе. Они создаются с помощью команды `ln`.

```
1  all: file.out write_a append_b
2  file.out:
3      touch file.out
4      ln -f file.out hardlink_a    hardlink_a и hardlink_b - жесткие
5      ln -f file.out hardlink_b    ссылки на file.out
6
7  write_a: file.out
8      echo 'a' > hardlink_a        'a' записывается в hardlink_a из цели
9                                  write_a
10 append_b: file.out
11     echo 'b' >> hardlink_b        'b' добавляется к hardlink_b из цели
                                   append_b
```

Журнал событий для приведенного выше примера:

```
1  ...
2  target file.out has performed write access on 'file.out'
3  ...
4  target write_a has performed write access on 'hardlink_a'
5  ...
6  target append_b has performed read-write access on 'hardlink_b'
```

Все три пути в журнале событий различаются. Состояние гонки не обнаружена.

Журнал событий для приведенного выше примера:

```
1  ...
2  target file.out has performed write access on 'file.out'
3  ...
4  target write_a has performed write access on 'hardlink_a'
5  ...
6  target append_b has performed read-write access on 'hardlink_b'
```

Все три пути в журнале событий различаются. Состояние гонки не обнаружена.

**Исправление:** Используйте **номера inode** вместо путей к файлам.

**inode** - это уникальный идентификатор файла в файловой системе. Он гарантированно будет иметь одно и то же значение для жестких ссылок на один и тот же файл.

Замените пути к файлам на номера inode:

```
1  ...
2  target file.out has performed write access on inum 100000
3  ...
4  target write_a has performed write access on inum 100000
5  ...
6  target append_b has performed read-write access on inum 100000
```

## Обновленный алгоритм:

1. Получить лог событий
  - Запустить сборку под strace с логированием номера inode.
  - Сопоставьте процессы с целями.
  - Переформатируйте журнал.
2. Получите граф зависимостей схемы сборки.
3. Для каждой пары конфликтующих доступов к одному и тому же пути файла номеру inode :
  - Найдите зависимость между соответствующими целями.
  - Если отсутствует, сообщить о состоянии гонки.

Этот алгоритм может найти состояния гонок даже при использовании жестких ссылок.

**Примечание:** номера inode могут быть повторно использованы.

- Гонка на содержимом файла.
- **Гонка на путь к файлу.**
- Гонка между созданием каталога и файлом в нем.

## Теоретический метод: Гонка на путь к файлу

В следующем Makefile присутствует гонка на пути tmp\_file

```
1  all: something something_else
2
3  something:
4      generate_something > tmp_file
5      do_something_with tmp_file
6      rm tmp_file
7
8  something_else:
9      generate_something_else > tmp_file
10     do_something_else_with tmp_file
11     rm tmp_file
```

tmp\_file используется для хранения некоторых промежуточных данных. Он удаляется после сборки цели.

То же имя используется для хранения промежуточных данных в другой независимой цели.

Это состояние гонка, потому что команда `rm` может быть выполнена, когда другая цель все еще использует файл.



Журнал событий для приведенного выше примера:

```
1  ...
2  target write_a has performed write access on inum 100000
3  target write_a has performed unlink access on inum 100000
4  ...
5  target append_b has performed write access on inum 100001
6  target append_b has performed unlink access on inum 100001
```

Файл `tmp_file` изменил свой номер `inode` после его удаления и повторного создания. Алгоритм, основанный на номерах `inode`, не сможет найти гонку здесь.

Журнал событий для приведенного выше примера:

```
1  ...
2  target write_a has performed write access on inum 100000
3  target write_a has performed unlink access on inum 100000
4  ...
5  target append_b has performed write access on inum 100001
6  target append_b has performed unlink access on inum 100001
```

Файл `tmp_file` изменил свой номер `inode` после его удаления и повторного создания. Алгоритм, основанный на номерах `inode`, не сможет найти гонку здесь.

**Исправление:** Используйте пути к файлам в дополнение к номерам `inode` только для обнаружения гонок, включающих удаление файла.

