Libgcc: When exceptions collide

Max Neunhöffer

A Rabbit Hole

This is the story of a rather deep rabbit hole...

- assembler debugging (endless loops),
- linked lists (always good for trouble),
- mutexes and concurrency,
- Linux perf tools,
- C++ exceptions and their runtime support,
- libmusl, static linking,
- weak references in ELF, ...

The problem

We use **continuous integration** with **Jenkins.**

A **PR must only** be merged, if Jenkins is green.

In my branch, I had **mysterious** shutdown problems.

More details about the testing

Jenkins runs:

- Lots of integration tests (distributed system)
- very high load (many concurrently running tests)
- Timeout would produce **gigabytes of core files**

Furthermore, we use:

- **Static binaries built with libmusl**

```
Program terminated with signal SIGABRT, Aborted.
#0  0x00007f4d1442cc57 in __deregister_frame_info_bases (begin=0x7f4d14adaf60)
    at /home/buildozer/aports/main/gcc/src/gcc-8.3.0/libgcc/unwind-dw2-fde.c:222
222 /home/buildozer/aports/main/gcc/src/gcc-8.3.0/libgcc/unwind-dw2-fde.c: No
such file or directory.
(gdb) bt
#0  0x00007f4d1442cc57 in __deregister_frame_info_bases (begin=0x7f4d14adaf60)
    at /home/buildozer/aports/main/gcc/src/gcc-8.3.0/libgcc/unwind-dw2-fde.c:222
#1  __deregister_frame_info_bases (begin=0x7f4d14adaf60)
    at /home/buildozer/aports/main/gcc/src/gcc-8.3.0/libgcc/unwind-dw2-fde.c:201
#2  0x00007f4d1199b6ce in __do_global_dtors_aux ()
#3  0x0000000000000027 in ?? ()
#4  0x00007f4d1199e192 in runServer (argc=0, argv=0x0, context=...)
    at /work/ArangoDB/arangod/RestServer/arangod.cpp:284
#5  0x00007ffddc78f898 in ?? ()
#6  0x00007f4d14447af5 in _fini ()
#7  0x0000000000000000 in ?? ()
```

The main function has already terminated!

Detective work in a core

By default, no source code for this is available.

This is **annoying**, one does not want to rebuild `libgcc` with debug symbols.

This meant **assembler debugging.**

```
0x7f4d1442cc46 <__deregister+103>      lea  0x28(%rbx),%rax
0x7f4d1442cc4a <__deregister+107>      mov  0x28(%rbx),%rbx
0x7f4d1442cc4e <__deregister+111>      test   %rbx,%rbx
0x7f4d1442cc51 <__deregister+114>      je   0x7f4d1442cc82 <__deregister+163>
0x7f4d1442cc53 <__deregister+116>      mov  0x18(%rbx),%rdi
0x7f4d1442cc57 <__deregister+120>      testb  $0x1,0x20(%rbx)
0x7f4d1442cc5b <__deregister+124>      jne  0x7f4d1442cc6b <__deregister+140>
0x7f4d1442cc5d <__deregister+126>      cmp  %rdi,%rbp
0x7f4d1442cc60 <__deregister+129>      jne  0x7f4d1442cc46 <__deregister+103>

%rip was 0x7f4d1442cc46,
%rbx was 0x7f4d1568e460 and x/6xg 0x7f4d1568e460 showed:

0x7f4d1568e460 :                  0x00007f4d11995000      0x0000000000000000
0x7f4d1568e470 <object.6149+16>: 0x0000000000000000      0x00007f4cd6f00b00
0x7f4d1568e480 <object.6149+32>: 0x000000002190d0d9      0x00007f4d1568e460
```

We are looking at an **endless loop!!!**

The `libgcc` source code

Is here:

https://github.com/gcc-mirror/gcc/blob/4ac50a4913ed81cc83a8baf865e49a2c62a5fe5d/libgcc/unwind-dw2-fde.c#L221

```
struct object
{
  void *pc_begin;
  void *tbase;
  void *dbase;
  union {
      const struct dwarf_fde *single;
      struct dwarf_fde **array;
      struct fde_vector *sort;
  } u;

  union {
      struct {
      unsigned long sorted : 1;
      unsigned long from_array : 1;
      unsigned long mixed_encoding : 1;
      unsigned long encoding : 8;
      unsigned long count : 21;
      } b;
      size_t i;
  } s;

  struct object *next;
};
```

Proving the code to be correct

Being a mathematician, the next I did was:

Prove the code to be correct.

Surely, I would find the bug in **`libgcc`**...

Catching the bug in the act

Monitoring showed (after the fact):

one CPU was spinning at 100%

How could I catch it?

