Automatic fuzzy-scheduling of threads in Google Thread Sanitizer to detect errors in multithreaded code

Oleg Doronin, Karina Dergun, and Andrey Dergachev

Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, Russian Federation
dorooleg@niuitmo.ru, 245704@niuitmo.ru, amd@corp.ifmo.ru

Abstract. The paper discusses several options for thread schedulers that are transparently embedded in the analyzed program at the code instrumentation stage, which increases the likelihood of data race detection. In addition, the work is originally interpreted by the technique of fuzzing-testing: if the traditional fuzzing involves working with data, this study proposes considering threads as data and manipulate them with the help of either other scheduler. The race found with this technique missed by Google Thread Sanitizer proves that the approach is promising.

Keywords: Google Thread Sanitizer · Relacy Race Detector · Bug detection · Multithreading · Schedulers · Data races · Libcds · Lock-free algorithms.

1 Introduction

The C++11 memory model [1] introduces potential for data race. The search for such races is expensive and difficult to implement, it primarily concerns lock-free algorithms and algorithms with use complex schemes of synchronization of threads on atomic variables (fine-grained locking) [2]. To detect data race in multithreaded code it is necessary for the analyzer to "stumble" on this race. This means that simultaneous execution of threads should be "met" on the shared variable. Only in this situation it is possible to analyze whether such a "meeting" is a race according to the memory model of C++11. A "meeting" of this kind can be quite rare in normal program execution, so the tools which increase the likelihood of such an event are in high demand and are actively developed at present.

There exist no production-ready solutions to address this problem. The Relacy Race Detector (RRD) library [3] [4] is among the developments in this direction. The library implements the functionality of thread management, but has a number of disadvantages: no ability to change the number of threads at the time of execution of the program, structural problems in the organization of the project, it does not work correctly with Thread Local Storage (TLS) [5], it does not take

into account the variability of global memory and others. A number of problems have been fixed in the work "Analiz problem v Relacy Race Detector s posleduyushchim ustraneniem" [6], but some shortcomings remain, namely:

- The tool is unpopular, which limits its further promotion;
- We need to change the code to use RRD, which takes a lot of time and effort, and causes potential errors;

A more popular tool is Google Thread Sanitizer (TSAN) [7], which at the compilation stage can intercept all calls to variables, function calls and other expressions, but does not contain the algorithms of scheduling in itself. Results obtained:

- The tool for automatic fuzzy-scheduling of threads in Google Thread Sanitizer is realized.
- The examples found where the original version of TSAN does not find potential errors, but they can be found using the automatic fuzzy-scheduling tool
- The detailed comparative analysis of various algorithms for planning of threads with the description of their pros and cons is provided.

2 Relacy Race Detector

Currently, there is an implementation that already knows how to manage thread execution. The library in which this functionality is implemented is called RRD (Relacy Race Detector). This library is external to the user which provides usability, but also there are a number of disadvantages, such as: the inability to change the number of threads at the time of execution, structural problems in organization of the project, not correct work with TLS (thread local storage), does not take into account the variability of global memory and other problems.

Multithreading is always difficult. Implementing synchronization primitives is even more difficult. And the most advanced synchronization primitives that use the weak memory model are incredibly complex to understand. It is sometimes necessary to use memory barriers because subtle synchronization is a matter of order of magnitude of performance difference and basic scaling capabilities (sometimes, of course). But such low-level synchronization primitives contribute to the appearance of errors related to memory order, which are very difficult to fix. Some time ago were developed the tool, which is called Relacy Race Detector to help developers cope with these problems.

Relacy Race Detector (RRD) is a verification algorithm for verifying sync - hronizations in the weak memory model. Physically it is a C++ library, which consists only of header files, but it can test not only C++ algorithms, but also Java, .net/CLI, x86, PowerPC, SPARC, etc.

