End-to-end Deadlock Debugging Tools at Facebook

Kenny Yu, Software Engineer, Facebook

Motivation: C++ and Deadlocks at Facebook

Facebook has an enormous and complex C++ codebase. **Facebook uses C++ primarily for developing backend services.** Examples of backend services include newsfeed ranking, ads personalization, search, and messaging.

With such a large codebase and huge scale of deployments, it can be difficult to detect and prevent bugs in production services at Facebook. **Deadlocks in particular are worrisome**—they can be difficult to detect, as the service may appear alive but is failing requests. Further, deadlocks can be easily introduced unintendedly and can be difficult to identify in a large codebase. **What tools can we use to make detecting, debugging, and preventing deadlocks easier?**

Approach: “End-to-end” Debugging Tools

Service developers at Facebook are responsible for all phases of the service life cycle. **We have developed end-to-end tooling to help service developers detect deadlocks throughout all of the phases of the service life cycle (Figure 1).** At Facebook, these tools have helped catch existing deadlocks in production for multiple services, and have helped prevent new regressions from ever reaching production.

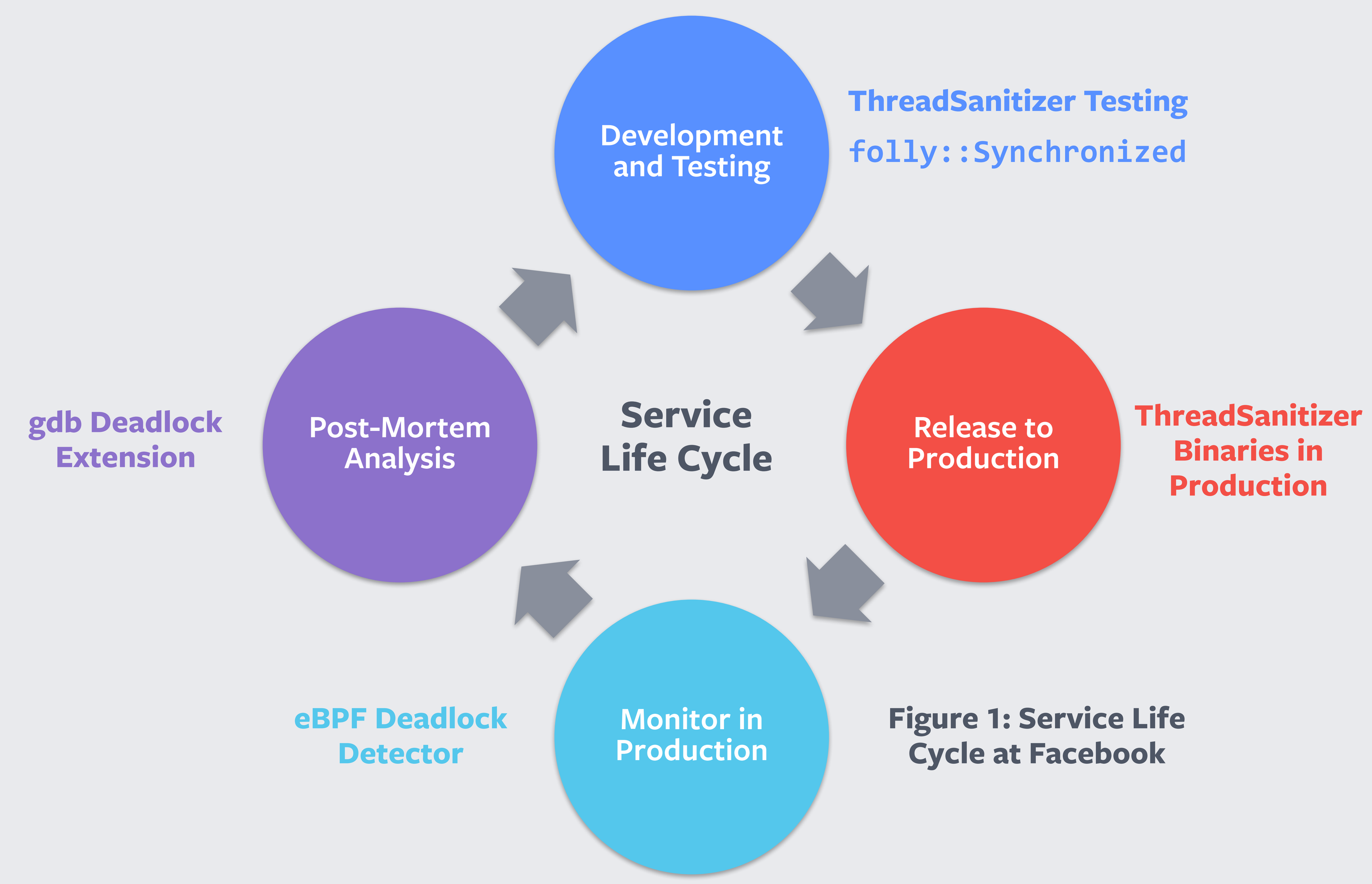

Figure 1: Service Life Cycle at Facebook

Deadlock Debugging Tools Summary

Tool	Need to recompile?	Can use on dead process?	Overhead?	Need to rewrite code?	Works for...	Bugs it can catch
folly::Synchronized	Yes	No	None	Yes	Lockable, SharedLockable traits	Data race
ThreadSanitizer	Yes	No	Huge. Up to 10x memory, 15x slowdown	No	std::mutex, and other mutexes with TSAN annotations	Many types of synchronization bugs, including lock inversion
eBPF deadlock detector	No	No	None when tool is not running. Huge when on.	No	pthread_mutex_t, std::mutex	Lock inversion
