Taming pthreads: Deterministic and Explorable Concurrency Fuzzing

Nathan Wachholz

June 17, 2023

Abstract

We introduce a deterministic `pthread` implementation to assist fuzzing of concurrent software, and demonstrate its performance gains on a suite of benchmarks. We also investigate adding concurrency-coverage metrics to the `centipede` fuzzer, to further improve performance when fuzzing with our implementation.

1 Introduction

Fuzzing multi-threaded applications is tricky because scheduling is non-deterministic. Race conditions mean identical inputs can result in different executions and feature sets. However, efficiently fuzzing these applications is desirable, as many interesting targets are multi-threaded, and multi-threaded code is bug-prone.

In this project, we introduce a deterministic `pthreads` implementation that is compatible with popular coverage-guided fuzzers. Specifically, our implementation[1] ensures that identical inputs result in the same execution (assuming the rest of the program is deterministic). Furthermore, it allows a fuzzer to explore potential schedules with fine-grained control.

This enables the discovery of bugs that only occur under specific (potentially quite unlikely) thread schedules (see Appendix 7.1 for an example). However, it also speeds up bug discovery in general, compared to running with a standard `pthreads` implementation. We evaluate our implementation on a suite of benchmarks, and show promising results.

Standard fuzzing feature-sets like code coverage can lead to schedule exploration, but there are a number of concurrency-coverage-metrics that directly incentivize it. We experiment with adding such metrics to the `centipede` fuzzer, making use of our `pthreads` implementation.

1.1 Prior Work

Concurrency is a well-studied problem in automated verification, and there are a variety of approaches.

In the realm of symbolic execution and formal model checking, the `por-se`[2] project is close to the state-of-the-art. It "combines Partial-Order Reduction to detect interleaving non-determinism, with Symbolic Execution (SE) to handle data non-determinism." The project is an extension of KLEE (a popular symbolic executor), and can verify programs that use `pthreads`. Although POR techniques drastically reduce the search space of thread interleavings, program

size is often a limiting factor. This is true of most static-analysis or symbolic-execution approaches–the search space of interleavings is gigantic.

Microsoft's CHESS[3] is a deterministic scheduler that exhaustively explores thread interleavings. It is usually run on an existing suite of unit tests, to ensure they pass regardless of thread schedule. CHESS has limits in debugging programs with missing or incomplete unit tests, and unit tests must remain somewhat small.

Applications that lack unit tests and are too large for symbolic execution are often fuzzed. On that front, Google's SockFuzzer[4] is an effort to fuzz Apple's multi-threaded XNU kernel. It uses Concurrence[5][6], to run a single kernel thread at a time, with all scheduling decisions controlled by fuzzer input. This allows traditional fuzzing to continue deterministically, while also allowing the fuzzer to explore different schedules by mutating scheduling data. This is a very interesting technique, but it unfortunately remains quite specific to XNU, and cannot be used to fuzz arbitrary `pthread` applications.

Our goal with this project is to generalize SockFuzzer-like fuzzing to arbitrary `pthread` applications. In-process, coverage-guided fuzzing has a strong track-record of finding bugs, and we want to allow these fuzzers to effectively target `pthread` applications.

This brings us to `unthread`[7], a deterministic `pthreads` implementation. `unthread` has some notable shortcomings: not being completely POSIX-compliant; nor allowing schedule configuration at runtime. Nevertheless, it is a drop-in `pthreads` replacement that was a valuable starting point for this project. We extended `unthread` to work with `centipede` (a modern fuzzer), and performed experiments to show that fine-grained schedule control is beneficial.

Academia and industry have turned to fuzzing for concurrency verification, and introduced various "concurrency-coverage metrics" to incentivize schedule exploration. This examined in more detail in Section 4, after we conclude our work with `unthread`.

2 Implementation

It is critical that our implementation is as simple to use as possible. Ease-of-use makes can help tool adoption, but it also makes it easier for us to benchmark and test on a large suite of examples.

To this end, target code does not need to be modified. `unthread` ships an `pthread.h` header, and a single compiler flag can have it replace the builtin one. This header file redirects all `pthread_XX` functions to `unthread_XX` ones. All POSIX `pthread_XX` functions are implemented, including synchronization primitives. Because all calls are intercepted, `unthread` can offer unique features such as deadlock detection: if all threads are blocked, the program will crash with a helpful error, rather than hanging.