- Гонка на содержимом файла.
- Гонка на путь к файлу.
- **Гонка между созданием каталога и файлом в нем.**

Другой распространенный  
шаблон состояния гонок:

```
1 all: build build/file.out
2
3 build:
4     mkdir -p build
5
6 build/a.out:
7     echo "a" > build/a.out
```

Если команда `echo` будет  
выполнена до команды  
`mkdir`, произойдет ошибка.

Другой распространенный шаблон состояния гонок:

```
1 all: build build/file.out
2
3 build:
4     mkdir -p build
5
6 build/a.out:
7     echo "a" > build/a.out
```

Если команда `echo` будет выполнена до команды `mkdir`, произойдет ошибка.

Для обнаружения таких состояний гонок необходимо записывать два события для каждого доступа к файлу:

- Сам доступ к файлу
- Специальное событие **dir\_lookup** для родительского каталога

Новый журнал событий:

```
1 build performed write on 'build'...
2 build/a.out performed dir_lookup on 'build'...
3 build/a.out performed unlink on 'build/a.out'...
```

Теперь алгоритм, основанный на номерах inode, обнаружит гонку на каталоге `build`

**Проблема:** В некоторых проектах каталог сборки создается в каждой цели отдельно с помощью `mkdir -p`:

```
1 all: library_1 ... library_9
2
3 library_1:
4     mkdir -p build
5     build_library build/lib1.a
6     ...
7
8 library_9:
9     mkdir -p build
10    build_library build/lib9.a
```

## Теоретический метод: Гонка на каталог

**Проблема:** В некоторых проектах каталог сборки создается в каждой цели отдельно с помощью `mkdir -p`:

```
1 all: library_1 ... library_9
2
3 library_1:
4     mkdir -p build
5     build_library build/lib1.a
6     ...
7
8 library_9:
9     mkdir -p build
10    build_library build/lib9.a
```

**Следствие:** Записывается несколько событий `write` для одного и того же каталога. Только первое успешно. Остальные завершаются с ошибкой `EEXIST`.

```
library_1 performed write on 'build'...
library_1 performed dir_lookup on 'build'...
library_1 performed write on 'build/lib1.a'...

library_2 failed write on 'build'...
library_2 performed dir_lookup on 'build'...
library_2 performed write on 'build/lib2.a'...
...
```

## Теоретический метод: Гонка на каталог

**Проблема:** В некоторых проектах каталог сборки создается в каждой цели отдельно с помощью `mkdir -p`:

```
1 all: library_1 ... library_9
2
3 library_1:
4     mkdir -p build
5     build_library build/lib1.a
6     ...
7
8 library_9:
9     mkdir -p build
10    build_library build/lib9.a
```

**Следствие:** Записывается несколько событий `write` для одного и того же каталога. Только первое успешно. Остальные завершаются с ошибкой `EEXIST`.

```
library_1 performed write on 'build'...
library_1 performed dir_lookup on 'build'...
library_1 performed write on 'build/lib1.a'...
```

```
library_2 failed write on 'build'...
library_2 performed dir_lookup on 'build'...
library_2 performed write on 'build/lib2.a'...
...
```

Алгоритм сообщит о ложной гонке для любой пары доступов к каталогу `build` из разных целей `library_n`. **Как избежать ложных положительных**



## Теоретический метод: Гонка на каталог

**Решение:** Создайте отдельный алгоритм для обнаружения состояний гонок с использованием **dir\_lookup**:

- Все попытки создать каталог сохраняются в наборе.
- Состояние гонки сообщается, если событие **dir\_lookup** для этого каталога конфликтует со **всеми** попытками создания.

Thread 1: (library\_1)

mkdir -p build



use build/lib1.a

...

...

...

