Argus: Understanding Livelocks Through Full-System Tracing

SOSP 2019 submission #XX

Abstract

1 Introduction

Deadlocks and livelocks are known to be significant challenges in developing distributed systems. A great deal of work has gone into formal analysis of call graphs to try to identify deadlocks. In practice, when faced with code that deadlocks or may deadlock, many developers will simply add timeouts to all code that obtains locks, converting a show-stopping hang into a program that will eventually unfreeze itself. However, the root cause of the cyclic dependency may go unaddressed, leading to significant user-facing delays in applications.

It is difficult for developers to diagnose livelocks with typical tracing or debugging tools (like `DTrace` or `lldb`) in user-facing applications for a number of reasons. First, many function calls happen asynchronously through events or inter-process communication, especially in graphical programs. This means that control flow travels through the kernel, and the root cause can be far away from an observed hang. Many processes and threads may be involved, and it is unclear which ones to target. Secondly, there is a vast number of events being triggered constantly, and sifting through all the innocuous events to find relevant ones is a Herculean task. Deadlocks that can be deterministically triggered are challenging enough, but this problem is exacerbated by livelock situations which by their nature involve the execution of a continuous stream of operations.

In this work, we present our system *Argus* which collects detailed tracing information across all processes on a Mac system. Argus collects relevant userspace and kernelspace events and messages, and automatically correlates them between processes. We allow trace points to be inserted at arbitrary locations in each process, allowing a developer to collect additional information as they hone in on the cause of a livelock or deadlock.

One important aspect of analyzing livelocks is that timeouts typically use real wall time, and any performance overhead incurred by an analysis tool may cause timeouts to expire more quickly, affecting program behaviour. Our framework incurs minimal overhead of XXX% in large-scale tests. We also present two case studies of Argus applied to real-world problems, including a livelock in Google Chrome. We believe that Argus represents an important step forward in debugging livelocks in user applications.

2 Design

- record system state (userspace + kernel)
- iteratively hone in on the problem
  - thread relationships (improve linkage between logs)
  - find shared variables (pthread mutexes, and manually)
  - add new trace points for callstacks
- find two distinct states (good, and buggy)
- verify root cause by adjusting timeout values
- patch the problem (in source, runtime, or at binary level)

3 Implementation

The XXX prototype consists of a set of tools: a tracing tool to collect system-wide events, an offline analyzer to identify event processing boundaries, reveal thread dependency with time information and narrow down the activities that correspond to the bug, and an online conditional debugging tool to verify our findings.

3.1 Tracing tool

Our tracing tool is built upon Apple's trace, a lightweight kernel debugging tool used to collect system-wide tracing data. It exposes API for adding tracing points in the system. Each tracing point has a type name to notate current activity in the system. Every time the tracing point gets invoked, data entry is generated. We call it a tracing event. Each event contains the timestamp and carries arbitrary attributes. The entries will be written to a memory buffer, and eventually printed to the standard output or dumped to the file in a batch.

In order to capture the performance anomaly now and then, the tracing tool should be able to run 24X7 without using up the storage. Therefore, the tracing tool is modified to dump data to file in a round-robin style. Only the most recently tracing data is saved. On our experiments, we set the file size 2GB to cover not only the perceived

buggy case but also preceding normal execution for further comparison.

3.1.1 Tracing points

To make the data pithy, as few tracing points as possible are included in our tool. Meanwhile tracing points that helps to diagnose the root cause can be added on demand.

Tracing Points Category	Tracing points Location	Recoding Purpose
IPC	mach_msg_send, mach_msg_recv	reveal connections
Thread Scheduling	wait, make_runnable	reveal connections
System calls	system_call	understanding
Interrupts	interrupts	identify boundaries

Table 1: Tracing Points from Apple

We take the tracing points in Table 1 shipped by Apple, and augment the infrastructure with additional ones shown in Table 2, in order to reveal request boundaries in a single thread and catch the relationships among threads. As shown in Table 1, tracing points on mach_msg are used to tracing the IPC in the system, to catch the relationships among the threads. Meanwhile, additional tracing points is needed to complete the connection of threads involved in the IPC, in that the mach msg is designed for uni-direction. The reply messages are not necessarily received by the same thread sending out request message, but a port name will be carried to receive reply. As is shown in Table 2, tracing points that carry the port name information are added in the mach_msg_send.