`unthread` is implemented using user-space threads, frequently termed "fibers". These fibers run within a single host thread using `makecontext`, `swapcontext`, `setjmp`, and `longjmp`. At any moment, only one fiber is running. This means

there is no preemption—context switches can only happen at specific "yield points". Almost any `pthread_XX` call is a yield point (including the recommended `sched_yield()`), which gives the scheduler the opportunity to switch to a different "runnable" thread.

A thread is runnable unless it is blocked (locking a mutex, joining another thread, etc.). At a yield point, the scheduler can switch to any runnable thread. If there is more than one runnable thread, then the scheduler has a "scheduling decision". The scheduler must pick one of the n runnable threads to start executing. This is a deterministic decision, made according to a configurable "data source".

There are two possible data sources (also called "scheduling control methods"), which are configured at runtime. The first is a stream of integers. We consume an integer k from the stream, and pick thread $k \,\%\, n$. The stream has a finite length, so the "end behavior" must also be defined. Options include looping, continuing with all zeros, terminating, etc.

The second option is a PRNG, which generates a stream of integers on the fly from a given "seed". Again, we generate an integer k, and pick thread $k \,\%\, n$. The PRNG is deterministic, depending only on the initial configured seed.

The user can configure `unthread` before each fuzzing input using the `unthread_configure` call. This should be called from the fuzzing driver. For an example, see Appendix 7.2.

Besides this call in the fuzzing driver and an additional compiler argument when building the target, there is no further user action required. Although not entirely compliant with POSIX (lack of preemption, fewer cancellation points, etc.), threading will become deterministic and reproducible. This is already any improvement over `pthreads`. In the next section, we will experimentally determine performance benefits.

3 Evaluation and Results

Although a deterministic `pthreads` implementation has other uses, the primary focus of this project is on fuzzing. Therefore, to evaluate performance, we will fuzz a suite of concurrent programs.

All programs in the suite take no input, but have a reachable bug (crash or deadlock), under some thread schedule. In theory, running any of these programs in a loop with `pthreads` will eventually result in a crash. We will check how the deterministic, explorable schedules provided by `unthread` can speed up this process.

We ran on a suite of 12 programs. They are described in Table 1. Most tests were adapted from existing sources, and information on each of those can be found through the SV-Comp project[8].

Tests were performed using the Centipede[9] fuzzer. For comparison, we fuzzed each program in one of three ways. (1) Using the standard `pthread`

[1] Mimics a concurrency bug in JDK1.4 StringBuffer, by Jie Yu (jieyu@umich.edu)

Table 1: Test Suite

Test Name	Source	Category
`bigshot_p`	sv-comp	mutex, trivial
`bounded_buffer`	sv-comp	mutex, trivial
`checkpoints`	-	very rare schedule
`deadlock_1`	sctbench	deadlock
`interleave`	-	very rare schedule
`lazy`	sv-benchmarks	rare schedule
`phase_1`	sctbench	deadlock
`singleton_b`	sv-benchmarks	very rare schedule
`stringbuffer`	Jie Yu[1]	rare schedule, mutex
`triangular`	sv-benchmarks	very rare schedule
`workstealqueue_mutex_1`	sv-comp	mutex
`workstealqueue_mutex_3`	sv-comp	mutex

implementation; (2) With the fuzzer varying `unthread`'s PRNG seed; and (3) With the fuzzer varying `unthread`'s schedule data.

3.1 Results

Each fuzzer-target-pair, was given 15 minutes to run as many times as possible. A run ends when a bug (crash or deadlock) is found. In the case of pthread, there is no deadlock detection, so we rely on a 10 second timeout per input. We record the average time-to-bug of each pair, which is presented in Table 2. If the fuzz-target-pair doesn't find a bug within 15 minutes, we record "timeout". The inverse of these times (a measure of speed) is illustrated in Figure 1.

Name	pthread	schedule	seed
`bigshot_p`	0.154	0.155	0.138
`bounded_buffer`	0.147	0.183	0.139
`checkpoints`	timeout	0.195	timeout
`deadlock_1`	16.940	0.140	0.137
`interleave`	timeout	0.202	timeout
`lazy`	0.424	0.144	0.138
`phase_1`	33.149	0.146	0.142
`singleton_b`	0.765	timeout	6.672
`stringbuffer`	1.138	0.147	0.145
`triangular`	timeout	0.146	timeout
`workstealqueue_mutex_1`	0.474	0.304	0.165
`workstealqueue_mutex_3`	0.540	0.238	0.163

Table 2: Average Fuzzing Time-to-bug (seconds)