Thread 9: (library\_9)

mkdir -p build



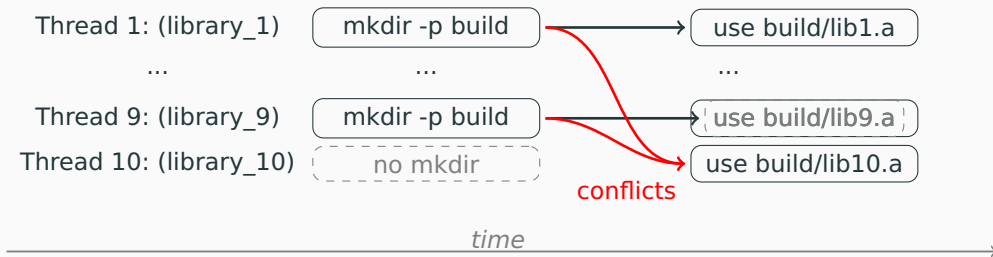
use build/lib9.a

*time*

## Теоретический метод: Гонка на каталог

**Решение:** Создайте отдельный алгоритм для обнаружения состояний гонок с использованием `dir_lookup`:

- Все попытки создать каталог сохраняются в наборе.
- Состояние гонки сообщается, если событие `dir_lookup` для этого каталога конфликтует со **всеми** попытками создания.

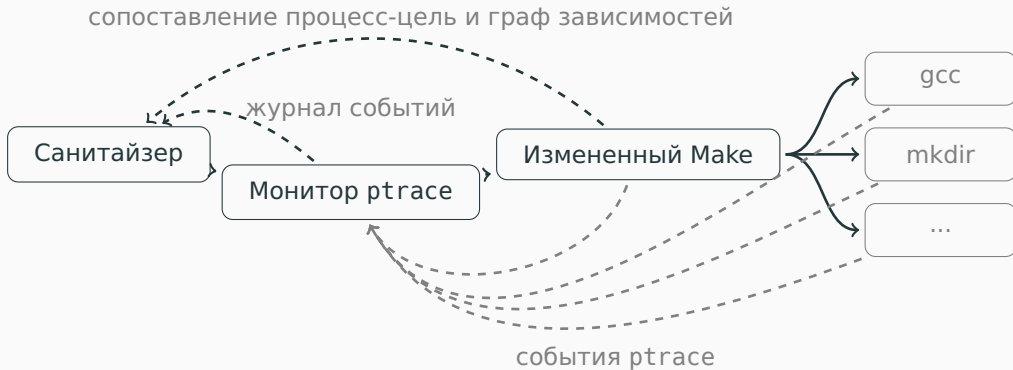


- Команда `mkdir build` не была выполнена для `library_10`, поэтому любое использование директории `build` будет сообщать о гонке.

**Реализация**

## Реализация: Архитектура

- Граф зависимостей, сопоставление процессов с целями: измененный Make.
- Журнал событий: монитор на основе ptrace вместо парсинга журнала strace.
- Компонент ptrace перемещен в отдельный процесс для изоляции процесса сборки от санитайзера.



## Реализация: Трейсинг системных вызовов

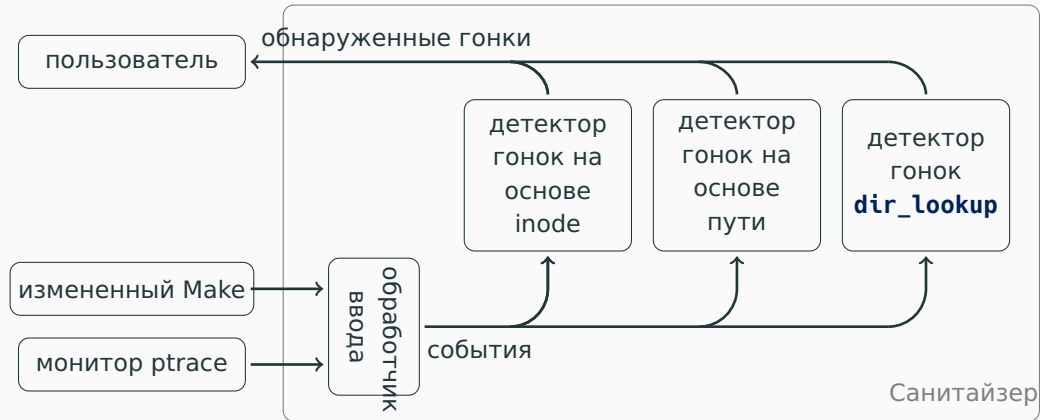
Перехватываемые системные вызовы:

системный вызов	событие для санитайзера
<code>open(at)(at2)</code>	доступ на <b>чтение</b> , <b>запись</b> или <b>чтение-запись</b>
<code>mkdir(at)</code>	доступ на <b>запись</b> к созданному каталогу
<code>creat</code>	доступ на <b>запись</b>
<code>rmdir</code>	доступ на <b>удаление</b>
<code>unlink(at)</code>	доступ на <b>удаление</b>
<code>rename(at)(at2)</code>	<b>удаление</b> целевого пути, если он существует.

