LPC17 - Supporting newer toolchains in the kernel

Bernhard “Bero” Rosenkränzer <bero@linaro.org>

Linaro

LMG
Mobile

Not too long ago, things were easy...

There was one compiler (gcc) and one linker (BFD ld), and one set of tools to extract information from binaries during builds… (binutils nm, objdump etc.)

But now there’s choice...

These days, clang is a great compiler, gold has become the default linker on many Linux distributions, lld 5.0 is the first version of LLVM’s linker that works well on Linux/ELF targets, llvm-nm, llvm-objdump, eu-nm and eu-objdump (from elfutils) are getting ready to replace their traditional counterparts.

Why not just pick one? gcc+ld.bfd work great!

True, but:

- Different compilers warn about different possible breakages.

  ```
  void bt_pair(char n[30]) {
      …
      if(!memcmp(buffer, n, sizeof(n))) {
          …
      }
  }
  ```

  From an Android vendor kernel -- first detected when trying to build it with clang

Why not just pick one? gcc+ld.bfd work great!

- Different compilers warn about different possible breakages.

```
UCHAR a[X];
…
for(int i=0; i<X; i++)
    b = a ? tagCpe++ : tagSce++;
```

From the MPEG reference encoder - bug present for more than 10 years, before we tried to build it with clang

Why not just pick one? clang works great!

- It works both ways - trying to build Android userspace (which is built with clang and only clang these days) with gcc 7.2 instantly finds some bugs.

Why not just pick one? gcc+ld.bfd work great!

- Supporting multiple options is always a good idea [unless one option simply sucks] -- what if the other compiler supports ARMv9, x86_128 or the next CPU architecture we want to support first?
- Can’t hurt to follow the standard - a future version of $SUPPORTED_COMPILER is likely to be stricter about standard adherence. Not fixing a nonstandard construct because one compiler currently supports it is just moving trouble to a future release.

Where are we?

- Great progress since last year -- 4.13.1 requires just a few extra patches to be buildable with clang (tested on aarch64 and x86_64):
    - VLAIS (variable length arrays in structs) - a gcc extension clang will never support - still used in:
        - Exofs
        - i2400 WIMAX driver
    - Nested function still present in drivers/md/bcache/sysfs.c
    - Code triggering clang warnings (and -Werror failures) still present in arch/mips
    -

Where are we?

- Building x86_64 kernels worked well with clang 3.8, but we're hitting a new bug with clang 5.0 (released a couple of days ago):

  https://bugs.llvm.org/show_bug.cgi?id=34537

  Affects various crypto drivers, should be fixed soon

What else can we do to support newer toolchains?

- Currently, the kernel uses -std=gnu89 -- C89 standard with GNU extensions
- Modern compilers (current versions of both gcc and clang) default to C11. This is what compilers have come to expect and what they optimize for (e.g. inlining algorithms are tuned for C99/C11 inline semantics)
- The kernel already uses some C99/C11 code (initializers etc.) that happen to be GNU extensions with gnu89 as well -- gcc and clang know about gnu89, the next compiler we might want to support might not (but if it is worthwhile it will certainly know C11)

What keeps us from going for -std=gnu11?

- Compile time errors:

```
drivers/staging/rtl8723bs/core/rtw_btcoex.o: In function `is_multicast_mac_addr':
rtw_btcoex.c:(.text+0x0): multiple definition of `is_multicast_mac_addr'
drivers/staging/rtl8723bs/core/rtw_ap.o:rtw_ap.c:(.text+0x9d0): first defined here
drivers/staging/rtl8723bs/core/rtw_btcoex.o: In function `is_broadcast_mac_addr':
rtw_btcoex.c:(.text+0x20): multiple definition of `is_broadcast_mac_addr'
```

- Caused by updated inline semantics -- easy to fix (just make is_multicast_mac_addr and is_broadcast_mac_addr `static inline` instead of plain `inline`)

What keeps us from going for -std=gnu11?

- Compile time errors:

```
arch/x86/entry/entry_64.o: In function `entry_SYSCALL_64_fastpath':
arch/x86/entry/.tmp_entry_64.o:(.entry.text+0x100): undefined reference to `syscall_return_slowpath'
arch/x86/entry/entry_64.o: In function `ret_from_fork':
(.entry.text+0x2e1): undefined reference to `syscall_return_slowpath'
arch/x86/entry/entry_64.o: In function `retint_user':
(.entry.text+0xa26): undefined reference to `prepare_exit_to_usermode'
lib/decompress_unlzo.o: In function `unlzo':
decompress_unlzo.c:(.init.text+0x253): undefined reference to `parse_header'
make: *** [Makefile:1000: vmlinux] Error 1
```

- Also caused by updated inline semantics -- easy to fix.
- Similar issue in video/fbdev/i810/i810_dvt.c

All it takes...

```
$ wget http://people.linaro.org/~bernhard.rosenkranzer/4.13.1-C11.patch
$ diffstat 4.13.1-C11.patch
Makefile                                      |    4 ++--
arch/x86/entry/common.c                       |    4 ++--
drivers/staging/rtl8192u/Makefile             |    2 +-
drivers/staging/rtl8723bs/include/ieee80211.h |    6 +++---
drivers/video/fbdev/i810/i810_dvt.c           |    2 +-
lib/decompress_unlzo.c                        |    2 +-
tools/testing/selftests/sync/Makefile         |    2 +-
7 files changed, 11 insertions(+), 11 deletions(-)
```

All it takes???

- 4.13 with this patch has been running on my x86_64 laptop for 2 weeks, a similarly patched kernel has been running on my aarch64 laptop prototype (QC 820 chipset) for 2 weeks - both without noticeable drawbacks
- But should get more testing under different workloads, different drivers in use etc.

What else can we do to support newer toolchains?

- The kernel build process can make use of some gcc plugins shipped in the kernel tree…
- Right now, the “fix” for clang is
  `# CONFIG_HAVE_GCC_PLUGINS is not set`
- Given the different underlying architectures in the compilers, it is probably not possible to create a common plugin interface that could generate a gcc plugin and a clang plugin based on a couple of `#ifdef` statements (but that would be nice…)

gcc plugins

- Relevant plugins need to be implemented for clang as well
- The cyclomatic complexity plugin probably isn’t relevant (just prints information - if you want the information, just run gcc even if you go on to build the kernel with clang…
- That leaves the latent entropy plugin -- to be done

What else can we do to support newer toolchains?

- Linkers:
- Linking the kernel with gold is generally possible, but triggers errors on some platforms and gold versions

```
LD      arch/x86/realmode/rm/realmode.elf
ld: warning: arch/x86/realmode/rm/header.o: missing .note.GNU-stack section implies executable stack
ld: internal error in do_layout, at ../../gold/object.cc:1821
```

- Linking the kernel with lld requires some tweaking - but mostly on lld’s side https://groups.google.com/forum/#!topic/llvm-dev/jVIJoC8ZXdc

What else can we do to support newer toolchains?

So far, looks like both gcc 8.x and clang 6.x will “just work” if gcc 7.x and clang 5.x are working -- no current plans to enforce extra strictness, rewrite plugin APIs, or the likes (but both are still in relatively early stages of development)

lld 6.x has a better chance of being supportable than 5.x (Linux/ELF support is a high priority for lld now)

What else can we do to support newer toolchains?

… And what can the toolchain communities do to help us support them?

Let’s collect some ideas...

Questions? Comments?

bero@linaro.org