Linux perf tools to the rescue:

```
sudo perf probe -x build/bin/arangod -a "unwind=unwind-dw2-fde.c:1072 ob p"
sudo perf record -e "probe_arangod*" -aR --call-graph dwarf
   (hit Control-C after the recording)
sudo perf script > perf.history
```

Gotcha! … Really???

After **many hours** of continuous running…

(using an identical machine and identical testing load)

It happened again!

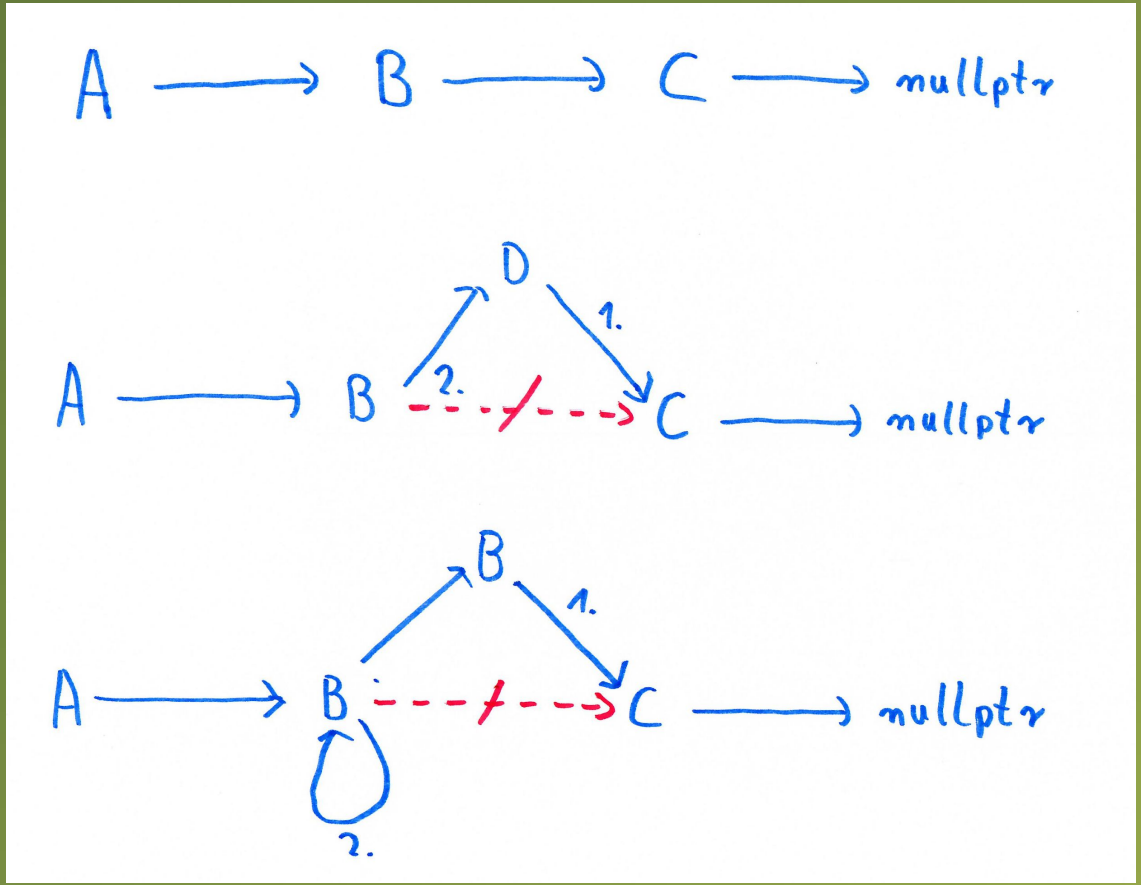
```
MaintenanceWork  8985 [025] 4713776.942241:   probe_arangod:unwind: (7f4d1442cd77) ob=0x7f4d1568e460
                                                                                   p=0x7f4d165b4590
     2c93d77 _Unwind_Find_registered_FDE (inlined)
     2c93d77 _Unwind_Find_FDE (/work/ArangoDB/build/bin/arangod)
     2c91474 uw_frame_state_for (/work/ArangoDB/build/bin/arangod)
     2c92065 uw_init_context_1 (/work/ArangoDB/build/bin/arangod)
     2c924a9 _Unwind_RaiseException (/work/ArangoDB/build/bin/arangod)
     2c3ea25 __cxa_throw (/work/ArangoDB/build/bin/arangod)
      520c33 replicationSynchronize (/work/ArangoDB/build/bin/arangod)
      5251d3 arangodb::maintenance::SynchronizeShard::first (/work/ArangoDB/build/bin/arangod)


MaintenanceWork  8981 [012] 4713776.992994:   probe_arangod:unwind: (7f4d1442cd77) ob=0x7f4d1568e460
                                                                                   p=0x7f4d1568e488
     2c93d77 _Unwind_Find_registered_FDE (inlined)
     2c93d77 _Unwind_Find_FDE (/work/ArangoDB/build/bin/arangod)
     2c91474 uw_frame_state_for (/work/ArangoDB/build/bin/arangod)
     2c92065 uw_init_context_1 (/work/ArangoDB/build/bin/arangod)
     2c924a9 _Unwind_RaiseException (/work/ArangoDB/build/bin/arangod)
     2c3ea25 __cxa_throw (/work/ArangoDB/build/bin/arangod)
      520c33 replicationSynchronize (/work/ArangoDB/build/bin/arangod)
      5251d3 arangodb::maintenance::SynchronizeShard::first (/work/ArangoDB/build/bin/arangod)
```