3 Google Thread Sanitizer

Relacy Race Detector has a number of disadvantages. One of which is the unpopularity of this tool, and to promote such tools is very difficult. The second disadvantage is that you need to change your code to use RRD, and these changes can be a lot of effort. However, there is already a tool that has earned popularity– Google Thread Sanitizer, it at compile time can intercept all calls to variables, function calls, etc., but does not contain the algorithms of planning in itself. Further work was related with improvement of algorithms of work the Google Thread Sanitizer.

One of the unpleasant kinds of multithreaded errors are data races. They are difficult to find and as a consequence to reproduce because there is no possibility of observing errors during the whole testing cycle, and in working applications they can be seen as rare unexplained failures. To find errors in multithreaded code, special programs are developed, one of which–Google Thread Sanitizer (TSAN). TSAN uses a hybrid algorithm (based on the ¡happens-before¿ and multiple locks). There are also special dynamic annotations in the implementation that let you report complex synchronizations in user code.

4 Platfroms

Special mechanisms are needed to simulate thread execution. Such mechanisms require either support from the operating system or fine-tune the processor registers (this is not always possible due to restrictions imposed by the operating system). Only mechanisms that require operating system support are considered in the work, on the example of Linux. To switch threads, two mechanisms are considered, the first of which is based on fiber (a lightweight thread running in user mode) and the second based on Pthread (the standard of POSIX-implementation of execution threads).

4.1 Fiber-based implementation

Fiber is a lightweight thread running in user mode. We do not need to access the operating system to manage these threads. In user mode, it is sufficient to store the thread context that includes the processor registers, TLS (thread local storage), and other fiber information. In order to change the thread, you need to save the context of the current executable thread, and then load the new values into the processor registers, set TLS and additional information about the fiber.

The Linux operating system has built-in support for fiber (makecontext / swapcontext), which facilitates the change of threads, but does not update TLS and has to contend with this problem separately.

The main problem with implementing fiber in Linux is that TLS is common to all fiber. This is actually the TLS of the main thread. One solution to this problem is to copy TLS to the local thread structure and then overwrite the main thread's TLS. The disadvantage of this approach is the high cost of copying.

Copy TLS

```
internal_memcpy(old_thread->GetTls(),
                reinterpret_cast<const void *>(tls_addr_),
                tls_size_);
internal_memcpy(reinterpret_cast<void *>(tls_addr_),
                new_thread->GetTls(),
                tls_size_);
```

The second disadvantage is not working with the following example:

TLS Copy counterexample

```
thread_local int a = 0;
int main()
{
    int *p = nullptr;
    thread t1([&]() {
        p = &a;
        while (a != 1);
        a = 2;
    });
    thread t2([&]() {
        while (p == nullptr);
        *p = 1;
        while (*p != 2);
    });
    t1.join(); t2.join();
    cout << "Finish" << endl;
}
```

The problem here is that we pass a pointer to TLS from thread 1 to thread 2. And if implemented correctly, these areas will be different and the program ends with a "Finish" display. When we copy TLS, we get that the TLS variable will refer to the memory area associated with thread 2. Then, if we change the value of the variable p, we get a local variable a owned by thread 2, which is unexpected behavior. But in most cases TLS is not passed between threads, so this decision justifies itself.

Note that when a new thread is created it gets the values of local variables that correspond to the values at the time of program startup. Consider the following example:

Initial TLS State

```
thread_local int a = 0;
int main()
{
    thread t1([&]() {
        ++a;
        thread t2([&]() {
            cout << a << endl;
        });
        t2.join();
        cout << a << endl;
    });
    t1.join();
}
```

The output of the program will be 01, because the newly created thread will have values in TLS corresponding to the moment of program startup. To do

this, we must put TLS at the appropriate initial state when creating the thread. Obtaining such a condition is a technical task and will not be considered.