Tracing points on thread scheduling are useful on identification of the general correlations among threads. The scenario that thread A wakes up thread B usually indicates thread B depends on thread A. However, it is not always true. For example, the kernel may clear wait condition for specific threads, but they are not logically correlated. Another example is that timer interrupt in any thread may wake up the kernel thread to process the timer. Therefore we add tracing points with the scheduling reason in the assert_wait and make runnable, which can rule out the false connections.

It is noted that wait and wake-up often occur inside the system calls. Thus, we include the tracing points on system calls, with the user passed in arguments, to make the tracing data more comprehensible. Moreover, the arguments in the synchronization system calls help to connect threads synchronized.

As mentioned above, the interrupts will cause the kernel to post tracing event on the thread whose control is taken over. As a result, the tracing points in the begin and end of interrupt help to isolate the unrelated task.

In addition to the existing tracing points, we also realized that userspace programming paradigms would mess up the dependency of threads with false positive and false negative connections.

Asynchronous and batch processing are everywhere, especially in GUI apps. We add tracing points in the libraries that implement such mechanisms to our best knowledge. The submission of the tasks and the drainage of them are traced to reveal connections of threads, while the begin and end of each task are also traced in order to identify the execution boundary in a worker thread.

Continuously processing irrelevant tasks are common in kernel thread and daemons like WindowServer. In order to exclude the false positive correlation of threads, tracing points are required to identify the boundary. We apply the heuristic that if they are processing messages from different applications, they should be working for different tasks.

Backtrace helps in understanding the semantics of the tracing data. As mach_msg_trap is the most widely used system calls to implement the communications among threads and processes, we add lightweight backtrace for it. Dump out the full call stack is costly. Subsequently, the lightweight backtrace only delivers the frames unwinded towards the valid rbps.

3.1.2 Instrumentation

The libraries as well as lots of the daily Apps on MacOS are closed source. Techniques, such as library preload with trampolines over the targeted functions, don't meet the requirements due to the two-layer namespace of dynamic libraries in Mac. If the instrumented target is invoked inside the same library, the function in the original library will be executed without going through the tracing point, which results in missing of tracing data. We hence implement the binary instrumentation lib for developers to add tracing points anywhere.

To complete the instrumentation, the developer only needs to point out the address of instruction where the desired attributes are accessible from its thread state. Meanwhile, the length of the instruction is long enough to hold a function call, and it can be simulated in the callee. We call such instructions that meet the condition as victim instructions, and the function call as a shell function. The shell function is required to add tracing points and simulate the victim instruction.

With the input of the address of victim instruction and its corresponding shell function, our tool will generate a new library, where the init function in Figure 1 is defined to calculate offsets and replace the victim instruction with the shell function. The shell function stores the return address 5 bytes after the victim instruction. If the victim instruction is more than 5 bytes, the rest bytes will be filled with nops. The replacement only happened once in the memory by calling the init externally with dispatch_once.

Finally, the generated library will re-export all the symbols from the original library and replace the orig-

Tracing Points Category	Tracing points Location	Recoding Purpose
IPC	mach_msg_send with ports name info	reveal connections
Thread Scheduling	code called assert_wait with wait resource), code called make_runnable with waken reason	reveal connection
System calls	system_call arguments and return value	understanding
Batch processing	submit tasks to dipatch queue, kqueue, runloop, timer armed	identify execution boundaries
Asynchronous mechanism	drain tasks from dispatch queue, kqueue, runloop, timer fired	connecitons
Kernel interferes	timer interrupt processing, time share maintainance	identify execution boundaries
Deamon interferes	Windowsever message processing	identify execution boundaries
Shared varialbes	system_calls with arguments NSEvents in CoreFoundations manually picked variables, like is_main_thread_spinning pending message in WindowServer processing	reveal connections, comparision reveal connections indicate GUI hangs, comparision identify execution boundaries
Lightweigt backtrace	mach_msg_trap	understanding, comparision

Table 2: Augmented Tracing Points

Figure 1 psuedo code for init

Input: $VictimFuncs_1 \ldots VictimFuncs_N$
$VictimInstOffsets_1 \ldots VictimInstOffsets_N$
$ShellFuncPtrs_1 \ldots ShellFuncPtrs_N$