These results align with our expectations (with some exceptions). First of

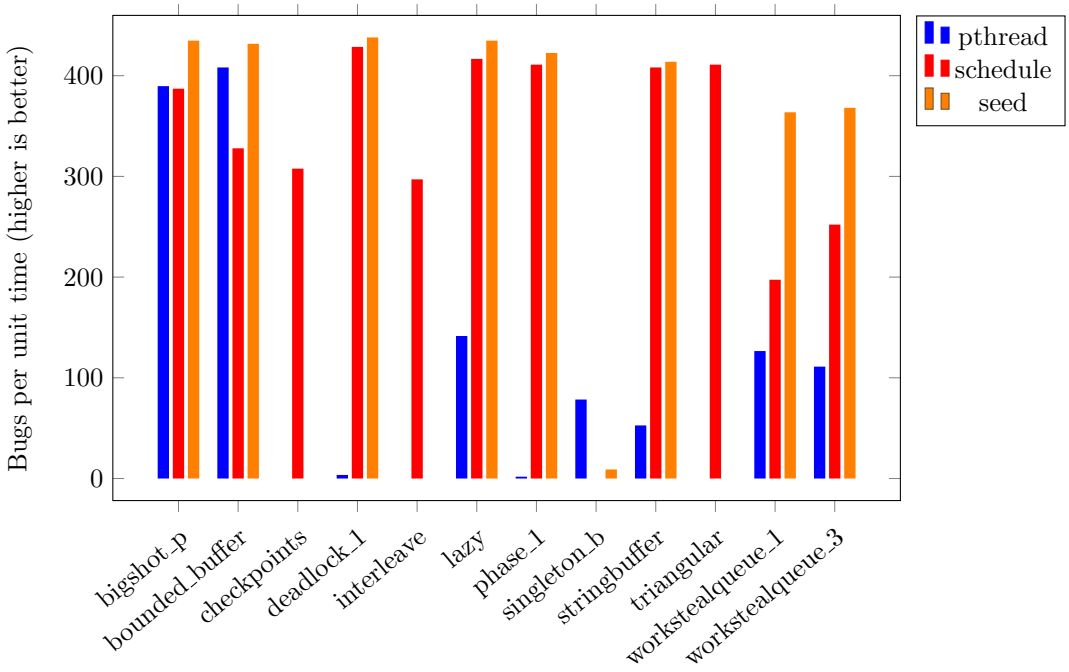

Figure 1: A comparison of fuzzing speed on a variety of targets using three scheduling control methods.

all, the two deadlock tests (`deadlock_1` and `phase_1`) competed much faster with `unthread`, due to its deadlock-reporting capabilities. Of course, the exact performance difference depends on the configured test timeout (we used 10 seconds, the default is 60).

Second, we expect `unthread` to outperform `pthread` in all but the most simple tests (`bigshot_p`, `bounded_buffer`). While this is generally true, the exact speedup varies. In some cases, `pthread` could not find the bug within 15 minutes, whereas `unthread` found it in under a second (`interleave`, `checkpoints`, `triangular`).

Lastly, we expected schedule data to outperform PRNG seed. This is not so clear-cut. Generally, seed was faster than schedule, unless the schedule required was exceedingly rare (`interleave`, `checkpoints`, `triangular`) and seed couldn't get it at all.

There was one unusual result with the `singleton_b` example. This case creates 11 threads. Each thread sets a constant to `'Y'` except one, which sets it to `'X'`. The X thread must run last for the crash to occur. It's unclear why `pthread` excels in this case, but it is clear why `unthread` struggles: there are no features to distinguish different schedules. In the following section, we discuss our attempts to add "concurrency coverage metrics" to solve this problem.

4 Concurrency Coverage Metrics

At this point, we have succeeded in creating a `pthread` implementation to speed up fuzzing of concurrent programs with a deterministic schedule. However, the fuzzer has no direct measure of "concurrency coverage". Even if two test inputs result in a different interleaving order, the actual lines of code executed (and the resulting fuzzing features) may be the same. There is no incentive for the fuzzer to try different interleavings, unless it gets lucky and they result in different features.

This is a well studied problem, and there have been a few proposed "concurrency-coverage metrics". [10] proposes Concurrent Function Pairs (CFP), which count how often each pair of functions are executed in parallel. Another is Concurrent Basic Block Pairs, which is the same but for basic blocks. And recently, Concurrent Context Pairs[11] (CCP), which is like CFP but includes a few more levels of the call stack.

We attempt to implement something similar to CCP. Each thread T maintains a hash of its call stack ($callstack(T)$, using an array of PCs from `__builtin_return_address(level)`). The depth of the callstack is configurable.