Impossible - protected traffic by mutex

What must have happened:

- The same exception was thrown **for the first time** at **approximately the same time** (50 ms difference)**.**
- **The same object** was inserted into the linked list **twice.**



Mutex - or no mutex - that is the question

BUT: There is a Mutex! Or is there?

https://github.com/gcc-mirror/gcc/blob/4ac50a4913ed81cc83a8baf865e49a2c62a5fe5d/libgcc/unwind-dw2-fde.c#L1046

```
static inline int
__gthread_mutex_lock (__gthread_mutex_t *__mutex)
{
  if (__gthread_active_p ())
    return __gthrw_(pthread_mutex_lock) (__mutex);
  else
    return 0;
}
```

https://github.com/gcc-mirror/gcc/blob/4ac50a4913ed81cc83a8baf865e49a2c62a5fe5d/libgcc/gthr-posix.h#L744

```
#ifdef __GLIBC__
__gthrw2(__gthrw_(__pthread_key_create),
     __pthread_key_create,
     pthread_key_create)
# define GTHR_ACTIVE_PROXY __gthrw_(__pthread_key_create)
#elif defined (__BIONIC__)
# define GTHR_ACTIVE_PROXY __gthrw_(pthread_create)
#else
# define GTHR_ACTIVE_PROXY __gthrw_(pthread_cancel)
#endif

static inline int
__gthread_active_p (void)
{
  static void *const __gthread_active_ptr
    = __extension__ (void *) & GTHR_ACTIVE_PROXY;
  return __gthread_active_ptr != 0;
}
```

The crucial finding...

The music plays here:

https://github.com/gcc-mirror/gcc/blob/4ac50a4913ed81cc83a8baf865e49a2c62a5fe5d/libgcc/gthr-posix.h#L156

and here:

https://github.com/gcc-mirror/gcc/blob/4ac50a4913ed81cc83a8baf865e49a2c62a5fe5d/libgcc/gthr-posix.h#L89

Linking is black magic of the worst kind

A **weak reference** to sth in an object file is one

- that is resolved to something, **if that sth is present**
- and to **`nullptr`** otherwise.

Here, on Linux, a weak ref to **`pthread_create`** is used.

Rationale:

- A multi-threaded program links **`pthread_create`**,
- A single-threaded program **does not**.

When exactly does it happen?

- We are on **Linux**,
- we are **not** using **GLibC**,
- we are not on Android with the Bionic C-library,
- we **statically link** the executable,
- we do not explicitly use `pthread_cancel`,
- the **first exception** that happens, happens **in two threads exactly at the same time**
- in one thread the execution is suspended at an unlucky time during the list manipulation in `_Unwind_Find_FDE`.

Whose fault is it?

This is an **unholy alliance** between

- over-optimization for the single-threaded case,
- subtle differences between `GLibC` and `libmusl`,
- using black linking magic and the preprocessor and
- exception handling complexity.

Alles wird gut...

I reported this problem:

- to **`libgcc`** first, who said it is a bug in **`libmusl`**,
- then to **`libmusl`**, who said it is a bug in **`libgcc`**,
- then an argument broke out and it turned out that a corresponding fix was done previously for **`libgfortran`** and **`libstdc++`**
- finally fixed for gcc-10 in **`libgcc.`**

Lessons learned

- **Do not over-optimize** at the cost of unnecessary complexity.
- C++ runtime and **exception handling is complicated**.
- C++ exceptions can be quite expensive when actually thrown
- **C-code** can be **hard to understand**, in particular in the presence of lots of platform-specific special cases and **preprocessor magic**.
- The **Linux perf tools** are an incredible **asset** for debugging.
- **Open source** makes such an investigation possible in the first place.
- The teams developing these fundamental tools like **gcc** and **libmusl** are **very responsive** and actually help when problems arise.

Thank you!

Resources

- Blog: https://www.arangodb.com/2019/09/when-exceptions-collide/
- Gcc bug report: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=91737
- Libmusl bug report: https://www.openwall.com/lists/musl/2019/09/17/1