1: **procedure** INIT
2: **for** $i \leftarrow 1$ to N **do**
3: $InstrVaddr \leftarrow$ *virtual addr of* $VictimFuncs_i + VictimInstOffsets_i$
4: $Offset \leftarrow InstrVaddr + 5 - ShellFuncPtrs_i$
5: $CallqInstr \leftarrow callq\ Offset$
6: $mprotect(page\ of\ InstrVaddr, R|W|E)$
7: $memcpy(InstrVaddr, CallqInstr)$
8: $mprotect(page\ of\ InstrVaddr, R|E)$

inal one. Although one of the shortcomings is that only the short distance function call is supported in our implementation, as the new library and the original one are usually loaded close in memory, we do not need to apply techniques to achieve the long jump in instrumentation.

3.2 Offline Analyzer

With the tracing data, previous works[**? ?**] generates dependency graphs for every user request. However, generating a perfect dependency graph for a request is not easy, especially for systems with closed source libraries, daemons, and apps and widely used asynchronous programming paradigms. Our offline analyzer is designed to extract a feasible dependency graph out of the large tracing data. The dependency graph is not necessarily complete and correct, event not neccessarily for exactly one user transaction, but provides insight into the root cause of the problem.

3.2.1 Dependancy graph

Tracing points in Table 1 and Table 2, such as IPC, thread scheduling, asynchronous programming and shared variables, reveal the thread relationships. For example, the receiver can only proceed after receiving the message from the sender. Our analyzer explored the dependency among threads which send out a message, receive the message, send out the reply and receive the reply message respectively. The tracing data also cover the dependency of two threads that one thread submits an asynchronous work and the other thread executes it. Thread scheduling, that thread A wakes up thread B, usually reveals thread B depends thread A. However, under specific circumstance, the dependency may not hold. One example happens when a thread wakes up the kernel thread for interrupt processing. Therefore we generate the dependency graph with the heuristic to exclude the false positive connection.

On the other hand, not all dependency among threads can be revealed with current tracing data. Shared variables usually reflect the data dependency of two threads. Discover all of them is extremely hard, due to various style, such as arrays, flags in structure, and so on. Fortunately, it is not necessary to find all either, as long as it isn't related to the performance bug. Therefore, the missing connections can be added gradually if the developer believes it is a concern. Our tool can easily set the hardware watchpoints to tracing the operations on the variables.

Moreover, to get a more concise dependency graph for the developer, execution boundaries should be identified to shrink the size of the graph. A worker thread can work for irrelevant requests continuously. As a result, it can expose temporal relationships with various threads. Without boundaries for individual requests, all of them will be included in the one sizeable graph. We make use of tracing points added before and after a request in the asynchronous work to divide the activities in a thread into execution segments. For threads that work for multiple requests, it is also common that the thread may wait for a new request. Therefore, we leverage the tracing point in wait as an execution boundary.

However, no tracing points can be added to universally indicate an execution boundary. From our experience, those tracing points can only be added incrementally based on the observation and validation of the execution segments.

3.2.2 Heuristics for Discovery

Our offline analyzer leverages heuristics to discover the over-connection and missing-connection in our dependency graph. Tracing points can be further added make the dependency graph concise, and to cover as much information related to the buggy case as possible.

To exclude irrelevant tracing data and make the graph concise, we need to discover false connections. Our analyzer will check every execution segments, and if one segment includes communications with two or more user space applications, it is likely not atomic for one request. The result will be reported to the developer, and tracing points can be added to identify the boundary. In our experiment, we discovered multiple programming paradigms in the libraries that are not realized before. WindowServer will compact the request of send and receive in a mach_msg_trap system call, althoug they are on behalf of different userspace applications. The function mig_dispatch_service, as well as the calling out functions in runloop, will finish works one by one in a loop without blocking between them. This procedure to discover such programming paradigm can be repeated on regular executions before tracing for the buggy case.

On the contrary, the tracing data from the buggy case are needed to get the missing information for the bug diagnosis. Gathering the missing information is a time-consuming task in that some bugs could only be reproduced occasionally and such information usually required manually exploring. Checking the call stacks of the threads when the GUI app is stuck provides hints to find shared variables that are bug related.

