LF *live*

MENTORSHIP SERIES

LF live MENTORSHIP SERIES

Using Clang and LLVM to Build the Linux Kernel

Nathan Chancellor, Maintainer, ClangBuiltLinux

LF live MENTORSHIP SERIES

Introduction

- Upstream kernel contributor since 2018 (~950 patches)
- Started formally maintaining LLVM build support in the kernel in 2021
- Slides: https://nathanchance.dev/lf_mentorship.pdf

Agenda

- What is LLVM?
- How to use it to build the Linux kernel
- Why would you want to use it?
- Fixing issues encountered with LLVM
- Features in the kernel that work due to LLVM

What is LLVM?

- The LLVM Project is “a collection of modular and reusable compiler and toolchain technologies”
- Ultimately, to an end user, it is a toolchain that contains a C compiler, linker, assembler, and other binary utilities
- The modular and reusable aspect of LLVM allows someone to write a language front end and gain access to powerful language-independent optimizations and easy code generation for many architectures (although not as many as `gcc`)

What is LLVM?

	LLVM	GNU
C compiler	`clang`	`gcc`
Assembler	“Integrated”	`as`
Linker	`ld.lld`	`ld`
Binary utilities	`llvm-nm`, `llvm-objcopy`, `llvm-objdump`, `llvm-strip`, ...	`nm`, `objcopy`, `objdump`, `strip`, ...

clang

- C language frontend for LLVM (other language frontends exist, such as `rustc`)
- Advertised as generally drop-in compatible with GCC
  - Somewhat the case in reality, some flags may not be implemented, so kernel handles this in a few different ways
- Multi-targeted binary, change target with `--target` instead of separate binaries
  - `clang --target=aarch64-linux-gnu` vs. `aarch64-linux-gnu-gcc`
- Takes C files, generates LLVM IR (intermediate representation), mutates that through a series of optimization passes, then hands it off for a target specific backend to perform target specific optimizations and actually generate the machine code

ld.lld

- LLVM's linker
- Advertised as being compatible with the GNU linker (in a similar manner as `clang`)
- Multi-targeted binary like `clang`
- Generally faster at linking than GNU ld, especially when debug information is involved

How to get it easily

- Distribution package manager
  - Arch (and derivatives): `pacman -S clang lld llvm`
  - Debian / Ubuntu: `apt install clang lld llvm` (or apt.llvm.org for newer releases)
  - Fedora: `dnf install clang lld llvm`
- kernel.org tarballs (optimized for building the kernel)
- Build it from source (tc-build makes it easy)
  - Example quick start command: `./build-llvm.py --build-targets distribution --build-stage1-only --ref llvmorg-19.1.3`

Using LLVM to build a Linux kernel

- Supported versions: LLVM 13.0.1 and newer (prefer newer versions whenever possible)
- Supported architectures:
  - arm64 / arm
  - hexagon
  - loongarch
  - mips
  - powerpc
  - riscv
  - s390
  - um
  - x86_64 / i386

Using LLVM to build a Linux kernel

- Official kernel documentation
- Kbuild has several variables for various tools that are invoked during the kernel build
  - `CC` for the C compiler
  - `LD` for the linker
  - `AR`, `NM`, `OBJCOPY`, `OBJDUMP`, `STRIP`, `READELF` for binutils
  - `HOSTCC`, `HOSTLD`, `HOSTAR` for building host programs

Using LLVM to build a Linux kernel

```
$ make AR=llvm-ar CC=clang LD=ld.lld NM=llvm-nm OBJCOPY=llvm-objcopy \
OBJDUMP=llvm-objdump READELF=llvm-readelf STRIP=llvm-ar HOSTCC=clang \
HOSTCXX=clang++ HOSTAR=llvm-ar HOSTLD=ld.lld
```

Little long huh? Enter the LLVM make variable to automatically set these values. The above command is equivalent to just

```
$ make LLVM=1
```

Makefile implementation

Using LLVM to build a Linux kernel

- `LLVM=-#` sets `CC=clang-#` `LD=ld.lld-#` `NM=llvm-nm-#` ...
  - Example: `LLVM=-19` to use `CC=clang-19` `LD=ld.lld-19` ...
  - Useful for apt.llvm.org packages

- `LLVM=<prefix>/bin/` sets `CC=<prefix>/bin/clang` `LD=<prefix>/bin/ld.lld`
  - Example: `LLVM=$HOME/toolchains/llvm-19/bin/` to use `CC=$HOME/toolchains/llvm-19/bin/clang` ... (trailing slash is important!!)
  - Useful for toolchains that are install somewhere on disk but not in `PATH`, such as the kernel.org toolchains mentioned earlier
  - Absolute paths are helpful for verifying that the correct toolchain is being used, as an incorrect file path in `PATH` means you might fallback to a different toolchain version.