In order to avoid problems with TLS copying, we can change the pointer to the local memory of the stream. The x64 address calculation is as follows:

TLS Address Calculation

```
movq %fs:0, %rax
movq x@tpoff(%rax), %rax
```

To obtain and change TLS, the following wrappers were written:

Functions for changing TLS

```
static unsigned long get_tls_addr()
{
    unsigned long addr;
    asm("mov %%fs:0, %0" : "=r"(addr));
    return addr;
}
static void set_tls_addr(unsigned long addr)
{
    asm("mov %0, %%fs:0" : "+r"(addr));
}
```

Unfortunately, this implementation is not secure because the compiler can optimize and not request TLS from the FS register, but in practice this has not been observed. The benefits of this implementation are fixing performance issues, TLS addresses are now truly different, but there is a new problem associated with the possibility of non-correct behavior.

Unfortunately fiber is not possible to execute in parallel as they are actually executed on one real operating system thread. A pthread-based approach will be considered to correct this problem.

4.2 Pthread-based implementation

To implement a thread management platform using pthread, add a variable to the stream context that will prevent the thread from executing. Then the expectation of the thread execution capability will look like this:

Waiting for thread on a variable

```
while(old_thread->GetWait())
{
    internal_sched_yield();
}
```

If we want to allow a thread to execute, we set the Wait variable for the thread to false and it starts its execution. In the current implementation, only one thread can work at a time, but there are ideas as to how this could be improved and implemented in the follow-up work.

Let's consider advantages and disadvantages of the described platform. The main advantages of this platform: ability to run threads in parallel, can work faster than fiber in sequential mode, no problems with TLS.

This implementation allows threads to run in parallel, so we need to set wait = false for multiple threads. Because these threads correspond to the operating system threads, they can be executed in parallel, which increases the speed of the algorithm.

The main disadvantage is the loss of the ability to take snapshots, because the fork mechanism copies only one real thread. So on one-nuclear processors can work slower than fiber, because context switching occurs with the help of OS.

5 Scheduling algorithms

5.1 Random Scheduler

Random Scheduler-the idea of the algorithm is based on random selection of threads while the program is running. This algorithm is similar to the operating system scheduling algorithm. However, thread-switching points remain synchronous and are located in interesting places that maximize the probability of finding an error.

The random scheduler implementation uses a uniform distribution. The formula of the distribution density function is as follows:

$$f(x) = \begin{cases} \frac{1}{b-a}, x \in [a, b] \\ 0, \notin [a, b] \end{cases} \tag{1}$$

$[a, b]$ - distribution segment of the random value.

Such a scheduling algorithm can be useful for its speed, low memory consumption and the probability of quickly finding an error without even regardless its shortcomings.

5.2 Scheduler with different distributions of random values

Scheduler with different distributions of random values - Such a scheduler is partially similar to an algorithm of random planning in which there was only a uniform distribution.

The planning algorithm looks like this: We have a number of random value generators. We add this set to a cyclic list. A circular list is a list in which the last element has a pointer to the next element that is not empty, but points to the first element. Number generators of random values from $1..n$, then the first iteration will use a 1 random value generator, on the second iteration of 2-th ... on n-th iteration n, but already on the $n + 1$ we will use 1 generator random values. We denote through the i iteration number of the algorithm pass, then the number of the random value generator is calculated using the following formula $(i \mod N) + 1$. An important note is that random value generators retain their old state (it can be considered that every new seed launch will always be different).

Random value generators that were used in the implementation of the planning algorithm with different distributions of random values: uniform distribution, bionomial distribution, negative bionomial distribution, geometric distribution, poisson distribution, exponential distribution, weibull distribution, normal distribution, lognormal distribution.

If we look at the probability of finding errors, it will be more due to the fact that the first generator of random values corresponds to a uniform distribution as in a random scheduler, and other generators are completely different from it, which will allow us to find new types of errors. Similarly, the locality of finding errors is ensured by a uniform distribution, and the spread between the work of the threads is achieved using other distributions.

The main disadvantages of such a scheduler is that it can not cover all possible cases, so there is a possibility that we will miss the error. The second disadnvatage is that we have no criterion when the algorithm can be stopped.

If to sum up the results it is possible to make a conclusion that generators of random values are similar among themselves and that it would be necessary to find different errors fine-tuning of algorithms. But the main advantage of such an algorithm is that it can find errors very quickly with some probability. This search is not only in the local area of the threads, but also takes into account the scatter between the execution of threads.