- Каждый доступ сообщается с указанием **процесса**, номера **inode** и **пути к файлу**.

## Реализация: Санитайзер

Санитайзер получает все сообщения и обрабатывает их с помощью трех алгоритмов:



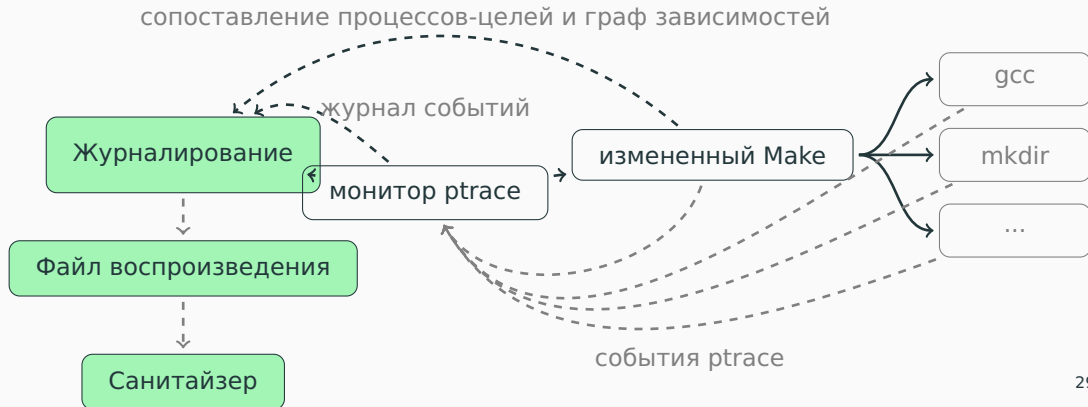
## Реализация: Воспроизведение сборки

- Структура инструмента позволяет сохранять журнал сборки в файл и воспроизводить его после.



## Реализация: Воспроизведение сборки

- Структура инструмента позволяет сохранять журнал сборки в файл и воспроизводить его после.
- Это позволяет быстро запускать санитайзер несколько раз с разными опциями или точками останова.





# Тестирование

Сергей Трофимович с помощью `make --shuffle` обнаружил условия гонок в 29 проектах с открытым исходным кодом, включая:

1. Vim
2. GCC
3. strace
4. Ispell

<https://trofi.github.io/posts/249-an-update-on-make-shuffle.html>

- Сообщаемая ошибка сборки Vim:

```
1 cp: cannot create regular file '../bin/vimtutor': No such file or directory
2 make[1]: *** [Makefile:2546: installtutorbin] Error 1
3 make[1]: Leaving directory '/build/source/src'
```

- Сообщаемая ошибка сборки Vim:

```
1 cp: cannot create regular file '../bin/vimtutor': No such file or directory
2 make[1]: *** [Makefile:2546: installtutorbin] Error 1
3 make[1]: Leaving directory '/build/source/src'
```

- `bin/vimtutor` кажется создается слишком поздно для цели **installtutorbin**.
- Журнал событий:

```
1 target inst_dir/bin has performed write access on 'inst_dir/bin'...
2 target installtutorbin has performed dir_lookup access 'inst_dir/bin'...
```

## Тестирование: Vim

- Сообщаемая ошибка сборки Vim:

```
1 cp: cannot create regular file '../bin/vimtutor': No such file or directory
2 make[1]: *** [Makefile:2546: installtutorbin] Error 1
3 make[1]: Leaving directory '/build/source/src'
```

- `bin/vimtutor` кажется создается слишком поздно для цели `installtutorbin`.
- Журнал событий:

```
1 target inst_dir/bin has performed write access on 'inst_dir/bin'...
2 target installtutorbin has performed dir_lookup access 'inst_dir/bin'...
```