Using LLVM to build a Linux kernel

- Change targets only through ARCH variable
  - `make LLVM=1` builds a kernel for the same architecture as your host machine
  - `make ARCH=arm64 LLVM=1` builds a kernel for an AArch64 machine
  - `CROSS_COMPILE` is generally unused when using LLVM (powerpc is an exception for now)

- Make sure that `LLVM=` is consistently set in all `make` invocations, as different Kconfig options can be selected or disabled based on toolchain support or versions. Especially important for non-interactive configuration changes (such as `scripts/config` or `sed -i ... .config`), as it will not always be apparent that the correct options are not set correctly.

Example: Building an in-tree configuration target

arm64 virtual machine configuration

```
$ make -j$(nproc) ARCH=arm64 LLVM=1 virtconfig
```

Build kernel image and modules (omitting the make target uses the default, which is “all”)

```
$ make -j$(nproc) ARCH=arm64 LLVM=1
```

Can be combined into a single command like so

```
$ make -j$(nproc) ARCH=arm64 LLVM=1 virtconfig all
```

Example: Building a distribution kernel

Grab current running configuration (could be in `/boot` instead)

```
$ zcat /proc/config.gz > .config
```

If you want to be prompted for new options available because of switching toolchains, skip this step
To see what options were set to their default, diff .config and .config.old

```
$ make -j$(nproc) LLVM=1 olddefconfig
```

Build distro package (such as for Arch Linux, Fedora, or Debian based distributions respectively)

```
$ make -j$(nproc) LLVM=1 pacman-pkg
$ make -j$(nproc) LLVM=1 binrpm-pkg
$ make -j$(nproc) LLVM=1 bindeb-pkg
```

Why build the Linux kernel with LLVM?

Using a whole separate toolchain implementation gives developers and users access to

- A different set and implementation of warnings
  - GCC and Clang implement many of the same warnings but there are warnings that are unique to each compiler and the same warnings may have implementation differences
- Features
  - Optimizations, such as link time optimization (LTO) or profile guided optimization (PGO)
  - Sanitizers, such as the memory sanitizer or kernel control flow integrity (kCFI)
  - Security, such as structure layout randomization and automatic stack variable initialization
    - Recently, security features are being implemented in both compilers simultaneously

Why build the Linux kernel with LLVM?

- Turning both projects against each other can result in more robust code or better supported features; the Linux kernel is one of the largest and most expansive C code bases in the world, which is a great stressor for clang's C support

- `asm goto` was implemented in LLVM because of the Linux kernel and other features have been fixed or improved based on their deployment in the kernel

- Numerous issue have been fixed due to differences between compiler implementations, such as optimization differences around undefined behavior, such as casting away `const`.

Warnings

- Warnings point out potential problems but they have heuristics, which may or may not always be correct. It is always important to think about what exactly the warning is pointing out and look at the context of the code, not just silencing it. When in doubt, ask the code author or maintainer!

- One major difference between GCC warnings and Clang warnings is that the vast majority of Clang's warnings happen in the frontend, whereas GCC implements many warnings in the middle end, which can be influenced by the outcome of optimizations

- This difference can cause Clang to miss problems in code that GCC catches but it often reduces the number of false positives it has

-Wuninitialized

```
int foo(int);
int bar(void)
{
    int x;
    return foo(x);
}
```

```
$ clang -fsyntax-only -Wuninitialized test.c

test.c:5:13: warning: variable 'x' is uninitialized when used here [-Wuninitialized]
    5 |         return foo(x);
      |                    ^
test.c:4:7: note: initialize the variable 'x' to silence this warning
    4 |         int x;
      |             ^
      |              = 0
1 warning generated.
```

-Wsometimes-uninitialized

```
int foo(int);
int bar(int val)
{
    int x;
    if (val > 0)
        x = 1;
    return foo(x);
}
```

```
$ clang -fsyntax-only -Wsometimes-uninitialized test.c

test.c:5:6: warning: variable 'x' is used uninitialized whenever 'if' condition is false
    5 |         if (val > 0)
      |             ^~~~~~
test.c:7:13: note: uninitialized use occurs here
    7 |         return foo(x);
      |                    ^
test.c:5:2: note: remove the 'if' if its condition is always true
    5 |         if (val > 0)
      |         ^~~~~~~~~~~
    6 |                 x = 1;
test.c:4:7: note: initialize the variable 'x' to silence this warning
    4 |         int x;
      |             ^
      |              = 0
1 warning generated.
```

Uninitialized warnings

Unfortunately, quite common since GCC's version is disabled under normal conditions (benefit of having side by side builds!)