3.2.3 Comparision

3.3 Online Debugger

3.3.1 Conditional debugging

4 Case Studies

We apply our tools to two realworld bugs. Both of them are long listed in public. The chromium bug has been reported multiple times in the chromium bug report. It is essentially a deadlock bug. Timeout was added and it turned out a livelock bug. Cache mechanism is added to alieviate but not solve it ultimately. The bug of the System Preferences is exposed by an app called DisableMonitor. Basiclly the app can change some system settings and eventually reveal the inappropriate displayer management in the Core Graphics.

4.1 Chromium IME responsive

One of the long-lasting performance issues in chromium is hanging caused by the non-English input. When users try to type non English to textFields, such as search box, the main thread of the browser becomes not responsive. With lldb, it is not hard to tell that the main thread gets stuck on FindFirstRect, where the main thread waits for the signal of the condition variable. According to the history in the bug report, the developers realized there were deadlocks somewhere. However, it was hard to pinpoint due to the multiprocess and multithread programming paradigms. As a result, a timeout was added to prevent the deadlock, but not the long latency. Although a further bug patch introducing cache helps to eliminate the long hanging mostly, the performance issue still appears from time to time. The scenario we can reproduce is to launch the website of Yahoo and immediately quickly type Simplified Chinese.

The ground truth we revealed with our tool is as shown in picture XXX. In chromium, there are one browser process and multiple renderer processes. The main thread of the browser process tries to get the caret position. It sends out the message and anticipates for the reply message with a condition variable. Usually, a worker thread in the browser process will return the firstrect and wake up the main thread. However, it requires the message from the main thread of a renderer process to proceed. Without the message from the renderer process, the worker thread is not able to signal the main thread. Thus, the main thread will always time out.

Our trace tool will collect the data system-wide. Therefore, all the thread relationships are captured. With the trace log size, both the hanging case and non-hanging case are recorded. From the shared condition variable between threads, we are able to align the logs of the two cases, and discover the missing message in the hanging case.

As we have known the unresponsive of the main thread in the renderer process, we further consult the analyzed trace log and observe that it is waiting on a semaphore, and eventually waken up by the main thread of the browser process.

Our tool further explains the root cause of the livelock with conditional debugging. We can either apply the binary instrument or modify the source code to make the renderer thread accept the attachment of lldb. The concrete call stacks from lldb disclose the task processing in the renderer thread is related to running javascript.

4.2 System Preferences spin

System Preferences is the application in MacOS for users to modify various settings. The Preference Pane named Displays allows the user to rearrange the position of displays, the location of the menu bar and set parameters for display, such as the resolution, brightness, and rotation, but to disable a monitor online is not supported in Mac. DisableMonitor is an app to easily disable/enable a monitor. It is implemented by calling to the API from Apple, CGSBeginDisplayConfiguration, CGSConfigureDisplayEnabled and CGSComplete-

DisplayConfiguration. The bug appears when an external monitor is diabled with DisableMonitor, and the user drags the windows under the tab of arrangement in System Preference. It makes the System Preferences window freeze for a few seconds.

One straightforward method for the single process app is to attach lldb, resume the process until the system throws out the spinning cursor. However, in the very case, it only tells when the spinning cursor shows up, the main thread is busy calling thread_switch from function CGSCompleteDisplayConfiguration in the call stack. Further static analysis, for example, reverse engineering on the binary, cast light on the programming paradigms of the repeating. In our case, it reveals the main thread is stuck in a loop repeating thread_switch until a variable is set or time out. Nevertheless, it is still hard to answer the question of why and how the variable gets set or if it is cleared by mistake.

Only the comparison of the buggy case and the normal execution can exhibit what is not expected. One advantage of our tool is it is lightweight and does not impact the responsiveness of the thread, we can identify the buggy case and normal case, compared to using lldb to trace through the whole API execution step by step. Another advantage is that it records the activities systemwide, and the data covers from the beginning of the user input, which makes the case more explainable. By adding the hardware breakpoints with our instrumentation tool, we also record the whole history of the variable activity. The tracing on the variale helps to make alignment of the graphs produced in the normal execution and freeze case, which makes the further comparison feasible. It reveals datagram from the WindowServer will post notifications for the application. The particular datagram makes the application to finish reconfiguration and set the variable, while in the spinning case the reconfiguration gets initiated but not completed. The handler display_notify_proc is not implemented properly to make the application exit the loop when there is an exception in the reconfiguration.

With the findings of our tool, as is shown in Figure **??**, we can now explain the root cause with the psuedo code in Figure **??** and Figure **??**, which provides insight for the developer to fix the bug.

References