gdb deadlock extension	No	Yes	None	No	pthread_mutex_t, std::mutex, other mutexes with metadata about ownership	Deadlock

Figure 2: Pros and cons of the deadlock debugging tools and when to use each tool

How can we prevent deadlocks during development and testing?

Development and Testing

Approach: Use `folly::Synchronized`.

- C++ library makes it impossible to access data without acquiring corresponding mutex.
- **API encourages smaller critical sections, reducing the probability of deadlocks.**

```
class Counter {
  void add(int n) {
    state_->wlock([&n](auto& state) {
      state.counter += n;
      state.deltas.push_back(n);
    });
  }
  int get() const {
    return state_->rlock([](const auto& state) {
      return state.counter;
    });
  }

private:
  struct State {
    int counter;
    std::vector<int> deltas;
  };
  folly::Synchronized<
    State, std::shared_mutex> state_;
};
```

Figure 3: `folly::Synchronized` example to implement a counter with the history of changes to the count.

The lines in red are the critical sections.

Approach: Run all tests with ThreadSanitizer (TSAN).

- ThreadSanitizer instruments all memory access and mutex acquisitions and can report data races, lock inversions, and other types of synchronization bugs.
- **Recompile and run all existing tests with TSAN.** Developers must wait for all tests to pass during code review before landing the change.

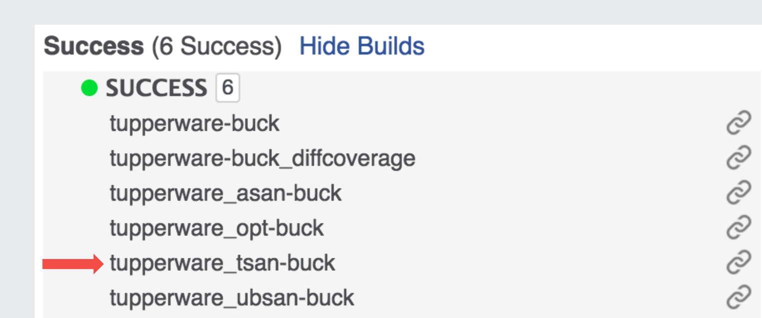

Figure 4: Screenshot of Facebook's continuous integration builds during code review

How can we detect deadlocks before releasing to all of production?

Release to Production

Approach: Push ThreadSanitizer builds of release candidates to a subset of production (canary).

- Send a subset of production traffic or *shadow traffic* to ThreadSanitizer builds.
- This allows us to **exercise production code paths** with ThreadSanitizer and find bugs that cannot be caught with normal testing.