Can result in some funny fixes...

```
diff --git a/arch/powerpc/xmon/ppc-dis.c b/arch/powerpc/xmon/ppc-dis.c
index 9deea5ee13f6..27f1e6415036 100644
--- a/arch/powerpc/xmon/ppc-dis.c
+++ b/arch/powerpc/xmon/ppc-dis.c
@@ -156,11 +156,11 @@ int print_insn_powerpc (unsigned long insn, unsigned long memaddr)

   if (cpu_has_feature(CPU_FTRS_POWER9))
     dialect |= (PPC_OPCODE_POWER5 | PPC_OPCODE_POWER6 | PPC_OPCODE_POWER7
         | PPC_OPCODE_POWER8 | PPC_OPCODE_POWER9 | PPC_OPCODE_HTM
         | PPC_OPCODE_ALTIVEC | PPC_OPCODE_ALTIVEC2
-        | PPC_OPCODE_VSX | PPC_OPCODE_VSX3),
+        | PPC_OPCODE_VSX | PPC_OPCODE_VSX3);

   /* Get the major opcode of the insn.  */
   opcode = NULL;
   insn_is_short = false;
```

-Wheader-guard

```
#ifndef _FOO_H_
#define _FOOO_H_

int foo(void);
#endif
```

```
$ clang -fsyntax-only -Wheader-guard test.h

test.h:1:9: warning: '_FOO_H_' is used as a header guard here, followed by #define of a different macro
    1 | #ifndef _FOO_H_
      |         ^~~~~~
test.h:2:9: note: '_FOOO_H_' is defined here; did you mean '_FOO_H_'?
    2 | #define _FOOO_H_
      |         ^~~~~~~
      |         _FOO_H_
1 warning generated.
```

LF live MENTORSHIP SERIES

-Wbitwise-conditional-parentheses

```
#define FLAG1 (1U << 1)
#define FLAG2 (1U << 2)

int get_flags(_Bool val)
{
    return FLAG1 |
           val ? FLAG2 : 0;
}
```

```
$ clang -fsyntax-only -Wbitwise-conditional-parentheses test.c

test.c:7:13: warning: operator '?:' has lower precedence than '|'; '|' will be evaluated first
    6 |         return FLAG1 |
      |                ~~~~~~~
    7 |                val ? FLAG2 : 0;
      |                ~~~ ^
test.c:7:13: note: place parentheses around the '|' expression to silence this warning
    6 |         return FLAG1 |
      |                ~~~~~~~
    7 |                val ? FLAG2 : 0;
      |                ~~~ ^
test.c:7:13: note: place parentheses around the '?:' expression to evaluate it first
    7 |                val ? FLAG2 : 0;
      |                    ^
      |                (             )
1 warning generated.
```

-Wbitwise-instead-of-logical

```
int func_1(void);
int func_2(void);

int wrong_operation(void)
{
    return func_1() > 2 &
           func_2() > 4;
}
```

```
$ clang -fsyntax-only -Wbitwise-instead-of-logical test.c

test.c:6:9: warning: use of bitwise '&' with boolean operands [-Wbitwise-instead-of-logical]
    6 |         return func_1() > 2 &
      |                ^~~~~~~~~~~~
      |                           &&
    7 |                func_2() > 4;
      |                ~~~~~~~~~~~
test.c:6:9: note: cast one or both operands to int to silence this warning
1 warning generated.
```

-Wtautological-compare

Collection of warnings to point out always true or false conditions

```
$ diagtool tree -Wtautological-compare
-Wtautological-compare
  -Wtautological-constant-compare
    -Wtautological-constant-out-of-range-compare
  -Wtautological-pointer-compare
  -Wtautological-overlap-compare
  -Wtautological-bitwise-compare
  -Wtautological-undefined-compare
  -Wtautological-objc-bool-compare
  -Wtautological-negation-compare
```

-Wtautological-compare

```
bool should_have_been_logical_or(int a)
{
    return a < -1 && a > 2;
}
```

```
$ clang -fsyntax-only -Wtautological-compare test.c

test.c:3:16: warning: overlapping comparisons always evaluate to false
    3 |         return a < -1 && a > 2;
      |                ~~~~~~~^~~~~~~
1 warning generated.
```

-Wtautological-compare

```
bool impossible_check(unsigned char a)
{
    return a > 256;
}
```

```
$ clang -fsyntax-only -Wtautological-constant-out-of-range-compare test.c

test.c:3:11: warning: result of comparison of constant 256 with expression of type 'unsigned char' is always false
    3 |         return a > 256;
      |                ~ ^ ~~~
1 warning generated.
```

Link time optimization (LTO)