- Цели `inst_dir/bin` и `installtutorbin` являются **неупорядоченными**
- Санитайзер сообщил об гонке:

```
race found on file 'inst_dir/bin':
- write at target inst_dir/bin
- dir_lookup at target installtutorbin
```

- Санитайзер также сообщил о ранее неизвестных гонках:

**race found** at file `'src/po/LINGUAS'`:

- **write** at target `gvim.desktop`
- **write** at target `vim.desktop`

## Тестирование: Другие гонки в Vim

- Санитайзер также сообщил о ранее неизвестных гонках:

**race found** at file `'src/po/LINGUAS'`:

- **write** at target `gvim.desktop`
- **write** at target `vim.desktop`

- Оказалось, что LINGUAS является временным файлом, используемым в двух независимых целях:

```
216 vim.desktop: ...  
217     echo ... > LINGUAS  
218     $(MSGFMT) ...  
219     rm -f LINGUAS  
220  
221 gvim.desktop: ...  
222     echo ... > LINGUAS  
223     $(MSGFMT) ...  
224     rm -f LINGUAS
```

## Тестирование: Другие гонки в Vim

- Санитайзер также сообщил о ранее неизвестных гонках:

**race found** at file `'src/po/LINGUAS'`:

- **write** at target **gvim.desktop**
- **write** at target **vim.desktop**

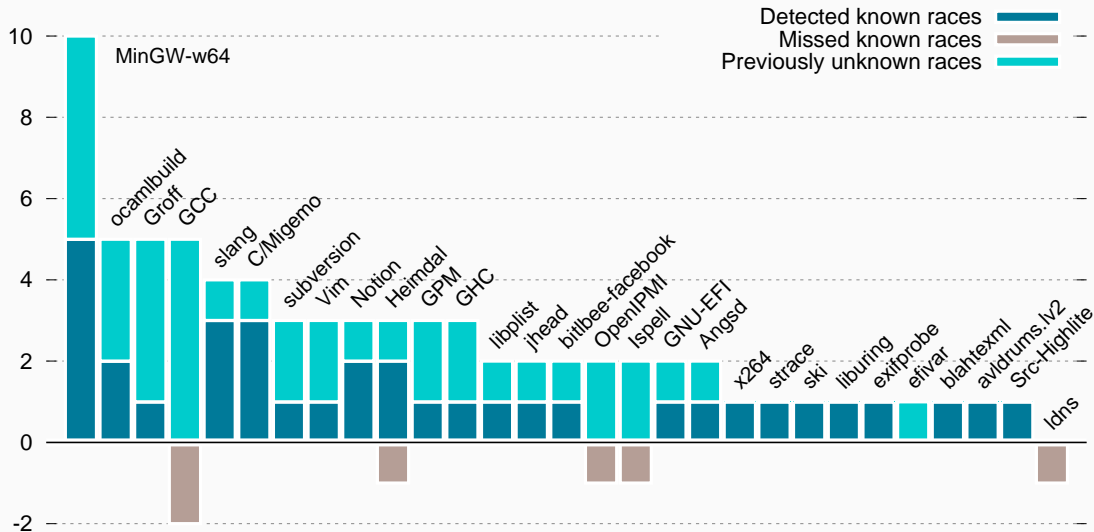
- Оказалось, что LINGUAS является временным файлом, используемым в двух независимых целях:

```
216 vim.desktop: ...
217     echo ... > LINGUAS
218     $(MSGFMT) ...
219     rm -f LINGUAS
220
221 gvim.desktop: ...
222     echo ... > LINGUAS
223     $(MSGFMT) ...
224     rm -f LINGUAS
```

- Эта гонка не может быть обнаружена с помощью `make --shuffle`



# Тестирование: Результаты



Новый инструмент продемонстрировал свою надежность и помог достичь цели исследования. Он был назван **parmasan** — **Parallel make sanitizer**.

### Дальнейшие улучшения:

- Улучшение учета символических ссылок в алгоритме поиска гонок.
- Интеграция инструмента в системы непрерывной интеграции (Tinderbox-cluster)
- Добавление поддержки системы сборки **Ninja**.

### Репозитории:

- <https://github.com/ispras/parmasan>
- <https://github.com/ispras/parmasan-remake>

**Благодарю за внимание!**