At each context switch, we record a features. When switching from thread S to thread T, we record a hash of the tuple $(callstack(S), callstack(T))$.

Centipede makes user-defined features easy. `unthread` declares an array in the special `section("__centipede_extra_features")`, and populates the array with integers corresponding to each custom feature.

Unfortunately, after implementing the custom features, there was little to no affect on performance. While the fuzzer was recording and reporting the new features for each input, the number of features was much higher than expected, and they would change between runs (so they were not deterministic). After more investigating, we realized some issues with our approach. Although we did not have time to address them, they are outlined in the Section 5.3.

5 Challenges

In this section, we describe challenges encountered during this project. We hope this explains our design decisions, and gives ideas for future work or improvements.

5.1 Integrating `unthread`

Initially, `unthread` used the same symbol names as `pthreads`. This made switching between implementations easy. No code needed to be modified. However, because `unthread` uses different, simpler types and structures, recompilation was still required. Recompilation only required using a different `pthread.h` header, which could be done with an additional compiler argument.

However, we quickly ran into issues with this approach. The C++ standard library uses `pthread` calls, and was not compiled with our header. This caused

crashes very early in program startup.

Our next approach was similar. Each thread would maintain a thread-local variable denoting if the thread was from `pthread` or `unthread`. `unthread` would continue to use `pthread` symbol names, but would optionally forward the call to `pthreads` using `dlsym`, based on the thread-local variable. There was a function to switch a thread to using `unthread`, which would be called by the fuzzing driver prior to executing the target.

This worked well for simple targets. However, we quickly realized that instrumented code uses `pthreads` (such as ASAN). We want to redirect the instrumentation to `pthreads`, because that's how it was initialized, but we could easily determine where calls were coming from. The edge cases and complexity of this solution was too much, and we reevaluated again.

We finally settled on the current approach. Since recompiling with a different header is already required, we simply `#define` each `pthread_XX` function to the corresponding `unthread_XX` one. This works well.

5.2 Memory Leak

After connecting `unthread` to our fuzzing harness for the first time, we would get an out-of-memory crash within a few hundred executions of the fuzzer. Some debugging uncovered a bug in `unthread`'s memory management, causing it to never free a thread's stack space. The fix was fairly simple, and was merged back into `unthread` via PR #4[12].

5.3 Concurrency Coverage Metrics

As mentioned in Section 4, we attempted to add concurrency-coverage metrics to centipede. Unfortunately, we experienced a blowup of features, and non-determinism between runs.

Our hashes and features are determined by walking up the call stack and recording the PC as reported by `__builtin_return_address(level)` and `__builtin_extract_return_addr`. Unfortunately, this doesn't account for PIE. The same function will be loaded into a different place for each run. This means that the hashes will be completely different each time, explaining our results.

There was no solution to this issue we could implement within the time allotted. Theoretically, we could use `dladdr` to determine the object base address. Then all addresses could be hashed relative to their object's base address. This would undoubtedly get slow, so we would likely need to add a cache.

Another option is to reuse `centipede`'s existing thread stack features, modifying them to work with `unthread`. This might be cleaner, but would necessitate a closer integration between the two projects.

6 Conclusion

This project introduced an ready-to-use solution to improve concurrency fuzzing. Our technique can quickly find bugs that can't be found otherwise. It is able to instantly detect deadlocks–and can describe the deadlock to help with debugging. We showed promising results, demonstrating sizeable speedups on 10 of 12 benchmarks. Although concurrency-coverage metrics aren't working, we believe they could with a little more effort.

There's a lot of potential here, and a few promising next steps:

- Use `unthread` with more realistic targets, such as adding it to OSS-Fuzz[13].
- Fix concurrency-coverage metrics to overcome PIE.
- Closer integration with centipede (or other fuzzers). For instance, separating schedule data from the rest of the fuzzing input.

References

[1] Nathan Wachholz. *n-wach/unthread*. URL: https://github.com/n-wach/unthread.

[2] Daniel Schemmel et al. "Symbolic Partial-Order Execution for Testing Multi-Threaded Programs". In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 376–400. ISBN: 978-3-030-53288-8.

[3] Madan Musuvathi, Shaz Qadeer, and Thomas Ball. *CHESS: A systematic testing tool for concurrent software*. Tech. rep. MSR-TR-2007-149. 2007, p. 16. URL: https://www.microsoft.com/en-us/research/publication/chess-a-systematic-testing-tool-for-concurrent-software/.

[4] Google Project Zero. *SockFuzzer*. URL: https://github.com/googleprojectzero/SockFuzzer.