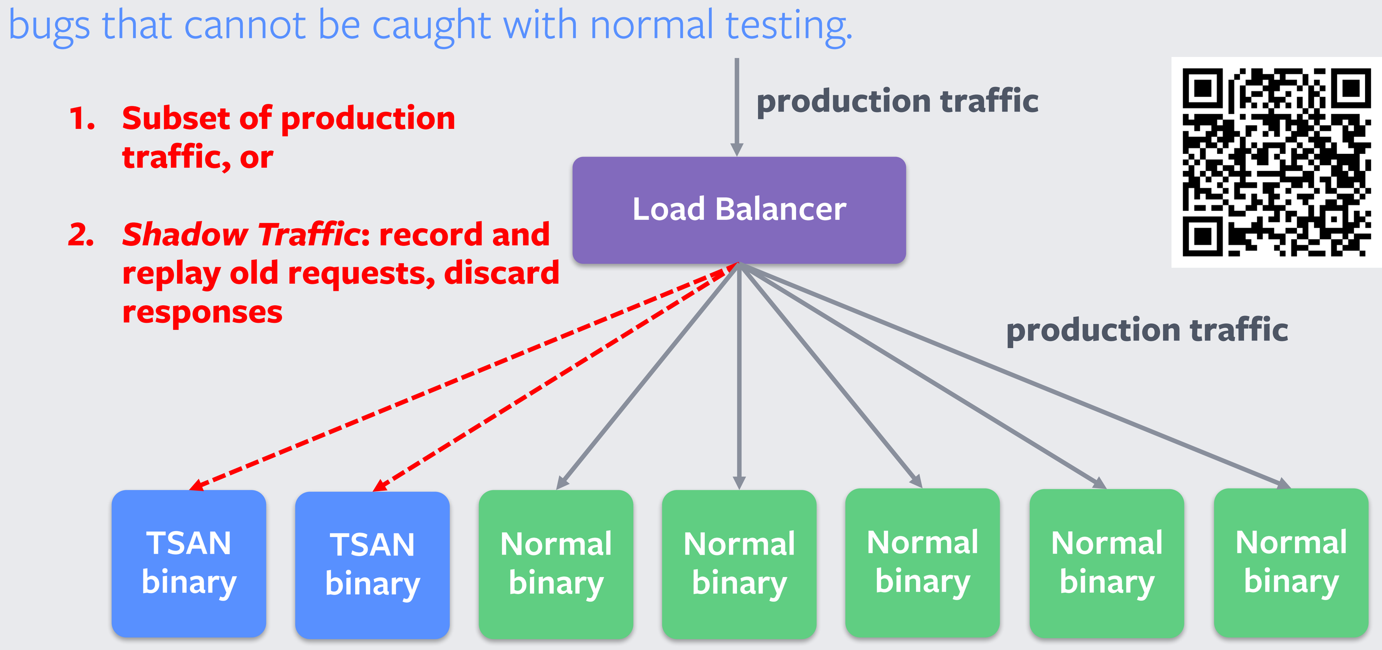

Figure 5: How we canary TSAN builds to a subset of production

How can we monitor for deadlocks in a running production service?

Monitor in Production

Constraints: We cannot restart or recompile the service.

Approach: Use *Linux eBPF* to trace mutex acquisitions.

- *Linux eBPF* allows us to run custom hooks whenever a process hits the specified userspace or kernel symbol.
- Maintain a map of thread to currently held mutexes for that thread. **Anytime a mutex B is acquired while mutex A is already held, add edge `(A,B)`. There is a lock inversion (potential deadlock) if there is a cycle in the graph.**
- To monitor for deadlocks in production, **we periodically run the detector on a subset of production machines** and log the results if a lock inversion is detected.

