A deadlock bug fix solution for gperftools issues 775

Jingyu Yang@HKU
April 2, 2016

Introduction

An issue(775) of gperftools (https://github.com/gperftools/gperftools/issues/775) is reproduced several times when I enabled heap and CPU profiler together to analyse Redis (http://redis.io/). In this document, I will try to figure out the bug and to explain my patch solution.

Envionment

Linux: Ubuntu 14.04.1 LTS
GCC: gcc (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4
gperftools: Sun Mar 20 12:29:40 2016, 9fd6d2687914a1f58a8ce457d6a1bd3d55ea0747
glibc: (Ubuntu EGLIBC 2.19-0ubuntu6.6) 2.19

Usage:
I statically linked libtcmalloc_and_profiler.a into redis and called ProfilerStart() and HeapProfilerStart() at the first line of main() in the source code of the redis. Then I called ProfilerStop() and HeapProfilerStop() when redis will be closed. When the redis-server is running, an environment variable is set CPUPROFILE_FREQUENCY=1000000.

Problem

The redis-server process will hung when the following command runs.
env CPUPROFILE_FREQUENCY=1000000 ../bin/redis-server ./redis.conf

I used gdb to attach the process of the redis-server, then I found a typical deadlock occurred when I use bt command to show the backtrace. The output of gdb is attached as Figure 1 for reference.

The reason of why this is a deadlock is that in frame#2 and frame#12, the program called pthread_mutex_lock() at the same mutex object(0x8315a0), in this case the program will hung at the second pthread_mutex_lock() forever.

```
(gdb) bt
#0  __lll_lock_wait () at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007f00828fd657 in _L_lock_909 () from /lib/x86_64-linux-gnu/libpthread.so.0
#2  0x00007f00828fd480 in __GI___pthread_mutex_lock (mutex=0x8315a0
<object_mutex>)
   at ../nptl/pthread_mutex_lock.c:79
#3  0x000000000055ddea in _Unwind_Find_FDE ()
#4  0x000000000055a1e1 in uw_frame_state_for ()
#5  0x000000000055b75d in uw_init_context_1 ()
#6  0x000000000055c348 in _Unwind_Backtrace ()
#7  0x0000000000557b2a in GetStackTraceWithContext_libgcc (result=<optimized out>,
   max_depth=<optimized out>, skip_count=<optimized out>, ucp=<optimized out>)
   at src/stacktrace_libgcc-inl.h:100
#8  0x00000000005582bc in GetStackTraceWithContext
(result=result@entry=0x7ffc82d12f88,
   max_depth=max_depth@entry=63, skip_count=skip_count@entry=3,
uc=0x7ffc82d13200)
   at src/stacktrace.cc:305
#9  0x00000000005447c1 in CpuProfiler::prof_handler (signal_ucontext=<optimized out>,
   cpu_profiler=0x808200 <CpuProfiler::instance_>, sig=<optimized out>) at
src/profiler.cc:360
#10 0x000000000054508b in ProfileHandler::SignalHandler (sig=27,
sinfo=0x7ffc82d13330,
   ucontext=0x7ffc82d13200) at src/profile-handler.cc:530
#11 <signal handler called>
#12 0x00007f00828fd47a in __GI___pthread_mutex_lock (mutex=0x8315a0
<object_mutex>)
   at ../nptl/pthread_mutex_lock.c:79
#13 0x000000000055ddea in _Unwind_Find_FDE ()
#14 0x000000000055a1e1 in uw_frame_state_for ()
#15 0x000000000055c379 in _Unwind_Backtrace ()
#16 0x0000000000557b7a in GetStackTrace_libgcc (result=<optimized out>,
max_depth=<optimized out>,
   skip_count=<optimized out>) at src/stacktrace_libgcc-inl.h:100
#17 0x0000000000558234 in GetStackTrace (result=result@entry=0x7ffc82d13bc0,
   max_depth=max_depth@entry=42, skip_count=skip_count@entry=1) at
src/stacktrace.cc:294
#18 0x000000000054fa66 in MallocHook_GetCallerStackTrace (result=0x7ffc82d13d50,
max_depth=32,
   skip_count=<optimized out>) at src/malloc_hook.cc:645
#19 0x000000000054b915 in RecordAlloc (skip_count=0, bytes=16, ptr=0x18df6e0)
   at src/heap-profiler.cc:319
#20 NewHook (ptr=0x18df6e0, size=16) at src/heap-profiler.cc:342
#21 0x000000000054fe32 in MallocHook::InvokeNewHookSlow (p=p@entry=0x18df6e0,
s=s@entry=16)
   at src/malloc_hook.cc:498
#22 0x000000000055e176 in InvokeNewHook (s=16, p=<optimized out>) at
src/malloc_hook-inl.h:127
#23 tc_malloc (size=16) at src/tcmalloc.cc:1604
```

```
---Type <return> to continue, or q <return> to quit---
#24 0x0000000000433266 in zmalloc (size=16) at zmalloc.c:105
#25 0x000000000044c439 in createObject (type=0, ptr=0x2b2) at object.c:40
#26 0x00000000004200c0 in createSharedObjects () at server.c:1439
#27 0x0000000000421d96 in initServer () at server.c:1889
#28 0x000000000042ce4d in main (argc=2, argv=0x7ffc82d14178) at server.c:4105
(gdb)
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 0x7f008331c780 (LWP 9448) "redis-server" __lll_lock_wait ()
    at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
(gdb)
```

Figure 1: The output of gdb when redis hungs

Analisys

The main reason is that GetStackTrace() and GetStackTraceWithContext() are not signal reentrant safe.