[5] Google Project Zero. *Concurrence*. URL: https://github.com/googleprojectzero/SockFuzzer/blob/1bf18a775ee886af7cfe1c0fc858b19c09e1352e/third_party/concurrence/.

[6] Ned Williamson. *Catch Me If You Can*. URL: https://github.com/googleprojectzero/SockFuzzer/blob/1bf18a775ee886af7cfe1c0fc858b19c09e1352e/third_party/concurrence/presentations/catch_me_if_you_can.pdf.

[7] Mike Pederson. *mpdn/unthread*. URL: https://github.com/mpdn/unthread.

[8] *Competition on Software Verification (SV-COMP)*. URL: https://sv-comp.sosy-lab.org.

[9] Google. *Centipede - a distributed fuzzing engine. Work-in-progress*. URL: https://github.com/google/fuzztest/tree/main/centipede#readme.

[10] Dongdong Deng, Wei Zhang, and Shan Lu. "Efficient Concurrency-Bug Detection across Inputs". In: *SIGPLAN Not.* 48.10 (2013), pp. 785–802. ISSN: 0362-1340. DOI: 10.1145/2544173.2509539. URL: https://doi.org/10.1145/2544173.2509539.

[11] Zu-Ming Jiang et al. "Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection". In: *Network and Distributed Systems Security (NDSS) Symposium 2022* (). DOI: 10.14722/ndss.2022.24296. URL: https://par.nsf.gov/biblio/10351217.

[12] Nathan Wachholz. *Fix memory leaks due to incorrectly set stack ownership*. URL: https://github.com/mpdn/unthread/pull/4.

[13] Google. *OSS-Fuzz: continuous fuzzing for open source software*. URL: https://github.com/google/oss-fuzz.

7 Appendix

7.1 Interleave Example

Here is the `interleave` example from our benchmark suite. It illustrates an extreme example of the kind of bug we would like to detect. Although contrived, the following program has a bug that is only reachable under a *very* specific schedule:

```
void* buggy(int *val) {
  // buggy must be the first
  // to execute
  if(*val != 0) return 0;
  pthread_yield();
  // incr must execute 2 times,
  // then back to buggy
  if(*val != 2) return 0;
  pthread_yield();
  // incr must execute 3 times,
  // then back to buggy
  if(*val != 5) return 0;
  pthread_yield();
  // etc.
  if(*val != 7) return 0;
  pthread_yield();

  if(*val != 12) return 0;
  pthread_yield();

  if(*val != 15) return 0;
  pthread_yield();

  if(*val == 17) {
    __builtin_trap(); // a "bug"!
  }
}
```

```
void* incr(int* val) {
  while(*val < 20) {
    pthread_yield();
    (*val)++;
  }
}

void interleave() {
  int val = 0;

  pthread_t a, b;
  pthread_create(&a, NULL,
                 incr, &val);
  pthread_create(&b, NULL,
                 buggy, &val);
  pthread_join(a, NULL);
  pthread_join(b, NULL);
}

int main() {
  interleave();
}
```

Specifically, to reach the "bug", the scheduler needs to make the correct choice of thread to resume at each of at least 25 yield points[2]. It's reasonable to say that a random schedule has a probability of *at best* 1 in 2^{25} $(1/33,554,432)$ to trigger the bug.

Using `centipede` and `unthread`, the bug is reached in 0.1s. After running with `pthreads` for 5 days, it still hadn't found the bug. In a real target, the "bug" could be from a sanitizer (ASAN, MSAN, TSAN, etc.), an assertion failure, or any other crash.

[2] at least 2 in `main` during thread creation, 6 in `buggy`, 17 in `incr`

7.2 Fuzzing Drivers

Here is an example of our fuzzing drivers, and demonstrating how to configure `unthread`.

```c
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  // Schedule data
  unthread_configure((struct entropy_configuration){
    .entropy_source = ENTROPY_SCHEDULE,
    .schedule = {
      .data = (uint32_t*) data,
      .data_len = size / sizeof(uint32_t),
      .end_behavior = SCHEDULE_END_ZEROS,
    },
  });

  interleave();
}

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  // PRNG Seed
  uint32_t state[4] = {0, 0, 0, 0};
  if(size > sizeof(state)) {
    return 0;
  }
  memcpy((void *) state, data, size);

  unthread_configure((struct entropy_configuration){
    .entropy_source = ENTROPY_PRNG_SEED,
    .prng_seed = {
      .state = {state[0], state[1], state[2], state[3]},
    },
  });

  interleave();
}
```