5.3 Full search scheduler

Full search scheduler - the main disadvantage of schedulers based on random value generators is that they do not cover all possible scheduling paths. Therefore, the implementation of the scheduler algorithm was invented, which iterates through all the variants, but with some limitations. For such a scheduler, we make an important assumption that if we run the iteration several times on the same sequence of threads, then all iterations will have the same execution paths for the program. In practice, this condition is not always met, but it is rare cases.

FullPathScheduler

```
1  procedure FullPathScheduler
2      for i from 1 to CountIterations() do
3          while not IsStopProgram do
4              AsynYield(i)
5          end while
6      end for
7  end procedure
```

Let's consider the general structure of the algorithm on the example of the procedure FullPathScheduler. Suppose we have some iterator i which is able to iterate through all possible variants of threads execution (it will be considered how it can be implemented), then we will iterate over all possible variants of program execution line 2. At each iteration of the program execution we will handle the calls where a change of flow can occur. In line 3, we check that the program has not completed yet, and in line 4, we are processing an asynchronous

call to AsyncYield which allows you to change the execution thread at this point in time.

```
AsyncYield
1  procedure AsyncYield(iteration)
2      s = wait Yield(iteration)
3      if s != Nill then
4          context = GetThread(iteration[s])
5          SysCallYield(context)
6      end if
7  end procedure
```

Let's dwell on the description of the AsyncYield procedure. Wait type design means that we are waiting for the asynchronous occurrence of some event, in our example it is an opportunity to choose a thread for execution. The variable s will be the value that corresponds to the call number of the wait function Yield. That is, at the first call it will be 0, at the second one, etc. This number will be called the logical execution time of the program. If this number equals Nill it means that the program has ended, otherwise we can request from our iterator, which iterates through all possible combinations of thread execution number that should be executed at the moment. Then, in line 4, we take the context of the thread that will execute after the AsyncYield procedure completes, and in line 5 we run it for execution.

Table 1. An array of values describing the iteration

Time	0	1	2	3	4	5	6	7	8	9
Tid	0	0	0	0	0	0	0	0	0	0
MaxTid	0	1	1	2	2	2	2	3	2	0

The most difficult part of the algorithm is to learn to define all possible ways of program execution. In the 1 table, the string $Time$ means a logical time. The values in this row increase each time by 1 starting at 0 and up to a certain number of n. The Tid line shows the thread numbers that should be executed in the current iteration. The Maxtid line has the maximum Tid stream that could be at a logical time i, where i any value from 0 to n. If we knew such a table and it was static then we could use the standard algorithms to iterate through all the combinations, (the description of the standard algorithm for iterating through all the combinations can be found in the article or the detailed description in the book) or add 1 to the number in the system in which each digit has a dimension of Maxtid. In our case, the situation is complicated by the fact that can change: n, maxtid and can still fall out incorrect combinations of threads. To get a new iteration, add 1 from the end in the mixed number system.

The ability to iterate through paths allows us to make sure that the code is correct for all possible paths, provided that the same code statements are executed at multiple startup with the same parameters.

This approach consumes large memory resources, because at any time it is necessary to store information about the maximum *tid* and the current thread. It also has a long working time, is interpreted in terms of the number of iterations. It is impossible to achieve a full enumeration of variants for large applications because we are limited by computational resources. This algorithm is recommended for small test scenarios.

5.4 Scheduler based on full paths with floating window

The main problem with the full-path scheduler is that we cannot wait for the algorithm to complete because of the limited computational resources. In this regard, we want to think of an algorithm that would allow to iterate through all the variants in some local area. This algorithm was given the name "Scheduler based on full paths with floating window"

Consider the algorithm in detail. In the last step we have considered the algorithm of full path iteration to which we will refer. Let's start some fixed window for certainty take size 3 and this window will be moved along the entire time axis. Then inside this window we will iterate through all the variants using the algorithm described above, and outside the window will move randomly. Then in the local area we will be able to iterate through all possible cases, which can help to find new mistakes

The main advantage of this algorithm is that if we choose a small window size, we can get the final algorithm even from a practical point of view. Therefore, the working time of the algorithm is considered finite.