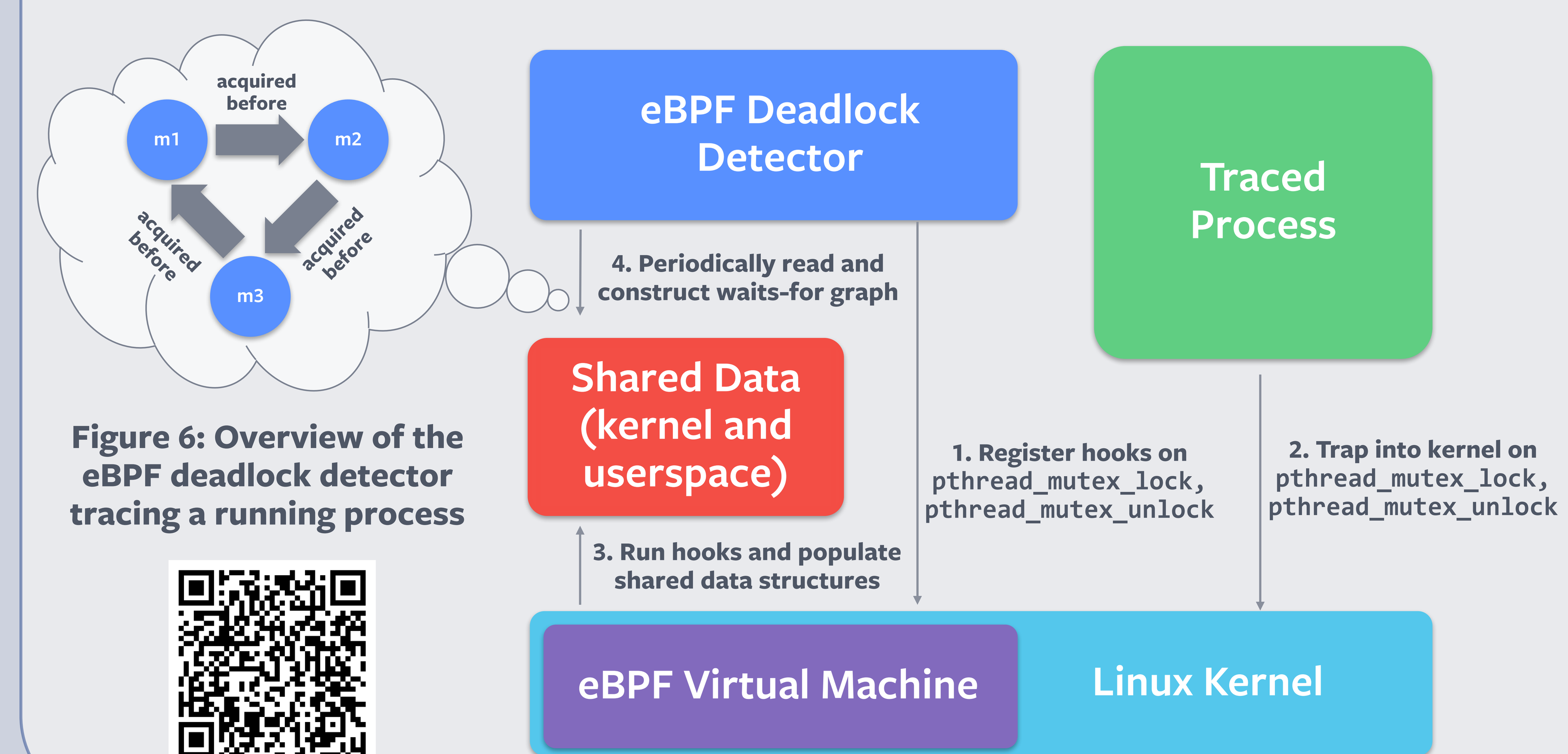

Figure 6: Overview of the eBPF deadlock detector tracing a running process

How can we detect deadlocks after the service has deadlocked or crashed?

Post-Mortem Analysis

Approach: Create a gdb extension to analyze mutex internals.

- Our gdb extension introduces a new **`deadlock`** command that can be used with a hung process or core dump.
- The NPTL implementation of **`pthread_mutex_t`** keeps track of the **owner** of the mutex. The extension uses this information to build the waits-for graph.
- **Edge `(A,B)` exists if thread A is waiting on a mutex held by thread B. There is a deadlock if there is a cycle in this graph.**

```
$ gdb -p 3406029
(gdb) deadlock
Found deadlock!
Thread 4 (LWP 3406032) is
waiting on mutex
(0x00007fffd0bed7c8) held
by Thread 5 (LWP 3406033)
Thread 5 (LWP 3406033) is
waiting on mutex
(0x00007fffd0bed818) held
by Thread 3 (LWP 3406031)
Thread 3 (LWP 3406031) is
waiting on mutex
(0x00007fffd0bed7f0) held
by Thread 4 (LWP 3406032)
```

```
(gdb) thread 3
(gdb) bt
#0  __lll_lock_wait ()
#1  0x00007fb051d4cf44 in
pthread_mutex_lock
(mutex=0x7fffd0bed7f0)
...
(gdb) frame 1
(gdb) p *mutex
$1 = {__data = {__lock = 2,
__count = 0, __owner = 3406032,
... } ...},
(gdb) thread find 3406032
Thread 4 has target id 'Thread
0x7fb04f14a700 (LWP 3406032)'
```

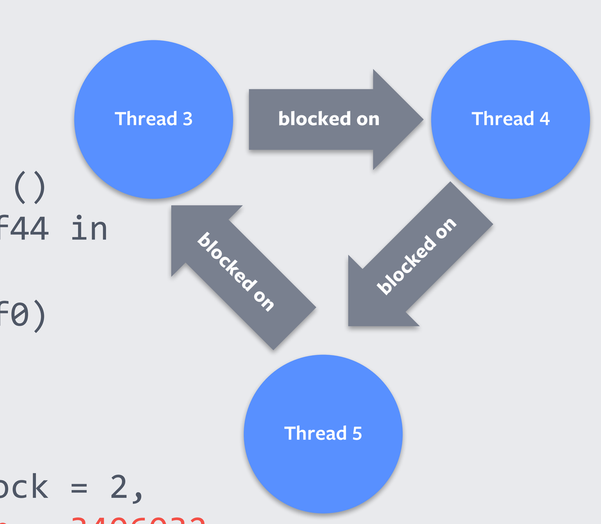

Figure 7: Example output from the gdb `deadlock` extension. The extension inspects the mutex internals to find the current owner of a mutex.