- Optimization technique where `clang` compiles C code to LLVM bitcode instead of machine code so that `ld.lld` can perform additional optimizations on the entire program, such as inlining across translation units (which may allow additional optimizations at the cost of code size) or dead code elimination

- Two different modes selectable from Kconfig
  - `CONFIG_LTO_CLANG_FULL` (Full LTO): more optimization potential but slower to link (especially as kernel image size increases) due to single threading
  - `CONFIG_LTO_CLANG_THIN` (ThinLTO): faster from multithreading and gets most of the optimization of full LTO

Kernel Control Flow Integrity (kCFI)

- Functions can be indirectly called through pointers, which can be dangerous if an attacker is able to get control of one of those pointers, as they might be able to arbitrarily call functions
- Control flow integrity restricts the number of functions that an indirect call can target by checking that the prototype of the called function matches the expected prototype of the indirect call at run time; some warnings exist to highlight some of these issues at compile time
- See `lkdtm_CFI_FORWARD_PROTO()` in `drivers/misc/lkdtm/cfi.c` for simple example of what problematic code might look like in the kernel
- Enable in Kconfig: `CONFIG_CFI_CLANG`

Kernel Control Flow Integrity (kCFI)

Can run that same test from LKDTM to verify that kCFI works. Using `echo | cat` ensures that your shell does not get killed from the test failing.

```
$ echo CFI_FORWARD_PROTO | cat >/sys/kernel/debug/provoke-crash/DIRECT
[ 2761.562356] lkdtm: Performing direct entry CFI_FORWARD_PROTO
[ 2761.562750] lkdtm: Calling matched prototype ...
[ 2761.562861] lkdtm: Calling mismatched prototype ...
[ 2761.563199] CFI failure at lkdtm_indirect_call+0x2c/0x44 (target:
lkdtm_increment_int+0x0/0x18; expected type: 0x7e0c52a5)
[ 2761.563984] Internal error: Oops - CFI: 00000000f2008228 [#1] PREEMPT
SMP
```

Stack variable initialization

- As previously seen, using variables while they are uninitialized is very easy to do in C. This is always a logic problem but it has security implications as well, as stack contents may be unintentionally revealed!

- Enter `-ftrivial-auto-var-init=` to automatically initialize stack variable contents deterministically!
  - `-ftrivial-auto-var-init=zero` initializes all variables to zero
  - `-ftrivial-auto-var-init=pattern` initializes all variables with an obvious, repeated pattern (like 0xAA or 0xFE)

Stack variable initialization

- Two configuration options to set the requested mode: `CONFIG_INIT_STACK_ALL_PATTERN` and `CONFIG_INIT_STACK_ALL_ZERO`
- Not a substitution for listening to `-Wuninitialized` or `-Wsometimes-uninitialized` and properly initializing stack variables! It aims to mitigate the security impact of these problems but there may still be a logic bug, leading to unexpected (rather than undefined) behavior
- May have a performance impact, especially pattern (`clang-18+` may mitigate some of that). `__uninitialized` attribute macro can be used to opt out on a per-variable basis, or disable it for a whole translation unit

```
CFLAGS_unit.o := $(call cc-option, -ftrivial-auto-var-init=uninitialized)
```

Static analyzer

- Clang has a separate static analyzer that can find many more problems than the compile time warnings can. However, they are often not full compiler warnings because their false positive rate is much higher

- Two Kbuild targets to run different checks (very noisy right now, I'd recommend outputting to a file for easier filtering and fixing)

```
$ make clang-analyzer
$ make clang-tidy
```

See implementation in [scripts/clang-tools/run-clang-tools.py]

Static analyzer

```
drivers/bus/mhi/host/main.c:1546:17: warning: Value stored to 'dev' during its
initialization is never read [clang-analyzer-deadcode.DeadStores]
 1546 |         struct device *dev = &mhi_cntrl->mhi_dev->dev;
      |                        ^~   ~~~~~~~~~~~~~~~~~~~~~~~~
drivers/bus/mhi/host/main.c:1546:17: note: Value stored to 'dev' during its
initialization is never read
 1546 |         struct device *dev = &mhi_cntrl->mhi_dev->dev;
      |                        ^~   ~~~~~~~~~~~~~~~~~~~~~~~~
```

Configuration dependent problem but could possibly be fixed by changing the first argument of the call to `pr_err()` in `mhi_mark_stale_events()`? First patch, anyone? :)

Get involved

- Issue tracker
- Mailing list: llvm@lists.linux.dev (web archive)
- `#clang-built-linux` in LLVM's Discord
- `#clangbuiltlinux` on Libera
- Video meeting every two weeks (calendar, meeting link)

LF live MENTORSHIP SERIES

Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.