The second GetStackTraceWithContext () in #8 was called but the first one GetStackTraceWithContext() in #17 had not finished yet.

The implement of both functions is provided by the same function _Unwind_Backtrace() in #6 and #15 in glibc. Unfortunately, the inner implementation of _Unwind_Backtrace() will obtain a pthread_mutex object by calling pthread_mutex_lock(), the source code is available at glibc-2.19\sysdeps\generic\unwind-dw2-fde.c:1018 in Figure 2.

```
fde *
_Unwind_Find_FDE (void *pc, struct dwarf_eh_bases *bases)
{
 struct object *ob;
 fde *f = NULL;

 init_object_mutex_once ();
 __gthread_mutex_lock (&object_mutex);
```

Figure 2: The source code of the implement of Backtrace in glibc.

In Figure2, the mutex object object_mutex will be the same in the same thread. So the second pthread_mutex_lock() in #2 will be hung forever, because the mutex has not been unlocked at the time.

Solution

I plan to solve this reentrancy bug by modifying the source code src/stacktrace.cc of gperftools.

1. A TLS variable named is_entry is introduced as a flag to prevent GetStackTrace*() reentry.

2. This flag will be set true when the first time GetStackTrace*() will be called.

3. This flag will be set false when a GetStackTrace*() returned.

4. The flag setting will be implemented in StacktraceScope, where the constructor and destructor function will be called at the beginning and end of the GetStackTrace*().

5. The value of the flag is_entry will indicate the value of stacktrace_allowed in StacktraceScope to control whether to call GetStackTrace*().

The reason of my solution to prevent signal reentrancy is that:

1. I do want to use a lock to solve the problem of deadlock.
2. GetStackTrace*() may give up to collect trace information if scope.IsStacktraceAllowed() is returned as false, so this solution will not change too much from the original design.
3. The flag is_entry is a TLS variable, so in multi-threading environment, the code will still get the benefits of multi-threading.

The patch code is attached as Figure 3.

```
diff --git a/src/stacktrace.cc b/src/stacktrace.cc
index 395d569..5c98ebf 100644
--- a/src/stacktrace.cc
+++ b/src/stacktrace.cc
@@ -243,13 +243,30 @@ namespace tcmalloc {
namespace {
  using tcmalloc::EnterStacktraceScope;
  using tcmalloc::LeaveStacktraceScope;
+  // A patch for isuess 775. https://github.com/gperftools/gperftools/issues/775
+  __thread bool is_entry = false; // A TLS variable is aimming to prevent GetStackTrace*()
reentry, which will introduce deadlock.
+  bool EnterStacktraceScope_reentry_check(){
+    if (is_entry){          // which means GetStackTrace*() was called and not finished.
+        //printf("[reentry detection] GetStackTrace reentry is detected. Can not do
GetStackTrace() now\n");
+        return false;       // It is unable to do get stack trace because reentry will introduce
deadlock.
+    }else{                  // which means GetStackTrace*() was not called before.
+        is_entry = true;    // Setting a flag will indicate GetStackTrace*() will be called.
+        return true;        // It is safe to call GetStackTrace*()
+    }
+  }
```

```
+  void LeaveStacktraceScope_reentry_release(){
+    is_entry = false;       // It will leave GetStackTrace*(), so clear the flag.
+  }
  class StacktraceScope {
    bool stacktrace_allowed;
  public:
    StacktraceScope() {
      stacktrace_allowed = true;
      stacktrace_allowed = EnterStacktraceScope();
+       if (stacktrace_allowed){
+         stacktrace_allowed = EnterStacktraceScope_reentry_check(); // To check whether it
is a reentry of GetStackTrace*()
+       }
    }
    bool IsStacktraceAllowed() {
      return stacktrace_allowed;
@@ -257,6 +274,7 @@ namespace {
    ~StacktraceScope() {
      if (stacktrace_allowed) {
        LeaveStacktraceScope();
+         LeaveStacktraceScope_reentry_release(); // GetStackTrace*() finished.
      }
    }
  };
```

Figure 3: the patch code for this bug

Testing

After applying this modification, I can get both CPU and heap profile successfully.

Conclusion

I found redis-server process will hung forever when I use gproftools to do CPU and heap profile together. I found there is a deadlock bug after I attached the process by gdb. The reason of the deadlock is that GetStackTrace() and GetStackTraceWithContext() are not signal reentrant safe. The solution for this bug is to introduce a flag named is_entry to prevent GetStackTrace*() reentry. I provided the patch as Figure 3. Any comment is highly appreciated.