This algorithm does not cover all possible ways if we compare it with the previous example, but we can choose an infinitely large window and then it will correspond to the previous algorithm

5.5 Brute force based scheduler

All the above algorithms have a lack of working time. Even a fixed window scheduler can take a long time to complete local iteration. We want to introduce some criteria that would check the operation of all threads, but it required not a large number of iterations (as a variant of the 64 iteration).

The idea of the algorithm is as follows: At every moment we want to give the opportunity to work all the threads that could be executed, thus it will not be considered all possible paths, and it will be all possible states of threads, then the number of iterations will be limited to the maximum number of threads from all maximum quantities at some logical point in time. Variable *used* will store a list of threads that have already been executed, *variants* a list of threads that can be executed at a certain logical point in time. If we look at the tables 2 and 3, we will need only two iterations to iterate through all the options. After the first iteration in the 0th moment of time only the 0-th thread will work, and after the second iteration all possible threads will work at this point of time, and this is 0 and 1. If at some point in time we do not have any threads that can be executed, a random thread is selected.

Table 2. First iteration

Time	0	1	2	3
used	0	1	1	0
variants	0,1	0,1	0,1	0,1

Table 3. Second iteration

Time	0	1	2	3
used	0,1	1,0	1,0	0,1
variants	0,1	0,1	0,1	0,1

The main disadvantage of this algorithm is that storing the list of threads for each moment of time can be a costly operation, in order to reduce the amount of memory will perform such a search for a maximum of 64 threads. And all the information about the threads will be stored as bitmasks.

From advantages it is desirable to allocate that such algorithm allows to iterate all states of threads, that is at each moment of time will be able to work all threads, but thus not all paths will be covered. This algorithm has a limited number of iterations, which is easy to calculate at the time of program execution.

6 Test results

Testing was conducted in several steps:

1. Using existing Tests Google Thread Sanitizer
2. Find cases for which schedulers can be useful
3. Testing the libcds library

Tests provided by Google Thread Sanitizer required further refinement after adding new functionality. Because the code can be executed multiple times, and the TSAN tests rely on exactly one pass, only a small part of code could be tested using existing tests.

The following example was found on which TSAN does not cope:

Data race on variable a

```
1  std::atomic_int d;
2  int a;
3  int main()
4  {
5      std::thread t1([]() { ++d; ++a; ++d; });
6      std::thread t2([]() { ++d; ++a; ++d; });
7      t1.join(); t2.join();
8      std::cout << d << a << std::endl;
9  }
```

In lines 5, 6 on the variable a there is a data race. Several thousand iterations using the operating system scheduler did not find this problem. All the developed schedulers were tested on this example and each was able to detect the data race on the variable a

Let's look at an interesting example:

Rare case

```
1  std::atomic_int value { 0 };
2  int main()
```

```
{
    auto f = [&]() {
            for (int j = 0; j < 5; j++) {
                auto r = value.load();
                r++;
                value.store(r);
            }
        };
    std::thread t1(f), t2(f);
    t1.join(); t2.join();
    std::cout << value.load() << std::endl;
}
```

In two threads we will increase the value of the variable r per iterataion. The operating system scheduler produces values ranging from 4 to 10. A scheduler with different distributions of random values and a full search scheduler were able to get all the possible value of the variable r,namely in the range of 2 to 10. Here are the histogram frequencies:

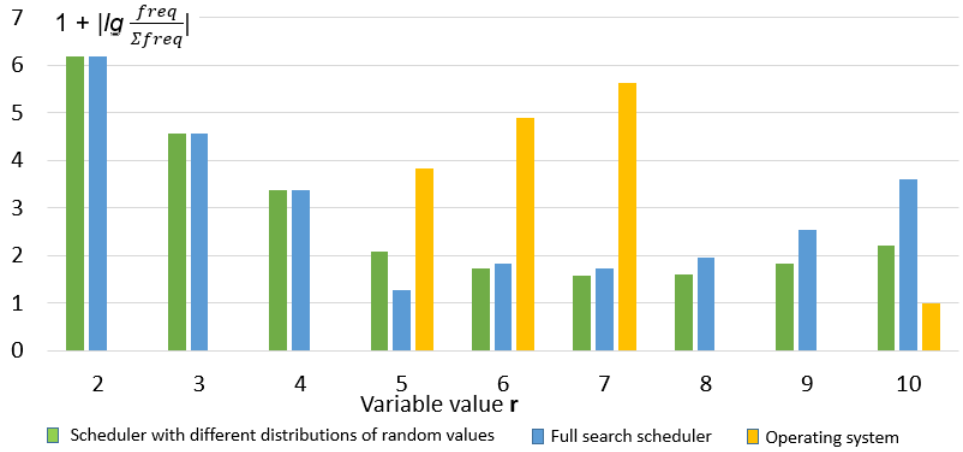

Fig. 1. Scheduling comparison

When testing the libcds library, it is currently not found on simple data structures such as lists of problems. But testing of the library continues, in particular it is planned to check BronsonAVLTreeMap and EllenBintree

7 Related work

In the future it is planned to try the developed tool on such projects as: Google Chrome Browser, Boost Library, Liblfds Library, research of other projects

There are also a number of tasks for further improvements in the project: support for correct work with mutex, handling hangs on spin lock, adding a

parallel scheduling algorithm [8], acceleration of developed algorithms, Modeling the order of visibility of variables [9] [1], development of a scheduler to check the properties of lock free algorithms

8 Conclusion

Various platforms and scheduling algorithms were considered in this paper. And as a result, the following tasks were completed:

1. TSAN has developed three platforms for managing threads
2. Implemented schedulers in TSAN
3. Bugs found and fixed in TSAN
4. Testing of developed tools

Currently, there are examples where Google Thread Sanitizer does not find errors using the operating system scheduler, and the developed infrastructure allows to find missed errors. This shows that the goal is achieved, the developed tool helps to find errors in multithreaded code that was previously difficult to detect.

References

1. Batty M., Owens S., Sarkar S., Sewell P., Weber T. Mathematizing C++ Concurrency // POPL, 2011
2. Khizhinsky M. CDS C++ library [Electronic Resource]. - Access mode: https://github.com/khizmax/libcds, free. Language English (access date 04.12.2018)
3. Vyukov D. Relacy Race Detector [Electronic Resource]. - Access mode: https://github.com/dvyukov/relacy, free. Language English (access date 15.11.2018)
4. Vyukov D. RRD: Introduction [Electronic Resource]. - Access mode: http://www.1024cores.net/home/relacy-race-detector/rrd-introduction, free. Language English (access date 25.11.2018)
5. Drepper U. ELF Handling For Thread-Local Storage, Red Hat Inc, 2005, 79 pages
6. Doronin O.V., Kalishenko E.L., Kalishenko E.L. Analiz problem v Relacy Race Detector s posleduyushchim ustraneniem // Sbornik tezisov dokladov kongressa molodyh uchenyh. [Electronic Resource]. - Access mode: http://kmu.ifmo.ru/collections_article/7307/analiz_problem_v_Relacy_Race_Detector_s_posleduyuschim_ustraneniem.htm, free. Language Russian (access date 20.11.2018)
7. Google Thread Sanitizer [Electronic Resource]. - Access mode: https://github.com/google/sanitizers/wiki, free. Language English (access date 20.11.2018)
8. S. Nagarakatte, S. Burckhardt, M. M. Martin and M. Musuvathi, Multicore acceleration of priority-based schedulers for concurrency bug detection, ACM, 2012.
9. C. Lidbury and A. F. Donaldson, Dynamic Race Detection for C++11, POPL, 2016